Micca: Translating XUML Models

A Single Threaded Model Execution Domain Targeting "C"

Andrew Mangogna

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	November 20, 2015	Initial coding.	GAM
1.0a1	January 24, 2017	Release of 1.0a1.	GAM
1.0a2	February 8, 2017	Release of 1.0a2.	GAM
1.0b1	April 4, 2017	Release of 1.0b1.	GAM
1.0b2	June 5, 2017	Release of 1.0b2.	GAM
1.0b3	June 24, 2017	Release of 1.0b3.	GAM
1.0b4	July 1, 2017	Release of 1.0b4.	GAM
1.0b5	August 1, 2017	Release of 1.0b5.	GAM
1.0b6	August 12, 2017	Release of 1.0b6.	GAM
1.0	September 18, 2017	Release of 1.0.	GAM
1.0.1	October 12, 2017	Release of 1.0.1. Clean up of conditional compilation for ARM targets in the run-time code.	GAM
1.0.2	November 1, 2017	Added conditional compilation to help with compilers that do not meet C11 standards.	GAM
1.1	December 5, 2017	Added ability to specify zero initialized attribute. This helps deal with problems of specifying more complexly structured attributes which can then be initialized in a constructor if necessary.	GAM
1.1.1	December 13, 2017	Corrected a problem in the code generation for assigning zero initialized attributes. Corrected a problem in the code generation for reclassifying subclass instances.	GAM

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.1.2	December 30, 2017	Corrected problem with error message on missing attributes. Corrected problem with default values on required attribute population.	GAM
1.1.3	March 11, 2018	Improvements in the sync queue processing for the run time code.	GAM
1.1.4	August 17, 2018	Correction to runtime code to fix possible dereference of a NULL pointer when creating simple relationship links for associative classes and to fix logic associated with determining whether there are duplicated association class instances.	GAM
1.1.5	March 20, 2019	Corrected a problem with the way instance counts were handled for union subclasses. The instance count in the class structure was set at the number of subtype instances in the initial instance population. This failed to account for subtype migration and that external instance ID numbering is based on the instances of the ultimate supertype of a generalization.	GAM
1.1.6	June 16, 2019	Corrected a problem with the declaration of the storage pool for multiple assigners. Minor improvements in man page documents.	GAM
1.1.7	December 5, 2019	Corrected an error in the generated code for conditional relationship traversal. The code assumed a loop control structure was generated and used a break statement. This was not the case for singular relationships.	GAM
1.1.8	April 5, 2020	Trim white space on type names given to the typealias command. This caused unintended parsing errors. Improved code generation for the findByIdInstance command by introducing another temporary variable.	GAM

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.2.0	November 16, 2020	Corrected an error in referential integrity checking for union based generalizations. The case of a subclass instance having been deleted and its superclass instance remaining was not caught. Corrected a problem with multiple assigners partitioned by a class which is also a subclass of a union-based generalization. Added referential integrity checks for domain operations. Now referential integrity is enforced both at the end of a thread of control and after a domain operation.	GAM
1.3.0	December 31, 2020	Add the ability to signal events periodically. This implied changes to the run time and an additional embedded command to access the feature. The run time handling of expiring the delayed event queue was also changed to simplify the processing and allow periodic events to be requeued immediately upon expiration.	GAM
1.3.1	March 17, 2021	Corrected an error when parsing "C" type names of the form: "data-type [identifier]" where "type" is not a primitive type specifier (e.g. char). These declarations occur in the context of variable length array arguments, typically as function arguments.	GAM
1.3.2	June 21, 2021	Reworked the run time code to be more precise about how threads of control are executed. In particular, foreground synchronization now takes place only on thread of control boundaries. Previously, foreground synchronization happened after each event was processed. This potentially caused the sync functions to execute at a time when the domain data could have been inconsistent. Since most sync functions invoke domain operations, executing such a sync function could have resulted in a referential integrity error.	GAM

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.3.3	March 21, 2023	Corrected a problem in the delayed event queue operation. When the queue is stopped, it is possible to lose the race with the timer and discover that the remaining time for the event at the head of the queue is actually zero. This can happen when the platform specific function which stops the timing resource happens just when the timer expires or the residual time is less than 1 ms. Such a situation leaves caused the delay time for the first delayed event in the queue to have a delay time of 0, with the queue timing stopped. This triggered an assertion. The solution is to expire the delayed timing queue before restarting it. This solution also allows for properly ordering delayed event posts with a time value of 0.	GAM
1.4.0	April 2, 2023	Adapted micca platform model to operate with rosea version 1.11.1. This version of rosea insists that all association that are not total functions must use an associative class to realize the association. There were only three associations affected, but the changes were required in the area where state and event signatures were checked for matching.	GAM

Contents

I Translation 1

1 Introduction 2

Translation Reference Model . . . 2

Micca Translation Model . . . 4

Comparing Translation Techniques . . . 5

II An Example 6

2 Controlling a Washing Machine 8

Introduction . . . 8

Translation Overview . . . 8

Domain Data . . . 9

Washing Machine Class . . . 10

Washing Cycle Class . . . 11

Clothes Tub Class . . . 11

Water Valve Class . . . 12

Motor Class . . . 12

Water Level Sensor . . . 13

Domain Dynamics . . . 13

Washing Machine State Model . . . 13

Clothes Tub State Model . . . 15

Domain Processing . . . 17

Washing Machine State Activities . . . 17

Clothes Tub State Activities . . . 21

Class Operations . . . 26

Washing Machine Operations . . . 26

Washing Cycle Operations . . . 27

Instance Operations . . . 27

Domain Operations . . . 28
Initialize . . . 30
Initial Instance Population . . . 30
Stubbing the External Operations . . . 32
Running the Example . . . 33
Example Program . . . 34
Example Run Results . . . 34

III The Micca Platform 36

3 Domain Subsystem 38
Introduction . . . 38
Domain . . . 40
Domain Element . . . 41
Checking for "C" Identifiers . . . 41
R1 . . . 42
Class . . . 42
Relationship . . . 43
R2 . . . 43
Domain Operation . . . 44
R5 . . . 45
Domain Operation Parameter . . . 45
R6 . . . 46
External Entity . . . 46
External Operation . . . 47
R10 . . . 48
External Operation Parameter . . . 48
R11 . . . 49
Constructor . . . 49
R8 . . . 50
Destructor . . . 50
R9 . . . 51
Operation . . . 51
R3 . . . 53
Operation Parameter . . . 53
R4 . . . 54
Type Alias . . . 54
R7 . . . 55

4 Classes Subsystem **56**

Introduction . . . 56
Class Component . . . 58
R20 . . . 59
R25 . . . 59
Populated Component . . . 59
Generated Component . . . 60
R21 . . . 60
Attribute . . . 61
Independent Attribute . . . 61
Dependent Attribute . . . 62
R29 . . . 63
R19 . . . 63
Value Initialized Attribute . . . 63
Zero Initialized Attribute . . . 64
Default Value . . . 65
R22 . . . 65
Reference . . . 65
R23 . . . 66
Association Reference . . . 66
Associator Reference . . . 67
Superclass Reference . . . 68
R24 . . . 68
Complementary Reference . . . 68
Link Container . . . 69
Subclass Container . . . 70
Subclass Reference . . . 71
R26 . . . 71
Singular Reference . . . 71
Array Reference . . . 72
Link Reference . . . 73
R27 . . . 73
R28 . . . 74
Forward Reference . . . 74
Backward Reference . . . 75

5 Relationship Subsystem **76**

Mapping Referential Attributes to Pointers . . . 76
Naming Conventions on Relationship Traversal . . . 95
Conventions Used for Class Based Associations . . . 95
Class Diagram . . . 96
Association . . . 98
Generalization . . . 98
R30 . . . 99
R31 . . . 99
Class Based Association . . . 99
Simple Association . . . 100
R43 . . . 100
Reference Generalization . . . 101
Union Association . . . 101
Simple Referring Class . . . 102
R32 . . . 103
Simple Referenced Class . . . 103
R33 . . . 104
Source Class . . . 104
R34 . . . 105
Target Class . . . 106
R35 . . . 107
Associator Class . . . 107
R42 . . . 108
R38 . . . 108
Destination Class . . . 108
Referenced Superclass . . . 109
R36 . . . 110
Referring Subclass . . . 110
R37 . . . 111
Union Superclass . . . 111
R44 . . . 112
Union Subclass . . . 112
R45 . . . 113
Superclass . . . 113
R48 . . . 114
Subclass . . . 115
R47 . . . 115
Class Role . . . 116
R40 . . . 116
R41 . . . 117

6 References Subsystem **118**
Introduction . . . 118
R90 . . . 120
R91 . . . 120
R94 . . . 120
R95 . . . 120
R92 . . . 121
R93 . . . 121
R96 . . . 121

7 State Model Subsystem **122**
Introduction . . . 122
State Model . . . 124
R58 . . . 125
R59 . . . 125
R50 . . . 125
Instance State Model . . . 125
R51 . . . 126
Assigner State Model . . . 126
R52 . . . 127
State . . . 127
R55 . . . 128
Creation State . . . 128
R56 . . . 129
State Place . . . 129
R57 . . . 130
Transition Rule . . . 130
R53 . . . 131
Single Assigner . . . 131
Multiple Assigner . . . 132
R54 . . . 132

8 Event Subsystem **133**
Introduction . . . 133
Event . . . 136
R80 . . . 137
Deferred Event . . . 137
R81 . . . 138
Deferral Path . . . 138

R86 . . . 139
Transitioning Event . . . 139
R82 . . . 140
R87 . . . 140
Polymorphic Event . . . 140
Inherited Event . . . 141
Mapped Event . . . 142
R84 . . . 143
Local Event . . . 143
R83 . . . 144
NonLocal Event . . . 144
R85 . . . 145

9 Transition Subsystem 146
Introduction . . . 146
Transition Place . . . 148
R70 . . . 149
R71 . . . 149
State Transition . . . 149
R72 . . . 150
NonState Transition . . . 150
R73 . . . 151
Argument Signature . . . 152
Parameter Signature . . . 152
R76 . . . 153
R78 . . . 154
R69 . . . 155
Argument . . . 155
R75 . . . 156
Parameter . . . 156
R77 . . . 157
R79 . . . 158

10 Population Subsystem 159
Introduction . . . 159
Population . . . 161
R100 . . . 161
Value Element . . . 161
R104 . . . 162

Element Population . . . 162
R101 . . . 163
R105 . . . 163
Class Population . . . 163
Assigner Population . . . 164
Class Instance . . . 164
R102 . . . 165
Class Component Value . . . 165
R103 . . . 166
R109 . . . 167
Specified Component Value . . . 167
Unspecified Component Value . . . 168
Multiple Assigner Instance . . . 169
R106 . . . 170
R107 . . . 170
R108 . . . 170

IV Configuring a Domain 171

11 Configuration Commands 173
Configure . . . 173
ConfigureFromChan . . . 173
ConfigureFromFile . . . 174
Clearing the Configuration . . . 174

12 Configuration Namespace Layout 176
Evaluating Configuration Scripts . . . 176
Clearing the Micca Model Population . . . 178

13 Defining a Domain 180
Domain . . . 180

14 Defining Domain Components 182
Interface . . . 182
Prologue . . . 183
Epilogue . . . 183
Append To Domain Attribute . . . 184
Class . . . 184
Association . . . 185
Generalization . . . 194
External Entity . . . 197
Type Alias . . . 198
Domain Operation . . . 199

15 Defining Class Components **200**
Attribute . . . 200
Polymorphic Events . . . 202
Propagating Polymorphic Events . . . 203
Class Operation . . . 212
Instance Operation . . . 212
Define Operation . . . 213
Constructor . . . 214
Destructor . . . 215

16 Defining Class State Models **216**
Statemodel . . . 216
State . . . 218
State Parameters . . . 220
FindParameterSignature . . . 221
FindArgumentSignature . . . 223
Event . . . 224
Transition . . . 225
Initialstate . . . 227
Defaulttrans . . . 228
Final . . . 228

17 Defining Assigners **229**
Assigner State Model . . . 230
Identify By Class . . . 231

18 Defining External Entities **232**
External Operation . . . 232

19 Populating a Domain **234**
Population . . . 234
Class . . . 236
Class Storage Allocation . . . 237
Specifying Instance Values . . . 238
Setting Instance Values . . . 239
Resolving Initial Values . . . 243
Tables of Instance Values . . . 243
Populating Multiple Assigners . . . 245
Assigner . . . 245
Populating An Assigner Instance . . . 246

Verifying Population Integrity . . . 247
Checking Completeness . . . 247
Checking Class Allocation . . . 248
Checking Reference Values . . . 250
Checking Association Reference Values . . . 250
Checking Associator Reference Values . . . 252
Checking Subclass Reference Values . . . 257
Calculating Generated Component Values . . . 259
Calculating Complementary Reference Values . . . 260
Singular Reference Values . . . 261
Singular Backward Reference Values . . . 261
Singular Forward Reference Values . . . 263
Array Reference Values . . . 265
Union Subclass Numbering . . . 272

20 Handling Configuration Errors 274

21 Helper Commands 279
DeclError . . . 279
Generate Numbers . . . 280
Parse C Type Names . . . 280
Checking for Duplicates . . . 283
Checking for Existance . . . 283

V Runtime Support 284

22 Introduction to the Micca Run Time 286
Limitation of the Run Time . . . 287
Conditional Compilation . . . 287

23 The Main Program 289
Runtime Use Cases . . . 290
Event Loop . . . 290
Dispatching a Thread of Control . . . 293
Running a Thread of Control . . . 295
Dispatching a Thread of Control Event . . . 296

24 Managing Data **298**
Instance Data . . . 298
Class Data . . . 299
Instance Allocation . . . 302
Instance Allocation Block . . . 302
Finding Instance Memory . . . 303
Instance Containment . . . 305
Creating an Instance . . . 307
Initializing Instance Memory . . . 310
Deleting an Instance . . . 311
Iterating Over Class Instances . . . 313
Instance Sets . . . 315
Iterating Over Instance Sets . . . 324

25 Managing Referential Integrity **329**
Describing Relationships . . . 330
Association Participants . . . 330
Simple Associations . . . 331
Class Based Associations . . . 331
Superclasses in a Generalization . . . 332
Reference Generalizations . . . 332
Union Generalizations . . . 333
Relationship Properties . . . 333
Data Transactions . . . 334
Synchronous Service Support . . . 334
Recording Transaction Details . . . 336
Verifying Referential Integrity . . . 340
Counting References . . . 345
Operations on Relationship Instances . . . 351
Creating Relationship Links . . . 353
Create Associator Links . . . 358
Deleting Relationship Linkage . . . 361
Reclassifying Subclasses . . . 366

26 Managing Execution **369**
State Machine Rules . . . 369
Event Types . . . 370
Event Control Block . . . 370
Event Parameter Storage . . . 371

Event Queues . . . 372
Event Signaling . . . 375
Obtaining An ECB . . . 375
Posting an Event . . . 376
One Step Event Signaling . . . 377
Asynchronous Instance Creation . . . 378
Delayed Events . . . 381
Posting A Delayed Event . . . 382
Posting A Periodic Event . . . 383
One Step Delayed Event Signaling . . . 384
One Step Periodic Event Signaling . . . 385
Inserting Delayed Events . . . 385
Removing Delayed Events . . . 386
Canceling Delayed Events . . . 386
Time Remaining for a Delayed Event . . . 387
Expired Events in the Delayed Event Queue . . . 389
Timing Considerations . . . 392
Event Dispatch . . . 393
Dispatching An Event From a Queue . . . 393
Transition Event Dispatch . . . 395
Polymorphic Event Dispatch . . . 398
Polymorphic Event Mapping . . . 399
Creation Event Dispatch . . . 402

27 Bridging Domains **404**
Portal Data Structures . . . 405
Portal Access Functions . . . 406
Portal Errors . . . 406
References to Attributes . . . 408
Reading Attributes . . . 410
Updating Attributes . . . 411
Signaling Events . . . 413
Signaling Delayed Events . . . 414
Canceling Delayed Events . . . 416
Remaining Delay Time . . . 417
Synchronous Instance Creation . . . 418
Asynchronous Instance Creation . . . 419
Deleting Instances . . . 420
Signaling Events To Assigners . . . 421
Obtaining Class Current State . . . 422
Obtaining Assigner Current State . . . 423
Domain Introspection . . . 424

28 Asynchronous Execution **435**
Sync Function and Parameters . . . 436
Sync Queue . . . 437
Dispatching Synchronization Functions . . . 439

29 Event Dispatch Tracing **440**
Trace Information . . . 440
Common Trace Data . . . 441
Transition Event Trace Data . . . 441
Polymorphic Event Trace Data . . . 441
Creation Event Trace Data . . . 442
Access to Trace Information . . . 442
Obtaining Tracing Output . . . 444
Tracing Strategies . . . 446

30 Error Handling **448**
Avoiding Fatalities . . . 453
Checking for Available Instance Space . . . 454
Checking for Event Blocks . . . 454
Non-Fatal Background Synchronization . . . 455
Causing Fatalities . . . 455

31 Linked List Operations **457**

32 POSIX Specific Code **459**
POSIX Critical Sections . . . 459
POSIX Timing Interfaces . . . 461
POSIX Async Execution Interface . . . 463
POSIX I/O Interface . . . 464
POSIX Suspending Execution . . . 467
POSIX Tracing . . . 469
POSIX Initialization . . . 470
POSIX Include Files . . . 470
POSIX Files . . . 470
Run Time Header Header for POSIX . . . 470
Run Time Code File for POSIX . . . 471

33 ARM v7-M Specific Code **473**
ARM v7-M Constants . . . 473
ARM v7-M Critical Sections . . . 473
ARM v7-M Async Execution Interface . . . 474
ARM v7-M Initialization . . . 474
EFM32GG Includes . . . 475
EFM32GG Timing Interfaces . . . 475
EFM32GG Initialization . . . 477
EFM32GG Suspending Execution . . . 478
EFM32GG Tracing . . . 479
EFM32GG Files . . . 479
MSP432 Includes . . . 480
MSP432 Timing Interfaces . . . 481
MSP432 Initialization . . . 482
MSP432 Suspending Execution . . . 483
MSP432 Tracing . . . 483
MSP432 Files . . . 483

34 MSP430 Specific Code **486**
MSP430 Critical Sections . . . 486
MSP430 Timing Interfaces . . . 486
MSP430 Async Execution Interface . . . 490
MSP430 Suspending Execution . . . 490
MSP430 Initialization . . . 491
MSP430 Tracing . . . 491
MSP430 Files . . . 491

35 Common Header File Definitions **493**

36 Internal Header File **494**

VI Generating “C” Code **496**

37 Introduction **498**
Generating a Domain . . . 498
Generating Run Time Data . . . 499
Common Class Queries . . . 501

38 Header Files **507**

Header File Guard . . . 508
Domain Interface . . . 508
Type Aliases . . . 508
Domain Operation Declarations . . . 510
External Operation Declarations . . . 511
Event Parameter Declarations . . . 511
Portal Function Constants . . . 512
Portal Data Structure Declaration . . . 515

39 Generating Code Files **516**

Including the Domain Header File . . . 518
Domain Name as a String . . . 518
Domain Prologue . . . 518
Implementation Type Aliases . . . 518
Forward Class Declarations . . . 519
Forward Relationship Declaration . . . 519
Class Declarations . . . 520
Assigner Declarations . . . 524
State Parameter Declarations . . . 525
Operation Declarations . . . 526
Constructor Declarations . . . 526
Destructor Declarations . . . 527
Formula Declarations . . . 527
Activity Declarations . . . 527
Storage Declarations . . . 528
Name Definitions . . . 529
Instance Allocation Block Definitions . . . 530
Event Dispatch Block Definitions . . . 532
Polymorphic Dispatch Block Definitions . . . 536
Class Description Definitions . . . 538
Assigner Definitions . . . 541
Relationship Description Definitions . . . 543
Class Instance Definitions . . . 548
Assigner Instance Definitions . . . 554
Operation Definitions . . . 555
Constructor Definitions . . . 556
Destructor Definitions . . . 557
Formula Definitions . . . 557

Activity Definitions . . . 558
Domain Constructor Definition . . . 560
Domain Operation Definitions . . . 561
External Operation Definitions . . . 563
Portal Data Definitions . . . 564
Epilogue Declarations . . . 564

40 Generating Activity Code 565
Expanding the Embedded Activity Commands . . . 565
Creating the Embedded Commands . . . 566
Symbol Table . . . 571
Verifying Symbols . . . 573
Tracking Code Blocks . . . 575
Expansion Context . . . 576
Template Expansion of Activity Code . . . 577
Invoking External Operations . . . 578
Instance Macro Commands . . . 579
Reading Instance Attribute Values . . . 579
Updating Instance Attribute Values . . . 580
Assigning Instance Attributes to Variables . . . 581
Deleting Instances . . . 582
Invoking Instance Based Operations . . . 583
Iterating Across Related Instances . . . 584
Conditional Iteration Across Related Instances . . . 592
Conditionally Find a Single Related Instance . . . 593
Finding a Single Related Instance . . . 594
Selecting Related Instances . . . 596
Conditionally Selecting Related Instances . . . 597
Signaling an Event . . . 598
Delayed Signaling of an Event . . . 600
Periodic Signaling of an Event . . . 600
Cancel a Delayed Signal . . . 601
Time Remaining for a Delayed Event . . . 602
Obtaining an Instance Identifier . . . 603
Instance Set Commands . . . 604
Iterating Over an Instance Set . . . 604
Selecting an Arbitrary Instance from an Instance Set . . . 604
Testing for an Empty Set . . . 605
Testing for an Non-Empty Set . . . 605

Number of Instances in a Set . . . 606
Instance Set Equality . . . 606
Instance Set Inequality . . . 607
Adding to an Instance Set . . . 607
Removing from an Instance Set . . . 607
Instance Set Membership . . . 608
Instance Set Union . . . 608
Instance Set Intersection . . . 609
Instance Set Difference . . . 609
Class Commands . . . 610
Synchronous Instance Creation . . . 610
Synchronous Instance Creation in a Given State . . . 615
Asynchronous Instance Creation . . . 616
Iterating Over Class Instances . . . 619
Conditional Iteration Over Class Instances . . . 620
Searching Class Instances . . . 621
Conditionally Selecting Class Instances . . . 622
Declaring Instance Reference Variables . . . 623
Converting an ID to an Instance Reference . . . 623
Finding an Initial Instance by Name . . . 624
Invoking a Class Operation . . . 624
Creating an Instance Set . . . 625
Relationship Commands . . . 625
Association Commands . . . 626
Assigner Commands . . . 631

41 Helper Commands **633**

VII Code Organization **636**

42 Legal Information **638**
Copyright Information . . . 638
Version Information . . . 639
Edit Warning . . . 639

43 Source Code **640**
Micca Starpack Application . . . 641

VIII Reference Materials **644**

A Literate Programming **645**

Bibliography **646**

Books 646

Articles 646

Index **647**

List of Figures

1.1 Automatic Translation Reference Model . . . 3
1.2 Micca Translation Workflow . . . 4

2.1 Washing Machine Class Diagram . . . 9
2.2 Washing Machine State Model Diagram . . . 14
2.3 Clothes Tub State Model Diagram . . . 16

3.1 Domain Subsystem Class Diagram . . . 39

4.1 Classes Subsystem Class Diagram . . . 57

5.1 Simple Non-Reflexive 1:1 Association . . . 77
5.2 Simple Reflexive 1:1 Association . . . 78
5.3 Simple Non-Reflexive M:1 Dynamic Association . . . 79
5.4 Simple Reflexive M:1 Dynamic Association . . . 80
5.5 Simple Non-Reflexive M:1 Static Association . . . 81
5.6 Simple Reflexive M:1 Static Association . . . 82
5.7 Class Based Non-Reflexive 1:1 Association . . . 83
5.8 Class Based Reflexive 1:1 Association . . . 84
5.9 Class Based Non-Reflexive M:1 Dynamic Association . . . 85
5.10 Class Based Reflexive M:1 Dynamic Association . . . 86
5.11 Class Based Non-Reflexive M:1 Static Association . . . 87
5.12 Class Based Reflexive M:1 Static Association . . . 88
5.13 Class Based Non-Reflexive M:M Dynamic Association . . . 89
5.14 Class Based Reflexive M:M Dynamic Association . . . 90
5.15 Class Based Non-Reflexive M:M Static Association . . . 91
5.16 Class Based Reflexive M:M Static Association . . . 92
5.17 Reference Based Generalization . . . 93
5.18 Union Based Generalization . . . 94
5.19 Decomposition of Class Based Associations . . . 96
5.20 Relationship Subsystem Class Diagram . . . 97

6.1 References Subsystem Class Diagram 119

7.1 State Model Subsystem Class Diagram 123

8.1 Event Subsystem Class Diagram 135

9.1 Transition Subsystem Class Diagram 147

10.1 Population Subsystem Class Diagram 160

23.1 Micca Run Time Event Loop 291

List of Tables

2.1 Washing Machine Transition Table 15
2.2 Clothes Tub Transition Table 16

Preface

This book is about a program named, `micca`. `Micca` is program that translates a platform dependent specification of an Executable UML domain into "C" code. It is part of an strategy to translate xUML models into code.

This book is also a literate program. As such it contains all the source code for `micca` interspersed with design information. The literate program syntax used is similar to the "noweb" approach. This document provides several roots for extraction of code. This includes not only the code for `micca` itself, but also for the run-time code and test cases.

There are four major components to `micca`.

1. A platform model that describes how the translation will operate on the target platform.
2. A domain specific language (DSL) that is used to populate the platform model.
3. Run-time code that provides data and execution sequencing according to the xUML execution rules.
4. An embedded command language that is used by domain actions to bridge between model level operations and the implementation provided by run-time code.

This book is divided into eight parts:

Part I — Translation
An overview of xUML translation and how `micca` fits into the workflow.

Part II — An Example
An example domain, translated by `micca`.

Part III — The Micca Platform
The platform model for `micca`.

Part IV — Configuring a Domain
The configuration DSL used to define a domain to `micca`.

Part V — Runtime Support
The "C" code for the run-time component.

Part VI — Generating "C" Code
`Micca` generates data structures to use with the runtime and expands embedded commands used to interface domain actions to the run-time code.

Part VII — Code Organization
The organization of `micca` code files.

Part VIII — Reference Materials
Additional background and reading materials.

This book contains a lot of material. You are not expected to read it front to back. Skipping around when reading this much material is encouraged. For readers new to xUML, the example in Part II is a good place to start. Those readers with more exposure to similar translation schemes (*e.g.* `pycca` or `rosea`) may find Part IV a good starting point in order to compare the configuration DSL between the various schemes. For hard core "C" programmers, Part V on the run-time code has some interesting aspects as to how referential integrity is enforced. For those more interested in how platform model data may be queried and used to generate "C" code, Part VI gives the details of how the embedded macro commands for domain actions are translated into code.

This book is a literate program. See Appendix A for a description of the literate programming syntax. The syntax is not complicated, but you will need to know it to make much sense of any code sequences in the book. Being a literate program document means that all the source code for `micca` is included here. The `micca` program (and many other components) is built by extracting the source from the document source. Including the source code and all the design discussion makes the text rather long and skipping around when reading is encouraged.

Part II of the book presents a simple example. This will give you a general sense of how model translation with `micca` works.

After the example, we consider the four major components of `micca`:

1. The platform model. This is a complete model of the platform specifics that the translation targets. This model is populated by the micca configuration process and is the information source for the code generation.
2. The configuration language is a command language where the elements of a domain are defined.
3. The run-time code implements in "C" the model execution rules and any other things that are not directly supported by the implementation language.
4. Finally, the code generation portion creates the data needed by the run-time code and expands the embedded commands in domain activities to "C" code.

Part I

Translation

Chapter 1

Introduction

In this section, we give an overview of model translation and how `micca` fits into the larger scheme of modeling and translation. First, we consider an idealized view of software development by modeling and translation. Next, we compare how a `micca` translation workflow compares to the idealized one.

Translation Reference Model

Figure 1.1 shows an idealized workflow for translating and executable model.

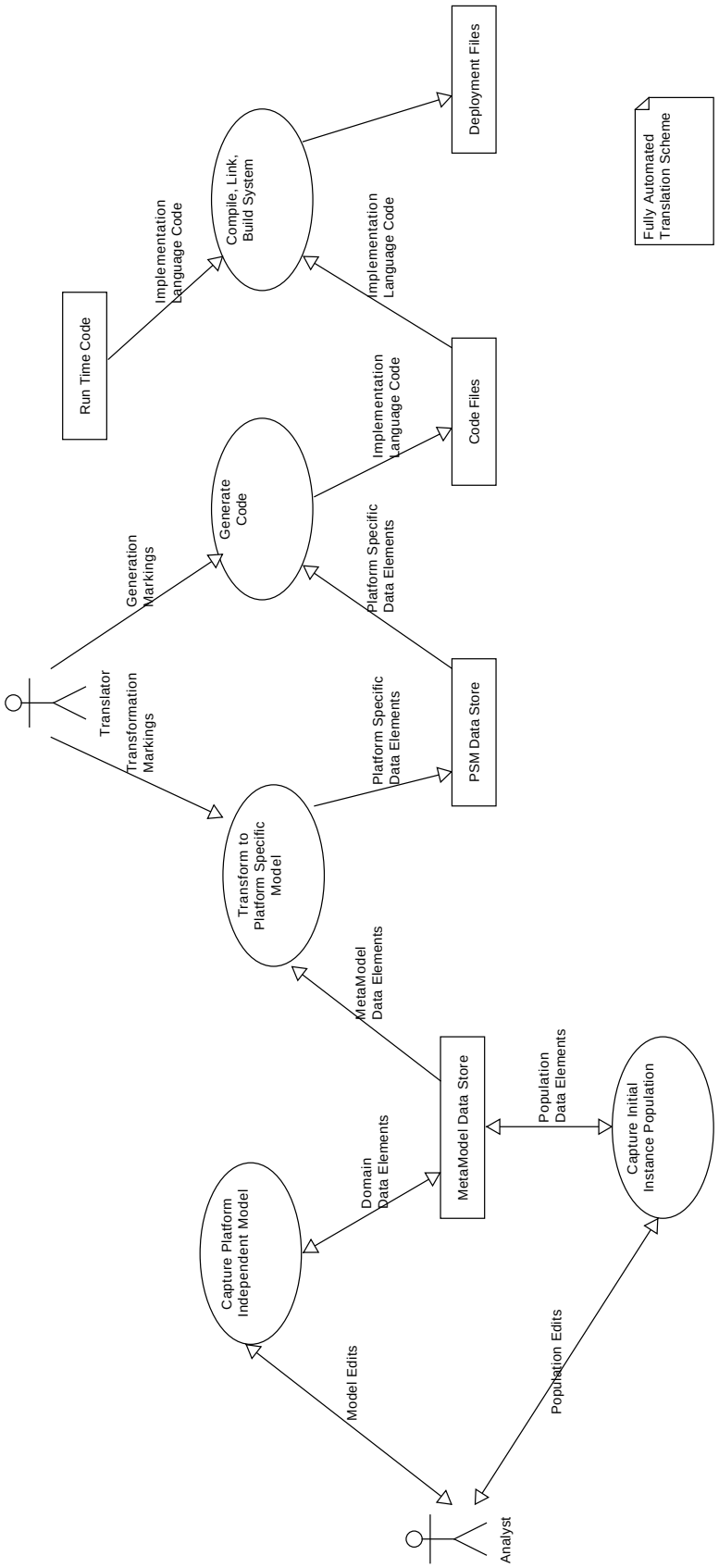


Figure 1.1: Automatic Translation Reference Model

Micca Translation Model

Figure 1.2 shows the workflow when `micca` is used to accomplish the translation of an executable model.

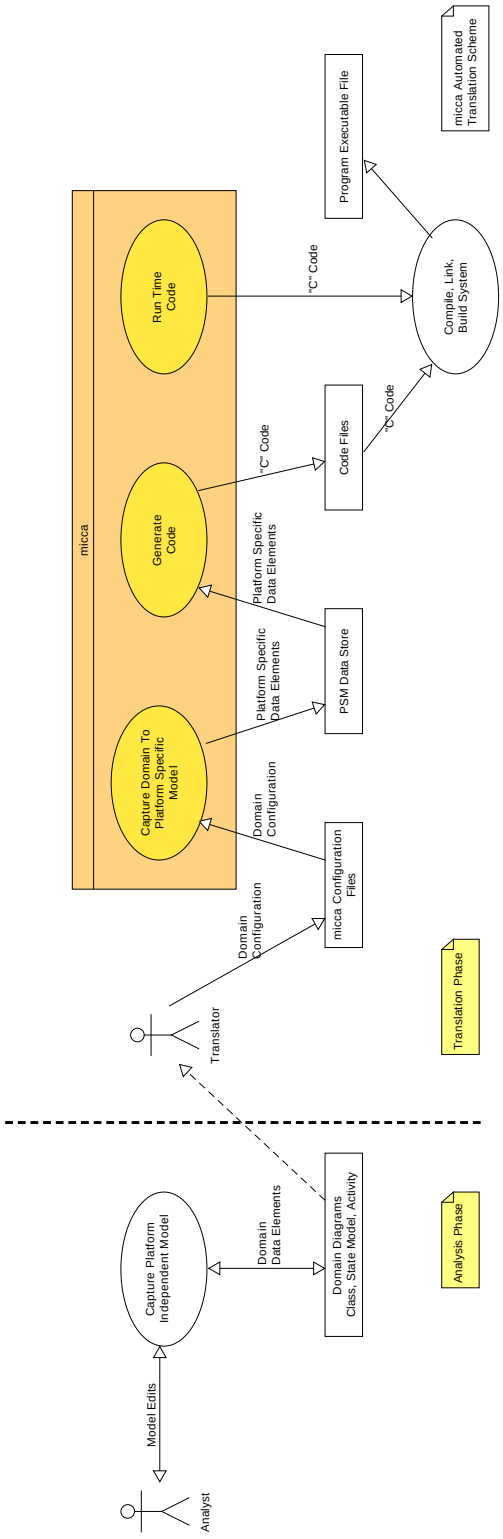


Figure 1.2: Micca Translation Workflow

Comparing Translation Techniques

When comparing the two workflows, note that `micca` requires a human to play significant role in the translation. None of the information about the plaform independent models is transferred into the translation scheme. Rather, a human translator uses the modeling artifacts to construct a domain configuration file. The domain configuration file is an ordinary text file that consists of commands that describe the elements of the domain. For example, as class might be defined as:

```
class WashingCycle {
    attribute CycleType char[32] ;
    attribute WashTemp int ;
}
```

In the process of preparing the domain configuration file, the human translator performs two distince roles:

1. Structural information about the domain elements is translated to the configuration file syntax. State models require little implementation consideration since the states and transitions are captured in a declarative way. Class definitions require addition consideration as some attributes from the platform independent model may not have a role in a platform specific model.
2. State activites are translated from the action language of the model to “C” code and use an embedded command language to provide model level actions. For example, navigating a relationship might appear as:

   ```
   <%my findOneRelated dog R14%>
   if (dog != NULL) {
       // related dog found code
   } else {
       // no related dog was found
   }
   ```

Once the domain model is transcribed to the `micca` domain configuration language, the remaining process is similar to the fully automated workflow. `Micca` parses and populates a platform specific model store and generates the required code to interface to run-time code and the expand the embedded commands within the domain activities.

The obvious disadvantage of this approach is lack of integration to the front end modeling tools. Strangely, it is also one of the strengths of the approach. UML modeling tools vary considerably in what and how they draw the various UML diagram types. In xUML, we are interested in only a few diagram types and define a much different set of semantics for the meaning of the graphical symbols than conventional UML. Finding a tool that expressing the precise modeling semantics of the xUML dialect in which we are interested is difficult. General agreement on the xUML meta-model does not exist and that meta-model is pivital in any automated translation from a platform independent model to a platform specific one. Here, we have simply side-stepped the entire issue by using a human translator to transcribe the platform independent model into a domain specific language that maps to the plaform dependent model.

Part II

An Example

In this part, we show an example of a simple domain being translated using `micca`.

Chapter 2

Controlling a Washing Machine

Introduction

It is traditional in technical books to start off with a simple example. This example is not quite as simple as the classic *Hello World!* example but suffices to give a summary of many of the characteristics of `micca` translated domain. We anticipate that there will be many questions after reading the example and those questions are answered in later chapters.

Unfortunately, there is rather a lot of background material that we do not cover here. We do not explain how to create an XUML model nor do we spend much time explaining why the example model was designed the way it was. There are many good books that explain XUML in detail and teach you the basics of modeling[1]. We suggest you read at least one of them. However, if your role is to perform the translation of a model, then the details of how the model was created are not particularly important. As long as you understand basic model execution rules, the translation can be done ignoring the details of what the model actually does. As long as you preserve all the application logic in the translation, the details of that logic are are secondary. It is also the case that this package represents one particular piece of a larger workflow and this means that readers who may not be completely familiar with model-driven, translation-oriented development may have many questions that remain.

The subject matter of our example is an automatic clothes washer. This is a very simple washing machine, especially compared to modern commercially available washers. The intent is to select a subject that most people would be familiar with from ordinary experience so that we don't have to devote too much time explaining the problem. One word of caution. This model is an example for pedagogical purposes and probably has little correspondence with the way *real* washing machines operate or are designed to operate. You can also notice a lack of any attention paid to *what can go wrong*. For industrial strength programs, handling probable failure cases is very important but we have dispensed with those considerations here to focus on how the model is translated into the implementation code using `micca`.

Translation Overview

To translate an XUML model using `micca`, the model is encoded in a domain specific language. You may draw the model with your favorite UML drawing tool. This example was done using UMLet as a drawing tool. For this translation, we choose to keep the domain description in one file and an initial instance population in another file. This is a common way to be able to provide several different populations to the same domain, an important consideration for keeping testing and deployment separate.

In the file for the domain, we use the `domain` command to specify the details of the model.

```
<<wmctrl.micca>>=
<<copyright info>>

domain wmctrl {
    <<WM class>>
    <<WC class>>
    <<CT class>>
```

[1] Mellor and Balcer, Chris Raistrick et.al and Leon Starr all are worthy of a close reading.

```
    <<WV class>>
    <<MTR class>>
    <<WLS class>>
    <<wmctrl relationships>>
    <<wmctrl operations>>
    <<external operations>>
    <<type aliases>>
    epilogue {
        <<main program>>
    }
}
```

Within the `domain` script, we specify the details of the classes, relationship and operations.

The `population` command is used to specify the initial class instances. These are instances that exist when the domain starts to execute.

```
<<wmctrl_pop.micca>>=
population wmctrl {
    <<WM population>>
    <<WC population>>
    <<CT population>>
    <<MTR population>>
    <<WV population>>
    <<WLS population>>
}
```

Domain Data

The first facet of a domain to translate is always the data. The data defines the basis on which the code can operate. The figure below shows a class diagram for the washing machine control domain in UML graphical notation.

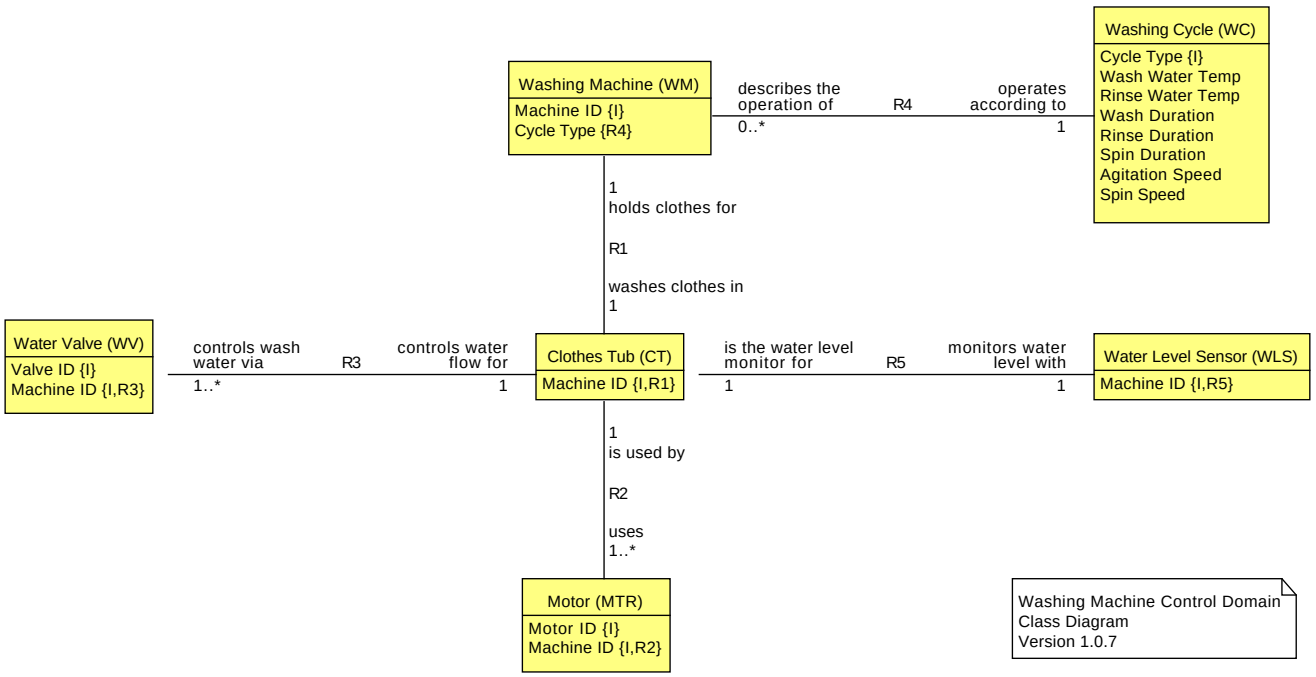


Figure 2.1: Washing Machine Class Diagram

In our world, a Washing Machine operates according to some Washing Cycle. The Washing Cycle is a set of parameters that specifies aspects of the washing that turns dirty clothes into clean ones. The Washing Machine itself has a Clothes Tub into which the dirty laundry is placed. There are also Water Valves to control the flow of water into and out of the Clothes Tub and Motors to run a water pump, agitate the Clothes Tub and rotate the Clothes Tub to spin excess water out of the clean laundry. Rounding out the machinery, there is a Water Level Sensor that tells us when the Clothes Tub is filled with water or empty of water.

For a well engineered model, the class diagram must also have a set of descriptions of what the attributes and relationship actually mean, what the value domains of the attributes are and many other aspects that describe how the problem is represented in the model. These descriptions are vital to understanding a class diagram. Here again, in the interests of space, we have to suffice ourselves with more casual descriptions included along with the example as it translated into the implementation and an admonition that writing the model descriptions is an essential aspect of a well engineered solution.

The class diagram shows the static aspects of our domain and is always the first aspect of the model that must be considered when deriving the implementation. The class diagram facet of the model is static in the sense that at any point in time, the logical predicates you can infer from the diagram are true. The class attributes hold those properties of the domain whose values can vary and the relationships state how the entities abstracted by the classes are associated with each other. Later, we visit the dynamic and algorithmic facets of the domain but, for now, we endeavor to represent the classes and relationships in terms of the `micca` constructs.

Washing Machine Class

By examining the graphics of the model we can see that the Washing Machine class has attributes, operations and a state model.

```
<<WM class>>=
class WashingMachine { # ❶
    attribute MachineID {char[32]}

    statemodel {
        <<WM statemodel>>
    }

    <<WM operations>>
}
```

❶ The basic anatomy of a class specification is the same. Classes usually (but not always) have attributes. Active classes have a state model. Some classes define class-based or instance-based operations as a means of factoring common code into one location. We have chosen to include the state model and operations as literate program chunks to highlight the way the components of a class are specified.

Looking at the Washing Machine class on the class diagram, we see that there are two attributes. The MachineID attribute is an identifier and we make its data type a simple string. Usually, attributes that are used purely for identification purposes are excluded from the implementation of the domain. In this case, Machine ID is used for other purposes and so must be retained as an attribute of the Washing Machine class. `Micca` makes up its own identifier for instances of the class. Since we still want Machine ID to be an identfier, we must take on that responsibility. We need a class operation that can search the instances of the Washing Machine class to determine if one already exists with a given Machine ID value.

The CycleType attribute is referential attribute and we do not need it. Generally, attributes that are purely referential in nature are excluded as attributes when a class is defined to `micca`. This is because `micca` generates its own identifier for instances and assumes the responsibility to store the necessary information to realize relationships. Since in this case the Cycle Type attribute is only used to realize relationship, R4, we need not include this attribute and can let `micca` manage the information needed for R4.

While we are discussing R4, we can define its characteristics. We can represent the participating classes, multiplicity and conditionality as presented in the class diagram by the following domain configuration statement.

```
<<wmctrl relationships>>=
association R4 WashingMachine 0..*--1 WashingCycle
```

The syntax of the `association` command is intended to follow directly from the model graphic. In this case, `R4` is an association where instances of the `WashingMachine` class refer to instances of the `WashingCycle` class. Specifically, each instance of the `WashingMachine` class refers to exactly one instance of the `WashingCycle` class and, conversely, each instance of the `WashingCycle` class is referred to by zero or more instances of the `WashingMachine` class. The `micca` run time insures that this declarative statement of the multiplicity and conditionality of the association instances remains true during the run time of the program.

Washing Cycle Class

Examining the class diagram and other graphics for the domain shows that the Washing Cycle class is a simple class consisting entirely of descriptive attributes.

```
<<WC class>>=
class WashingCycle {
    attribute CycleType {char[32]}
    attribute WashWaterTemp WaterTemp_t
    attribute RinseWaterTemp WaterTemp_t
    attribute WashDuration unsigned
    attribute RinseDuration unsigned
    attribute SpinDuration unsigned
    attribute AgitationSpeed WashSpeed_t
    attribute SpinSpeed WashSpeed_t

    <<WC operations>>
}
```

We need only specify the attributes and their data types. In order to determine the proper data types of the attributes we would need to consult the descriptive text associated with the class diagram. As we have no such text in this example, we infer a set of data types based on how the attribute values are to be used in the processing. We can declare the data types to `micca`; in this case some enumerations.

```
<<type aliases>>=
typealias WaterTemp_t {enum {WT_Hot, WT_Cold, WT_Warm}}
typealias WashSpeed_t {enum {WS_Low, WS_Medium, WS_High}}
```

Clothes Tub Class

By now the pattern of examining the model graphics to determine the class characteristics is established. The Clothes Tub class is an active class with a state model. A Clothes Tub only has identifying and referential attributes and they are elided as `micca` provides the necessary identifier and reference storage.

```
<<CT class>>=
class ClothesTub { # ❶
    statemodel {
        <<CT statemodel>>
    }
    <<CT operations>>
}
```

❶ Note in this case there are no attributes of `ClothesTub`. The `MachineID` attribute shown in the class diagram is both an identifier and referential. Since we only ever refer to instances of `ClothesTub` as they apply to their corresponding `WashingMachine` this single attribute can be eliminated and `micca` assumes the responsibility of identifying the instances and insuring enough information is stored to navigate the `R1` association.

The `R1` association itself can be transcribed from the graphic as follows. `R1` states that there is an exact one-to-one correspondence between `ClothesTub` instances and `WashingMachine` instances. This is formal statement of a fundamental rule of the Washing Machine physical design.

```
<<wmctrl relationships>>=
association R1 ClothesTub 1--1 WashingMachine
```

Water Valve Class

The Water Valve class represents the means to control the movement of water into and out of a Clothes Tub. Since a Clothes Tub can have several valves, we need an attribute to distinguish them. So, we keep the `ValveID` attribute. The `MachineID` attribute is discarded as it is purely referential.

```
<<WV class>>=
class WaterValve {
    attribute ValveID ValveType_t

    <<WV operations>>
}
```

The type of a valve comes from a fixed, well-known set.

```
<<type aliases>>=
typealias ValveType_t {enum {WV_Hot, WV_Cold, WV_Drain}}
```

The R3 relationship represents the design of the washing machine hardware that includes three valves on each machine. Notice the we don't concern ourselves here with whether a washing machine has three or three thousand valves on it. That is dealt with when we populate the class instances. The important point here is that there is one or more valves associated with each tub and each water valve is connected to exactly one tub. The exact number is not represented in the model because the logic of the model does not change if there are three or three thousand water valves attached to the Washing Machine. Of course, when we implement computing operations on the Water Valves, whether there are three or three thousand might have a very significant impact on the algorithms we might choose.

```
<<wmctrl relationships>>=
association R3 WaterValve 1..*--1 ClothesTub
```

Motor Class

The Motor class is similar to the Water Valve class.

```
<<MTR class>>=
class Motor {
    attribute MotorID MotorType_t

    <<MTR operations>>
}
```

Our washer has three types of motors.

```
<<type aliases>>=
typealias MotorType_t {enum {MTR_Pump, MTR_Agitator, MTR_Spin}}
```

The R2 relationships states the hardware design of using one or more motors to operation a clothes tub.

```
<<wmctrl relationships>>=
association R2 Motor 1..*--1 ClothesTub
```

Water Level Sensor

The Water Level Sensor class is also simple. It has no attributes. It is closely associated with the Clothes Tub for which it senses.

```
<<WLS class>>=
class WaterLevelSensor {
    <<WLS operations>>
}
```

It is R5 that sets the rule about one Water Level Sensor per Clothes Tub.

```
<<wmctrl relationships>>=
association R5 WaterLevelSensor 1--1 ClothesTub
```

Domain Dynamics

The second facet of the model that is considered during translation is the dynamics. The model encodes the sequences of domain execution as state models attached to classes. In this model, there are two classes for which state models are defined, Washing Machine and Clothes Tub. The control is partitioned by the model between these classes. The Washing Machine class is given the responsibility for coordinating the actions needed to clean clothes according to the particulars specified by the Washing Cycle. In this example, that coordination primarily involves timing the various components of the washing cycle. The Clothes Tub class deals with sequencing the mechanics of the washer to perform a specific activity. Together, the classes coordinate their activities to achieve clothes washing.

For each state model, we present a graphical representation of the state model. We also show the state transition table. It is important to have both representations. In the graphic, it is conventional not to show ignored or error transitions. However, in the transition table all possible transitions and their outcomes are exposed.

Washing Machine State Model

The state model for the Washing Machine class is shown below. Notice in the graphic that the states contain language statements that specify the processing to be performed when the state is entered. We say more about the state activities when we take up the domain processing below. For now, it is convenient to have the actions present on the diagram in order to better understand exactly what the washing machine does as it responds to events and thereby better understand how the state machines achieve the overall result of producing clean clothes.

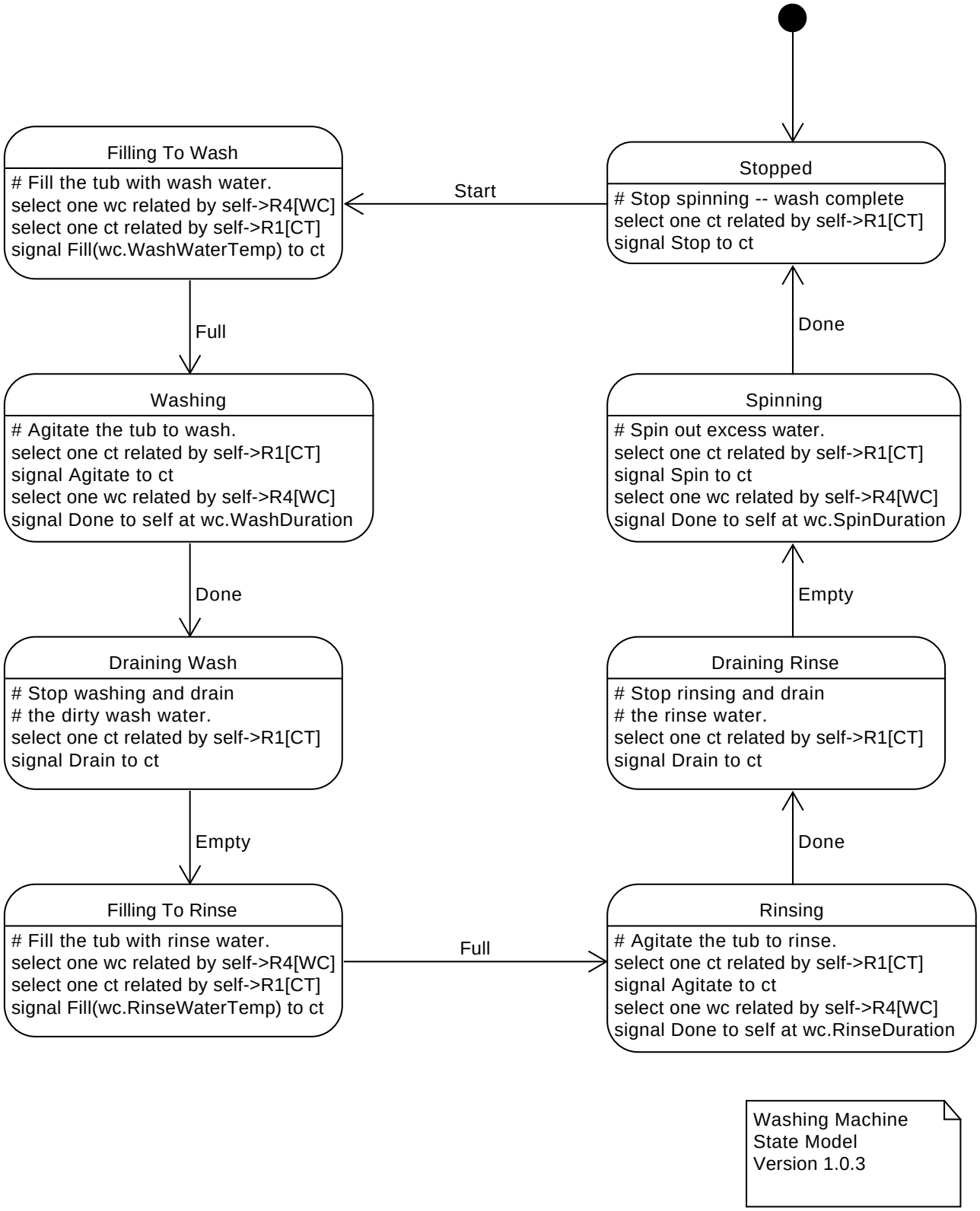


Figure 2.2: Washing Machine State Model Diagram

From the diagram we can see how the washing machine cycles in a rather simple circular form filling, washing, draining, rinsing and spinning to convert dirty clothes into clean ones. The state activities are primarily concerned with obtaining the particular details of the next step and requesting the Clothes Tub to carry out that step. The various durations are handled by signaling delayed events to march things along to the next step in the cleaning.

The transition table corresponding to the diagram is shown next.

Table 2.1: Washing Machine Transition Table

	Start	**Full**	**Done**	**Empty**
Stopped	Filling To Wash	CH	CH	CH
Filling To Wash	CH	Washing	CH	CH
Washing	CH	CH	Draining Wash	CH
Draining Wash	CH	CH	CH	Filling To Rinse
Filling To Rinse	CH	Rinsing	CH	CH
Rinsing	CH	CH	Draining Rinse	CH
Draining Rinse	CH	CH	CH	Spinning
Spinning	CH	CH	Stopped	CH

The transition table may be directly translated into the required `micca` configuration statements. Below, we have not shown state activity code. We return to the processing below. For now, we simply transliterate the transition table into the required `transition` statements.

```
<<WM statemodel>>=
initialstate Stopped
defaulttrans CH

transition Stopped - Start -> FillingToWash

transition FillingToWash - Full -> Washing

transition Washing - Done -> DrainingWash

transition DrainingWash - Empty -> FillingToRinse

transition FillingToRinse - Full -> Rinsing

transition Rinsing - Done -> DrainingRinse

transition DrainingRinse - Empty -> Spinning

transition Spinning - Done -> Stopped
```

Clothes Tub State Model

The diagram below shows the state model for the Clothes Tub class. This model is *not* circular like that of the Washing Machine class. There are two paths through the states. One corresponds to agitating the tub for the purposes of either washing or rinsing. The other path corresponds to spinning the tub to remove excess water from the clean clothes. Both paths start at the Empty state.

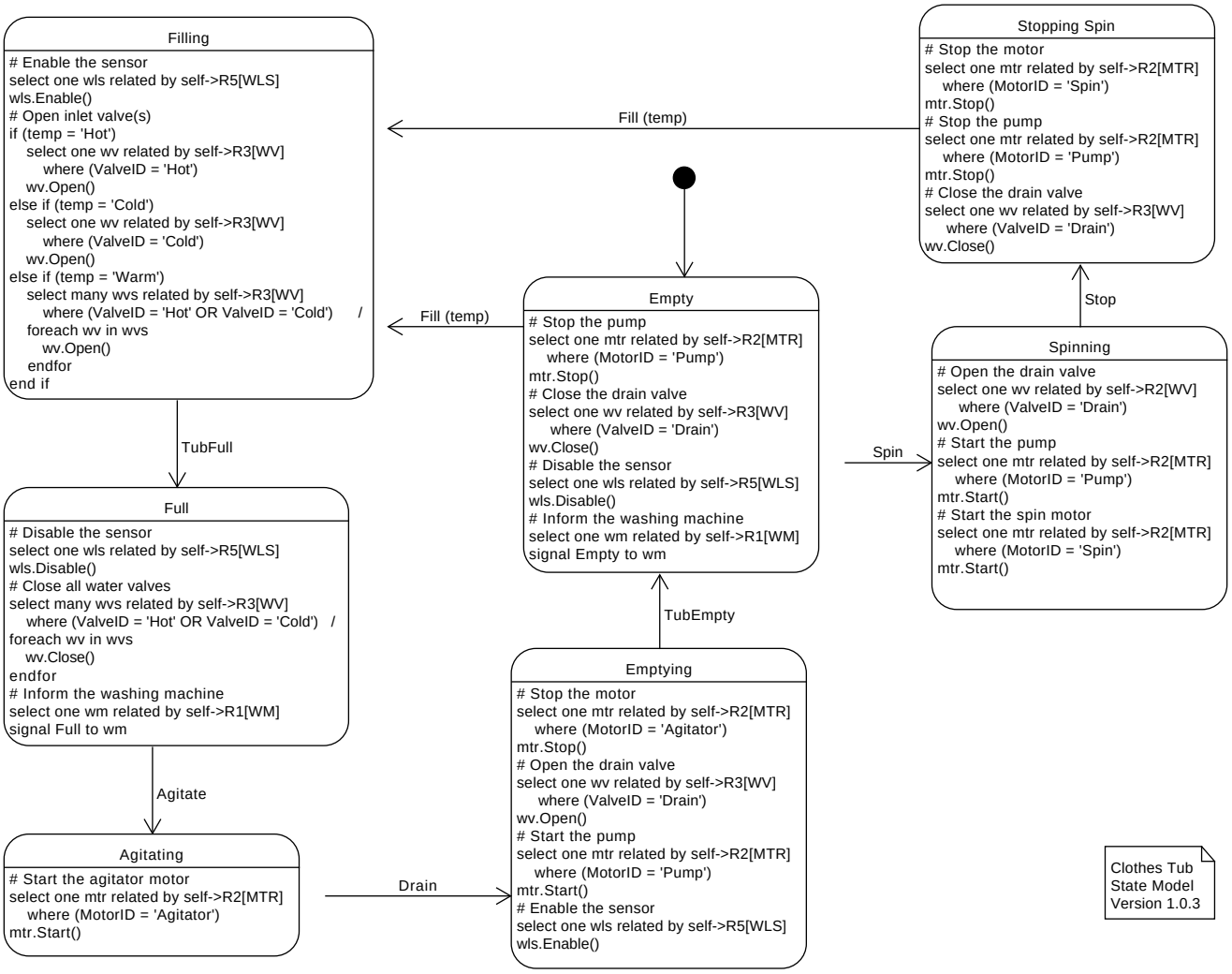


Figure 2.3: Clothes Tub State Model Diagram

The transition table follows immediately from the diagram.

Table 2.2: Clothes Tub Transition Table

	Fill	**TubFull**	**Agitate**	**Drain**	**TubEmpty**	**Spin**	**Stop**
Empty	Filling	CH	CH	CH	CH	Spinning	CH
Filling	CH	Full	CH	CH	IG (1)	CH	CH
Full	CH	CH	Agitating	CH	CH	CH	CH
Agitating	CH	CH	CH	Emptying	CH	CH	CH
Emptying	CH	IG (1)	CH	CH	Empty	CH	CH
Spinning	CH	CH	CH	CH	CH	CH	Stopping Spin
Stopping Spin	Filling	CH	CH	CH	CH	CH	CH

(1) For the **Filling** and **Emptying** states, we allow for the fact that when the water level sensor is enabled it may send events that reflect its current state. So we simply ignore those sensor events in which we are not currently interested.

The `micca` configuration statements follow directly from the transition table.

```
<<CT statemodel>>=
initialstate Empty
defaulttrans CH

transition Empty - Fill -> Filling
transition Empty - Spin -> Spinning

transition Filling - TubFull -> Full
transition Filling - TubEmpty -> IG

transition Full - Agitate -> Agitating

transition Agitating - Drain -> Emptying

transition Emptying - TubEmpty -> Empty
transition Emptying - TubFull -> IG

transition Spinning - Stop -> StoppingSpin

transition StoppingSpin - Fill -> Filling
```

Domain Processing

The third facet of the model that must be translated is the processing. Processing is executed in state activities or the various operations of the domain or classes. We represent the processing as action language. There are several different action languages that are in use. In the example, we follow an established syntax for the actions. All the action languages allow for writing expressions and for performing model level processing. Since we are not automatically parsing the action language in this context, we are not particular about the syntax as long as the action statements convey the required processing unambiguously. In the `micca` scheme, action language translation is done manually with substantial support from `micca` for specifying model level types of processing.

For state activities, the diagrams above show the action language that is to be executed when the state is entered. In the translations below, we duplicate the action language from the state and write the translation to "C" code immediately following.

The model level actions performed by the state activities, such as access to attribute values, navigating the class diagram or signaling events, is accomplished by invoking embedded macro commands that are expanded into "C" code. The commands are contained between the "<%" and "%>" markers. All the content between the markers (and the markers themselves) is replaced with the "C" code that implements the model level execution semantics implied by the embedded macro command. Otherwise, ordinary "C" code is passed along unmodified. The `micca` scheme uses directly coded "C" for flow of control, variables and expression evaluation. Access to model level execution actions is provided by the embedded macro commands.

In the code below, we strive for a clear correspondence between the action language of the activity and its translation into "C" and macro commands. This is not the only translation possible and not necessarily the most efficient. The intent is to show the correspondence between the actions and the translation into "C". In the interest of clarity, no attempt has been made to shorten or optimize the "C" code. There are cases where some variable assignments might seem superfluous.

Washing Machine State Activities

Below is the processing for the Washing Machine state activities. For the first few states, we make several comments on the correspondence between the action language statements and the "C" code. After the translation pattern is established, we present the remaining state with minimal explanation.

The Stopped state of the Washing Machine is entered when the cycle is complete and we must signal the Clothes Tub to stop spinning. The action consists of finding the related Clothes Tub instance and signaling the Stop event to it.

Stopped Activity Action Language

```
# Stop spinning -- wash complete
select one ct related by self->R1[CT]
signal Stop to ct
```

Stopped State Implementation

```
<<WM statemodel>>=
state Stopped {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Stop%>

    // This code is not part of the model activity. ❶
    MRT_InstId selfid = <%my instid%> ;
    <%NOTIFY complete washerid selfid%> ;
}
```

❶ We have added a call to an external operation to notify the outside world that the washer had completed it cycle. This is strictly for the benefit of running the example. Below we show how this is used to terminate the example run.

All state activities have an implicit `self` argument defined for them. `Self` is an reference to the instance upon which the activity is operating. The `<%my findOneRelated ct ~R1%>` command navigates from `self` along `~R1` and assigns the resulting Clothes Tub instance into a variable named `ct` that is typed as a reference to a Clothes Tub. With the Clothes Tub instance in hand, the `Stop` event is signaled to it using the `signal` command. There are a couple things to note here.

1. The use of the direction syntax, *i.e.* `~R1`, says that we navigate the `R1` relationship in the **reverse** direction. Supplying a direction obviates the need to say that we are navigating to an instance of the `ClothesTub` class in the manner that the action language statement does. Because of the syntax convention and the manner in which the association was defined, the system knows the destination class.
2. The question arises whether the instance reference returned from `findOneRelated` in the above traversal across `~R1` can ever be the empty reference or ever reference multiple instances. The answer is no! The `R1` association, as shown in the class diagram, is singular and unconditional on the Clothes Tub side. Every traversal from Washing Machine to Clothes Tub along `~R1` is guaranteed to return exactly one instance. Any operation in the domain that would perturb that state of affairs is rejected as violating the referential integrity of the class diagram. Consequently, no test of the multiplicity of the returned reference is necessary as the system is already making that test and so it would be strictly redundant. In general, traversing unconditional relationships never requires a test to determine if we obtained an instance reference and traversing a conditional relationship should always be followed by a test of the returned reference to see if it is empty. The fact that a relationship is conditional implies that there is to be conditional processing associated with traversing the relationship.

The Filling To Wash state is entered when the washing machine wishes to fill it clothes tub with water in preparation for a period of clothes washing. In the action, it is necessary to find out the water temperature that is to be used for washing. That information is stored as an attribute of the related washing cycle and passed as the `temp` parameter of the Fill event. We know the name of the parameter to the Fill event is `temp` because that is the name of the parameter of the Filling state into which the Fill event causes a transition.

Filling To Wash Activity Action Language

```
# Fill the tub with wash water.
select one wc related by self->R4[WC]
select one ct related by self->R1[CT]
signal Fill(wc.WashWaterTemp) to ct
```

Filling To Wash State Implementation

```
<<WM statemodel>>=
state FillingToWash {} {
    <%my findOneRelated wc R4%>
    <%my findOneRelated ct ~R1%>
    <%instance wc assign {WashWaterTemp washtemp}%>
    <%instance ct signal Fill temp washtemp%>   // ❶
}
```

❶ Note that embedded macros *cannot* be nested, so it is necessary to assign the wash water temperature to a variable. There is no efficiency concern here. Optimizations by the compiler remove any unneeded local variables.

While in the Washing state, clothes are being agitated in the tub. The state uses a delayed event to determine when the washing part of the cycle is finished. Note that the delay time of the signal is another attribute value obtained from the Washing Cycle class. Here Washing Cycle serves a role of holding attributes that just specify conditions for another class This is a common arrangement in models.

Washing Activity Action Language

```
# Agitate the tub to wash.
select one ct related by self->R1[CT]
signal Agitate to ct
select one wc related by self->R4[WC]
signal Done to self at wc.WashDuration
```

Washing State Implementation

```
<<WM statemodel>>=
state Washing {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Agitate%>
    <%my findOneRelated wc R4%>
    <%instance wc assign {WashDuration washtime}%>
    washtime *= 1000 ;
    <%my delaysignal washtime Done%> // ❶
}
```

❶ We are implicitly assuming the units of WashDuration are seconds. In truth, the units are minutes, but we are not patient enough to wait that long for the example to execute.

The delayed signal is accomplished by using the `my delaysignal` macro command. The event is directed to the `self` instance (the `my` command implies this), albeit delayed by the washing cycle time. So this state sends two signals, one to the Clothes Tub to tell it to start agitating the clothes and a delayed signal to itself so it when know when the clothes have been washing for the amount of time defined by the related Washing Cycle.

The remaining Washing Machine state activities follow the established pattern.

Draining Wash Activity Action Language

```
# Stop washing and drain
# the dirty wash water.
select one ct related by self->R1[CT]
signal Drain to ct
```

Draining Wash State Implementation

```
<<WM statemodel>>=
state DrainingWash {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Drain%>
}
```

Filling To Rinse Activity Action Language

```
# Fill the tub with rinse water.
select one wc related by self->R4[WC]
select one ct related by self->R1[CT]
signal Fill(wc.RinseWaterTemp) to ct
```

Filling To Rinse State Implementation

```
<<WM statemodel>>=
state FillingToRinse {} {
    <%my findOneRelated wc R4%>
    <%my findOneRelated ct ~R1%>
    <%instance wc assign {RinseWaterTemp rinsetemp}%>
    <%instance ct signal Fill temp rinsetemp%>
}
```

Rinsing Activity Action Language

```
# Agitate the tub to rinse.
select one ct related by self->R1[CT]
signal Agitate to ct
select one wc related by self->R4[WC]
signal Done to self at wc.RinseDuration
```

Rinsing State Implementation

```
<<WM statemodel>>=
state Rinsing {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Agitate%>
    <%my findOneRelated wc R4%>
    <%instance wc assign {RinseDuration rinsetime}%>
    rinsetime *= 1000 ;
    <%my delaysignal rinsetime Done%>
}
```

Draining Rinse Activity Action Language

```
# Stop rinsing and drain
# the rinse water.
select one ct related by self->R1[CT]
signal Drain to ct
```

Draining Rinse State Implementation

```
<<WM statemodel>>=
state DrainingRinse {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Drain%>
}
```

Spinning Activity Action Language

```
# Spin out excess water.
select one ct related by self->R1[CT]
signal Spin to ct
select one wc related by self->R4[WC]
signal Done to self at wc.SpinDuration
```

Spinning State Implementation

```
<<WM statemodel>>=
state Spinning {} {
    <%my findOneRelated ct ~R1%>
    <%instance ct signal Spin%>
    <%my findOneRelated wc R4%>
    <%instance wc assign {SpinDuration spintime}%>
    spintime *= 1000 ;
    <%my delaysignal spintime Done%>
}
```

Clothes Tub State Activities

The other state model in our example is for the Clothes Tub class

Because the Clothes Tub interacts frequently with its related Motor and Water Valve instances, we create some instance based operations to help us locate the required Motor or Water Valve instance. It is common to factor out such code as it usually corresponds to an action language `where` clause. This allows us to create type specific and type safe code for the query.

Factoring out the search code also allows us to control the algorithm used. Below we just use a simple linear search as the `micca` embedded macro commands provide iterators for the class instances. This works well for a small number of instances. However, if the number of instances was large or the frequency of the search was high, then you might consider a more sophisticated search technique such as binary searching or hash based searching.

```
<<CT operations>>=
instop {struct Motor *} findRelatedMotor {mtrid MotorType_t} {
    <%my findRelatedWhere mtr {mtr->MotorID == mtrid} ~R2%>
    assert(mtr != NULL) ;
    return mtr ;
}
```

```
<<CT operations>>=
instop {struct WaterValve *} findRelatedValve {valveid ValveType_t} {
    <%my findRelatedWhere valve {valve->ValveID == valveid} ~R3%>
    assert(valve != NULL) ;
    return valve ;
}
```

```
<<CT operations>>=
instop MRT_InstSet findRelatedWaterValves {} {
    <%my selectRelatedWhere watervalves vid\
            {vid->ValveID == WV_Hot || vid->ValveID == WV_Cold} ~R3%>

    assert(<%instset watervalves cardinality%> == 2) ;
    return watervalves ;
}
```

With these helper operations, the Clothes Tub state activities require much less boilerplate code.

Empty Activity Action Language

```
# Stop the pump
select one mtr related by self->R2[MTR]
    where (MotorID = 'Pump')
mtr.Stop()
# Close the drain valve
select one wv related by self->R3[WV]
     where (ValveID = 'Drain')
wv.Close()
# Disable the sensor
select one wls related by self->R5[WLS]
wls.Disable()
# Inform the washing machine
select one wm related by self->R1[WM]
signal Empty to wm
```

Empty State Implementation

```
<<CT statemodel>>=
state Empty {} {
    <%Motor refvar mtr%>
    mtr = <%my operation findRelatedMotor mtrid MTR_Pump%> ;
    <%instance mtr operation Stop%> ;

    <%WaterValve refvar wv%>
    wv = <%my operation findRelatedValve valveid WV_Drain%> ;
    <%instance wv operation Close%> ;

    <%my findOneRelated wls ~R5%>
    <%instance wls operation Disable%> ;

    <%my findOneRelated wm R1%>
    <%instance wm signal Empty%>
}
```

Filling Activity Action Language

```
# Enable the sensor
select one wls related by self->R5[WLS]
wls.Enable()
# Open inlet valve(s)
if (temp = 'Hot')
    select one wv related by self->R3[WV]
        where (ValveID = 'Hot')
    wv.Open()
else if (temp = 'Cold')
    select one wv related by self->R3[WV]
        where (ValveID = 'Cold')
    wv.Open()
else if (temp = 'Warm')
    select many wvs related by self->R3[WV]
        where (ValveID = 'Hot' OR ValveID = 'Cold')
    foreach wv in wvs
        wv.Open()
    endfor
end if
```

Filling State Implementation

```
<<CT statemodel>>=
state Filling {temp WaterTemp_t} {
    <%my findOneRelated wls ~R5%>
    <%instance wls operation Enable%> ;
    <%WaterValve refvar wv%>

    if (temp == WT_Hot) {
        wv = <%my operation findRelatedValve valveid WV_Hot%> ;
        <%instance wv operation Open%> ;
    } else if (temp == WT_Cold) {
        wv = <%my operation findRelatedValve valveid WV_Cold%> ;
        <%instance wv operation Open%> ;
    } else if (temp == WT_Warm) {
        <%WaterValve instset watervalves%>
        watervalves = <%my operation findRelatedWaterValves%> ;
        <%instset watervalves foreachInstance wv%>
            <%instance wv operation Open%> ;
        <%end%>
    }
}
```

Full Activity Action Language

```
# Disable the sensor
select one wls related by self->R5[WLS]
wls.Disable()
# Close all water valves
select many wvs related by self->R3[WV]
    where (ValveID = 'Hot' OR ValveID = 'Cold')
foreach wv in wvs
    wv.Close()
endfor
# Inform the washing machine
select one wm related by self->R1[WM]
signal Full to wm
```

Full State Implementation

```
<<CT statemodel>>=
state Full {} {
    // Disable the sensor
    <%my findOneRelated wls ~R5%>
    <%instance wls operation Disable%> ;

    // Close all water valves
    <%WaterValve instset watervalves%>
    watervalves = <%my operation findRelatedWaterValves%> ;
    <%instset watervalves foreachInstance wv%>
        <%instance wv operation Close%> ;
    <%end%>

    // Inform the washing machine
    <%my findOneRelated wm R1%>
    <%instance wm signal Full%>
}
```

Agitating Activity Action Language

```
# Start the agitator motor
select one mtr related by self->R2[MTR]
    where (MotorID = 'Agitator')
mtr.Start()
```

Agitating State Implementation

```
<<CT statemodel>>=
state Agitating {} {
    // Start the agitator motor
    <%Motor refvar mtr%>
    mtr = <%my operation findRelatedMotor mtrid MTR_Agitator%> ;
    <%instance mtr operation Start%> ;
}
```

Emptying Activity Action Language

```
# Stop the motor
select one mtr related by self->R2[MTR]
    where (MotorID = 'Agitator')
mtr.Stop()
# Open the drain valve
select one wv related by self->R3[WV]
     where (ValveID = 'Drain')
wv.Open()
# Start the pump
select one mtr related by self->R2[MTR]
    where (MotorID = 'Pump')
mtr.Start()
# Enable the sensor
select one wls related by self->R5[WLS]
wls.Enable()
```

Emptying State Implementation

```
<<CT statemodel>>=
state Emptying {} {
    // Stop the motor
    <%Motor refvar mtr%>
    mtr = <%my operation findRelatedMotor mtrid MTR_Agitator%> ;
    <%instance mtr operation Stop%> ;

    // Open the drain valve
    <%WaterValve refvar wv%>
    wv = <%my operation findRelatedValve valveid WV_Drain%> ;
    <%instance wv operation Open%> ;

    // Start the pump
    mtr = <%my operation findRelatedMotor mtrid MTR_Pump%> ;
    <%instance mtr operation Start%> ;

    // Enable the sensor
    <%my findOneRelated wls ~R5%>
    <%instance wls operation Enable%> ;
}
```

Spinning Activity Action Language

```
# Open the drain valve
select one wv related by self->R2[WV]
     where (ValveID = 'Drain')
wv.Open()
# Start the pump
select one mtr related by self->R2[MTR]
    where (MotorID = 'Pump')
mtr.Start()
# Start the spin motor
select one mtr related by self->R2[MTR]
    where (MotorID = 'Spin')
mtr.Start()
```

Spinning State Implementation

```
<<CT statemodel>>=
state Spinning {} {
    // Open the drain valve
    <%WaterValve refvar wv%>
    wv = <%my operation findRelatedValve valveid WV_Drain%> ;
    <%instance wv operation Open%> ;

    // Start the pump
    <%Motor refvar mtr%>
    mtr = <%my operation findRelatedMotor mtrid MTR_Pump%> ;
    <%instance mtr operation Start%> ;

    // Start the spin motor
    mtr = <%my operation findRelatedMotor mtrid MTR_Spin%> ;
    <%instance mtr operation Start%> ;
}
```

Stopping Spin Activity Action Language

```
# Stop the motor
select one mtr related by self->R2[MTR]
    where (MotorID = 'Spin')
mtr.Stop()
# Stop the pump
select one mtr related by self->R2[MTR]
    where (MotorID = 'Pump')
mtr.Stop()
# Close the drain valve
select one wv related by self->R3[WV]
     where (ValveID = 'Drain')
wv.Close()
```

Stopping Spin State Implementation

```
<<CT statemodel>>=
state StoppingSpin {} {
    // Stop the motor
    <%Motor refvar mtr%>
    mtr = <%my operation findRelatedMotor mtrid MTR_Spin%> ;
    <%instance mtr operation Stop%> ;

    // Stop the pump
    mtr = <%my operation findRelatedMotor mtrid MTR_Pump%> ;
    <%instance mtr operation Stop%> ;

    // Close the drain valve
    <%WaterValve refvar wv%>
    wv = <%my operation findRelatedValve valveid WV_Drain%> ;
    <%instance wv operation Close%> ;
}
```

Class Operations

Class operations are those defined to operate on the class as a whole. Consequently, class operation do *not* have an implied `self` variable. A common use for class operation is to select an instance of the class based on supplied attribute values (*i.e.* to implement `select from instances where` type action language statements).

Washing Machine Operations

The operation below searches the instances of the WashingMachine class for any instance which matches the `id` argument value.

```
<<WM operations>>=
classop {struct WashingMachine *} findByMachineID {id {char const *}} {
    <%WashingMachine findWhere wm {strcmp(id, wm->MachineID) == 0}%>
    return wm ;
}
```

The `<%WashingMachine findWhere wm%>` command expands to a loop to iterate across each instance of the Washing Machine class. The `wm` variable is assigned a reference (or pointer value) to an instance of Washing Machine. If the where clause (the invocation of `strcmp()` in this case) evaluates to true, then iteration stops and `wm` holds the matching machine. Otherwise, `wm` is `NULL` if no match is found.

Washing Cycle Operations

We construct a similar class operation for the Washing Cycle class. It is a common idom to construct such class operations for attributes that are identifiers of the class. This allows activities to verify an instance exists with a given value for the identifier or to insure that no duplicated instances are created.

```
<<WC operations>>=
classop {struct WashingCycle *} findByCycleType {cycleType {char const *}} {
    <%WashingCycle findWhere wc {strcmp(cycleType, wc->CycleType) == 0}%>
    return wc ;
}
```

Instance Operations

The Water Valve, Motor and Water Level Sensor classes have instance based operations that serve as an interface to external operations that are intended to perform the physical actions associated with the hardware. The external operations specify the detailed dependencies this domain assumes are performed by some other domain.

Water Valve Operations

```
<<WV operations>>=
instop void Open {} {
    MRT_InstId selfid = <%my instid%> ;
    <%VALVE open valveid selfid%> ;
}

instop void Close {} {
    MRT_InstId selfid = <%my instid%> ;
    <%VALVE close valveid selfid%> ;
}
```

Motor Class Operations

```
<<MTR operations>>=
instop void Start {} {
    MRT_InstId selfid = <%my instid%> ;
    <%MOTOR start motorid selfid%> ;
}

instop void Stop {} {
    MRT_InstId selfid = <%my instid%> ;
    <%MOTOR stop motorid selfid%> ;
}
```

Water Level Sensor Operations

```
<<WLS operations>>=
instop void Enable {} {
    <%my findOneRelated ct R5%>
    MRT_InstId ctid = <%instance ct instid%> ;
    <%SENSOR enable tubid ctid%> ;
}

instop void Disable {} {
    <%my findOneRelated ct R5%>
```

```
    MRT_InstId ctid = <%instance ct instid%> ;
    <%SENSOR disable tubid ctid%> ;
}
```

Domain Operations

In this section we show the code for the domain operations. We assume that there is some entity, such as a user interface, in the overall system that invokes these operations. To make our example run, we contrive to make that happen even though we do not intend to supply a user interface, *per se*.

Create Washer

Since there several classes involved in creating a functioning washer, a domain operation is useful to localize the construction of a washer.

```
<<wmctrl operations>>=
domainop int createWasher {washer {char const *}} {
    <%WashingMachine refvar wm%>
    wm = <%WashingMachine operation findByMachineID id washer%> ;
    if (wm == NULL) {
        <%WashingCycle findByName Normal nwc%>  // ❶
        <%WashingMachine create wm MachineID washer R4 nwc%>

        <%ClothesTub create ct R1 wm%>

        <%WaterValve create wv ValveID WV_Hot R3 ct%>
        <%WaterValve create wv ValveID WV_Cold R3 ct%>
        <%WaterValve create wv ValveID WV_Drain R3 ct%>

        <%Motor create mtr MotorID MTR_Pump R2 ct%>
        <%Motor create mtr MotorID MTR_Agitator R2 ct%>
        <%Motor create mtr MotorID MTR_Spin R2 ct%>

        <%WaterLevelSensor create wls R5 ct%>
        return <%instance wm instid%> ;
    } else {
        MRT_DEBUG("washer, %s, already exists\n", washer) ;
        return -1 ;
    }
}
```

❶ By default, the Washing Cycle is set to `Normal`.

Delete Washer

If we can create a washer, it is also necessary to be able to delete one. Here we must be careful to leave the data model referentially consistent.

```
<<wmctrl operations>>=
domainop bool deleteWasher {washer {char const *}} {
    bool status ;
    <%WashingMachine refvar wm%>
    wm = <%WashingMachine operation findByMachineID id washer%> ;
    if (wm != NULL) {
        <%instance wm findOneRelated ct ~R1%>
        <%instance ct findOneRelated wls ~R5%>
        <%instance wls delete%>
```

```
        <%instance ct foreachRelated wv ~R3%>
            <%instance wv delete%>
        <%end%>
        <%instance ct foreachRelated mtr ~R2%>
            <%instance mtr delete%>
        <%end%>
        <%instance ct delete%>
        <%instance wm delete%>
        status = true ;
    } else {
        MRT_DEBUG("unknown washer, %s\n", washer) ; // ❶
        status = false ;
    }

    return status ;
}
```

❶ `micca` provides a `printf` style "C" preprocessor macro for debugging output.

Start Washer

To start a washing machine we must supply the identifier of the washer so we can know which one is to be started. Although our instance population only included a single WashingMachine instance, as we stated before, the models run with an arbitrary number of washing machine instances.

The implementation of the operation first searches all the instances of WashingMachine to find the correct one to start. It is possible to request an unknown washer to start. After finding the correct instance, the **Start** event kicks things off.

```
<<wmctrl operations>>=
domainop bool startWasher {washer {char const *}} {
    bool status ;
    <%WashingMachine refvar wm%>
    wm = <%WashingMachine operation findByMachineID id washer%> ;
    if (wm != NULL) {
        <%instance wm signal Start%>
        status = true ;
    } else {
        MRT_DEBUG("unknown washer, %s\n", washer) ;
        status = false ;
    }

    return status ;
}
```

Select Cycle

On the class diagram, relationship `R4` determines which Washing Cycle is used to control the operations. Selecting a wash cycle means we must reform the `R4` relationship, *i.e.* relate the existing cycle to a different one. The implementation of the `selectCycle` domain operation does just that.

```
<<wmctrl operations>>=
domainop void selectCycle {washer {char const *} cycle {char const *}} {
    <%WashingMachine refvar wm%>
    wm = <%WashingMachine operation findByMachineID id washer%> ;
    if (wm == NULL) {
        MRT_DEBUG("unknown washer, %s\n", washer) ;
        return ;
    }
```

```
    <%WashingCycle refvar reqCycle%>
    reqCycle = <%WashingCycle operation findByCycleType cycleType cycle%> ;
    if (reqCycle == NULL) {
        MRT_DEBUG("unknown washing cycle, %s\n", cycle) ;
        return ;
    }

    <%R4 reference wm reqCycle%>
}
```

Initialize

Most domains need to have an initialization operation that can be invoked as the system is started. Here we start all the washing machines in the initial instance population.

```
<<wmctrl operations>>=
domainop void init {} {
    <%WashingMachine foreachInstance wm%>
        <%instance wm signal Start%>
    <%end%>
}
```

Initial Instance Population

In this section we supply an initial instance population for the domain. `Micca` arranges for the initial instances to be in place when the domain begins execution.

We intend to have only a single WashingMachine instance and have chosen to populate it directly rather than invoke `createWasher` in order to show how an initial instance population is specified. Note that the model runs correctly regardless of how many washing machines we are trying to control even though we are creating only a single instance here.

```
<<WM population>>=
class WashingMachine {
    instance wm1 MachineID {"wm1"} R4 PermPress ; # ❶
}
```

❶ Note that although we do not have any referential attributes in the WashingMachine class, we must still account for how the R4 relationship is to be handled. Here we state that the WashingMachine instance named, “wm1”, is related across R4 to the WashingCycle instance named, “PermPress”. The `PermPress` name is of our own invention and is the instance name in the initial population. It is defined below. Notice that in the class model for the domain, the Washing Machine class had a referential attribute that referenced the Washing Cycle class. The `R4` specification here accomplishes the same purpose, namely, this instance of Washing Machine references a specific instance of Washing Cycle.

```
<<WC population>>=
class WashingCycle {
    table {
        CycleType WashWaterTemp RinseWaterTemp
        WashDuration RinseDuration SpinDuration
        AgitationSpeed SpinSpeed
    } Normal {
        {"Normal"} WT_Cold WT_Cold
        20 10 10
        WS_Medium WS_Medium
    } Whites {
```

```
        {"Whites"} WT_Hot WT_Cold
        20 10 20
        WS_High WS_High
    } PermPress {
        {"PermPress"} WT_Warm WT_Cold
        15 10 15
        WS_Medium WS_Medium
    } Delicate {
        {"Delicate"} WT_Cold WT_Cold
        15 10 10
        WS_Low WS_Low
    } ; # ❶
}
```

❶ Note that the character string literals are enclosed in braces ({}). This is necessary since double quote marks (") are significant to the Tcl embedded macro processor and we need to make sure that the double quote characters make is all the way to the "C" code. Enclosing text in braces prevents any interpretation of the enclosed text.

The class model dictates that each washer have exactly one ClothesTub.

```
<<CT population>>=
class ClothesTub {
    instance ct1 R1 wm1 ; # ❶
}
```

❶ Note the `R1` attribute. This is not a real attribute, but rather it determines the instance of `WashingMachine` to which the `ClothesTub` instance is related. Any class that contains referential attributes must supply an instance reference to realize the relationship at the time the referring instance is created. For `ClothesTub`, `R1` is realized by referring to an instance of `WashingMachine` and hence the `R1` pseudo-attribute must be supplied with an instance reference to a `WashingMachine`.

Each Washing Machine also has three motors.

```
<<MTR population>>=
class Motor {
    table   {MotorID        R2}\
    mtr1    {MTR_Pump       ct1}\
    mrt2    {MTR_Agitator   ct1}\
    mtr3    {MTR_Spin       ct1}
}
```

Each washing machine, also by design, has three valves to control hot and cold water and draining.

```
<<WV population>>=
class WaterValve {
    table   {ValveID    R3}\
    wv1     {WV_Hot     ct1}\
    wv2     {WV_Cold    ct1}\
    wv3     {WV_Drain   ct1}
}
```

Finally, each washing machine has a sensor that can determine whether the tub is full or empty.

```
<<WLS population>>=
class WaterLevelSensor {
    instance wls1 R5 ct1
}
```

Stubbing the External Operations

External operations invoked by the domain must be resolved. In this section we stub out those operations with sufficient code to be able to run our example program.

Normally, external operation code is supplied outside of the domain specification. It is typically resolved by code used to bridge one domain to another. For convenience, `micca` allows code to be associated with an external operation and, with the correct options to the code generator, that code is passed to the output "C" file. This is convenient for testing, demonstration purposes and running a domain in isolation, but should *not* be the primary means of supplying external operation code for domain.

For the Motor and Valve operations, we content ourselves simply to log the fact that they were invoked. The control that is implied by the operation is "open loop" and no feed back is assumed. So when we say "Open a Valve" we assume that the value does what it is told. This leads us to the following implementation.

```
<<external operations>>=
eentity MOTOR {
    operation void start {motorid MRT_InstId} {
        MRT_DEBUG("%s: starting motor %u\n", __func__, motorid) ;
    }
    operation void stop {motorid MRT_InstId} {
        MRT_DEBUG("%s: stopping motor %u\n", __func__, motorid) ;
    }
}

eentity VALVE {
    operation void open {valveid MRT_InstId} {
        MRT_DEBUG("%s: opening valve %u\n", __func__, valveid) ;
    }
    operation void close {valveid MRT_InstId} {
        MRT_DEBUG("%s: closing valve %u\n", __func__, valveid) ;
    }
}
```

The Water Level Sensor operations present a bit more difficulty to stub. In this case, there *is* feedback from the interaction. We must signal back the state of tub as being full or empty. In some sense we must *simulate* the action of the sensor. To accomplish simulating the sensor, we record data in a variable indexed by the washing machine instance ID. This allows us to simulate an arbitrary number of washing machines. We delay the announcement of the new state for some time to simulate the water filling or draining. To make the example run in reasonable times, we assume the tub fills or empties in 3 seconds. That's quick!

Delivering the indication that the Water Level Sensor has detected a change in the water level really means we want to send the appropriate ClothesTub instance either the **TubFull** or **TubEmpty** event. So we use a boolean to track the state of the tub, toggling that state when the sensor is enabled.

```
<<external operations>>=
prologue {
    static bool tubFull[WMCTRL_CLOTHESTUB_INSTCOUNT] ;
}
```

The enable of the sensor then complements the state held in our variable and signals a delayed event to the domain via the portal interface. The value of the state variable for the sensor is used to select which event, Full or Empty, is sent to the Clothes Tub.

```
<<external operations>>=
eentity SENSOR {
    operation void enable {tubid MRT_InstId} {
        MRT_DEBUG("%s: enable sensor for tub %u\n", __func__, tubid) ;

        tubFull[tubid] = !tubFull[tubid] ;

        int pcode = mrt_PortalSignalDelayedEvent(&wmctrl__PORTAL,
            WMCTRL_CLOTHESTUB_CLASSID,
            tubid,
```

```
            tubFull[tubid] ?    WMCTRL_CLOTHESTUB_TUBFULL_EVENT :
                                WMCTRL_CLOTHESTUB_TUBEMPTY_EVENT,
            NULL, 0,
            3000) ; // ❶
        assert(pcode == 0) ;
    }
}
```

❶ Since this an external operation, we use the bridging portal of `micca` to signal the event. The portal functions supplied by the run time provide a convenient way to access many model level actions inside of a domain.

To disable the sensor, we just cancel the delayed event. Given that the state activities disable the sensor after it has signaled it state, canceling the event usually results in no real operation.

```
<<external operations>>=
eentity SENSOR {
    operation void disable {tubid MRT_InstId} {
        MRT_DEBUG("%s: disable sensor for tub %u\n", __func__, tubid) ;

        int pcode = mrt_PortalCancelDelayedEvent(&wmctrl__PORTAL,
            WMCTRL_CLOTHESTUB_CLASSID,
            tubid,
            tubFull[tubid] ? WMCTRL_CLOTHESTUB_TUBFULL_EVENT :
                             WMCTRL_CLOTHESTUB_TUBEMPTY_EVENT) ;
        assert(pcode == 0) ;
    }
}
```

Running the Example

Before we can start the example running, we explain how we are going to stop it. To dispatch state machine events we must **enter the event loop**. The event loop is provided by the run time code and is how events are signaled and dispatched. We invoke the `mrt_EventLoop` function to do that. But we need some way to break out of the event loop so that we can stop the example run. To do that, somewhere we must invoke the `mrt_SyncToEventLoop` function.

We invoke `mrt_SyncToEventLoop` in the external operation **cycleComplete**. This operation is invoked when the cycle is done and it is our intent to regain control of the execution flow after each washing cycle. Note that we are implementing the external operation in this manner solely to be able to run one washing cycle in our example and gain control after that cycle has completed. An actual application would most likely run forever or invoke `exit` based on some other condition or circumstance.

```
<<external operations>>=
eentity NOTIFY {
    operation void complete {washerid MRT_InstId} {
        MRT_DEBUG("%s: cycle complete for washer %u\n", __func__, washerid) ;
        /*
         * Exit the event loop for the purposes of the example.  This call is
         * used to cause the event loop to exit. This means that we only
         * run one cycle of the washer before the program exits the event
         * loop and then terminates.
         */
        mrt_SyncToEventLoop() ;
    }
}
```

Finally, yes truly finally, we are in a position to run the washer through the cycle. This is done by supplying a `main` program. The domain initialization done by `wmctrl_init` signals all the initial instances (one in our case) to start. After that, the event loop gains control and the execution sequencing begins.

```
<<main program>>=
int
main(
    int argc,
    char *argv[])
{
    mrt_Initialize() ; // ❶
    wmctrl_init() ;
    mrt_EventLoop() ;

    return EXIT_SUCCESS ;
}
```

❶ The run time must be initialized before initializing the domain since the domain initialization code signals an event. System initialization that does not use run time facilities, *e.g.* hardware initialization, may be done before initializing the run time.

Example Program

The example program must be generated by running `micca`. Normally, this would be done using the `micca` program. Here we are doing it using a Tcl script as this is more convenient during the development of `micca` itself.

```
<<build_wmctrl.tcl>>=
source ../../tcl/micca.tcl
micca configureFromFile wmctrl.micca
micca configureFromFile wmctrl_pop.micca
micca generate stubexternalops true lines true ; # ❶
```

❶ This option insures that the code we supplied to stub the external operations is passed along to the resulting "C" output file.

Example Run Results

After running the example we obtain the following output.

Output From Running the Example

```
2023-03-31T16:37:49.100.999: Transition: ?.? - Start -> WashingMachine.wm1:  ↩
    Stopped ==> FillingToWash
2023-03-31T16:37:49.101.060: Transition: WashingMachine.wm1 - Fill -> ClothesTub. ↩
    ct1: Empty ==> Filling
2023-03-31T16:37:52.101.242: Transition: ?.? - TubFull -> ClothesTub.ct1: Filling  ↩
    ==> Full
2023-03-31T16:37:52.101.311: Transition: ClothesTub.ct1 - Full -> WashingMachine. ↩
    wm1: FillingToWash ==> Washing
2023-03-31T16:37:52.101.345: Transition: WashingMachine.wm1 - Agitate ->  ↩
    ClothesTub.ct1: Full ==> Agitating
2023-03-31T16:38:07.101.428: Transition: WashingMachine.wm1 - Done ->  ↩
    WashingMachine.wm1: Washing ==> DrainingWash
2023-03-31T16:38:07.101.508: Transition: WashingMachine.wm1 - Drain -> ClothesTub. ↩
    ct1: Agitating ==> Emptying
2023-03-31T16:38:10.101.738: Transition: ?.? - TubEmpty -> ClothesTub.ct1:  ↩
    Emptying ==> Empty
```

```
2023-03-31T16:38:10.101.806: Transition: ClothesTub.ct1 - Empty -> WashingMachine. ↩
    wm1: DrainingWash ==> FillingToRinse
2023-03-31T16:38:10.101.828: Transition: WashingMachine.wm1 - Fill -> ClothesTub. ↩
    ct1: Empty ==> Filling
2023-03-31T16:38:13.102.059: Transition: ?.? - TubFull -> ClothesTub.ct1: Filling  ↩
    ==> Full
2023-03-31T16:38:13.102.128: Transition: ClothesTub.ct1 - Full -> WashingMachine. ↩
    wm1: FillingToRinse ==> Rinsing
2023-03-31T16:38:13.102.161: Transition: WashingMachine.wm1 - Agitate ->  ↩
    ClothesTub.ct1: Full ==> Agitating
2023-03-31T16:38:23.102.323: Transition: WashingMachine.wm1 - Done ->  ↩
    WashingMachine.wm1: Rinsing ==> DrainingRinse
2023-03-31T16:38:23.102.388: Transition: WashingMachine.wm1 - Drain -> ClothesTub. ↩
    ct1: Agitating ==> Emptying
2023-03-31T16:38:26.102.589: Transition: ?.? - TubEmpty -> ClothesTub.ct1:  ↩
    Emptying ==> Empty
2023-03-31T16:38:26.102.657: Transition: ClothesTub.ct1 - Empty -> WashingMachine. ↩
    wm1: DrainingRinse ==> Spinning
2023-03-31T16:38:26.102.705: Transition: WashingMachine.wm1 - Spin -> ClothesTub. ↩
    ct1: Empty ==> Spinning
2023-03-31T16:38:41.102.867: Transition: WashingMachine.wm1 - Done ->  ↩
    WashingMachine.wm1: Spinning ==> Stopped
2023-03-31T16:38:41.102.939: Transition: WashingMachine.wm1 - Stop -> ClothesTub. ↩
    ct1: Spinning ==> StoppingSpin
```

The trace consists of the chronological trace of the state machine event dispatch. The first column of the event dispatch trace is the time of day down to microseconds. The remainder of the trace show the details of the event dispatch. A *Transition* trace shows the event being dispatched from a source instance to a target instance. (The output here is reminiscent of the `transition` statements of the state model specification. The transition of the target instance, from current state to new state, is shown in the second portion of the trace output using the "$\Longrightarrow$" symbol. Instances are shown in the form of *class.instance* if such naming information is known. For events originating outside of a state activity, the instance is shown as "?.?". For instances that have no naming information (*i.e.* they were created dynamically), then the instance id number is shown.

Part III

The Micca Platform

In this part we present a class model of the platform specific model used by `micca`. Some readers will have studied platform independent models. These are sometimes referred to as *meta-models*. There is no single meta-model for XUML. You can express the rules of executable modeling in a number of ways, although the several meta-models that have been developed have, as would be expected, many of the same constructs in them. The model we present here is platform specific. The platform we are targeting is single threaded with "C" as the implementation language. The platform also assumes that all data will be held in primary memory and that all data storage is statically allocated. Consequently, some of the constructs in this model will be decidedly implementation oriented. It is, after all, meant to be specific to a particular computing technology. A single threaded, "C" based implementation is **not** an appropriate basis for all executable models. It is suitable, however, for a large class of applications. That is generally true of all translation technologies. You must be specific about the chosen class of applications and then choose appropriate computing technology that is applicable to that class of applications. There is no such thing as a universal software architecture any more than there is a universal programming language. All involve trade-offs of usually conflicting needs and requirements.

The role of the platform model is central to the overall translation scheme. Populating this model is the primary focus of the `micca` domain specific language. The code generation phase of the translation will query the populated platform model to produce "C" language output to implement the semantics of the model.

Consequently, we devote considerable effort to describing the platform model. We divide the discussion up into several sections focused on a particular subsystem of the platform model. The platform model itself is implemented as a `rosea` domain. Intermixed with the platform model description is the implementation of the model in `rosea`.

Chapter 3

Domain Subsystem

Introduction

The domain subsystem is concerned with defining the basic elements of what constitutes a domain Below is the UML class diagram for the domain subsystem of the platform model.

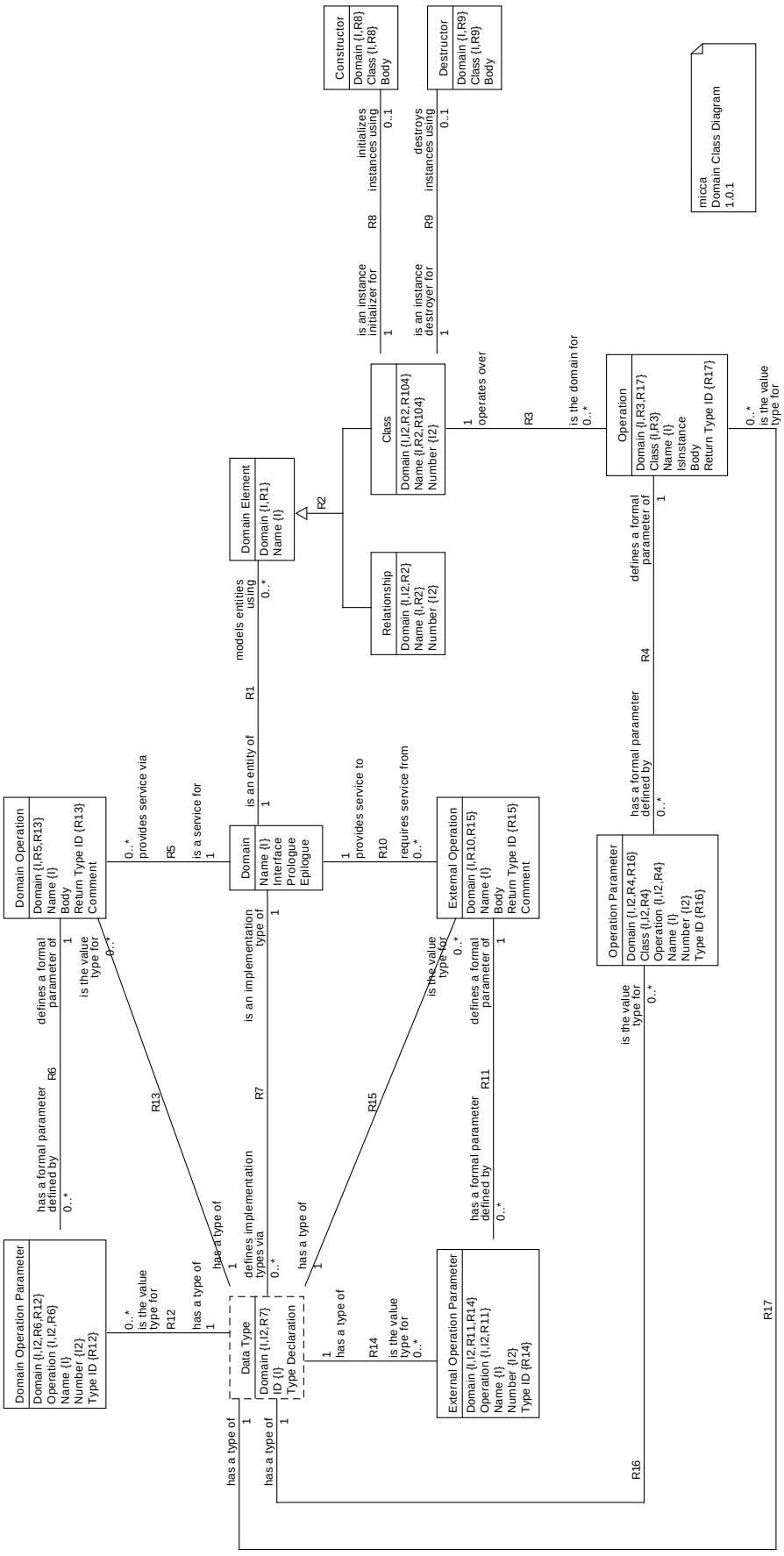


Figure 3.1: Domain Subsystem Class Diagram

A **Domain** is identified by a **Name**. A **Domain** consists of a set of **Domain Elements** (**R1**). **Domain Elements** are of two types (**R2**), **Class** and **Relationship**. **R2** insures that classes and relationships have different names and this insures there are no naming conflicts in the generated code. For the **Domain Element.Name** attribute and many other names, we will insist that the names be usable as "C" language identifiers, *i.e.* they must start with a letter followed by an arbitrary number of letters, digits or underscore characters. The code generation process will use these names directly since this eases the burden of finding model elements in the generated implementation.

A **Domain** may also define a set of **Domain Operation** (**R5**). These operations constitute the invocable interface functions for the **Domain**. A **Domain** is otherwise encapsulated and code generation will not expose symbol names used in the domain's implementation. A **Domain Operation** may also have **Domain Operation Parameter** (**R6**) associated with it. It is also typical for a **Domain** to have a set of **Type Alias** (**R7**). A **Type Alias** provides a convenient way to map model type names onto "C" implementation types.

A **Class** may also have a set of **Operation** (**R3**). An **Operation** may also have a set of **Operation Parameter** (**R4**). A **Class Operation** may be class based or instance based, the difference being that an instance based operation is invoked with a reference to the class instance upon which the operation is to happen.

Domain

A domain is a real or imagined world with its own set of rules.

Name {I}

Each domain in a system must have a unique name. Code generation will use the domain name to generate a component of the file system path name for the generated "C" files and the domain is prepended to external symbols to avoid name collisions.

Data Type string

Interface

A Domain will, in general, have dependencies on other aspects of the system. The Interface attribute is a string that will be inserted in the generated interface header file for the domain. Typically, the Interface will contain `#include` "C" preprocessor statements, but may contain any other compiler declarations needed.

Data Type string

Prologue

The Prologue for a Domain is inserted into the code file before any `micca` generated code. This gives the translation the opportunity to have external or forward references needed to satisfy the "C" compiler.

Data Type string

Epilogue

The Epilogue for a Domain is inserted into the code file after any `micca` generated code.

Data Type string

Implementation

```
<<micca configuration>>=
class Domain {
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Interface string -default {}
    attribute Prologue string -default {}
    attribute Epilogue string -default {}
}
```

Domain Element

A domain element is one of the constituent parts of a domain.

Domain {I,R1}
The name of the domain to which the element belongs.

Data Type references Domain.Name

Name {I}
Each domain elements is given a name that is unique within the domain.

Data Type c-identifier. In addition, the code generator reserves all names that end in two underscore characters (__) followed by one or more upper case or decimal numeric characters.

Implementation

```
<<micca configuration>>=
class DomainElement {
    attribute Domain string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name] &&\
            ![::regexp -- {__[A-Z]+\Z} $Name]} ; # ❶

    reference R1 Domain -link {Domain Name}
}
```

❶ In addition, the code generator reserves all names that end in two underscore characters (__) followed by one or more upper case or decimal numeric characters.

Checking for “C” Identifiers

Many names in the platform model will be directly used as an identifier in the generated “C” code. We insist they be of the correct form when the model is populated. To be a proper identifier, a name cannot be a keyword, must begin with an alphabetic character and contain zero or more alphabetic, numeric or underscore characters.

```
<<helper data>>=
set keywords {_Alignas _Alignof _Atomic _Bool _Complex _Generic _Imaginary
    _Noreturn _Static_assert _Thread_local alignas alignof auto bool break case
    char complex const continue default do double else enum extern float for
    goto if imaginary inline int long noreturn register restrict return short
    signed sizeof static static_assert struct switch thread_local typedef union
    unsigned void volatile while}
```

```
<<helper commands>>=
proc isIdentifier {name} {
    variable keywords
    return [expr {[lsearch -sorted -exact $keywords $name] == -1 &&\
            [regexp {\A[[:alpha:]]\w*\Z} $name]}] ; # ❶
}
```

❶ The regular expression gibberish is just a translation of what the narrative description says.

R1

- **Domain Element** is an entity of *exactly one* **Domain**
- **Domain** models entities using *zero or more* **Domain Element**

A domain is allowed to be empty of domain elements. This is certainly not an extremely useful situation but is allowed. Each element of a domain belongs strictly to a single domain, *i.e.* domain elements are never shared between domains.

Implementation

```
<<micca configuration>>=
association R1 DomainElement 0..*--1 Domain
```

Class

A Class represents a logical predicate about some aspect of a Domain.

Domain {I,I2,R2,R104}
The name of the domain to which the class belongs.

Data Type References DomainElement.Domain

Name {I,R2,R104}
Classes are identified by name within the domain to which they belong.

Data Type string

Number {I2}
It is useful during code generation to have a sequential integer number for classes within a given domain.

Data Type zero based sequential numeric

Implementation

```
<<micca configuration>>=
class Class {
    attribute Domain string -id 1 -id 2
    attribute Name string -id 1 ; # ❶
    attribute Number int -id 2

    reference R2 DomainElement -link Domain -link Name
    reference R104 ValueElement -link Domain -link Name
}
```

1. The question arises whether we need to check if **Class.Name** is a valid “C” identifier. The answer is no. Since **Class.Name** refers to **DomainElement.Name** unconditionally, the check on **DomainElement.Name** insures that **Class.Name** is a valid “C” identifier. Otherwise, we would fail the referential integrity check indicated by **R2**.

Relationship

A Relationship represents a real world association between class instances.

Domain {I,R2}
The name of the domain to which the relationsihp belongs.

Data Type References DomainElement.Domain

Name {I,R2}
Relationships are identified by name within the domain to which they belong. Traditionally, relationships are given names of the form, "R<digit>+", where <digit> is a decimal number. Although the convention is very commonly used in the executable modeling world, relationships may be given any valid “C” identifier.

Data Type Refers to Domain Element.Name

Number {I2}
It is useful during code generation to have a sequential integer number for relationships within a given domain.

Data Type sequential numeric

Implementation

```
<<micca configuration>>=
class Relationship {
    attribute Domain string -id 1 -id 2
    attribute Name string -id 1
    attribute Number int -id 2 -system 0

    reference R2 DomainElement -link Domain -link Name
}
```

R2

- **DomainElement** is a **Class** or **Relationship** or **External Entity**

There are three types of domain elements, Classes, Relationships and External Entities. These elements alone model the domain subject matter.

Implementation

```
<<micca configuration>>=
generalization R2 DomainElement Class Relationship ExternalEntity
```

Domain Operation

A Domain Operation is an executable body of code that provides a service access point into a domain. The set of domain operations of a domain constitute the invocable programming interface to the domain.

Domain {I,R5}
The name of the domain to which the domain operation belongs.

Data Type refers to Domain.Name

Name {I}
The name of the operation. Domain operations are given names that must be unique only within a given domain.

Data Type c-identifier

Body
The code that is to be executed when the domain operation is invoked. This code is presumed to be "C" language statements and, except for some preprocessing of the statements, is passed along in the generated output file for the domain.

Data Type c-code

ReturnDataType
The data type of the value returned by the domain operation.

Data Type c-typename

Comment
A text string that is passed only as a comment in the generated header file for the domain. This provides a means for documentation of the operation to be included in the generated output of the domain.

Data Type string

Implementation

```
<<micca configuration>>=
class DomainOperation {
    attribute Domain string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Body string
    attribute ReturnDataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName\
            $ReturnDataType]}
    attribute Comment string -default {}
    attribute File string
    attribute Line int

    reference R5 Domain -link {Domain Name}
}
```

R5

- **Domain Operation** is a service for *exactly one* **Domain**
- **Domain** provides service via *zero or more* **Domain Operation**

Domain operations only operate on a single domain. However, a domain may not provide an explicit service entry points and have no defined domain operations.

Implementation

```
<<micca configuration>>=
association R5 DomainOperation 0..*--1 Domain
```

Domain Operation Parameter

A Domain Operation may define a set of formal parameters. When invoked, the caller must supply an actual argument value for each formal parameter.

Domain {I,I2,R6}
The name of the domain to which the parameter applies.

Data Type refers to Domain Operation.Domain

Operation {I,I2,R6}
The name of the domain operation to which the parameter applies.

Data Type refers to Domain Operation.Name

Name {I}
The name of the parameter.

Data Type c-identifier

Number {I2}
The ordinal position of the parameter when it is supplied as part of a list of other parameters. In addition to names, parameters are given a number as in many cases supplying arguments for formal parameters is accomplished in an ordered list without reference to the parameter name.

Data Type numeric

DataType
The data type of the parameter.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class DomainOperationParameter {
    attribute Domain string -id 1 -id 2
    attribute Operation string -id 1 -id 2
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Number int -id 2
    attribute DataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName $DataType]}

    reference R6 DomainOperation -link Domain -link {Operation Name}
}
```

R6

- **Domain Operation Parameter** defines a formal parameter of *exactly one* **Domain Operation**
- **Domain Operation** has a formal parameter defined by *zero or more* **Domain Operation Parameter**

A domain operation parameter definition only applies to a single operation. However, domain operations may have no parameters as it is valid to have "C" functions with no parameters.

Implementation

```
<<micca configuration>>=
association R6 DomainOperationParameter 0..*--1 DomainOperation
```

External Entity

An External Entity is a proxy for a Domain to obtain services from outside of the Domain. A Domain may delegate functionality to other components of the system. An External Entity is a proxy for that delegation and serves as a collection point for a group of related operations. Note that how the services are resolved by bridges is not specified. It may be that all the operations defined for an External Entity are mapped to one other Domain, but is is also the case that satisfying the delegated requirements of a Domain through an External Entity may be mapped to multiple service Domains.

Domain {I,R10}
The name of the domain to which the external entity belongs.

Data Type refers to Domain.Name

Name {I}
The name of the entity. External entities are given names that must be unique only within a given domain.

Data Type c-identifier

Implementation

```
<<micca configuration>>=
class ExternalEntity {
    attribute Domain string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}

    reference R2 DomainElement -link Domain -link Name
}
```

External Operation

An External Operation represents a service provide by an External Entity. It is a executable body of code that provides the resolution of a service requirement for the Domain.

Domain {I,R12}
The name of the domain to which the external operation belongs.

Data Type refers to External Entity.Domain

Entity {I,R12}
The name of the external entity to which the external operation belongs.

Data Type refers to External Entity.Name

Name {I}
The name of the operation. External operations are given names that must be unique only within a given domain.

Data Type c-identifier

Body
A body of code may be specified for an external operation. Normally, the code generator ignores the external operation code since that functionality is being provided by something outside of the domain. However for testing purposes, the code will include an implementation of the external operation. This simplifies building an executable for the the domain. This code is presumed to be "C" language statements and, except for some preprocessing of the statements, is passed along in the generated output file for the domain if requested.

Data Type c-code

ReturnDataType
The data type of the value returned by the external operation.

Data Type c-typename

Comment
A text string that is passed only as a comment in the generated header file for the domain. This provides a means for documentation of the operation to be included in the generated output of the domain.

Data Type string

Implementation

```
<<micca configuration>>=
class ExternalOperation {
    attribute Domain string -id 1
    attribute Entity string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Body string -default {}
```

```
    attribute ReturnDataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName\
            $ReturnDataType]}
    attribute Comment string -default {}
    attribute File string
    attribute Line int

    reference R10 ExternalEntity -link Domain -link {Entity Name}
}
```

R10

- **External Operation** is a service provision point for *exactly one* **External Entity**
- **External Entity** provides services via *one or more* **External Operation**

External operations only provide service to a single domain. However, a domain may not require any explicit external services and have no defined external operations.

Implementation

```
<<micca configuration>>=
association R10 ExternalOperation 1..*--1 ExternalEntity
```

External Operation Parameter

A External Operation may define a set of formal parameters. When invoked, the caller must supply an actual argument value for each formal parameter.

Domain {I,I2,R11}
The name of the domain to which the parameter applies.

Data Type refers to External Operation.Domain

Entity {I,R12}
The name of the external entity to which the external operation paramter belongs.

Data Type refers to External Operation.Entity

Operation {I,I2,R11}
The name of the domain operation to which the parameter applies.

Data Type refers to External Operation.Name

Name {I}
The name of the parameter.

Data Type c-identifier

Number {I2}
The ordinal position of the parameter when it is supplied as part of a list of other parameters. In addition to names, parameters are given a number as in many cases supplying arguments for formal parameters is accomplished in an ordered list without reference to the parameter name.

Data Type numeric

DataType
The data type of the parameter.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class ExternalOperationParameter {
    attribute Domain string -id 1 -id 2
    attribute Entity string -id 1 -id 2
    attribute Operation string -id 1 -id 2
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Number int -id 2
    attribute DataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName $DataType]}

    reference R11 ExternalOperation -link Domain -link Entity\
            -link {Operation Name}
}
```

R11

- **External Operation Parameter** defines a formal parameter of *exactly one* **External Operation**
- **External Operation** has a formal parameter defined by *zero or more* **External Operation Parameter**

An external operation parameter definition only applies to a single operation. However, external operations may have no parameters as it is valid to have "C" functions with no parameters.

Implementation

```
<<micca configuration>>=
association R11 ExternalOperationParameter 0..*--1 ExternalOperation
```

Constructor

A Constructor is a body of code that is invoked by the system when a class instance is created. Note that the constructor concept in `micca` is very simplified. Constructors receive no arguments other than a reference to the newly created class instance nor is there any notion of a construction hierarchy with other classes. The primary use for construction is to initialize attributes that have some internal structure and represent a user defined data type.

Domain {I,R8}
The name of the domain to which the constructor applies.

Data Type refers to Class.Domain

Class {I,R8}
The name of the class to which the constructor applies.

Data Type refers to Class.Name

Body
The body of the constructor. This is assumed to be “C” code.

Data Type string

Implementation

```
<<micca configuration>>=
class Constructor {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Body string
    attribute File string
    attribute Line int

    reference R8 Class -link Domain -link {Class Name}
}
```

R8

- **Constructor** is an instance initializer for *exactly one* **Class**
- **Class** initializes instances using *at most one* **Constructor**

A class may have a constructor but is not required to. Any constructor only operates on a single class.

Implementation

```
<<micca configuration>>=
association R8 Constructor 0..*--1 Class
```

Destructor

A Destructor is a body of code that is invoked by the system when a class instance is destroyed. The destructor is invoked with a reference to the class instance being destroyed. Like constructors, the primary use for a destructor is to clean up resources or aggregate

Domain {I,R9}
The name of the domain to which the destructore applies.

Data Type refers to Class.Domain

Class {I,R9}
The name of the class to which the destructore applies.

Data Type refers to Class.Name

Body
The body of the constructor. This is assumed to be “C” code.

Data Type string

Implementation

```
<<micca configuration>>=
class Destructor {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Body string
    attribute File string
    attribute Line int

    reference R9 Class -link Domain -link {Class Name}
}
```

R9

- **Destructor** is an instance destroyer for *exactly one* **Class**
- **Class** destroys instances using *at most one* **Destructor**

A class may have a destructor but is not required to. Any destructor only operates on a single class.

Implementation

```
<<micca configuration>>=
association R9 Destructor 0..*--1 Class
```

Operation

An **Operation** is a executable body of code that computes on a class or class instance.

Domain {I,R3}
The name of the domain to which the operation belongs.

Data Type refers to Class.Domain

Class {I,R3}
The name of the class to which the operation belongs.

Data Type refers to Class.Name

Name {I}
The name of the operation. Operations are given names that must be unique only within a given class.

Data Type c-identifier

IsInstance
This attribute determine whether the operation applies to class instances. If true, then the generated code for the operation will have an implicitly declared parameter named `self`.

Data Type boolean

Body
The code that is to be executed when the operation is invoked.

Data Type c-code

ReturnDataType
The data type of the value returned by the operation.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class Operation {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute IsInstance boolean
    attribute Body string
    attribute ReturnDataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName\
            $ReturnDataType]}
    attribute File string
    attribute Line int

    reference R3 Class -link Domain -link {Class Name}
}
```

R3

- **Operation** operates over *exactly one* **Class**
- **Class** is the domain for *zero or more* **Operation**

Operations only operate on a single class. However, a class may not provide any explicit operations.

Implementation

```
<<micca configuration>>=
association R3 Operation 0..*--1 Class
```

Operation Parameter

An Operation may define a set of formal parameters. When invoked, the caller must supply an actual argument value for each formal parameter.

Domain {I,I2,R4}
The name of the domain to which the parameter applies.

Data Type refers to Operation.Domain

Class {I,I2,R4}
The name of the class to which the parameter belongs.

Data Type refers to Operation.Class

Operation {I,I2,R4}
The name of the domain operation to which the parameter applies.

Data Type refers to Operation.Name

Name {I}
The name of the parameter.

Data Type c-identifier

Number {I2}
The ordinal position of the parameter when it is supplied as part of a list of other parameters. In addition to names, parameters are given a number as in many cases supplying arguments for formal parameters is accomplished in an ordered list.

Data Type sequential numeric

DataType
The data type of the parameter.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class OperationParameter {
    attribute Domain string -id 1 -id 2
    attribute Class string -id 1 -id 2
    attribute Operation string -id 1 -id 2
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Number int -id 2
    attribute DataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName $DataType]}

    reference R4 Operation -link Domain -link Class -link {Operation Name}
}
```

R4

- **Operation Parameter** defines a formal parameter of *exactly one* **Operation**
- **Operation** has a formal parameter defined by *zero or more* **Operation Parameter**

Implementation

```
<<micca configuration>>=
association R4 OperationParameter 0..*--1 Operation
```

Type Alias

Typically, attributes in class models are given application specific data types that reflect the value semantics of the attribute. These must be mapped onto an implementation provided data type. A Type Alias is just such a mapping. In “C”, there are no mechanisms to implement a user defined data type. What is provided is a simple alias scheme whereby a new name may be given to a defined type.

Domain {I,R7}
The name of the domain to which the type alias belongs.

Data Type refers to Class.Domain

TypeName {I}
The new name of a type. Typically, this name will correspond to the type for a model attribute. The name must be a “C” identifier as it will be passed along by the code generation.

Data Type c-identifier

TypeDefinition
A “C” type name which represents the type alias in the implementation.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class TypeAlias {
    attribute Domain string -id 1
    attribute TypeName string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $TypeName]}
    attribute TypeDefinition string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName\
            $TypeDefinition]}

    reference R7 Domain -link {Domain Name}
}
```

R7

- **Type Alias** is an implementation type of *exactly one* **Domain**
- **Domain** defines implementation types via *zero or more* **Type Alias**

Domains may specify an arbitrary number of aliases for their attribute data types. Any alias specified by a domain applies only to that domain.

Implementation

```
<<micca configuration>>=
association R7 TypeAlias 0..*--1 Domain
```

Chapter 4

Classes Subsystem

Introduction

The classes subsystem defines how model level ideas of a class will be realized in `micca`. We will simplify considerably the relational view of classes that we find at the model level. Because we are holding all data in memory and we are using an implemenation language that exposes addresses, there are number of simplifying transformations on classes available to us.

- Since the address in memory (*.i.e.* a pointer to an object) is unique, we can use it as an identifier of each class instance. If the identifying attributes from the model serve no other descriptive role, then they can be eliminated altogether. Using the object pointer as an identifier has a number of other implementation benefits such as directly accessing attribute values.
- Referential attributes will be replaced by pointer values. Since referential attributes always refer to identifiers in the model, then implementing relationships can be accomplished by storing sets of pointers whose values refer to objects in memory.

Both of these considerations lead to the decision, common for platforms of the type `micca` is intended, to map model level class definitions onto "C" structure definitions. Storage for a class can then be allocated as an array of structures that correspond to the class definition and relationship information is stored as pointers into the instance arrays. Most of the classes subsystem is concerned with how the "C" structure definitions will be composed and the type of information that is held. We will also be concerned with how some relationship storage will be structured since there are choices that have different trade-offs.

Below is the UML class diagram for the domain subsystem of the platform model.

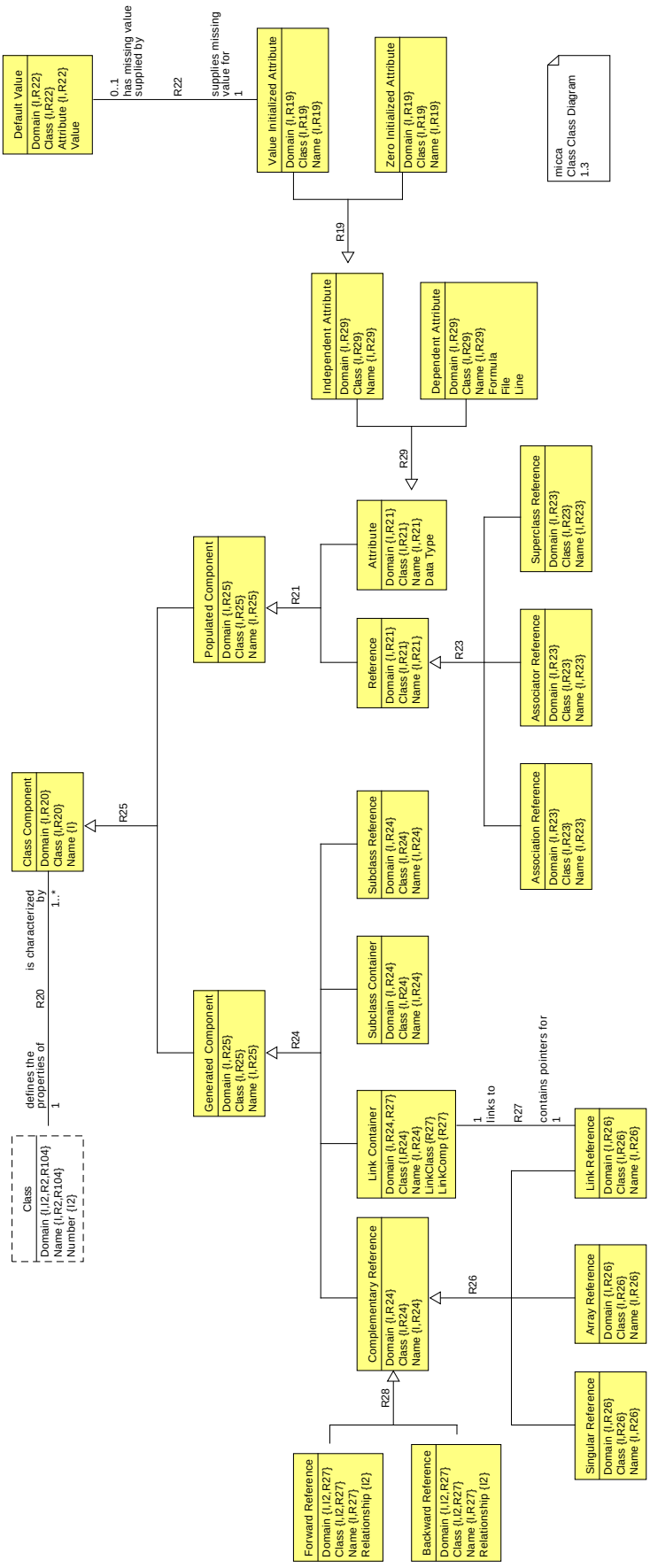


Figure 4.1: Classes Subsystem Class Diagram

A **Class** is composed of **ClassComponent** (**R20**). There are two types of **ClassComponent** (**R25**). **PopulatedComponent** are fundamental components whose values can be determined by population. **GeneratedComponent** are those that arise because of the design of how reference information is held. There are two types of **PopulatedComponent** (**R21**). An **Attribute** is a named, descriptive value of the **Class**. Attributes come in two types (**R29**). The value of an **IndependentAttribute** does not depend upon any other aspect of the domain and may have a **DefaultValue** (**R22**). A **DependentAttribute** depends upon other aspects of the domain and its value is computed by a formula. Consequently, you cannot update the value of a **DependentAttribute**. Dependent attributes are, strictly speaking, redundant, but are used by models to provide summary information in a more conveniently accessible form. A **Reference** is used to specify class relationships. There are three types of **Reference** (**R23**). An **AssociationReference** is a reference a class makes to realize a relationship. An **Associator Reference** is the pair of references that an associator class makes to map participants in a class based association. A **Superclass Reference** is the reference that a **Subclass** makes to it related **Superclass**. There are four types of **GeneratedComponent** (**R24**). A reference generalization supertype uses a **SubclassReference** to traverse a generalization relationship from superclass to subclass. A union generalization subclass may be held as a discriminated union in a **SubclassContainer**. When a class instance participates in a multiple relationship that uses linked lists, the memory for the list linkage is held in a **LinkContainer**. Traversal in a relationship in the direction opposite of the references is realized by a **ComplementaryReference**. A **ComplementaryReference** may also be one of three different types (**R26**). The types of references fulfill different roles and types of relationships. A **SingularReference** is used to store a pointer to a single class instance. An **ArrayReference** holds the references for static associations of multiplicity greater than one and a **LinkReference** serves the same role for dynamic associations. A **LinkReference** uses a **LinkContainer** to store pointers for chaining instances together (**R27**). A **ComplementaryReference** may also be used in one of two ways (**R28**). A **ForwardReference** is that reference set originated in a class based association when traversing the forward direction. A **BackwardReference** is that reference used when navigating any association in the reverse direction.

Class Component

A Class Component is an element of the implementation class structure and directly represents a member of the structure definition for the class as it is used in the generated code.

Domain {I,R20}
The name of the domain to which the class component belongs.

Data Type Refers to Class.Domain

Class {I,R20}
The name of the class to which the class component belongs.

Data Type Refers to Class.Name

Name {I}
The name of the component.

Data Type c-identifier

Implementation

```
<<micca configuration>>=
class ClassComponent {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R20 Class -link Domain -link {Class Name}
}
```

R20

- **ClassComponent** defines the properties of *exactly one* **Class**
- **Class** is characterized by *one or more* **ClassComponent**

A Class is made up of one or more components and each component can belong to only one class. Empty classes are not allowed as these are deemed to model nothing.

Implementation

```
<<micca configuration>>=
association R20 ClassComponent 1..*--1 Class
```

R25

- **ClassComponent** is a **GeneratedComponent** or **PopulatedComponent**

Some class components are fundamental in the sense that they can be given values as part of an initial instance population or when dynamically created. These types of components are **PopulatedComponents**. Other types of class components are created as part of the specific design of how model execution rules are mapped onto the implementation. The values of such components can be deduced from those of the **PopulatedComponents**. These types of components are **GeneratedComponents**.

Implementation

```
<<micca configuration>>=
generalization R25 ClassComponent GeneratedComponent PopulatedComponent
```

Populated Component

A Populated Component is that type of Class Component for which values can be specified in an initial instance population or during run time creation. These types of components are fundamental to the class model and carry values that are meaningful to the semantics of the model.

Domain {I,R25}
The name of the domain to which the populated component belongs.

Data Type Refers to Class Component.Domain

Class {I,R25}
The name of the class to which the populated component belongs.

Data Type Refers to Class Component.Name

Name {I,R25}
The name of the component.

Data Type Refers to Class Component.Name

Implementation

```
<<micca configuration>>=
class PopulatedComponent {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R25 ClassComponent -link Domain -link Class -link Name
}
```

Generated Component

A Generated Component is that type of Class Component that exists as a result of the design of how model execution rules are being mapped onto the implementation. The values of these components can be deduced from those of the Populated Components.

Domain {I,R25}
The name of the domain to which the generated component belongs.

Data Type Refers to Class Component.Domain

Class {I,R25}
The name of the class to which the generated component belongs.

Data Type Refers to Class Component.Name

Name {I,R25}
The name of the component.

Data Type Refers to Class Component.Name

Implementation

```
<<micca configuration>>=
class GeneratedComponent {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R25 ClassComponent -link Domain -link Class -link Name
}
```

R21

- **PopulatedComponent** is an **Attribute** or **Reference**

There are two types of components that may be populated for a class. An attribute corresponds to a model level attribute that holds data parameterizing the class (*i.e.* attributes whose roles are descriptive and not strictly referential or identifying). A reference is one or more pointers that are used to implement model level relationships.

Implementation

```
<<micca configuration>>=
generalization R21 PopulatedComponent Attribute Reference
```

Attribute

The Attribute class represent descriptive data that parameterizes a class.

Domain {I,R21}
The name of the domain to which the attribute belongs.

Data Type Refers to Populated Component.Domain

Class {I,R21}
The name of the class to which the attribute belongs.

Data Type Refers to Populated Component.Class

Name {I,R21}
The name of the attribute.

Data Type Refers to Populated Component.Name

DataType
The data type of the parameters.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class Attribute {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute DataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName $DataType]}

    reference R21 PopulatedComponent -link Domain -link Class -link Name
}
```

Independent Attribute

The Independent Attribute class represents those attributes whose value is independent of other components of the model. Values of independent attributes may be read or updated by domain activities.

Domain {I,R29}
The name of the domain to which the attribute belongs.

Data Type Refers to Attribute.Domain

Class {I,R29}
The name of the class to which the attribute belongs.

Data Type Refers to Attribute.Class

Name {I,R29}
The name of the attribute.

Data Type Refers to Attribute.Name

Implementation

```
<<micca configuration>>=
class IndependentAttribute {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R29 Attribute -link Domain -link Class -link Name
}
```

Dependent Attribute

The Dependent Attribute class represents those attributes whose value can be described by a formula or algorithm. Values of dependent attributes may only be read and the value read from the attribute is the result of evaluating a formula.

Domain {I,R29}
The name of the domain to which the attribute belongs.

Data Type Refers to Attribute.Domain

Class {I,R29}
The name of the class to which the attribute belongs.

Data Type Refers to Attribute.Class

Name {I,R29}
The name of the attribute.

Data Type Refers to Attribute.Name

Formula

A segment of “C” code that is executed whenever the attribute is read. The return value of the code matches the type given by the related Attribute.Data Type attribute.

Data Type string

Implementation

```
<<micca configuration>>=
class DependentAttribute {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1
    attribute Formula string
    attribute File string
    attribute Line int

    reference R29 Attribute -link Domain -link Class -link Name
}
```

R29

- **Attribute** is an **IndependentAttribute** or **DependentAttribute**

The R29 generalization reflects the distinction made between those attributes that are functionally independent of the other model components and those that are not.

Implementation

```
<<micca configuration>>=
generalization R29 Attribute IndependentAttribute DependentAttribute
```

R19

- **IndependentAttribute** is a **ValueInitializedAttribute** or **ZeroInitializedAttribute**

An Independent Attribute may be initialized in two different ways. It is initialized to zero or to a specified value of its type.

Implementation

```
<<micca configuration>>=
generalization R19 IndependentAttribute\
        ValueInitializedAttribute ZeroInitializedAttribute
```

Value Initialized Attribute

The Value Initialized Attribute class represents those attributes whose initial value is explicitly specified. The initial value for the attribute must be supplied when it is created (including the initial instance population) or by a default.

Domain {I,R19}
The name of the domain to which the attribute belongs.

Data Type Refers to Independent Attribute.Domain

Class {I,R19}
The name of the class to which the attribute belongs.

Data Type Refers to Independent Attribute.Class

Name {I,R19}
The name of the attribute.

Data Type Refers to Independent Attribute.Name

Implementation

```
<<micca configuration>>=
class ValueInitializedAttribute {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R19 IndependentAttribute -link Domain -link Class -link Name
}
```

Zero Initialized Attribute

The Zero Initialized Attribute class represents those attributes whose initial value is assumed to be zero. For these attributes, no initial value need be supplied and if missing is set to zero. In this case, zero initialization mean the memory space occupied by the attribute is set to all bits being zero. Under certain extreme cases, this may or may not be a proper zero value for the data type of the attribute. There are two reasonable uses for Zero Initialized Attributes:

- A data type where it is more convenient to initalized it to a value in a constructor function.
- A data type that represents the composition of another class within a containing class where zero initialization prevents having to supply a large initializer value.

Domain {I,R19}
The name of the domain to which the attribute belongs.

Data Type Refers to Independent Attribute.Domain

Class {I,R19}
The name of the class to which the attribute belongs.

Data Type Refers to Independent Attribute.Class

Name {I,R19}
The name of the attribute.

Data Type Refers to Independent Attribute.Name

Implementation

```
<<micca configuration>>=
class ZeroInitializedAttribute {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R19 IndependentAttribute -link Domain -link Class -link Name
}
```

Default Value

The default value of an attribute is one that will be given to the attribute when an class instance is created if no value is otherwise provided.

Implementation

```
<<micca configuration>>=
class DefaultValue {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Attribute string -id 1
    attribute Value string

    reference R22 ValueInitializedAttribute -link Domain -link Class\
            -link {Attribute Name}
}
```

R22

- **DefaultValue** supplies missing value for *exactly one* **ValueInitializedAttribute**
- **ValueInitializedAttribute** has missing value supplied by *zero or one* **DefaultValue**

An Value Initialized Attribute may be defined to have a default value that the system will use when the attribute's value is not otherwise specified when a class instance is created. There can only be one default value for an attribute and not all attributes will define default values.

Implementation

```
<<micca configuration>>=
association R22 DefaultValue 0..1--1 ValueInitializedAttribute
```

Reference

A reference is one or more pointer values that are used to implement the ability to navigate a relationship.

Domain {I,R21}
The name of the domain to which the reference belongs.

Data Type Refers to Populated Component.Domain

Class {I,R21}
The name of the class to which the reference belongs.

Data Type Refers to Populated Component.Class

Name {I,R21}
The name of the reference.

Data Type Refers to Populated Component.Name

Implementation

```
<<micca configuration>>=
class Reference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R21 PopulatedComponent -link Domain -link Class -link Name
}
```

R23

- **Reference** is a **AssociationReference**, **AssociatorReference**, or **SuperclassReference**

There are three types of references that specify class relationships.

Implementation

```
<<micca configuration>>=
generalization R23 Reference\
    AssociationReference AssociatorReference SuperclassReference
```

Association Reference

An Association Reference is the reference made by the class playing the referring role in a simple association type of relationship.

Domain {I,R23}
The name of the domain to which the assocation reference belongs.

Data Type Refers to Reference.Domain

Class {I,R23}
The name of the class to which the assocation reference belongs.

Data Type Refers to Reference.Class

Name {I,R23}
The name of the assocation reference.

Data Type Refers to Reference.Name

Implementation

```
<<micca configuration>>=
class AssociationReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R23 Reference -link Domain -link Class -link Name
}
```

Associator Reference

When a class is an associator class, it makes two references to the participants in a class based association. Note for associator references, the reference values may not be empty and must always refer to a class instance. This is because there is an instance of the associator class for each instance of the association relationship itself.

Domain {I,R23}
The name of the domain to which the associator reference belongs.

Data Type Refers to Reference.Domain

Class {I,R23}
The name of the class to which the associator reference belongs.

Data Type Refers to Reference.Class

Name {I,R23}
The name of the associator reference.

Data Type Refers to Reference.Name

Implementation

```
<<micca configuration>>=
class AssociatorReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R23 Reference -link Domain -link Class -link Name
}
```

Superclass Reference

A Superclass Reference is the reference that a subclass makes to the superclass instance to which it is related.

Domain {I,R23}
The name of the domain to which the superclass reference belongs.

Data Type Refers to Reference.Domain

Class {I,R23}
The name of the class to which the superclass reference belongs.

Data Type Refers to Reference.Class

Name {I,R23}
The name of the superclass reference.

Data Type Refers to Reference.Name

Implementation

```
<<micca configuration>>=
class SuperclassReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R23 Reference -link Domain -link Class -link Name
}
```

R24

- **Generated Component** is a **Complementary Reference**, **Link Container**, **Subclass Container** or **Subclass Reference**

There are four types of components that are generated. These components do not have independent values, *i.e.* the values of the component may be derived from the values of **PopulatedComponents**. The necessity of these types of components arise from the design decision as to how relationships are navigated and subclass instance may be stored.

Implementation

```
<<micca configuration>>=
generalization R24 GeneratedComponent ComplementaryReference LinkContainer\
        SubclassContainer SubclassReference
```

Complementary Reference

To facilitate navigating associations in the directions other than that indicated by direct referential attributes, a Complementary Reference component is created to hold reference pointers to the referring class of an association.

Domain {I,R24}
The name of the domain to which the complementary reference belongs.

Data Type Refers to Generated Component.Domain

Class {I,R24}
The name of the class to which the complementary reference belongs.

Data Type Refers to Generated Component.Class

Name {I,R24}
The name of the complementary reference.

Data Type Refers to Generated Component.Name

Implementation

```
<<micca configuration>>=
class ComplementaryReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R24 GeneratedComponent -link Domain -link Class -link Name
}
```

Link Container

When a class stores multiple references as part of a back reference and those references are threaded onto a linked list, then the classes residing on the linked list have a Link Container component defined for them to hold the link list pointers. It is possible for an instance to be threaded onto multiple linked lists and each will have a Link Container component defined to hold the pointer values.

Domain {I,R24}
The name of the domain to which the link container belongs.

Data Type Refers to Generated Component.Domain

Class {I,R24}
The name of the class to which the link container belongs.

Data Type Refers to Generated Component.Class

Name {I,R24}
The name of the link container.

Data Type Refers to Generated Component.Name

Implementation

```
<<micca configuration>>=
class LinkContainer {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1
    attribute LinkClass string
    attribute LinkComp string

    reference R24 GeneratedComponent -link Domain -link Class -link Name
    reference R27 LinkReference -link Domain -link {LinkClass Class}\
            -link {LinkComp Name}
}
```

Subclass Container

One option for storing generalization relationship information is as a union. The idea is that subclass storage is accomplished by creating a union of all the subclasses in the generalization. Then an component of the superclass is defined to be the subclasses union. This type of storage structure can save some memory space and traversal across the generalization relationships can be accomplished with pointer arithmetic.

Domain {I,R24}
The name of the domain to which the subclass container belongs.

Data Type Refers to Generated Component.Domain

Class {I,R24}
The name of the class to which the subclass container belongs.

Data Type Refers to Generated Component.Class

Name {I,R24}
The name of the subclass container.

Data Type Refers to Generated Component.Name

Implementation

```
<<micca configuration>>=
class SubclassContainer {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R24 GeneratedComponent -link Domain -link Class -link Name
}
```

Subclass Reference

For generalization relationships that used references (rather than a union), a Subclass Reference holds the pointer to the related subclass instance.

Domain {I,R24}
The name of the domain to which the subclass reference belongs.

Data Type Refers to Generated Component.Domain

Class {I,R24}
The name of the class to which the subclass reference belongs.

Data Type Refers to Generated Component.Class

Name {I,R24}
The name of the subclass reference.

Data Type Refers to Generated Component.Name

Implementation

```
<<micca configuration>>=
class SubclassReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R24 GeneratedComponent -link Domain -link Class -link Name
}
```

R26

- **Complementary Reference** is a **Singular Reference**, **Array Reference**, or **Link Reference**

There are three types of Complementary References that represent the three ways that reference pointer values are stored. The type of reference accommodates both the multiplicity of the association and its dynamic behavior.

Implementation

```
<<micca configuration>>=
generalization R26 ComplementaryReference SingularReference ArrayReference\
        LinkReference
```

Singular Reference

A singular reference holds a single pointer value as a scalar structure member. It is used when the back reference for an association is of multiplicity one.

Domain {I,R26}
The name of the domain to which the singular reference belongs.

Data Type Refers to Back Reference.Domain

Class {I,R26}
The name of the class to which the singular reference belongs.

Data Type Refers to Back Reference.Class

Name {I,R26}
The name of the singular reference.

Data Type Refers to Back Reference.Name

Implementation

```
<<micca configuration>>=
class SingularReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R26 ComplementaryReference -link Domain -link Class -link Name
}
```

Array Reference

An array reference holds multiple reference pointer values in the form of an array. It is used when the back reference for an association is of multiplicity greater than one and the association is static in nature.

Domain {I,R26}
The name of the domain to which the array reference belongs.

Data Type Refers to Back Reference.Domain

Class {I,R26}
The name of the class to which the array reference belongs.

Data Type Refers to Back Reference.Class

Name {I,R26}
The name of the array reference.

Data Type Refers to Back Reference.Name

Implementation

```
<<micca configuration>>=
class ArrayReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R26 ComplementaryReference -link Domain -link Class -link Name
}
```

Link Reference

A link reference holds the terminus of a linked list that is used to thread together multiple instances as part of a back reference. It is used when the back reference for an association is of multiplicity greater than one and the association is dynamic in nature.

Domain {I,R26}
The name of the domain to which the link reference belongs.

Data Type Refers to Back Reference.Domain

Class {I,R26}
The name of the class to which the link reference belongs.

Data Type Refers to Back Reference.Class

Name {I,R26}
The name of the link reference.

Data Type Refers to Back Reference.Name

Implementation

```
<<micca configuration>>=
class LinkReference {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Name string -id 1

    reference R26 ComplementaryReference -link Domain -link Class -link Name
}
```

R27

- **LinkContainer** contains pointers for *exactly one* **LinkReference**
- **LinkReference** links to *exactly one* **LinkContainer**

When class instances are threaded onto a linked list as part of a Link Reference, the reference is always between exactly two classes. One class contains the linked list terminus and the other (not necessarily distinct) class contains Link Containers to hold the linked list pointers.

Implementation

```
<<micca configuration>>=
association R27 LinkContainer 1--1 LinkReference
```

R28

- **Complementary Reference** is a **Forward Reference** or **Backward Reference**

A **Complementary Reference** is used in one of two ways. First, it is used when navigating the forward direction for class based associations. Second, it used when navigating in the reverse direction for all types of associations.

Implementation

```
<<micca configuration>>=
generalization R28 ComplementaryReference ForwardReference BackwardReference
```

Forward Reference

To facilitate navigating associations in the directions other than that indicated by direct referential attributes, a Forward Reference component is created to hold reference pointers to the referring class of an association.

Domain {I,I2,R27}
The name of the domain to which the forward reference belongs.

Data Type Refers to Generated Component.Domain

Class {I,I2,R27}
The name of the class to which the forward reference belongs.

Data Type Refers to Generated Component.Class

Name {I,R27}
The name of the forward reference.

Data Type Refers to Generated Component.Name

Relationship {I2}
The relationship name that the forward reference is used in navigating.

Data Type string

Implementation

```
<<micca configuration>>=
class ForwardReference {
    attribute Domain string -id 1 -id 2
    attribute Class string -id 1 -id 2
    attribute Name string -id 1
    attribute Relationship string -id 2

    reference R28 ComplementaryReference -link Domain -link Class -link Name
}
```

Backward Reference

To facilitate navigating associations in the directions other than that indicated by direct referential attributes, a Backward Reference component is created to hold reference pointers to the referring class of an association.

Domain {I,I2,R27}
The name of the domain to which the backward reference belongs.

Data Type Refers to Generated Component.Domain

Class {I,I2,R27}
The name of the class to which the backward reference belongs.

Data Type Refers to Generated Component.Class

Name {I,R27}
The name of the backward reference.

Data Type Refers to Generated Component.Name

Relationship {I2}
The relationship name that the backward reference is used in navigating.

Data Type string

Implementation

```
<<micca configuration>>=
class BackwardReference {
    attribute Domain string -id 1 -id 2
    attribute Class string -id 1 -id 2
    attribute Name string -id 1
    attribute Relationship string -id 2

    reference R28 ComplementaryReference -link Domain -link Class -link Name
}
```

Chapter 5

Relationship Subsystem

Mapping Referential Attributes to Pointers

When targeting a statically typed language such as "C" and assuming that all class data will be held in primary memory, we must devise a way to map relational notions of referential attributes and referential integrity onto "C"' language constructs. There are, needless to say, many ways this can be done. As we discussed earlier, we make the convenient decision to use the address of a class instance as an architecturally supplied identifier. This design decision has many benefits in the implementation. This also means we can store pointer values as a means of implementing relationship navigation.

Consider a many-to-one association between two classes. In a relational scheme, we would have referential attributes in the class that was on the **many** side of the relationship and the values of those attributes would match the values of identifying attributes on the **one** side. In a pointer based scheme we can do the same thing, *i.e.* place a pointer variable in the "C" structure of the class on the **many** side and have the value of the pointer match that of a class instance on the **one** side.

Now lets look at the computation required to navigate this relationship, first from the many side to the one side and then from the one side to the many side. For this discussion, we will assume the relationship is unconditional on both sides.

From the many side, given a pointer to an **A** instance we can navigate to the related **B** instance by simply accessing the structure member that points to the related **B** instance. When navigating from the one side to the many side, we will, in general, obtain more than one instance as a result of the navigation. We can find those instances by searching all of the many-side instances selecting those instances who have a stored pointer value that matches the address of the one-side class instance.

Having to perform a search for the one-side to many-side navigation is somewhat troubling. If the number of instances of the many-side class is small, then there is little concern. If it is larger, then we might explore ways to avoid costs associated with a straight sequential` search of the many-side instances. The search code is also rather inconvenient since we target a statically typed language. The function to iterate across the many-side instances looking for the one-side pointer value has to be specific to that particular relationship if we are to be strictly type safe and don't want to resort to extremes of type casting and pointer arithmetic.

One way to avoid the search altogether is to store the pointers of the multiple related instances. We will have, in effect, pre-computed the related set and, at the cost of the pointer storage, incur no run time computation to find it. In this arrangement, the multiple instances implied by traversing the relationship from one side to the many side are held in an appropriate data structure as part of the one side class structure.

It does take computation to maintain the set if the instances participating in the relationship change. The space vs. speed trade-off is such that we will always use some additional space to save a run time cost and additional code associated with computing a set of related instances. We will also find that the choice of data structure to hold multiple pointers will be better if we know the dynamics of the relationship. By choosing different data structures to store the pointer sets, we can, again at the cost of some additional space, make maintaining the related pointer sets easier in the face of relationship changes.

So we adopt the strategy to store relationship pointers that implement the navigation in both participating classes. This decision will also be helpful when we discuss referential integrity checking and transactions on the data model.

Analysis like that above can be applied to the other types of relationships and pointer storage schemes can be designed that map the relational notion of referential attributes to memory pointers. Most of the relationship subsystem is concerned with

categorizing the types of relationships from a model level and mapping those onto the roles the classes play in the relationship. This will then allows us to explicitly map the role a class plays in a relationship to the kind of storage required to implement that role and further to the class structure components that are required to hold the pointers. The references subsystem models this later part.

A careful analysis of the conditionality, multiplicity, reflexivity and dynamics of relationships shows that there are 18 distinct outcomes for how referential notions are mapped onto implementation pointer storage. The diagrams below show how each case is implemented. These diagrams are useful to understand both how the relationship subsystem models the various types of relationships but also how the code generator emits the required “C” structure element definitions for the classes.

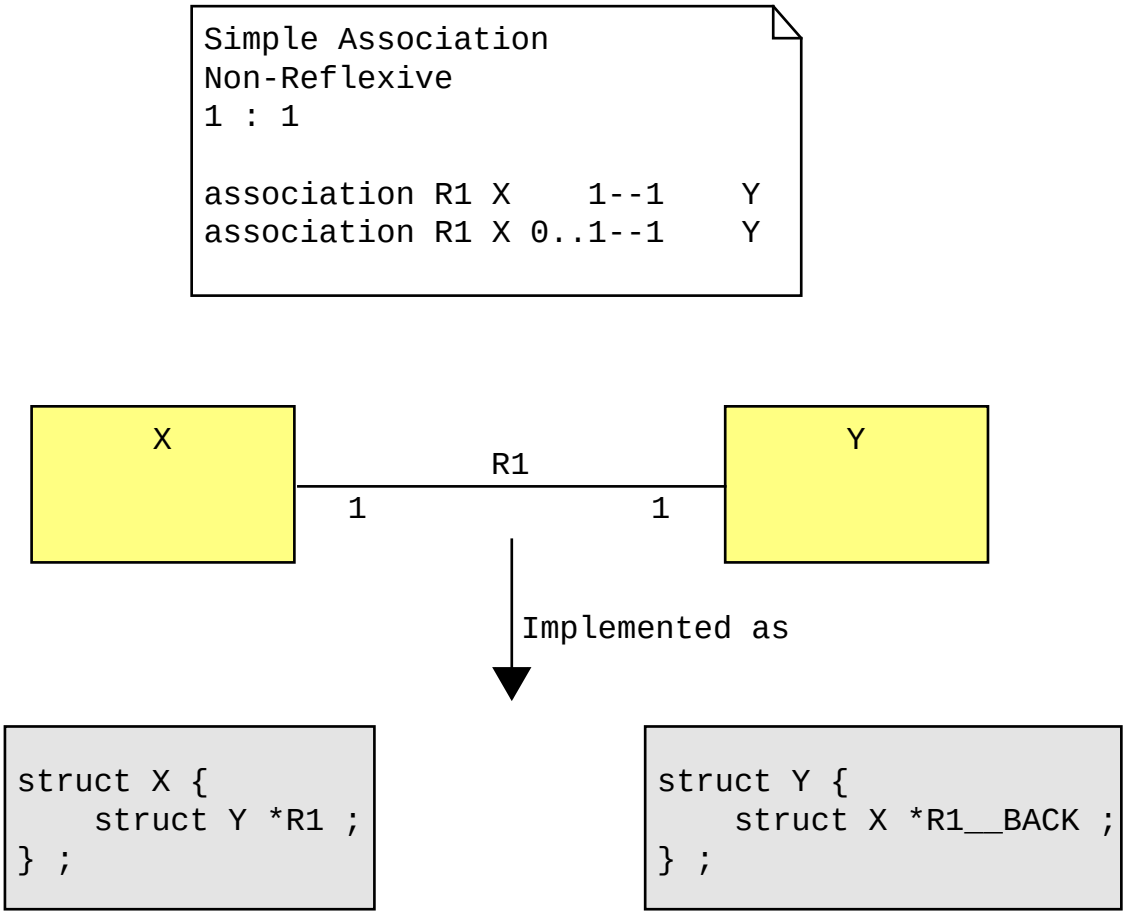


Figure 5.1: Simple Non-Reflexive 1:1 Association

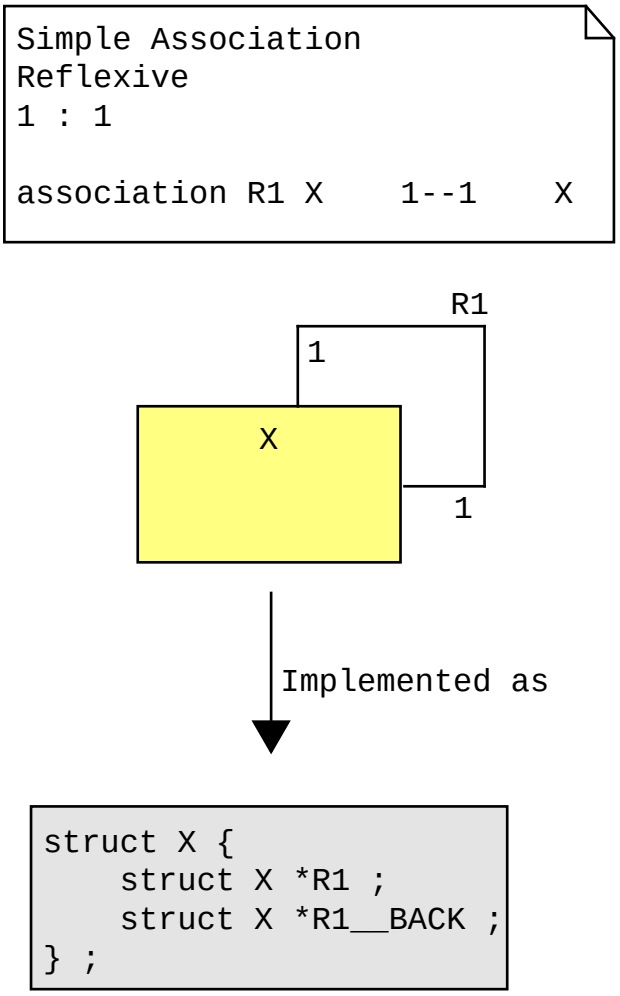


Figure 5.2: Simple Reflexive 1:1 Association

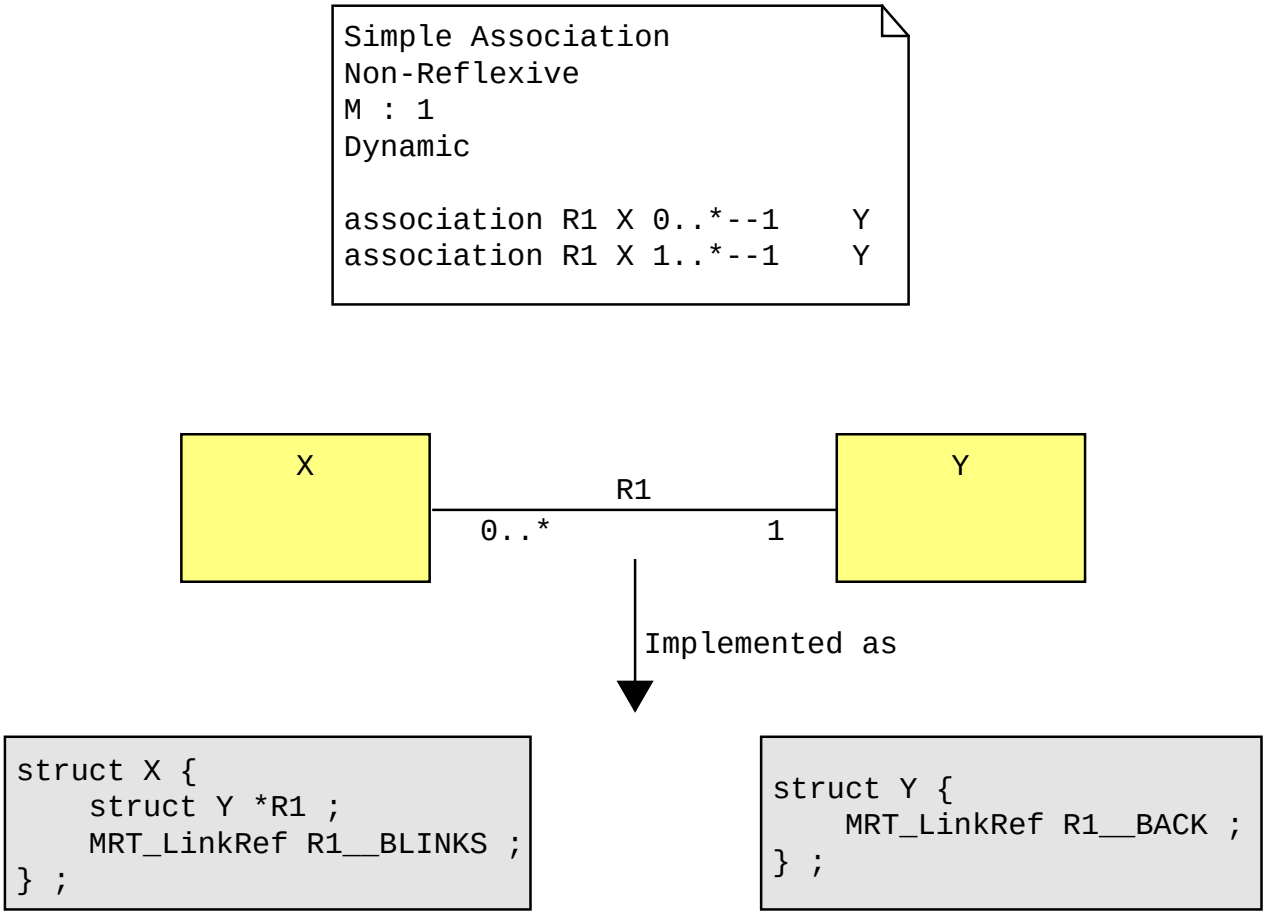


Figure 5.3: Simple Non-Reflexive M:1 Dynamic Association

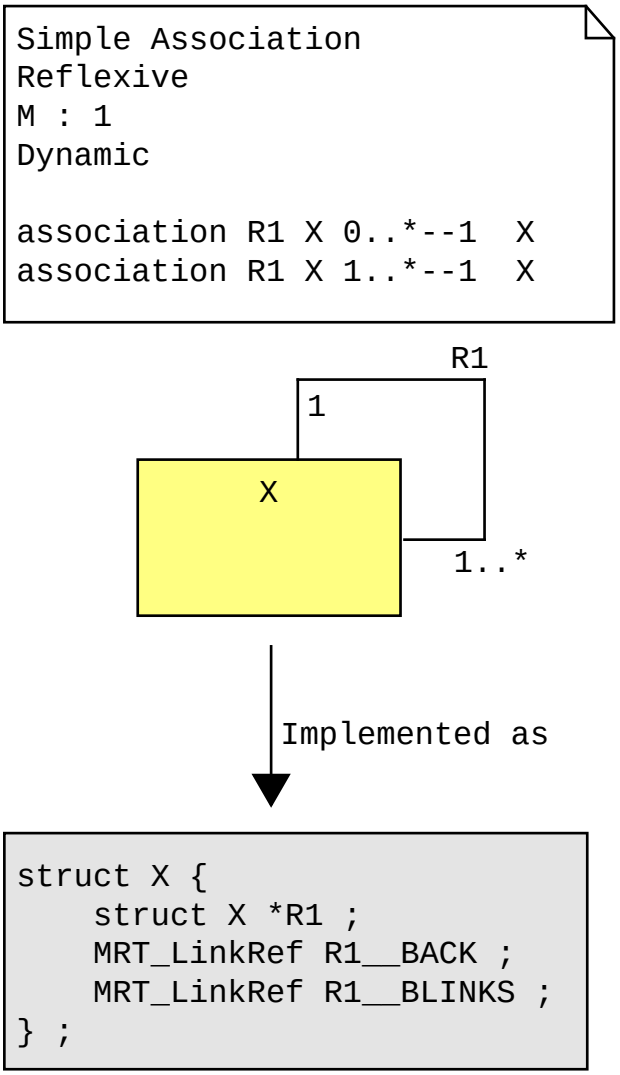


Figure 5.4: Simple Reflexive M:1 Dynamic Association

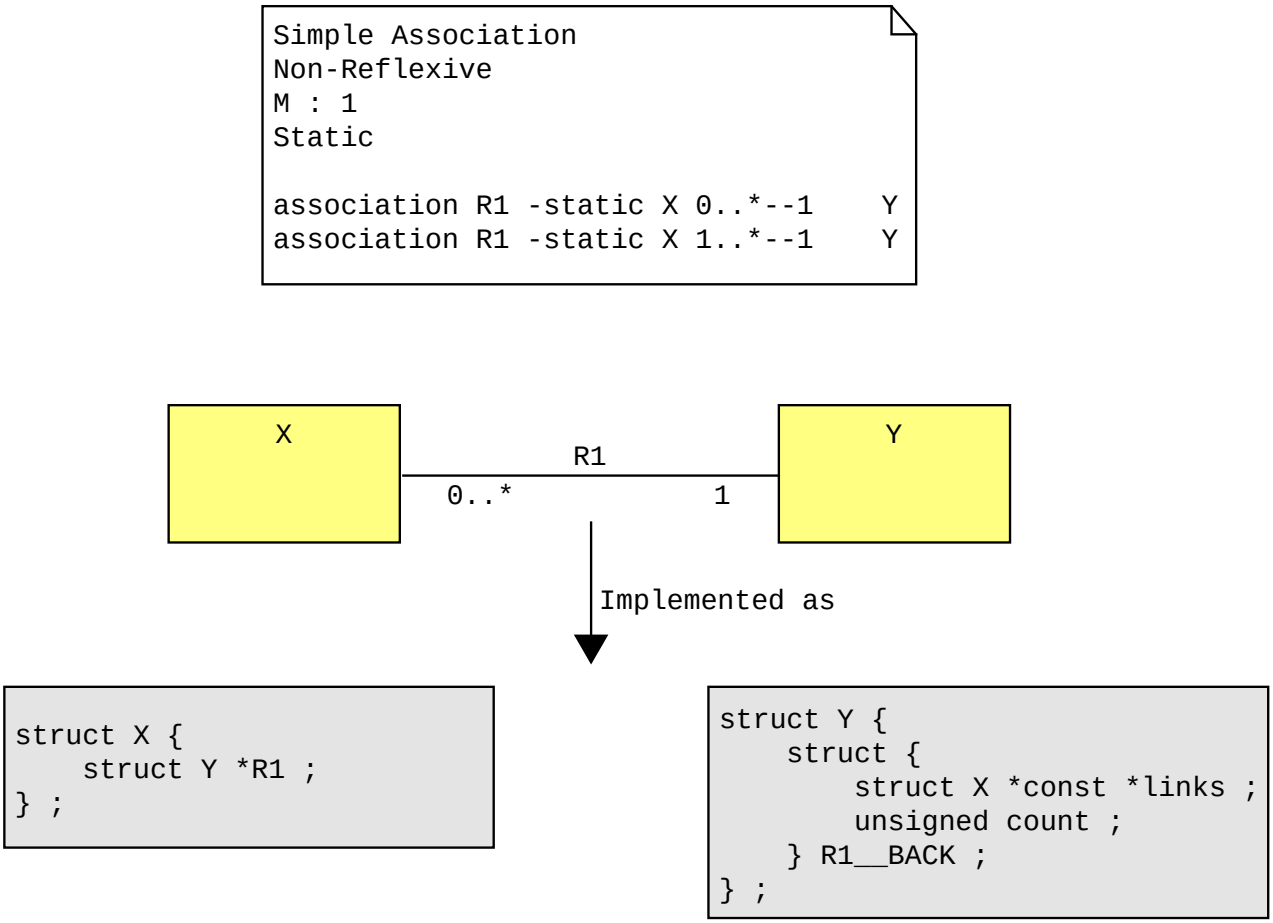


Figure 5.5: Simple Non-Reflexive M:1 Static Association

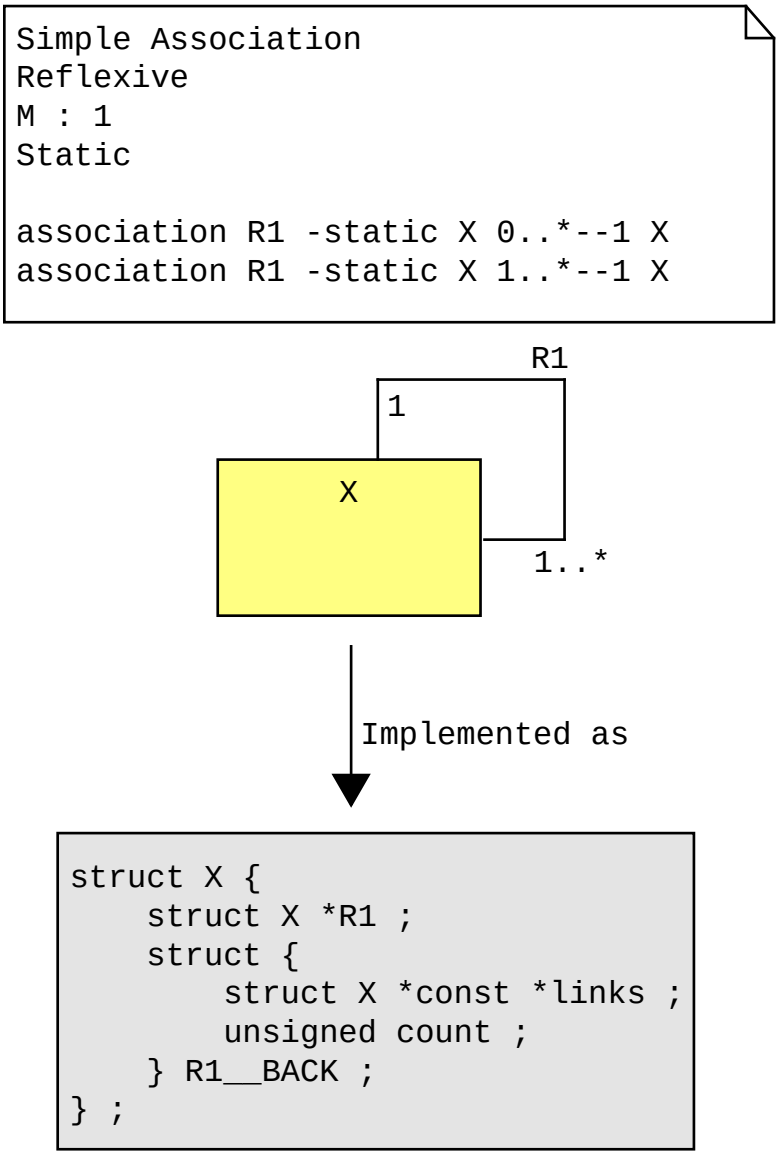


Figure 5.6: Simple Reflexive M:1 Static Association

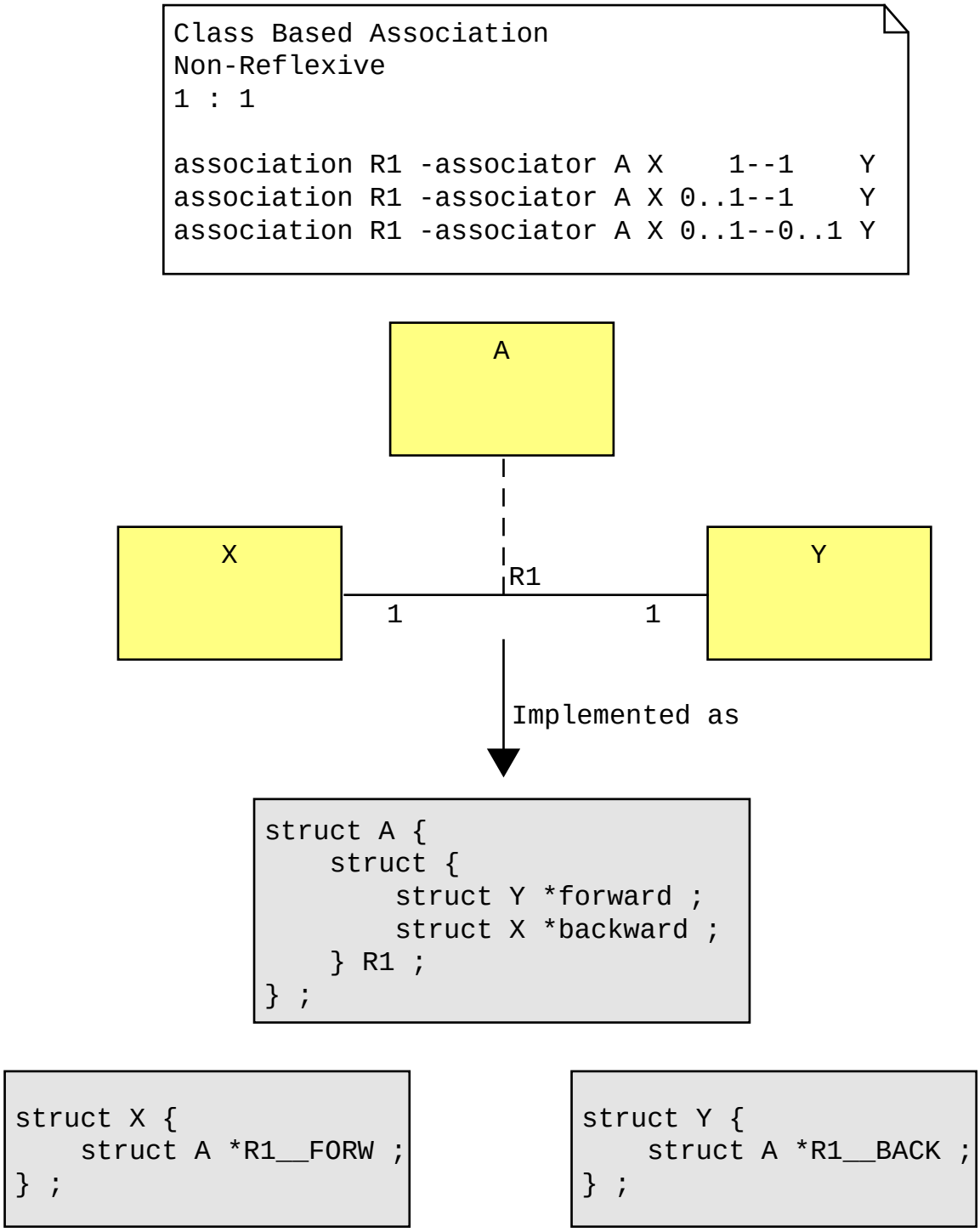


Figure 5.7: Class Based Non-Reflexive 1:1 Association

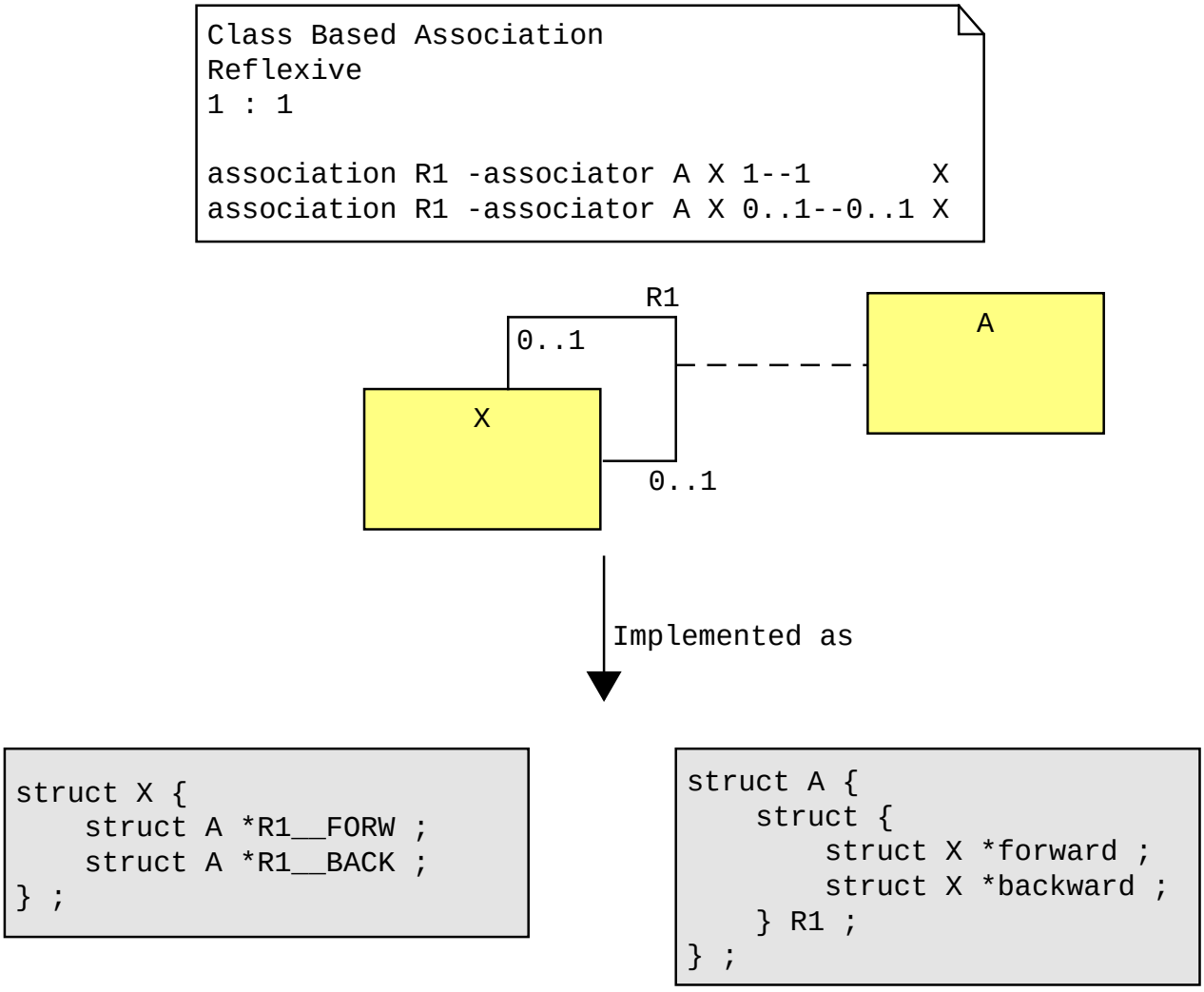


Figure 5.8: Class Based Reflexive 1:1 Association

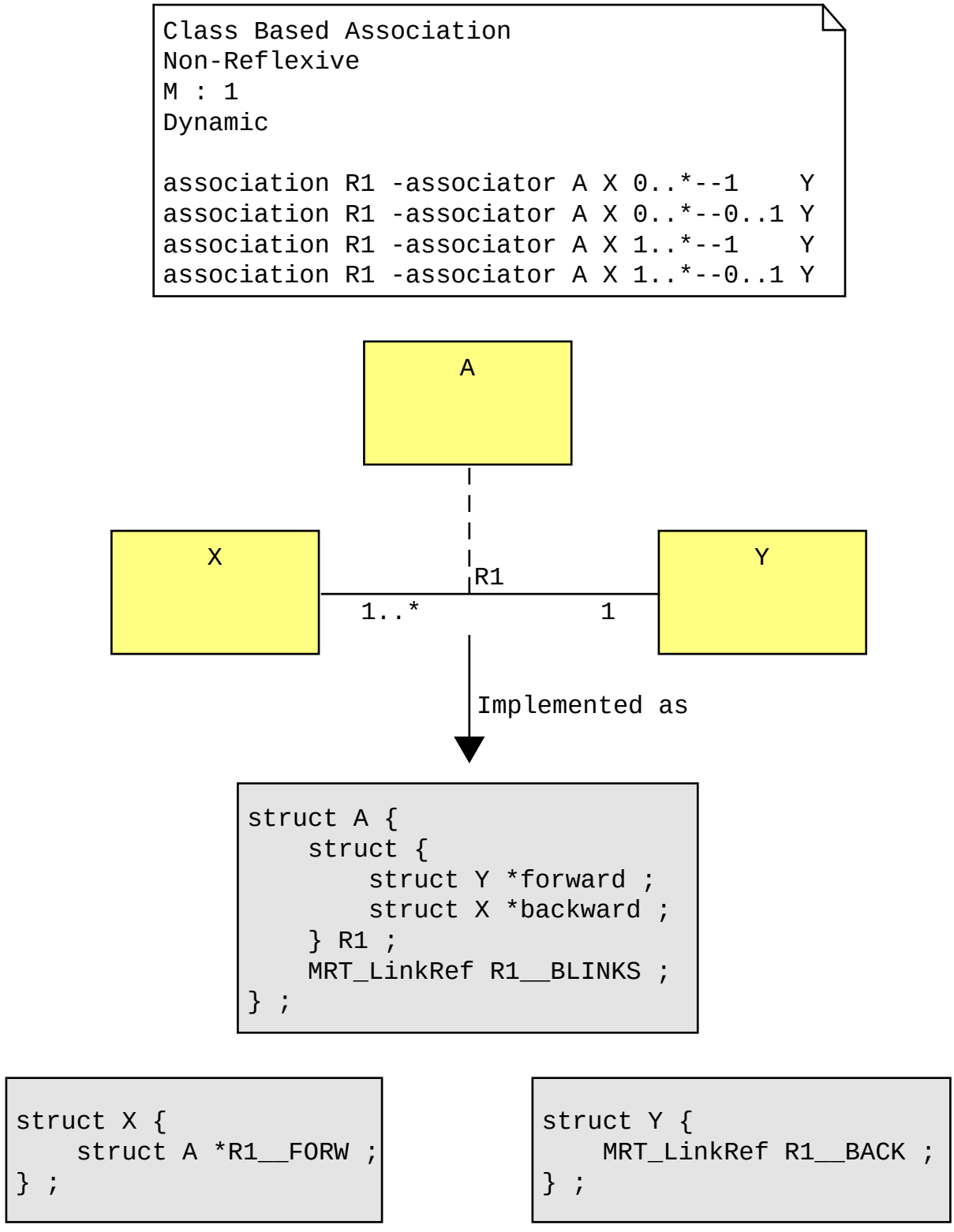


Figure 5.9: Class Based Non-Reflexive M:1 Dynamic Association

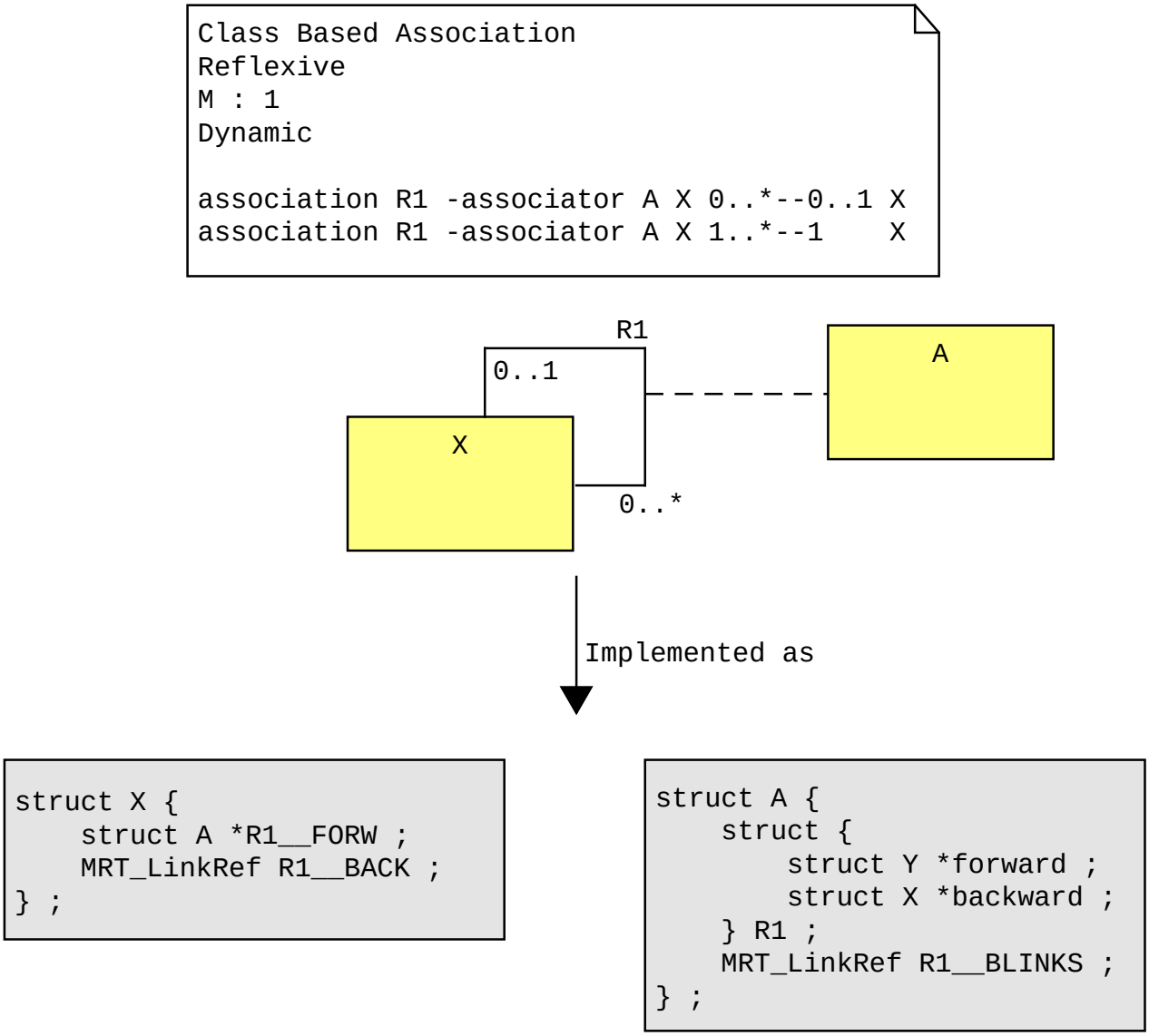
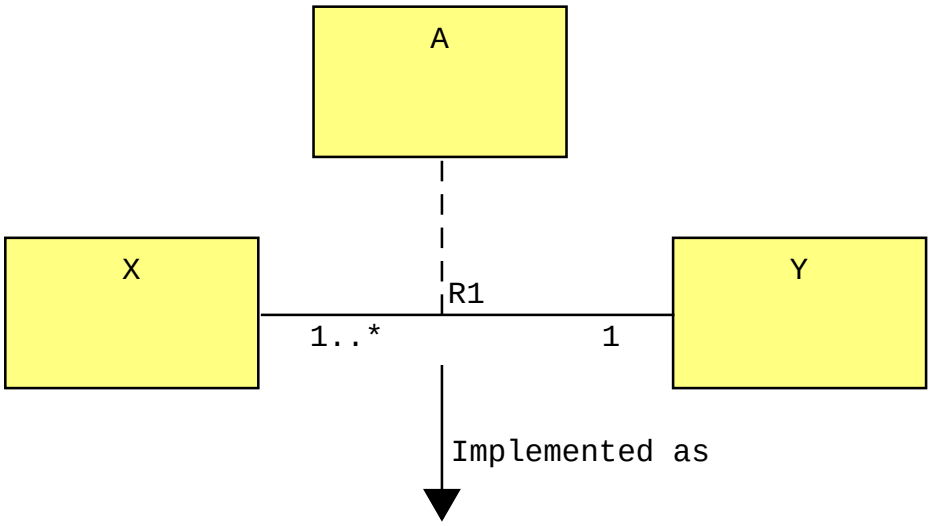


Figure 5.10: Class Based Reflexive M:1 Dynamic Association

```
Class Based Association
Non-Reflexive
M : 1
Static

association R1 -static -associator A X 0..*--1    Y
association R1 -static -associator A X 0..*--0..1 Y
association R1 -static -associator A X 1..*--1    Y
association R1 -static -associator A X 1..*--0..1 Y
```




```
struct A {
    struct {
        struct Y *forward ;
        struct X *backward ;
    } R1 ;
} ;
```

```
struct X {
    struct A *R1__FORW ;
} ;
```

```
struct Y {
    struct {
        struct A *const *links ;
        unsigned count ;
    } R1__BACK ;
} ;
```

Figure 5.11: Class Based Non-Reflexive M:1 Static Association

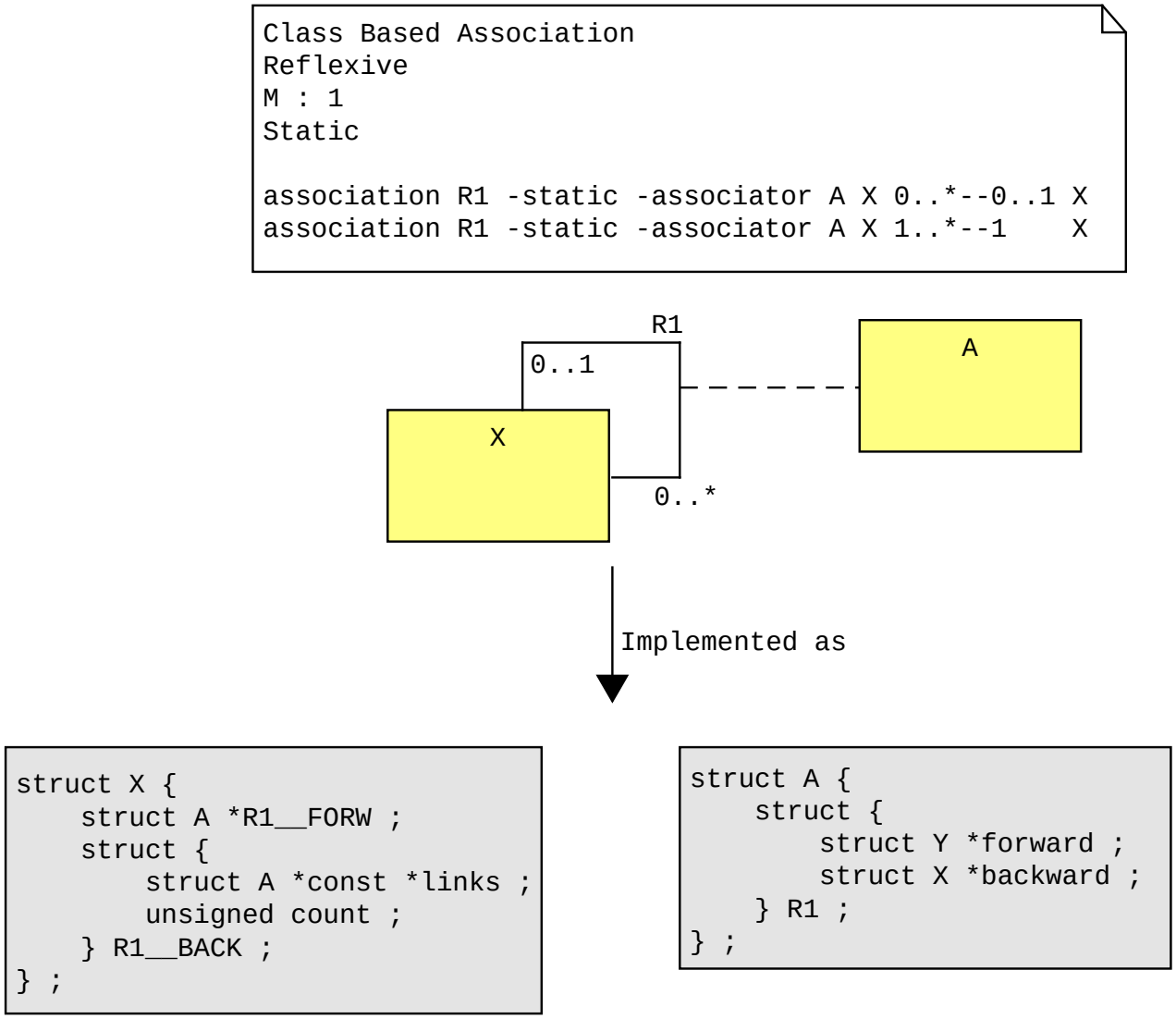
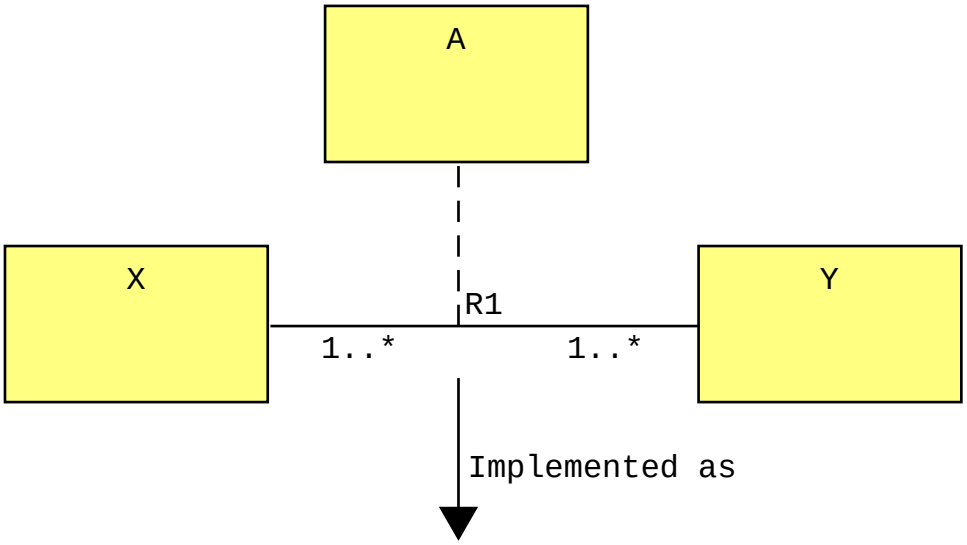


Figure 5.12: Class Based Reflexive M:1 Static Association

```
Class Based Association
Non-Reflexive
M : M
Dynamic

association R1 -associator A X 0..*--0..* Y
association R1 -associator A X 0..*--1..1 Y
association R1 -associator A X 1..*--0..* Y
association R1 -associator A X 1..*--1..1 Y
```




```
struct A {
    struct {
        struct Y *forward ;
        struct X *backward ;
    } R1 ;
    MRT_LinkRef R1__FLINKS ;
    MRT_LinkRef R1__BLINKS ;
} ;
```

```
struct X {
    MRT_LinkRef R1__FORW ;
} ;
```

```
struct Y {
    MRT_LinkRef R1__BACK ;
} ;
```

Figure 5.13: Class Based Non-Reflexive M:M Dynamic Association

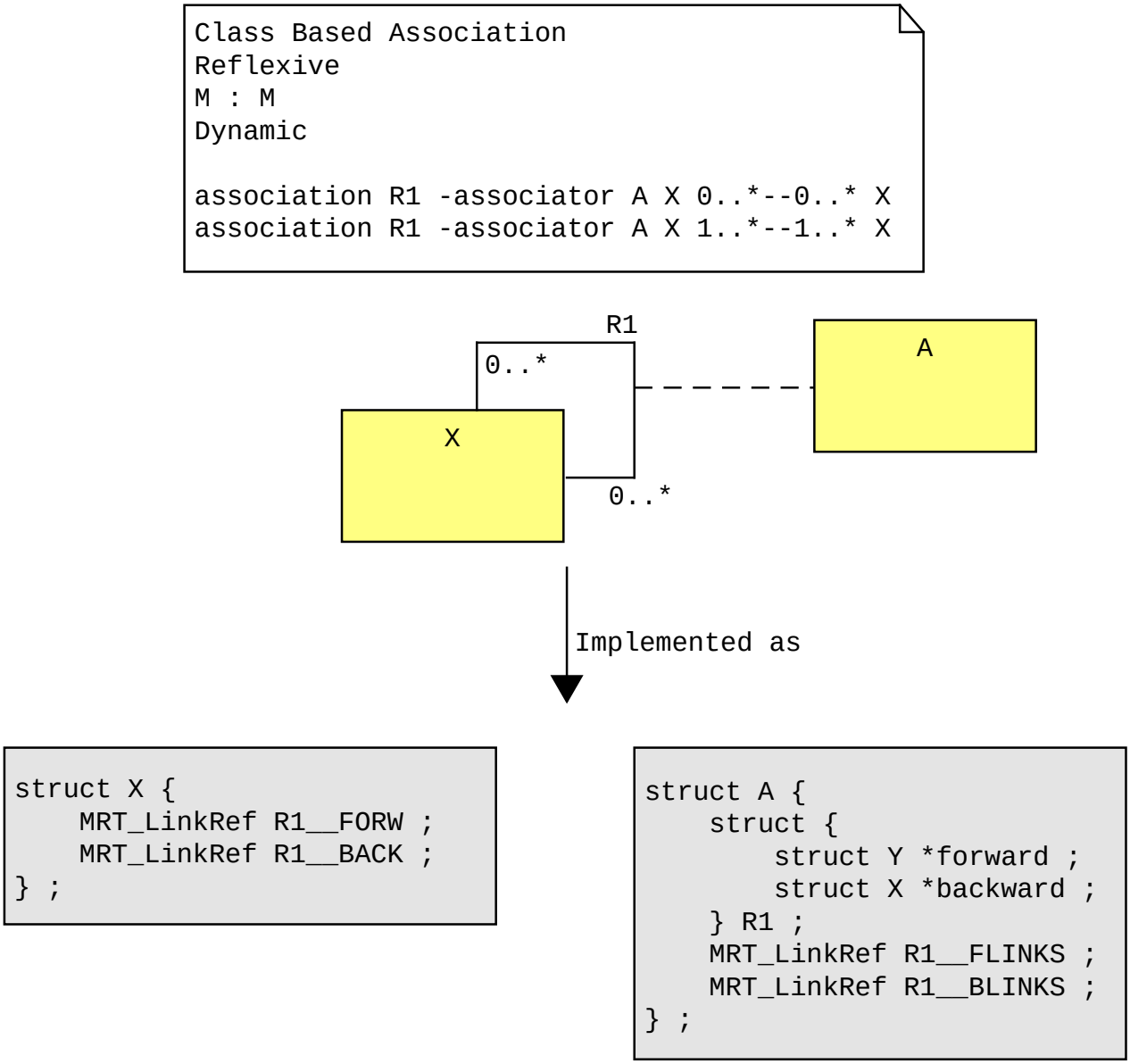
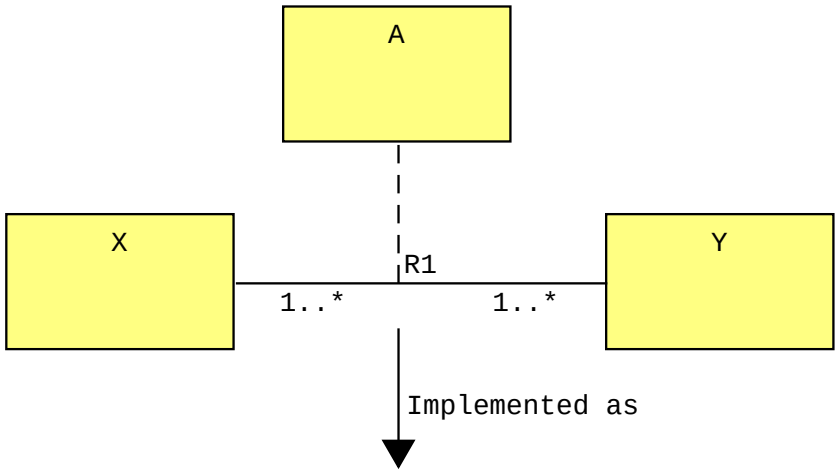


Figure 5.14: Class Based Reflexive M:M Dynamic Association

```
Class Based Association
Non-Reflexive
M : M
Static

association R1 -static -associator A X 0..*--0..* Y
association R1 -static -associator A X 0..*--1..1 Y
association R1 -static -associator A X 1..*--0..* Y
association R1 -static -associator A X 1..*--1..1 Y
```




```
struct A {
    struct {
        struct Y *forward ;
        struct X *backward ;
    } R1 ;
} ;
```

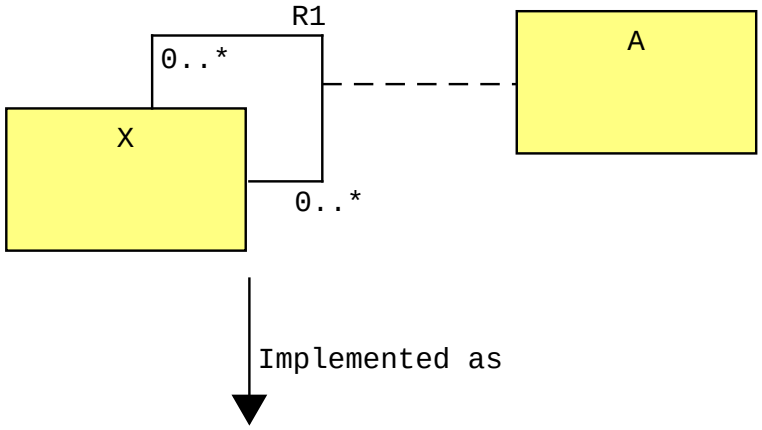
```
struct X {
    struct {
        struct A *const *links ;
        unsigned count ;
    } R1__FORW ;
} ;
```

```
struct Y {
    struct {
        struct A *const *links ;
        unsigned count ;
    } R1__BACK ;
} ;
```

Figure 5.15: Class Based Non-Reflexive M:M Static Association

```
Class Based Association
Reflexive
M : M
Static

association R1 -static -associator A X 0..*--0..* X
association R1 -static -associator A X 1..*--1..* X
```



Implemented as

```
struct X {
    struct {
        struct A *const *links ;
        unsigned count ;
    } R1__FORW ;
    struct {
        struct A *const *links ;
        unsigned count ;
    } R1__BACK ;
} ;
```

```
struct A {
    struct {
        struct Y *forward ;
        struct X *backward ;
    } R1 ;
} ;
```

Figure 5.16: Class Based Reflexive M:M Static Association

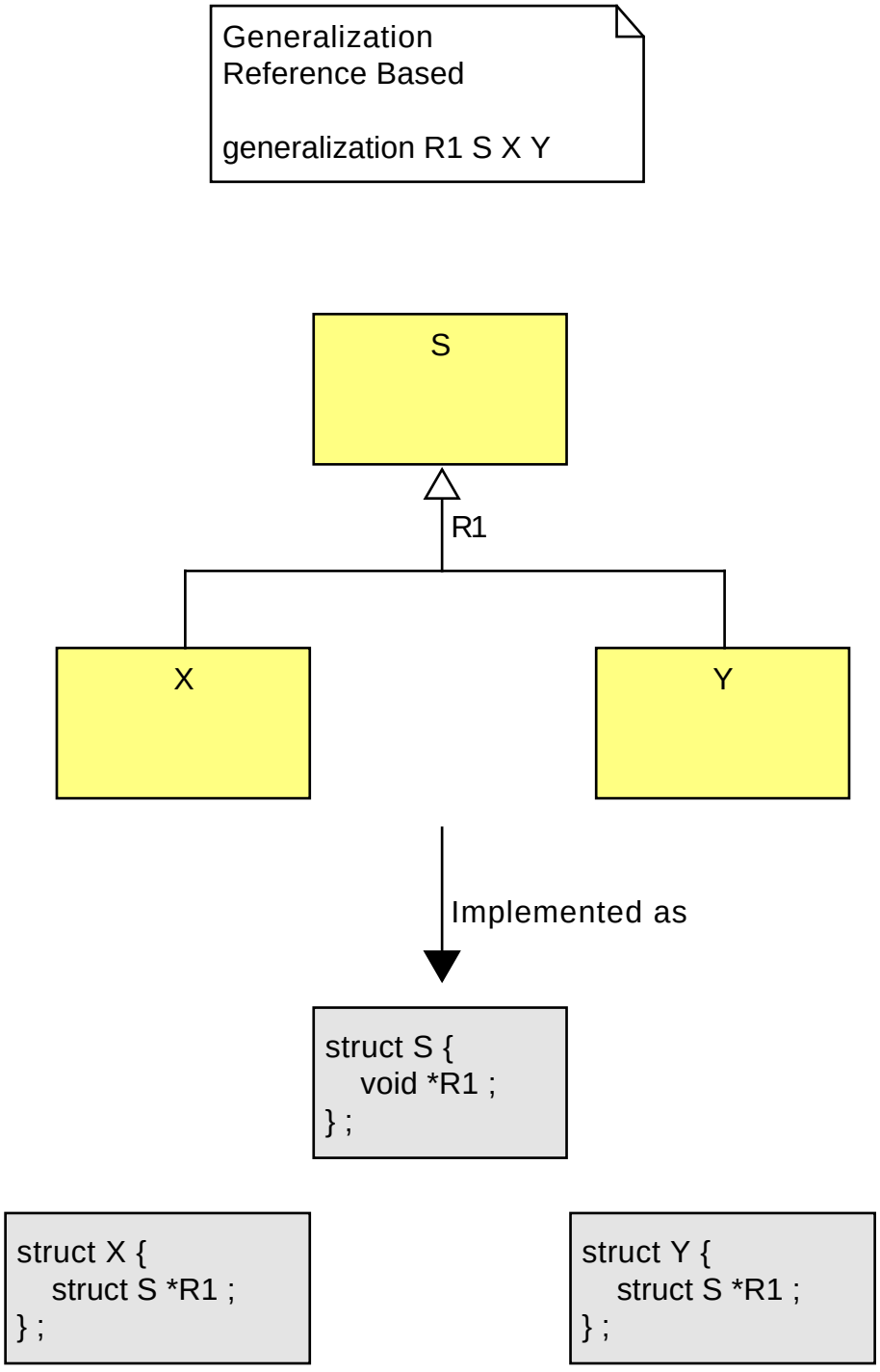


Figure 5.17: Reference Based Generalization

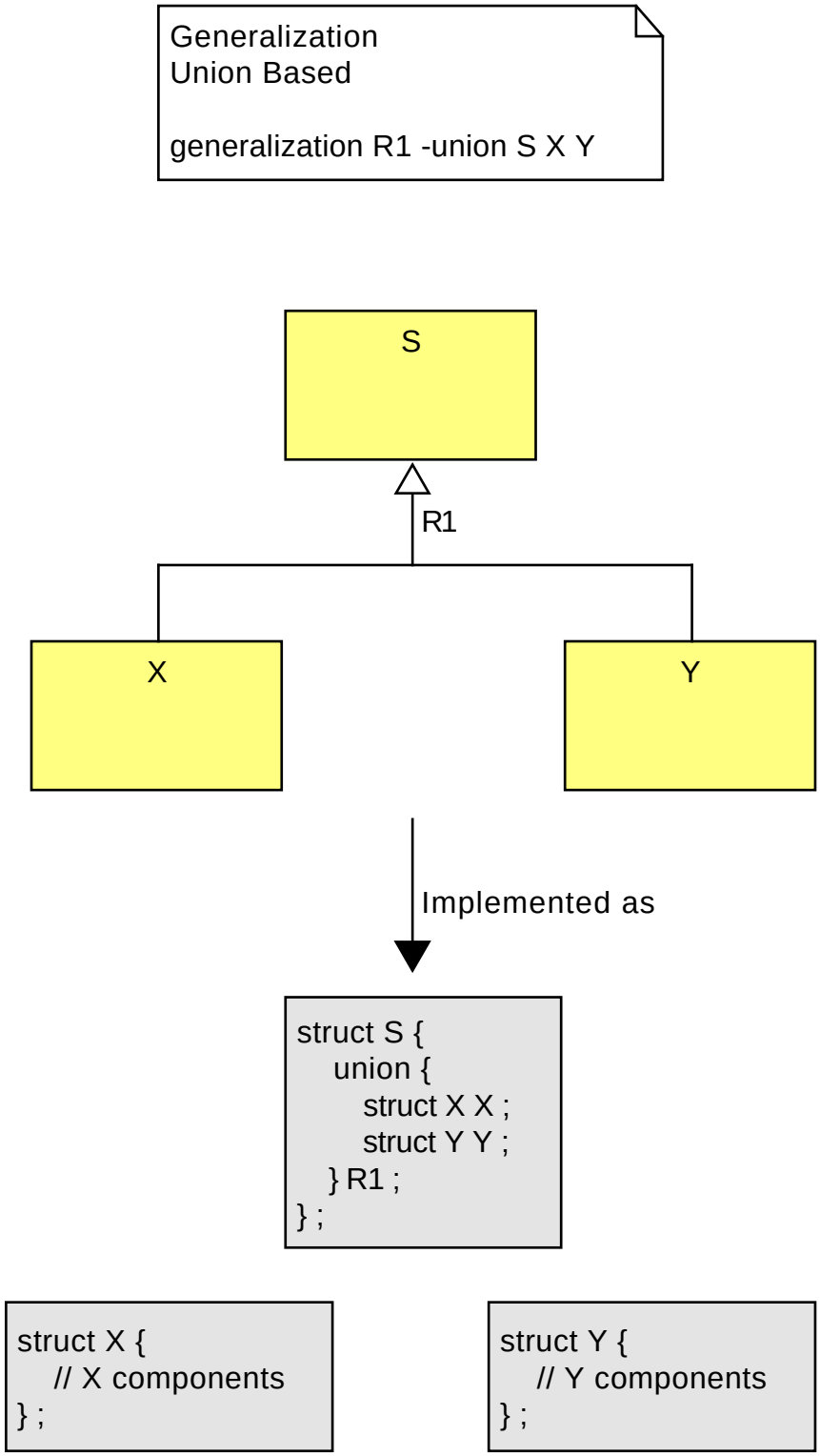


Figure 5.18: Union Based Generalization

Naming Conventions on Relationship Traversal

In addition to modeling the real world associations between classes, relationships also provide a means for the processing of the domain to navigate the class diagram to access related class instances. Here we discuss the syntax conventions we will use to indicate how this navigation will take place. We adopt the strategy that each relationship has a designated *direction*. Navigating in the *forward direction* starts with instances of a source class and ends with instances of a target class. Navigating in the *reverse direction* starts with target class instances and ends with source class instances. To designate the forward direction we will use the unadorned relationship name, *e.g.* `R22` and the reverse direction is indicated by placing a tilde character before the relationship name, *e.g.* `~R22`. The motivation behind this convention is to minimize the amount of specification that needs to be given for navigating relationships and to disambiguate the case of a reflexive relationship where the source class and target class are the same. Given the specification of a relationship direction, the system knows which classes are involved and that detail need not be restated when specifying the navigation. When specifying a direction to a relationship there are three cases to consider.

For simple assocations, the forward direction of the relationship is from the referring class to the referenced class. In a simple association, one class contains referential attributes and is hence the referring class. The other class contains the referenced identifying attributes and is deemed the referenced class. The forward direction is then the traversal from instances of the referring class to instances of the referenced class.

For generalization relationships, the forward direction of the relationship is also from the referring class to the referenced class. In a generalization, the subclass instances are always the referring instances. Navigating from the superclass to a subclass requires additional information, namely which subclass is to be the target. Navigating from a superclass instance to a subclass instance can yield at most one subclass instance and possibly zero if the superclass instance is not actually related to the requested subclass. So the syntax convention to navigate from a superclass instance to a subclass instance specifies navigating in the reverse direction as well as the desired subclass name, *e.g.* `~R42 TableLamp`.

For class based associations, it is the associator class that holds referential attributes that refer to both participating classes. In this case, the forward direction of the relationships must simply be specified and one class is designated the source and the other the target. Since it is also possible to navigate from a participating class to the associator class itself, we will specify that case by including the associator class name, similar to the way generalization relationship navigation is specified. Navigating from source to target is specified as the unadorned relationship name, *e.g.* `R14`. Navigating from target to source uses the tilde notation, *e.g.* `~R14`. Navigating from the source to the associator includes the associator class name, *e.g.* `R14 Ownership` and from target to associator uses the tilde and the associator class name, *e.g.* `~R14 Ownership`.

Only associations can be reflexive (*i.e.* the source and target are the same class). For generalization relationships, we insist that all the subclasses are distinct and that they are distinct from the superclass. The conventions used here serve to disambiguate association traversal in the reflexive case, however, additional specification is required to fix the precise traversal path for the case of one-to-one and many-to-many reflexive relationships. For a one-to-one reflexive association, we will need to specify which attributes are the referential ones and that will set the direction. For a many-to-many association an associator class is required and again we will have to specify the path from source to target.

Conventions Used for Class Based Associations

Class based association pose some additional complexity when describing the way in which they are mapped to the implementation realm. The associator class represents instances of the association between the participants. Each instance of the associator class corresponds one-to-one to instances of the association itself. As we discussed above, in a class based association one participant will be deemed the source and the other the target in order to specify the direction of the relationship for traversal. The associator class itself always contains the referential attributes and in our implementation strategy, will contain two pointers, one each to the participating class instances. The pointer values will never be `NULL` as the reference from the associator class to one of the participants is always unconditional. We will decompose a class based association to make the role of the associator class explicit. This is shown in the diagram below.

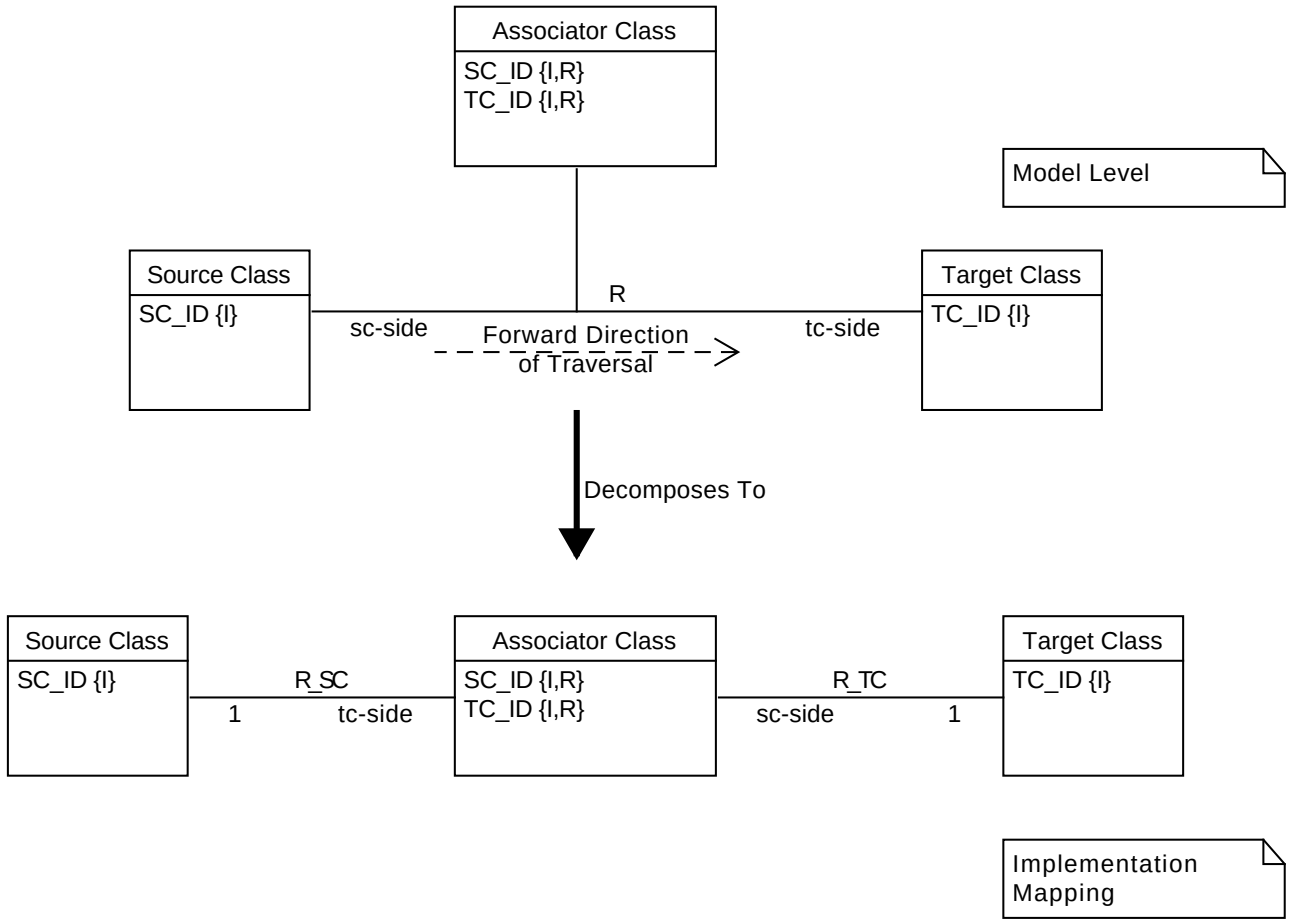


Figure 5.19: Decomposition of Class Based Associations

Note that as part of the decomposition, the multiplicity and conditionality of the source and target sides have been switched. Because the references from the associator class to the participants is unconditional, the multiplicity and conditionality of the decomposed association when going from the source class to the associator class is the same as that specified for the target side at the model level and conversely for the target class. The apparent switch of the multiplicity and conditionality results from both the decomposition, which makes the role of the associator class and its pointer references explicit, and the unconditional nature of the references from the associator class to the participant classes.

With this in mind, we describe the roles that classes play in a class based association below. We will have occasion to refer back to this explanation when we generate the code for class based associations.

Class Diagram

Below is the UML class diagram for the relationship subsystem of the platform model.

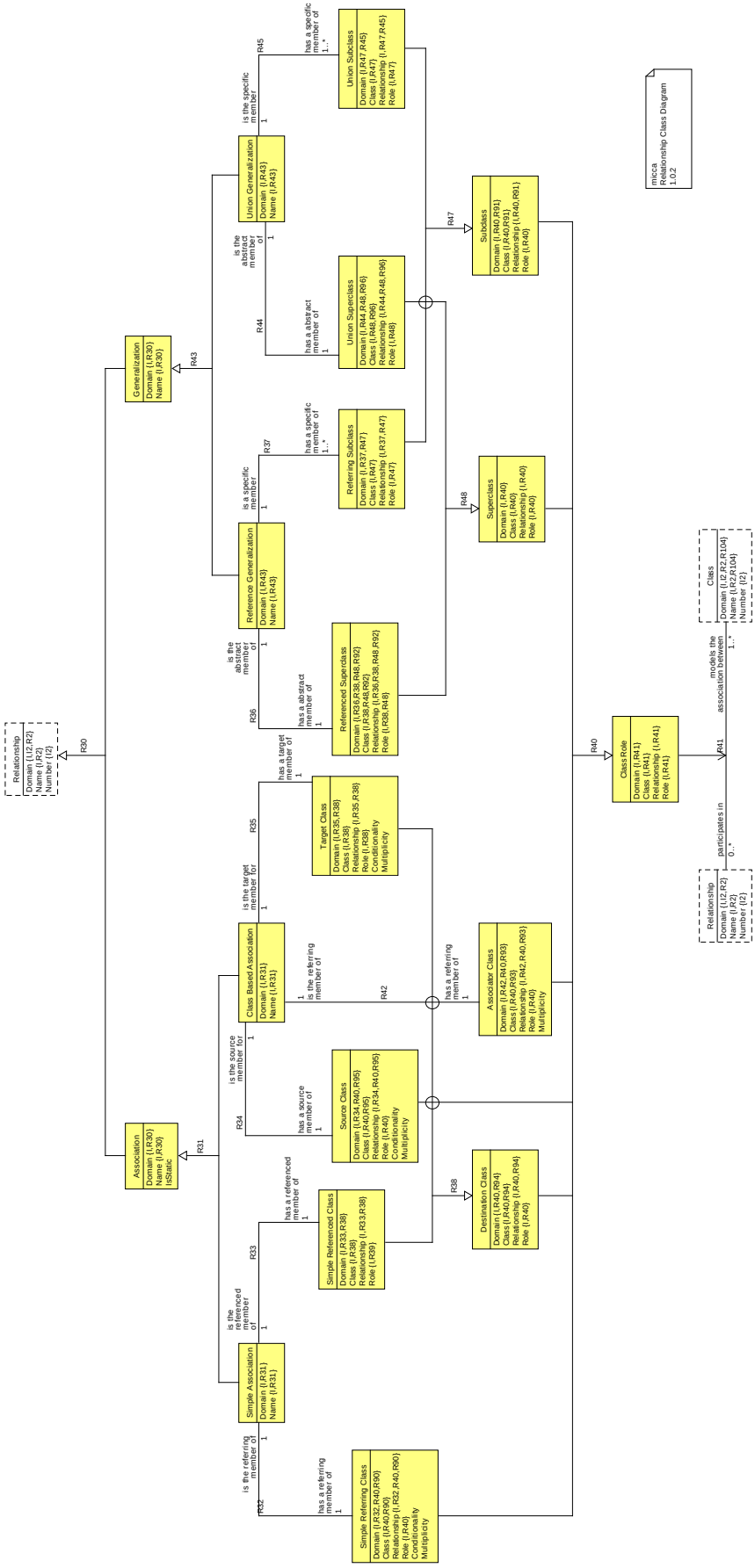


Figure 5.20: Relationship Subsystem Class Diagram

There are two types of **Relationship** (**R30**), namely **Association** and **Generalization**. In turn, there are two types of **Association** (**R31**), namely **Simple Association** and **Class Based Association**. A **Class Based Association** uses a distinct **Associator Class** (**R42**) to hold references to the participating **Source Class** (**R34**) and **Target Class** (**R35**). A **Simple Association** is one where the multiplicity and conditionality are such that all the references can be located in one of the participating classes. The **Simple Referring Class** houses the references (**R32**) while the **Simple Referenced Class** is the one referred to (**R33**). `Micca` supports two ways to handle generalization relationships (**R43**). A **Reference Generalization** uses pointer values to implement the relationship. In this arrangement, a **Referenced Superclass** (**R36**) holds a pointer and encoded type to a **Referring Subclass** (**R37**). The **Referring Subclass** also holds a pointer to its related superclass instances. A **Union Generalization** uses a discriminated union to implement the relationship. This leads to the complementary ideas of a **Union Superclass** (**R44**) and a **Union Subclass** (**R45**). All of these class roles are specialized as a **Class Role** (**R40**) which is the many-to-many-to-many correlation between **Relationship** and **Class** (**R41**).

Association

An Association is a mapping between the instances of two classes. The two classes need not be distinct. If the association is between instances of the same class, then it is called a *reflexive* association. If the association is between instances of distinct classes, then it is called a *non-reflexive* association.

Domain {I,R30}
The name of the domain to which the association belongs.

Data Type Refers to Relationship.Domain

Name {I,R30}
The name of the association.

Data Type Refers to Relationship.Name.

Implementation

```
<<micca configuration>>=
class Association {
    attribute Domain string -id 1
    attribute Name string -id 1
    attribute IsStatic boolean -default false

    reference R30 Relationship -link Domain -link Name
}
```

Generalization

An Generalization is a partitioning of a class into a disjoint union.

Domain {I,R30}
The name of the domain to which the generalization belongs.

Data Type Refers to Relationship.Domain

Name {I,R30}
The name of the generalization.

Data Type Refers to Relationship.Name.

Implementation

```
<<micca configuration>>=
class Generalization {
    attribute Domain string -id 1
    attribute Name string -id 1

    reference R30 Relationship -link Domain -link Name
}
```

R30

- **Relationship** is an **Association** or **Generalization**

There are two fundamental types of relationships. These types are complementary to each other. An Association is fundamentally related to the Cartesian product of the participants. A Generalization is related to the disjoint union of the particiants.

Implementation

```
<<micca configuration>>=
generalization R30 Relationship Association Generalization
```

R31

- **Association** is a **Simple Association** or **Class Based Association**

Associations are mappings between class instances. In its most general form, a separate class is used to hold the mappings and these are called **Class Based Associations**. For the special case of one-to-one and one-to-many multiplicities, the realization of the association can be simplified by placing the association mappings into one of the participating classes. These are known as **Simple Association**.

Implementation

```
<<micca configuration>>=
generalization R31 Association ClassBasedAssociation SimpleAssociation
```

Class Based Association

A Class Based Association is a that type of association where a distinct class is used to implement the relationship.

Domain {I,R31}
The name of the domain to which the class based association belongs.

Data Type Refers to Association.Domain

Name {I,R31}
The name of the class based association.

Data Type Refers to Association.Name.

Implementation

```
<<micca configuration>>=
class ClassBasedAssociation {
    attribute Domain string -id 1
    attribute Name string -id 1

    reference R31 Association -link Domain -link Name
}
```

Simple Association

A Simple Association is a that type of association where the relationship is implemented using attributes in one of the participating classes.

Domain {I,R31}
The name of the domain to which the simple association belongs.

Data Type Refers to Association.Domain

Name {I,R31}
The name of the simple association.

Data Type Refers to Association.Name.

Implementation

```
<<micca configuration>>=
class SimpleAssociation {
    attribute Domain string -id 1
    attribute Name string -id 1

    reference R31 Association -link Domain -link Name
}
```

R43

- **Generalization** is a **Reference Generalization** or **Union Generalization**

The `micca` platform supports two different techniques to hold generalization relationship information. One technique is to use pointer values in a way similar to association relationships. The other techique is to use a discriminated union to hold the subclass instance as part of the superclass instance.

Implementation

```
<<micca configuration>>=
generalization R43 Generalization ReferenceGeneralization UnionGeneralization
```

Reference Generalization

A Reference Generalization is that type of generalization where pointer references are used to implement the relationship.

Domain {I,R43}
The name of the domain to which the reference generalization belongs.

Data Type Refers to Generalization.Domain

Name {I,R43}
The name of the reference generalization association.

Data Type Refers to Generalization.Name.

Implementation

```
<<micca configuration>>=
class ReferenceGeneralization {
    attribute Domain string -id 1
    attribute Name string -id 1

    reference R43 Generalization -link Domain -link Name
}
```

Union Association

A Union Generalization is that type of generalization where the subclass is held as a discriminated union within the superclass instance. Because of the properties of a generalization relationship, a superclass is never related to more than one of its subclasses and so a union of all the subclass structures may be used directly as the subclass instance storage.

Domain {I,R43}
The name of the domain to which the union generalization belongs.

Data Type Refers to Generalization.Domain

Name {I,R43}
The name of the union generalization.

Data Type Refers to Generalization.Name.

Implementation

```
<<micca configuration>>=
class UnionGeneralization {
    attribute Domain string -id 1
    attribute Name string -id 1

    reference R43 Generalization -link Domain -link Name
}
```

Simple Referring Class

In a Simple Association between two classes, one class has the role of refering to the other class in the association. That role is termed the Simple Referring Class. This class is deemed to be the source of relationship navigation when going in the forward direction. From a relational point of view, the Simple Referring Class is the class in the Simple Association which contains referential attributes.

Domain {I,R32,R40,R90}
The name of the domain to which the simple referring class belongs.

Data Type Refers to Simple Association.Domain, Class Role.Domain and Association Reference.Domain

Class {I,R40,R90}
The name of the simple referring class.

Data Type Refers to Class Role.Domain and Association Reference.Domain

Relationship {I,R32,R40,R90}
The name of the relationship in which the simple referring class participates.

Data Type Refers to Simple Association.Name, Class Role.Relationship and Association Reference.Relationship

Role {I,R40}
The role the simple referring class plays in the relationship.

Data Type Refers to Class Role.Role

Conditionality
This attribute records whether the reference by the class is conditional. Conditional associations allow for there to be zero instances referred to. Unconditional associations insist that there must be at some instance referred to.

Data Type boolean

Multiplicity
This attribute records whether the reference by the class can be multiple. Muliple associations may refer to more than one associated instances. Singular association may refer to only one associated instance.

Data Type boolean

Implementation

```
<<micca configuration>>=
class SimpleReferringClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1
    attribute Conditionality boolean
```

```
    attribute Multiplicity boolean

    reference R32 SimpleAssociation -link Domain -link {Relationship Name}
    reference R40 ClassRole\
        -link Domain -link Class -link Relationship -link Role
    reference R90 AssociationReference -link Domain -link Class\
            -link {Relationship Name}
}
```

R32

- **Simple Referring Class** is the referring member of *exactly one* **Simple Association**
- **Simple Association** has a referring member of *exactly one* **Simple Referring Class**

A **Simple Association** is one where referential attributes are placed in one of participating class. That class then serves the role of a **Simple Referring Class**. Each **Simple Association** must have a class that acts as a **Simple Referring Class** and a class can only be a **Simple Referring Class** within the context of one **Simple Association** relationship.

Implementation

```
<<micca configuration>>=
association R32 SimpleReferringClass 1--1 SimpleAssociation
```

Simple Referenced Class

In a Simple Association between two classes, one class has the role of being referenced by the other class in the association. That role is termed the Simple Referenced Class. This class is deemed to be the target of relationship navigation when going in the forward direction. From a relational point of view, the Simple Referenced Class is the class in the Simple Association which contains identifying attributes referenced by the Simple Referring Class.

Domain {I,R33,R38}
The name of the domain to which the simple referenced class belongs.

Data Type Refers to Simple Association.Domain and Back Referring Class.Domain.

Class {I,R38}
The name of the simple referenced class.

Data Type Refers to Back Referring Class.Class

Relationship {I,R33,R38}
The name of the relationship in which the simple referenced class participates.

Data Type Refers to Simple Association.Name and Back Referring Class.Relationship

Role {I,R38}
The role the simple referenced class plays in the relationship.

Data Type Refers to Back Referring Class.Role

Note that there are no Conditionality and Multiplicity attributes. A Simple Referenced Class is always referred to by at exactly one Simple Referring Class. Simple Associations exhibit this property and all other associations are Class Based Associations where the conditionality of the association is based on whether there are instances of the Associator Class.

Implementation

```
<<micca configuration>>=
class SimpleReferencedClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R33 SimpleAssociation -link Domain -link {Relationship Name}
    reference R38 DestinationClass\
        -link Domain -link Class -link Relationship -link Role
}
```

R33

- **Simple Referenced Class** is the referenced member of *exactly one* **Simple Association**
- **Simple Association** has a referenced member of *exactly one* **Simple Referenced Class**

A **Simple Association** is one where referential attributes are placed in one of participating class that refer to the identifying attributes of the other class. The class containing the referenced identifying attributes serves the role of a **Simple Referenced Class**. Each **Simple Association** must have a class that acts as a **Simple Referenced Class** and a class can only be a **Simple Referenced Class** within the context of one **Simple Association** relationship.

Implementation

```
<<micca configuration>>=
association R33 SimpleReferencedClass 1--1 SimpleAssociation
```

Source Class

For a Class Based Association, the Source Class is the starting class from which the association can be traversed in the forward direction.

Domain {I,R34,R38}
The name of the domain to which the source class belongs.

Data Type Refers to Simple Association.Domain and Back Referring Class.Domain.

Class {I,R38}
The name of the source class.

Data Type Refers to Back Referring Class.Class

Relationship {I,R34,R38}
The name of the relationship in which the source class participates.

Data Type Refers to Simple Association.Name and Back Referring Class.Relationship

Role {I,R38}
The role the source class plays in the relationship.

Data Type Refers to Back Referring Class.Role

Conditionality
This attribute records whether the reference by the class is conditional. Conditional associations allow for there to be zero instances referred to. Unconditional associations insist that there must be at some instance referred to.

Data Type boolean

Multiplicity
This attribute records whether the reference by the class can be multiple. Muliple associations may refer to more than one associated instances. Singular association may refer to only one associated instance.

Data Type boolean

Implementation

```
<<micca configuration>>=
class SourceClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1
    attribute Conditionality boolean
    attribute Multiplicity boolean

    reference R34 ClassBasedAssociation -link Domain -link {Relationship Name}
    reference R40 ClassRole\
        -link Domain -link Class -link Relationship -link Role
    reference R95 ForwardReference -link Domain -link Class -link Relationship\
        -refid 2
}
```

R34

- **Source Class** is the source member for *exactly one* **Class Based Association**
- **Class Based Association** has a source member of *exactly one* **Source Class**

Each Class Based Association designates one of the participant classes to serve the role as the source of traversal.

Implementation

```
<<micca configuration>>=
association R34 SourceClass 1--1 ClassBasedAssociation
```

Target Class

For a Class Based Association, the Target Class is that class where, when the association is traversed in the forward direction starting at the Source Class, the instances related by the association are found. Alternatively, the Target Class can serve as the start of a traversal of the association in the reverse direction.

Domain {I,R35,R38}
The name of the domain to which the target class belongs.

Data Type Refers to Simple Association.Domain and Back Referring Class.Domain.

Class {I,R38}
The name of the target class.

Data Type Refers to Back Referring Class.Class

Relationship {I,R35,R38}
The name of the relationship in which the target class participates.

Data Type Refers to Simple Association.Name and Back Referring Class.Relationship

Role {I,R38}
The role the target class plays in the relationship.

Data Type Refers to Association Participant Class.Role

Conditionality
This attribute records whether the reference by the class is conditional. Conditional associations allow for there to be zero instances referred to. Unconditional associations insist that there must be at some instance referred to.

Data Type boolean

Multiplicity
This attribute records whether the reference by the class can be multiple. Muliple associations may refer to more than one associated instances. Singular association may refer to only one associated instance.

Data Type boolean

Implementation

```
<<micca configuration>>=
class TargetClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1
    attribute Conditionality boolean
    attribute Multiplicity boolean

    reference R35 ClassBasedAssociation -link Domain -link {Relationship Name}
    reference R38 DestinationClass\
        -link Domain -link Class -link Relationship -link Role
}
```

R35

- **Target Class** is the target member for *exactly one* **Class Based Association**
- **Class Based Association** has a target member of *exactly one* **Target Class**

Each Class Based Association designates one of the participant classes to serve the role as the target of traversal.

Implementation

```
<<micca configuration>>=
association R35 TargetClass 1--1 ClassBasedAssociation
```

Associator Class

For a Class Based Association, the Associator Class is that class which holds references to the participating classes. Each instance of an Associator Class represents an instance of the association between the participants.

Domain {I,R42,R40,R93}
The name of the domain to which the associator class belongs.

Data Type — Refers to Class Based Association.Domain, Class Role.Domain and Associator Reference.Domain.

Class {I,R40,R93}
The name of the associator class.

Data Type — Refers to Class Role.Domain and Associator Reference.Domain.

Relationship {I,R42,R40,R93}
The name of the relationship in which the associator class participates.

Data Type — Refers to Class Based Association.Name, Class Role.Relationship and Associator Reference.Name.

Role {I,R40}
The role the associator class plays in the relationship.

Data Type — Refers to Class Role.Role

Implementation

```
<<micca configuration>>=
class AssociatorClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1
    attribute Multiplicity boolean

    reference R42 ClassBasedAssociation -link Domain -link {Relationship Name}
    reference R40 ClassRole -link Domain -link Class -link Relationship -link Role
    reference R93 AssociatorReference -link Domain -link Class\
        -link {Relationship Name}
}
```

R42

- **Associator Class** is the referring member of *exactly one* **Class Based Association**
- **Class Based Association** has a referring member of *exactly one* **Associator Class**

Class Based Associations must have a class that serves the role of the associator and holds the references to the participating classes. That role is served by exactly one class in the association.

Implementation

```
<<micca configuration>>=
association R42 AssociatorClass 1--1 ClassBasedAssociation
```

R38

- **Back Referring Class** is a **Simple Referenced Class**, **Source Class** or **Target Class**

All the class roles that are referenced by other classes in relationships can be generalized to be a Back Referring Class. The common characteristic of these classes is that they may hold pointer references to other classes in the association that are used to navigate the relationship in the reverse direction.

Implementation

```
<<micca configuration>>=
generalization R38 DestinationClass\
    SimpleReferencedClass TargetClass
```

Destination Class

An Destination Class is that type of role in an association where the class is the destination of navigating the relationship in the forward direction.

Domain {I,R40,R94}
The name of the domain to which the association participant class belongs.

Data Type Refers to Class Role.Domain and Backward Referenced.Domain

Class {I,R40,R94}
The name of the association participant class.

Data Type Refers to Class Role.Domain and Backward Referenced.Class

Relationship {I,R40,R94}
The name of the relationship in which the association participant class participates.

Data Type Refers to Class Role.Relationship and Backward Referenced.Relationship

Role {I,R40}
The role the association participant class plays in the relationship.

Data Type Refers to Class Role.Role

Implementation

```
<<micca configuration>>=
class DestinationClass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R40 ClassRole -link Domain -link Class -link Relationship -link Role
    reference R94 BackwardReference -link Domain -link Class -link Relationship\
        -refid 2
}
```

Referenced Superclass

A Referenced Superclass is that role in a Reference Generalization assumed by the superclass.

Domain {I,R36,R46,R92}
The name of the domain to which the referenced superclass class belongs.

Data Type Refers to Superclass.Domain, Reference Generalization.Domain and Subclass Reference.Domain

Class {I,R46,R92}
The name of the referenced superclass class.

Data Type Refers to Superclass.Domain and Subclass Reference.Class

Relationship {I,R36,R46,R92}
The name of the relationship in which the referenced superclass class participates.

Data Type Refers to Superclass.Relationship, Reference Generalization.Name and Subclass Reference.Name

Role {I,R46}
The role the referenced superclass class plays in the relationship.

Data Type Refers to Superclass.Role

Implementation

```
<<micca configuration>>=
class ReferencedSuperclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R36 ReferenceGeneralization -link Domain -link {Relationship Name}
    reference R48 Superclass -link Domain -link Class\
            -link Relationship -link Role
    reference R92 SubclassReference -link Domain -link Class\
            -link {Relationship Name}
}
```

R36

- **Referenced Superclass** is the abstract member of *exactly one* **Reference Generalization**
- **Reference Generalization** has an abstract member of *exactly one* **Referenced Superclass**

Each generalization relationship has exactly one member that serves as the abstract or superclass member. We will also insist that superclass roles may not be played by any of the participant subclasses in the generalization.

Implementation

```
<<micca configuration>>=
association R36 ReferencedSuperclass 1--1 ReferenceGeneralization
```

Referring Subclass

For a Reference Generalization, a Referring Subclass is that role played by a subclass.

Domain {I,R37,R47}
The name of the domain to which the referring subclass class belongs.

Data Type Refers to Subclass.Domain and Reference Generalization.Domain

Class {I,R47}
The name of the referring subclass class.

Data Type Refers to Subclass.Class

Relationship {I,R37,R47}
The name of the relationship in which the referring subclass class participates.

Data Type Refers to Subclass.Relationship and Reference Generalization.Name

Role {I,R47}
The role the referring subclass class plays in the relationship.

Data Type Refers to Subclass.Role

Implementation

```
<<micca configuration>>=
class ReferringSubclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R37 ReferenceGeneralization -link Domain -link {Relationship Name}
    reference R47 Subclass -link Domain -link Class -link Relationship -link Role
}
```

R37

- **Referring Subclass** is a specific member of *exactly one* **Reference Generalization**
- **Reference Generalization** has a specific member of *one or more* **Referring Subclass**

Each generalization relationship has one or more participants that are the specific members or subclasses of the generalization. We will also insist that there are at least two subclass members since partitioning a set into a single improper subset serves no useful semantic reason.

Implementation

```
<<micca configuration>>=
association R37 ReferringSubclass 1..*--1 ReferenceGeneralization
```

Union Superclass

For a Union Generalization, the Union Superclass is that class that serves as a superclass in the relationship. This type of superclass will also be constructed so that the storage for any related subclass will be a part of the memory allocated to the superclass instance. This provides a platform specific optimization of memory space for simple generalizations.

Domain {I,R40,R44,R96}
The name of the domain to which the union superclass class belongs.

Data Type Refers to Class Role.Domain, Union Generalization.Domain and Subclass Container.Domain.

Class {I,R40,R96}
The name of the union superclass class.

Data Type Refers to Class Role.Class and Subclass Container.Class.

Relationship {I,R40,R44,R96}
The name of the relationship in which the union superclass class participates.

Data Type Refers to Class Role.Relationship, Union Generalization.Name, and Subclass Container.Name.

Role {I,R40}
The role the union superclass class plays in the relationship.

Data Type Refers to Class Role.Role

Implementation

```
<<micca configuration>>=
class UnionSuperclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R44 UnionGeneralization -link Domain -link {Relationship Name}
    reference R48 Superclass -link Domain -link Class\
            -link Relationship -link Role
    reference R96 SubclassContainer -link Domain -link Class\
        -link {Relationship Name}
}
```

R44

- **Union Superclass** is the abstract member of *exactly one* **Union Generalization**
- **Union Generalization** has an abstract member of *exactly one* **Union Superclass**

Each generalization relationship has exactly one member that serves as the abstract or superclass member. We will also insist that superclass roles may not be played by any of the participant subclasses in the generalization.

Implementation

```
<<micca configuration>>=
association R44 UnionSuperclass 1--1 UnionGeneralization
```

Union Subclass

For a Union Generalization, a Union Subclass is that role played by a subclass in the generalization.

Domain {I,R47,R45}
The name of the domain to which the union subclass class belongs.

Data Type Refers to Subclass.Domain and Union Generalization.Domain.

Class {I,R47}
The name of the union subclass class.

Data Type Refers to Subclass.Domain.

Relationship {I,R47,R45}
The name of the relationship in which the referring subclass class participates.

Data Type Refers to Subclass.Relationship and Union Generalization.Name.

Role {I,R47}
The role the union subclass class plays in the relationship.

Data Type Refers to Subclass.Role

Implementation

```
<<micca configuration>>=
class UnionSubclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R47 Subclass -link Domain -link Class -link Relationship -link Role
    reference R45 UnionGeneralization -link Domain -link {Relationship Name}
}
```

R45

- **Union Subclass** is a specific member of *exactly one* **Union Generalization**
- **Union Generalization** has a specific member of *one or more* **Union Subclass**

Each generalization relationship has one or more participants that are the specific members or subclasses of the generalization. We will also insist that there are at least two subclass members since partitioning a set into a single improper subset serves no useful semantic reason.

Implementation

```
<<micca configuration>>=
association R45 UnionSubclass 1..*--1 UnionGeneralization
```

Superclass

A Superclass is the specific member of a generalization relationship and serves the role of a superclass.

Domain {I,R40}
The name of the domain to which the superclass class belongs.

Data Type Refers to Class Role.Domain

Class {I,R40}
The name of the superclass class.

Data Type Refers to Class Role.Domain

Relationship {I,R40}
The name of the relationship in which the superclass class participates.

Data Type Refers to Class Role.Relationship

Role {I,R40}
The role the superclass class plays in the relationship.

Data Type Refers to Class Role.Role

Implementation

```
<<micca configuration>>=
class Superclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R40 ClassRole -link Domain -link Class -link Relationship -link Role

    instop findSubclasses {} {
        set refsubs [findRelated $self {~R48 ReferencedSuperclass} R36 ~R37 R47]
        set usubs [findRelated $self {~R48 UnionSuperclass} R44 ~R45 R47]
        return [refUnion $refsubs $usubs]
    }
}
```

R48

- **Superclass** is a **Referenced Superclass** or **Union Superclass**

The `micca` platform supports two different implementation means for representing a superclass in a generalization relationship. A generalization may be represented using pointer references or as a union.

Implementation

```
<<micca configuration>>=
generalization R48 Superclass ReferencedSuperclass UnionSuperclass
```

Subclass

A Subclass is the specific member of a generalization relationship and serves the role of a subclass.

Domain {I,R40,R91}
The name of the domain to which the subclass class belongs.

Data Type Refers to Class Role.Domain and Superclass Reference.Domain

Class {I,R40,R91}
The name of the subclass class.

Data Type Refers to Class Role.Domain and Superclass Reference.Class

Relationship {I,R40,R91}
The name of the relationship in which the subclass class participates.

Data Type Refers to Class Role.Relationship and Superclass Reference.Name

Role {I,R40,R91}
The role the subclass class plays in the relationship.

Data Type Refers to Class Role.Role

Implementation

```
<<micca configuration>>=
class Subclass {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R40 ClassRole -link Domain -link Class -link Relationship -link Role
    reference R91 SuperclassReference -link Domain -link Class\
            -link {Relationship Name}
}
```

R47

- **Subclass** is a **Referring Subclass** or **Union Subclass**

The `micca` platform supports two different implementation means for representing a subclass in a generalization relationship. A generalization may be represented using pointer references or as a discriminated union.

Implementation

```
<<micca configuration>>=
generalization R47 Subclass ReferringSubclass UnionSubclass
```

Class Role

A Class Role is an abstraction of the various ways that classes may participate in relationships. The major concept in the relationship subsystem is to define the various types of relationships and then abstract the way that classes participate in the relationships and the roles they play in holding referential attributes, etc.

Domain {I,R41}
The name of the domain to which the class role class belongs.

Data Type Refers to Class.Domain and Relationships.Domain.

Class {I,R41}
The name of the class role class.

Data Type Refers to Class.Name.

Relationship {I,R41}
The name of the relationship in which the class role class participates.

Data Type Refers to Relationship.Name.

Role {I,R40}
The role the class role class plays in the relationship. There are three roles available to a class. It may be a *source*, *target* or *associator*. Note that the Role attribute is part of the identifier of the class. This accounts for reflexive associations where the same class will play two different roles in the association.

Data Type {source, target, associator}.

Implementation

```
<<micca configuration>>=
class ClassRole {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1 -check {$Role in {source target associator}}

    reference R41 Relationship -link Domain -link {Relationship Name}
    reference R41 Class -link Domain -link {Class Name}
}
```

R40

- **Class Role** is a **Simple Referring Class**, **Destination Class**, **Source Class**, **Associator Class**, **Union Superclass**, or **Subclass**

Classes play various roles when that are involved in relationships. This generalization abstracts the various ways that classes are involved in relationships into a role.

Implementation

```
<<micca configuration>>=
generalization R40 ClassRole SimpleReferringClass DestinationClass\
        SourceClass AssociatorClass Superclass Subclass
```

R41

- **Class Role** *is an instance of* **Relationship** models the association between *one or more* **Class**
- **Class Role** *is an instance of* **Class** participates in *zero or more* **Relationship**

Classes may participate in relationships to model the real world associations between the classes. It is possible (but not usual) for a class not to participate in any relationship. Relationships alway involve the participation of one or more classes (usually two or three). Note that **Class Role** has an additional identifying attribute, **Role**, so that **R41** is actually a many-to-many-to-many association. This additional multiplicity is required since reflexive relationships have the same class playing more than one role in the relationship.

Implementation

```
<<micca configuration>>=
association R41 Relationship 0..*--1..* Class -associator ClassRole
```

Chapter 6

References Subsystem

Introduction

In the Relationship subsystem, we saw how the platform models relationships and how classes play a role in realizing the relationship. In the References subsystem, we map the roles a class plays in realizing a relationship to the components that the class instances must store. This mapping joins back with the Class Components seen in the Classes subsystem

Below is the UML class diagram for the references subsystem of the platform model.

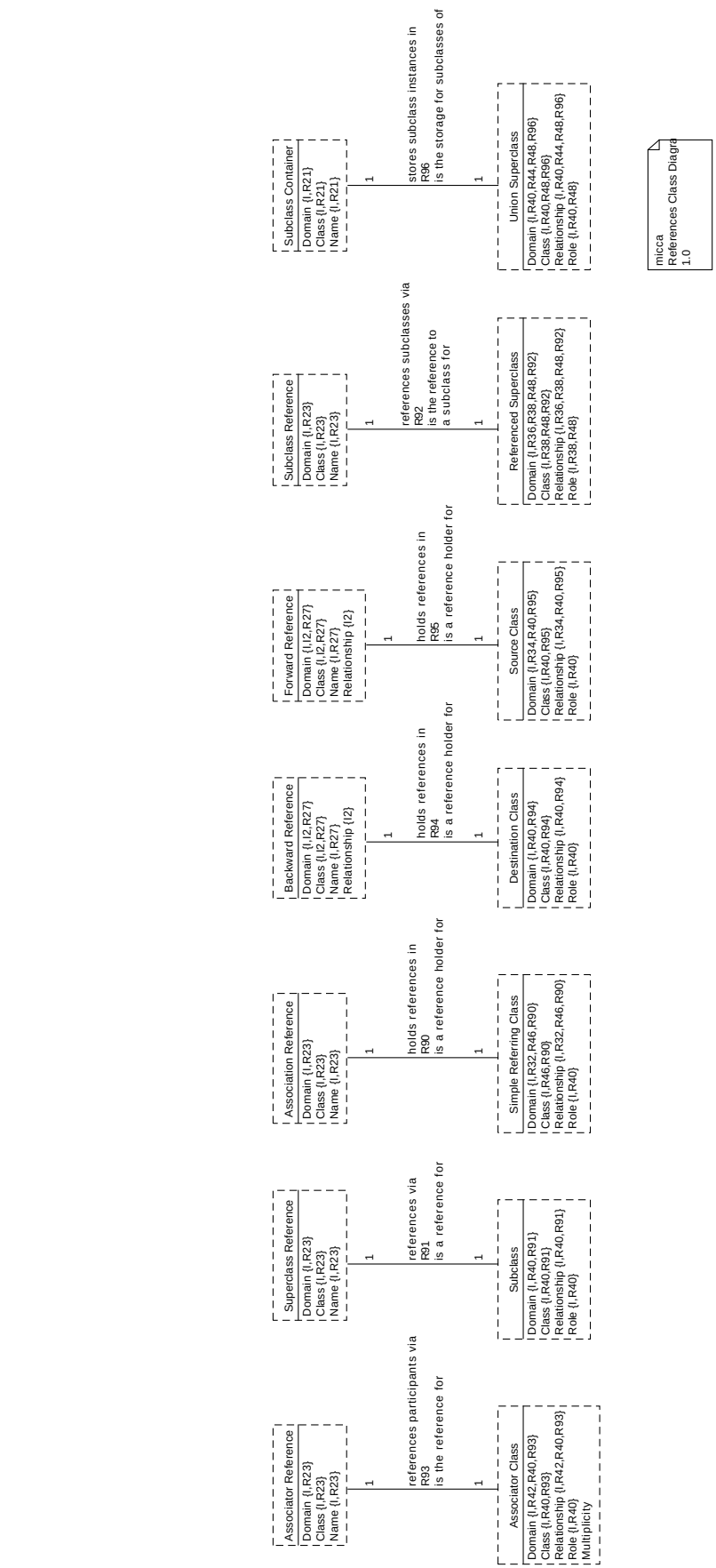


Figure 6.1: References Subsystem Class Diagram

R90

- **Simple Referring Class** holds references in *exactly one* **Association Reference**
- **Association Reference** is a reference holder for *exactly one* **Simple Referring Class**

When a class serves as an Simple Referring Class it may hold pointer references to the other class in the association. Association References hold the pointer information needed to traverse the relationship in the forward direction.

Implementation

```
<<micca configuration>>=
association R90 SimpleReferringClass 1--1 AssociationReference
```

R91

- **Subclass** references via *exactly one* **Superclass Reference**
- **Superclass Reference** is a reference for *exactly one* **Subclass**

A Subclass must always refer to its related Superclass and it does so by having a Superclass Reference as one of its components.

Implementation

```
<<micca configuration>>=
association R91 Subclass 1--1 SuperclassReference
```

R94

- **Destination Class** holds references in *exactly one* **Backward Reference**
- **Backward Reference** is a reference holder for *exactly one* **Destination Class**

When a class serves as an Destination Class it holds pointer references to the other class in the association. These pointer references aid in navigating the relationship in the reverse direction.

Implementation

```
<<micca configuration>>=
association R94 DestinationClass 1--1 BackwardReference
```

R95

- **Source Class** holds references in *exactly one* **Forward Reference**
- **Forward Reference** is a reference holder for *exactly one* **Source Class**

When a class serves as an Source Class it holds pointer references to the associator class in the class based association. These pointer references aid in navigating the relationship in the forward direction.

Implementation

```
<<micca configuration>>=
association R95 SourceClass 1--1 ForwardReference
```

R92

- **Referenced Superclass** references subclasses via *at most one* **Subclass Reference**
- **Subclass Reference** is the reference to a subclass for *exactly one* **Referenced Superclass**

When a Referenced Superclass in a reference generalization stores its reference to its currently related subclass in a **Subclass Reference**. However, a Referenced Superclass may exist before its related Subclass Reference is generated.

Implementation

```
<<micca configuration>>=
association R92 ReferencedSuperclass 1--1 SubclassReference
```

R93

- **Associator Class** references participants via *exactly one* **Associator Reference**
- **Associator Reference** is the reference for *exactly one* **Associator Class**

An Associator Class must hold references to both participants of the association and does so in an Associator Reference.

Implementation

```
<<micca configuration>>=
association R93 AssociatorClass 1--1 AssociatorReference
```

R96

- **Union Superclass** stores subclass instances in *at most one* **Subclass Container**
- **Subclass Container** is the storage for subclasses of *exactly one* **Union Superclass**

The use of a union to store associated subclass instances in a superclass instance memory structure eliminates the need for various reference pointers. The Union Superclass stores the currently related subclass instance in the Subclass Container that can be used to disinguish the type of the related subclass. However, a Union Superclass may exist before its related Subclass Container is generated.

Implementation

```
<<micca configuration>>=
association R96 UnionSuperclass 1--1 SubclassContainer
```

Chapter 7

State Model Subsystem

Introduction

The class, relationship and reference subsystems describe how model level concepts of classes and relationships are mapped to implementation level concepts of structures and pointers. In the state model subsystem we turn our attention to the dynamics and sequencing of execution of the model.

The classes in the state model subsystem capture the essential information of a Moore type state model along with the XUML conventions of how to specify ignored and error transitions. The sequencing of execution via dispatch of state machine events is handled by the run time mechanisms of `micca`. The run time is completely data driven. So our focus in the platform model is to capture that data needed to generate the data structures used by the run time mechanisms. The implications of this are that the state model subsystem is relatively platform independent. There are no particular "C" language implications of dispatching events to state machines since the there is no particular "C" language constructs specific to state machines. The run time mechanisms must supply all the logic around event dispatch. We will find it convenient to include some sequential numeric attributes since the run time will use array data structures to access some types of information and we would like a rather direct way of computing appropriate array indices.

Below is the UML class diagram for the state model subsystem of the platform model.

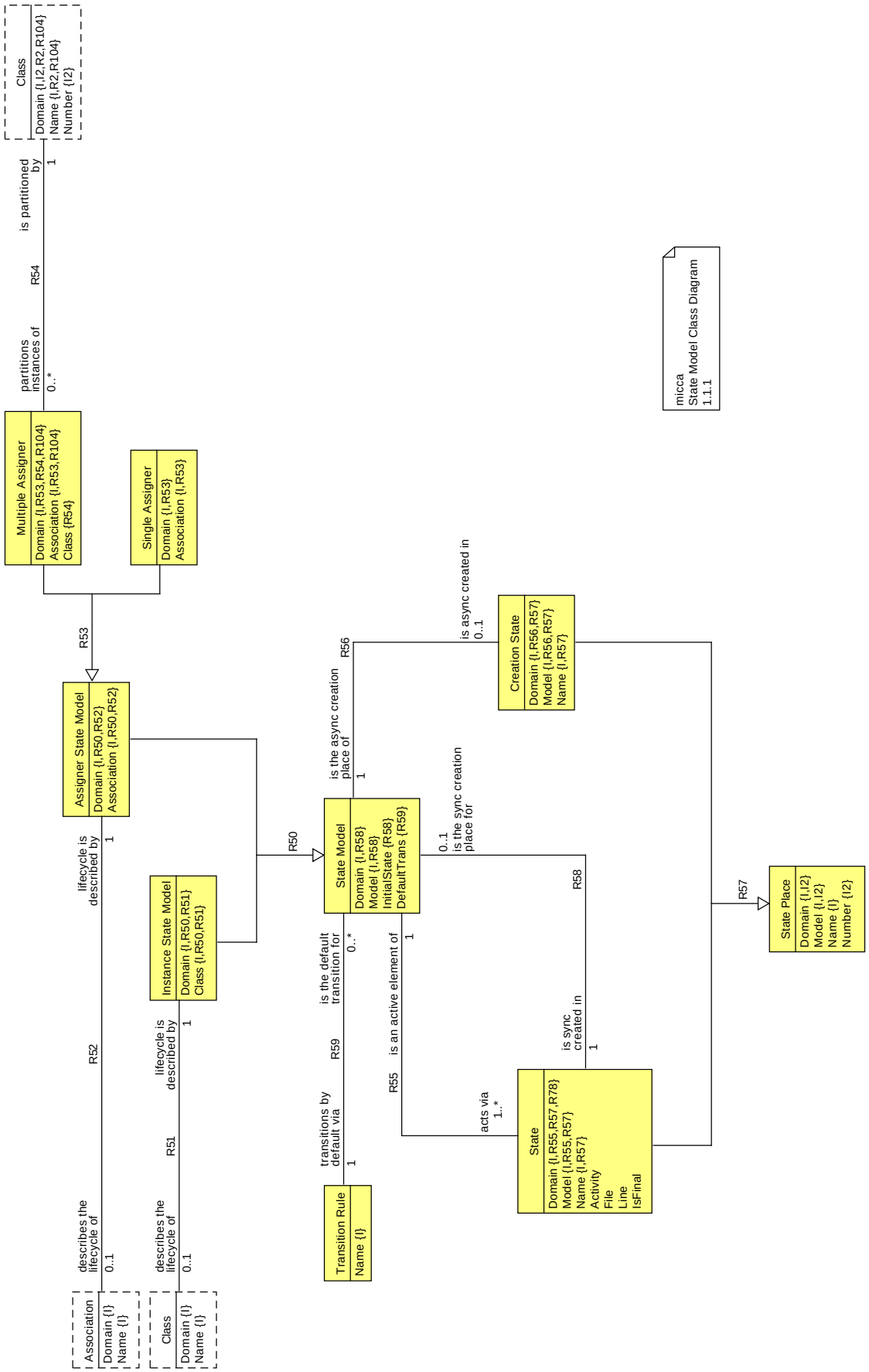


Figure 7.1: State Model Subsystem Class Diagram

A **State Model** is of one of two types (**R50**), an **Instance State Model** or an **Assigner State Model**. An **Instance State Model** is associated with a **Class** (**R51**) and an **Assigner State Model** is associated with an **Association** relationship (**R52**). Both types of state models operate the same. They contain a set of **States** (**R55**) and possibly a **Creation State** (**R56**). A **State Model** has a default **Transition Rule** (**R59**) to resolve unspecified transitions and an initial state (**R58**) to know where things start. All the **States** and the **Creation State** constitute a **State Place** where the state model can transition when it has an event dispatched to it (**R57**). An **Assigner State Model** also consists of two types (**R53**), **Single Assigner** and **Multiple Assigner**. A **Single Assigner** has only a single instance while a **Multiple Assigner** has a many instances as the **Class** that partitions is behavior (**R54**).

State Model

A State Model is a finite automaton, specifically a Moore type state machine. A State Model is used to define the sequencing of execution for the lifecycle of its assocated class or assigner. The State Model defines behavior that is common to all of the associated instances or assigners and each class instance or assigner instance has its own notion of its current state.

Domain {I,R58}
The domain in which the state model resides.

Data Type Refers to State.Domain

Model {I,R58}
The name of the class or association for which the state model operates.

Data Type Refers to State.Model

InitialState {R58}
The name of the state into which the a newly create state model is placed by default.

Data Type Refers to State.Name

DefaultTrans {R59}
The name of the default transition rule.

Data Type Refers to Transition Rule.Name

Implementation

```
<<micca configuration>>=
class StateModel {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute InitialState string
    attribute DefaultTrans string

    reference R58 State -link Domain -link Model -link {InitialState Name}
    reference R59 TransitionRule -link {DefaultTrans Name}
}
```

R58

- **State Model** is sync created in *exactly one* **State**
- **State** is the sync creation place for *at most one* **State Model**

When an entity that has an associated State Model is create synchronously, it is placed in an initial state. No state activity is run as part of the synchronous creation. Not all States of a State Model serve the role of being the default initial state.

Implementation

```
<<micca configuration>>=
association R58 StateModel 0..1--1 State
```

R59

- **State Model** transitions by default via *exactly one* **Transition Rule**
- **Transition Rule** is the default transition for *zero or more* **State Model**

When specifying the state transitions for a state model, not all transitions must be specified explicitly. Those transition not explicitly specified are are give the value of a Transition Rule. Since there multiple Transition Rules, any given Transition Rule may not be used as a default transition for a State Model.

Implementation

```
<<micca configuration>>=
association R59 StateModel 0..*--1 TransitionRule
```

R50

- **State Model** is a **Instance State Model** or **Assigner State Model**

The execution rules of XUML allow the lifecycle of either a class or an association to be described by a State Model. These are the only two domain entities that exhibit lifecycle behavior.

Implementation

```
<<micca configuration>>=
generalization R50 StateModel InstanceStateModel AssignerStateModel
```

Instance State Model

An Instance State Model is that type of state model associated with the lifecycle of a class instance.

Domain {I,R50,R51}
The domain in which the instance state model resides.

Data Type Refers to Class.Domain and State Model.Domain

Class {I,R50,R51}
The name of the class for which the instance state model operates.

Data Type Refers to Class.Name and State Model.Model

Implementation

```
<<micca configuration>>=
class InstanceStateModel {
    attribute Domain string -id 1
    attribute Class string -id 1

    reference R50 StateModel -link Domain -link {Class Model}
    reference R51 Class -link Domain -link {Class Name}
}
```

R51

- **Instance State Model** describes the lifecycle of *exactly one* **Class**
- **Class** lifecycle is described by *at most one* **Instance State Model**

An Instance State Model is the definition of the lifecycle behavior for only one class but not all classes will exhibit lifecycle behavior and so may not have an Instance State Model.

Implementation

```
<<micca configuration>>=
association R51 InstanceStateModel 0..1--1 Class
```

Assigner State Model

An Assigner State Model is that type of state model associated with the lifecycle of an association relationship. Sometimes the semantics of the domain subject matter are such that forming and breaking an association must be serialized. The archetypical example is where the association is competitive and parallel activity could create incorrect association linkages.

Domain {I,R50,R52}
The domain in which the assigner state model resides.

Data Type Refers to Association.Domain and State Model.Domain

Association {I,R50,R52}
The name of the association relationship for which the assigner state model operates.

Data Type Refers to Association.Name and State Model.Model

Implementation

```
<<micca configuration>>=
class AssignerStateModel {
    attribute Domain string -id 1
    attribute Association string -id 1

    reference R50 StateModel -link Domain -link {Association Model}
    reference R52 Association -link Domain -link {Association Name}
}
```

R52

- **Assigner State Model** describes the lifecycle of *exactly one* **Association**
- **Association** lifecycle is described by *at most one* **Assigner State Model**

An Assigner State Model is the definition of the lifecycle behavior for only one association relationship but not all associations will exhibit lifecycle behavior and so may not have an Assigner State Model.

Implementation

```
<<micca configuration>>=
association R52 AssignerStateModel 0..1--1 Association
```

State

A State is one of the components of a State Model. States have a name that is disctinct among the other states of the State Model and have an executable activity that is run by the system when the state is entered upon a transition.

Domain {I,R55,R57,R78}
The domain in which the state resides.

Data Type Refers to State Model.Domain, State Place.Domain and Parameter Signature .Domain

Model {I,R55,R57}
The name of the state model to which the state belongs.

Data Type Refers to State Model.Model and State Place.Model

Name {I,R57}
The name of the state.

Data Type c-identifier

Activity
The body of code that is executed when the state is entered.

Data Type string

IsFinal
Determines whether the class instance will be automatically deleted when the state activity is completed.

Data Type boolean

PSigID {R78}
The identifier of the parameter signature for the state.

Data Type Refers to Parameter Signature.SigID

Implementation

```
<<micca configuration>>=
class State {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Activity string
    attribute File string
    attribute Line int
    attribute IsFinal boolean

    reference R55 StateModel -link Domain -link Model
    reference R57 StatePlace -link Domain -link Model -link Name
}
```

R55

- **State** is an active element of *exactly one* **State Model**
- **State Model** acts via *one or more* **State**

A State is part of only a single State Model. Sharing states between state models is not allowed. Every State Model must have at least one state. The notion of an empty state model does not carry any behavior which it the purpose of a state model.

Implementation

```
<<micca configuration>>=
association R55 State 1..*--1 StateModel
```

Creation State

When a entity that has an associated State Model is created asynchonously by a creation event, it is placed in a pseudo-initial state. The creation event is then dispatched to the instance, causing a transition to a real state and the consequent execution of the destination state activity. The Creation State is the state the instance resides in temporarily until the creation event is dispatched. It differs from an ordinary state in that it can have no activity associated with it and can only have out-bound transitions (*i.e.* you cannot transition into a Creation State).

Domain {I,R56,R57}
The domain in which the creation state resides.

Data Type Refers to State Model.Domain and State Place.Domain

Model {I,R56,R57}
The name of the state model to which the creation state belongs.

Data Type Refers to State Model.Model and State Place.Model

Name {I,R57}
The name of the creation state. Since a State Model may have at most one Creation State (by R56) and since a Creation State does not result in creating any "C" language identifiers, we choose a distinctive name for the Creation State that is outside of the domain for ordination state names. It suffices to choose the literal string "@" to distinguish the Creation State.

Data Type the literal string "@"

Implementation

```
<<micca configuration>>=
class CreationState {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Name string -id 1

    reference R56 StateModel -link Domain -link Model
    reference R57 StatePlace -link Domain -link Model -link Name
}
```

R56

A State Model may have at most one Creation State and that Creation State is unique to a given State Model. Not all State Models have creation events defined for them, *i.e.* not all Classes support asynchronous creation.

Implementation

```
<<micca configuration>>=
association R56 CreationState 0..1--1 StateModel
```

State Place

A State Place is an abstract location from which an event transition may occur.

Domain {I,I2}
The domain in which the state place resides.

Data Type string

Model {I,I2}
The name of the state model to which the state place belongs.

Data Type string

Name {I}
The name of the state place.

Data Type Union of State.Name and Creation State.Name

Number {I2}
A sequential number given to the state place to aid in the encoding of states required during code generation.

Data Type sequential numeric

Implementation

```
<<micca configuration>>=
class StatePlace {
    attribute Domain string -id 1 -id 2
    attribute Model string -id 1 -id 2
    attribute Name string -id 1
    attribute Number int -id 2
}
```

R57

- **State Place** is a **State** or **Creation State**

State transitions can originate from either a State or a Creation State.

Implementation

```
<<micca configuration>>=
generalization R57 StatePlace State CreationState
```

Transition Rule

The execution rules for XUML state models allow for two types of transition specifications that do not actually cause a transition in a state machine. These rules simplify the specification of state models. One transition rule states that an event is to be ignored in a state and is named, “IG”. The other transition rule states that it is a logical impossiblity for the event to be received in the state and is named, “CH” (for “can’t happen”).

Name {I}
The name of the transition rule.

Data Type {IG, CH}

Implementation

```
<<micca configuration>>=
class TransitionRule {
    attribute Name string -id 1 -check {$Name in {IG CH}}
}
```

The number of Transition Rule instances is well known and constant so we will specify them here.

```
<<micca population>>=
class TransitionRule    {
    Name                } {
    IG
    CH
}
```

R53

- **Assigner State Model** is a **Single Assigner** or **Multiple Assigner**

When an association has an Assigner State Model, it may exist as a singleton or it may have multple instances of the assigner.

Implementation

```
<<micca configuration>>=
generalization R53 AssignerStateModel SingleAssigner MultipleAssigner
```

Single Assigner

A Single Assigner is a type of Assigner State Model that is used when a competitive association must be serialized through one execution path. This is the most frequent case.

Domain {I,R53}
The domain in which the single assigner resides.

Data Type Refers to Assigner State Model.Domain

Association {I,R53}
The name of the association relationship for which the single assigner operates.

Data Type Refers to Assigner State Model.Association

Implementation

```
<<micca configuration>>=
class SingleAssigner {
    attribute Domain string -id 1
    attribute Association string -id 1

    reference R53 AssignerStateModel -link Domain -link Association
}
```

Multiple Assigner

A Multiple Assigner is a type of Assigner State Model that is used when a competitive association is partitioned into distinct subsets wherein the assignment of the association is made.

Domain {I,R53,R54}
The domain in which the multiple assigner resides.

Data Type Refers to Assigner State Model.Domain and Class.Domain

Association {I,R53}
The name of the association relationship for which the multiple assigner operates.

Data Type Refers to Assigner State Model.Association

Class {R54}
The name of the class whose instances partition the multiple assigner.

Data Type Refers to Class.Name

Implementation

```
<<micca configuration>>=
class MultipleAssigner {
    attribute Domain string -id 1
    attribute Association string -id 1
    attribute Class string

    reference R53 AssignerStateModel -link Domain -link Association
    reference R54 Class -link Domain -link {Class Name}
    reference R104 ValueElement -link Domain -link {Association Name}
}
```

R54

- **Multiple Assigner** is partitioned by *exactly one* **Class**
- **Class** partions instance of *zero or more* **Multple Assigner**

A Multiple Assigner operates on association instances that are divided into sets associated with the instance of some Class. Not all Classes partition Multiple Assigners. The archetypical example of a Multiple Assigner is where a clerk services customers but the clerk only services customers who are shopping in the department where the clerk works. In this case the department serves to partition the assignment of clerks to customers to insure that assignments occur within the context of the department in which the clerk works.

Implementation

```
<<micca configuration>>=
association R54 MultipleAssigner 0..*--1 Class
```

Chapter 8

Event Subsystem

Introduction

The Event subsystem is concerned with classifying events that drive state machine transitions. The complexity here is that the execution rules for XUML allow for one form of polymorphism, namely polymorphic events.

A polymorphic event can arise only in the context of a generalization relationship. Events can be designated as polymorphic and when such an event is signaled to a superclass instance, it is mapped at run time to an event in the subclass instance that is currently related to superclass instance. The net effect is to allow the subclasses of a generalization all to respond to the same events and yet the events are signaled to the superclass instances of the generalization. This aleviates the burden on the signalling class to determine the subclass instance to which the superclass instance is currently related.

Strictly speaking event polymorphism is an optimization performed by the run time mechanisms since mapping an event to a subclass could be performed as part of the state activity. However, such code is repetitive and rather fragile in the wake of adding or deleting subclasses in a generalization. This is a case where the system can "know" the right thing to do and remove a significant burden from the model level processing.

In its most common usage, polymorphic events are quite straight forward. Events are designated as polymorphic by the superclass and they are used as transitioning events by the state models of the subclasses. In its most general usage however, we must account for both repeated generalization (*i.e.* where a subclass serves as the superclass for a futher generalization) and compound generalization (*i.e.* where a class is the superclass for multiple generalization relationships). The full set of rules and implications of polymorphic events can be rather daunting.

- Associating polymorphic events with a superclass does *not* imply that the superclass has no state behavior of its own. A superclass may have both a state model and polymorphic events since generating a polymorphic event to a superclass does not result in any behavior in the superclass.
- A given superclass may be the superclass of multiple generalizations. In this case, generating an event to an instance of such a superclass will cause an event to be generated to all generalizations for which the class is a superclass. In this way, signalling a single event may result in multiple events being dispatched.
- The state model for a subclass may respond to transitioning events that are not part of the polymorphic event set associated with the generalization. Such events may be signalled directly to instances of the subclass or they may be signalled by a subclass instance itself.
- A class that is a subclass may also be a superclass of another generalization, *i.e.* a subclass may be subject to repeated generalization. Such a mid-level class may designate additional polymorphic events associated with the generalization relationship for which it is the superclass. Also a mid-level class may delegate polymorphic events associated with its subclass role to any hierarchy for which it serves as a superclass.
- A mid-level class may have a state model that consumes an inherited polymorphic event. In that case, the event is not inherited down any other generalization for which the mid-level class is a superclass.

- All leaf classes, *i.e.* subclasses which are *not* the superclass of another generalization, must consume as transitioning events all polymorphic events delegated to them. It is sufficient to ignore an event or deem an event as can't happen, but polymorphic events are mapped ultimately to transitioning events when the bottom of the generalization hierarchy is reached.

Note also that generalizations are *not* subject to the so called "diamond" construct where a given class is a subclass of multiple generalizations which themselves have a common superclass ancestor. Blessedly, referential integrity simply does not allow such a beast to be constructed.

These rules give rise to the following class model for state model events.

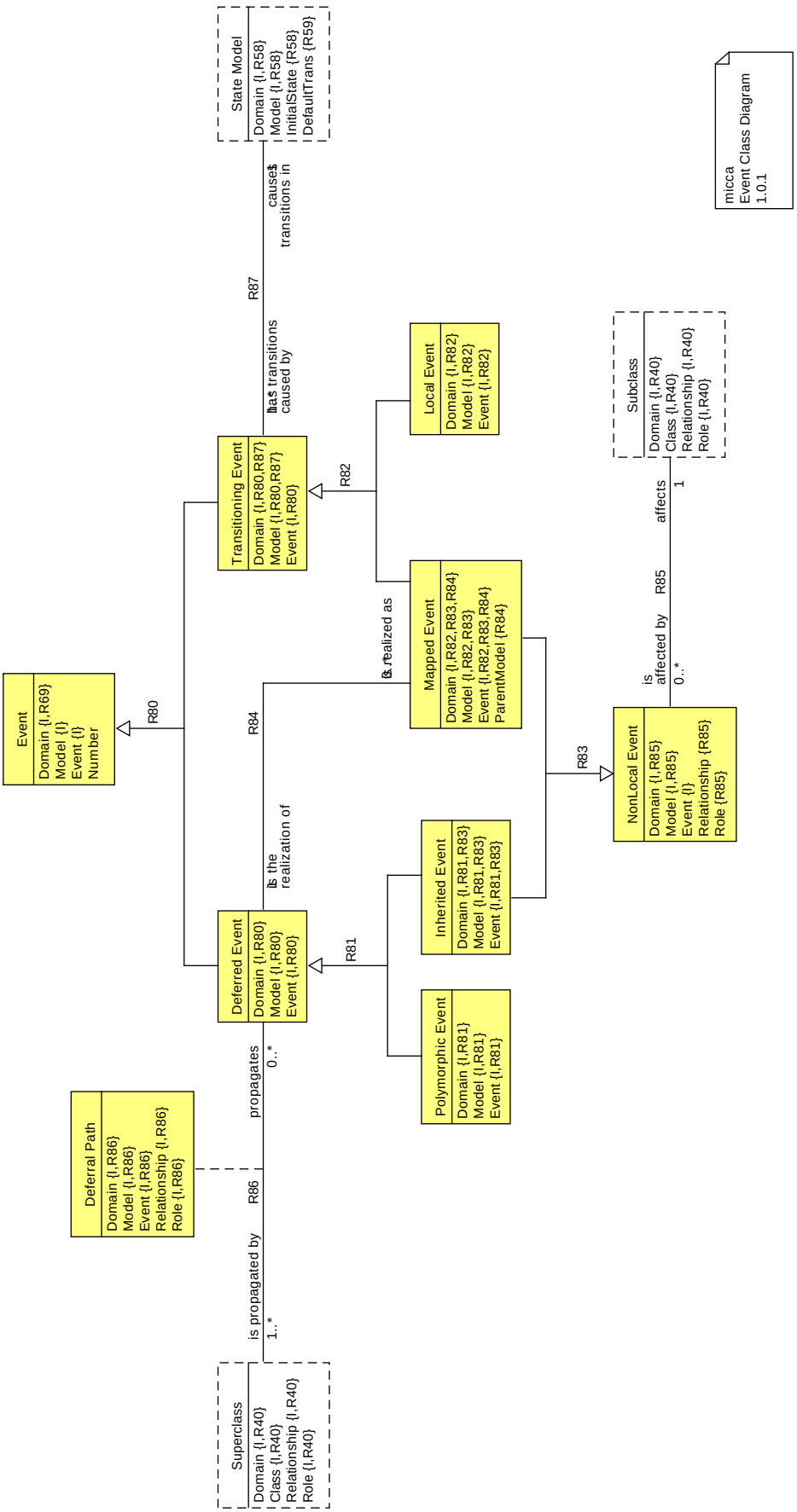


Figure 8.1: Event Subsystem Class Diagram

An **Event** can be of two types (**R80**), either a **Deferred Event** or a **Transitioning Event**. A **Transitioning Event** actually causes state transition in a **State Model** (**R87**). A **Deferred Event** arises from a **Superclass** across a **Deferral Path** (**R86**). Because of **R80**, transitioning events and polymorphic events in a superclass must have different names. We do consider this an advantage in avoiding confusion between polymorphic events and transitioning events in the superclass. A **Transitioning Event** is either a **Local Event** or a **Mapped Event** (**R82**). A **Local Event** is specified directly by a state model and a **Mapped Event** is a deferred event as it is consumed in a state model (**R84**). A **Deferred Event** is either a **Polymorphic Event** defined directly by a superclass or is delegated as an **Inherited Event** from another generalization for which the class was a subclass. Both an **Inherited Event** or a **Mapped Event** are forms of **NonLocal Event** (**R83**) and a **NonLocal Event** may only be defined to act upon a **Subclass** (**R85**).

Event

An Event is a stimulus to a state model that has to potential to cause a transition.

Domain {I,R69}
The domain in which the event resides.

Data Type Refers to Parameter Signature.Domain

Model {I}
The name of the state model to which the event belongs.

Data Type c-identifier

Event {I}
The name of the event.

Data Type string

PSigID {R69}
The identifier of the parameter signature for the event.

Data Type Refers to Argument Signature.SigID

Implementation

```
<<micca configuration>>=
class Event {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1
    attribute Number int

    reference R69 ParameterSignature -link Domain -link PSigID
}
```

R80

- **Event** is a **Deferred Event** or **Transitioning Event**

There are two types of events. Those that cause an immediate effect on a state model are **Transitioning Events**. Those that are mapped at run time to an event in a subclass are **Deferred Events**. The execution rules do not allow for any other types of events. Note that a creation event is just another form of **Transitioning Event**. The only difference is that the run time mechanism creates the class instance immediately before dispatching the creation event to it.

Implementation

```
<<micca configuration>>=
generalization R80 Event DeferredEvent TransitioningEvent
```

Deferred Event

A Deferred Event is that type of event which is specified in a Superclass but which is mapped to an event in a Subclass.

Domain {I,I2,R80}
The domain in which the deferred event resides.

Data Type Refers to Event.Domain

Model {I,I2,R80}
The name of the state model to which the deferred event belongs.

Data Type Refers to Event.Model

Event {I,R80}
The name of the deferred event.

Data Type Refers to Event.Event

Number
A non-negative number assigned to event and used by the code generator.

Data Type zero based sequential numeric

Implementation

```
<<micca configuration>>=
class DeferredEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1

    reference R80 Event -link Domain -link Model -link Event
}
```

R81

- **Deferred Event** is a **Polymorphic Event** or **Inherited Event**

There are two source of events that are deferred. Deferral happens in the sense that the events do not affect the behavior of the superclass to which they are sent. A superclass may declare an event as polymorphic or a subclass of a generalization may choose to not act upon the event if it is also the superclass of a different generalization. In the later case, the event not acted upon is inherited by the subclasses.

Implementation

```
<<micca configuration>>=
generalization R81 DeferredEvent PolymorphicEvent InheritedEvent
```

Deferral Path

A Deferral Path is traversal along a generalization relationship where an event is being deferred to a subclass of the generalization.

Domain {I,R80}
The domain in which the deferral path resides.

Data Type Refers to Superclass.Domain and Deferred Event.Domain

Model {I,R80}
The name of the state model to which the deferral path belongs.

Data Type Refers to Superclass.Class and Deferred Event.Model

Event {I,R80}
The name of the event that is being deferred along the deferral path.

Data Type Refers to Deferred Event.Event

Relationship {I,R86}
The generalization relationship along which the event is deferred.

Data Type Refers to Superclass.Relationship

Role {I,R86}
The role of the superclass which is deferring an event.

Data Type Refers to Superclass.Role

Implementation

```
<<micca configuration>>=
class DeferralPath {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1
    attribute Relationship string -id 1
    attribute Role string -id 1

    reference R86 DeferredEvent -link Domain -link Model -link Event
    reference R86 Superclass -link Domain -link {Model Class}\
        -link Relationship -link Role
}
```

R86

- **Deferral Path** is an instance of **Superclass** propagates *zero or more* **Deferred Event**
- **Deferral Path** is an instance of **Deferred Event** is propagated by *one or more* **Superclass**

Event polymorphism allows an event in a Superclass to be deferred to the subclasses. Not all Superclasses define polymorphic events. A given polymorphic event may be propagated along mulitple generalization relationships in the case of a compound generalization, *i.e.* a superclass that is the a superclass for multiple generalizations.

Implementation

```
<<micca configuration>>=
association R86 Superclass 1..*--0..* DeferredEvent -associator DeferralPath
```

Transitioning Event

A Transitioning Event is that type of event that has the potential to cause a state transition when dispatched to a state machine.

Domain {I,R80,R87}
The domain in which the transitioning event resides.

Data Type Refers to Event.Domain and State Model.Domain

Model {I,R80,R87}
The name of the state model to which the transitioning event belongs.

Data Type Refers to Event.Model and State Model.Model

Event {I,R80}
The name of the transitioning event.

Data Type Refers to Event.Event

Number
A non-negative number assigned to event and used by the code generator.

Data Type zero based sequential numeric

Implementation

```
<<micca configuration>>=
class TransitioningEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1

    reference R80 Event -link Domain -link Model -link Event
    reference R87 StateModel -link Domain -link Model
}
```

R82

- **Transitioning Event** is a **Mapped Event** or **Local Event**

Events that can cause state transitions arise from two circumstances. First, they may be locally defined. This is usually the preponderance of events. Second, the event could have been deferred from a superclass and when it dispatched to a subclass is mapped into an event defined for the subclass.

Implementation

```
<<micca configuration>>=
generalization R82 TransitioningEvent MappedEvent LocalEvent
```

R87

- **Transitioning Event** causes transitions in *exactly one* **State Model**
- **State Model** has transitions caused by *one or more* **Transitioning Event**

Events that cause transition in a state model are defined to be specific to that state model and all state models must respond to at least one event by the definition of a state machine. Of course, state models can, and usually do, respond to multiple events.

Implementation

```
<<micca configuration>>=
association R87 TransitioningEvent 1..*--1 StateModel
```

Polymorphic Event

A polymorphic event is an event defined for the superclass of a generalization that is to be mapped at run time to an event of one of the subclasses of the generalization. The run time mapping determines the subclass to which the superclass is currently related and maps the event into one by the same name in the subclass.

Domain {I,R81}
The domain in which the polymorphic event resides.

Data Type Refers to Deferred Event.Domain

Model {I,R81}
The name of the state model to which the polymorphic event belongs.

Data Type Refers to Deferred Event.Model

Event {I,R81}
The name of the polymorphic event.

Data Type Refers to Deferred Event.Event

Implementation

```
<<micca configuration>>=
class PolymorphicEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1

    reference R81 DeferredEvent -link Domain -link Model -link Event
}
```

Inherited Event

A polymorphic event defined for superclass becomes an inherited event in the subclasses of the generalization if a subclass does not act upon the polymorphic event. This can occur when the subclass is part of a repeated generalization where it serves the role of subclass in a generalization that defines the polymorphic event and serves the role of superclass in another generalization. The event can be thought of as being inherited down the hierarchy formed by the repeated generalizations.

Domain {I,R81,R83}
The domain in which the inherited event resides.

Data Type Refers to Deferred Event.Domain and NonLocal Event.Domain

Model {I,R81,R83}
The name of the state model to which the inherited event belongs.

Data Type Refers to Deferred Event.Model and NonLocal Event.Model

Event {I,R81,R83}
The name of the inherited event.

Data Type Refers to Deferred Event.Event and NonLocal Event.Event

Implementation

```
<<micca configuration>>=
class InheritedEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1

    reference R81 DeferredEvent -link Domain -link Model -link Event
    reference R83 NonLocalEvent -link Domain -link Model -link Event
}
```

Mapped Event

When a deferred event is consumed by a subclass it becomes a mapped event for the subclass state model. This allows the subclass to define other events to which it may respond.

Domain {I,R82,R83,R84}
The domain in which the polymorphic event resides.

Data Type Refers to Transitioning Event.Domain, NonLocal Event.Domain and Deferred Event.Domain

Model {I,R82,R83}
The name of the state model to which the polymorphic event belongs.

Data Type Refers to Transitioning Event.Model and NonLocal Event.Model

Event {I,R82,R83,R84}
The name of the polymorphic event.

Data Type Refers to Transitioning Event.Event, NonLocal Event.Event and Deferred Event.Event

ParentModel {R84}
The name of the state model from which the mapped event was deferred.

Data Type
Refers to Deferred Event.Model

Implementation

```
<<micca configuration>>=
class MappedEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1
    attribute ParentModel string

    reference R82 TransitioningEvent -link Domain -link Model -link Event
    reference R83 NonLocalEvent -link Domain -link Model -link Event
    reference R84 DeferredEvent -link Domain -link {ParentModel Model}\
        -link Event
}
```

R84

- **Mapped Event** is the realization of *exactly one* **Deferred Event**
- **Deferred Event** is realized as *at most one* **Mapped Event**

When a deferred event is consumed, it becomes a mapped event in the subclass where it is acted upon. A mapped event always starts out as one that is deferred from a superclass. Not all deferred events are necessarily mapped. It is possible to defer an event through muliple generalizations before it is finally consumed and becomes a mapped event.

Implementation

```
<<micca configuration>>=
association R84 MappedEvent 0..*--1 DeferredEvent
```

Local Event

A local event is an event defined by a class and directly acted upon by the state model of the class. This forms the vast majority of defined events and all the events for classes that are not subclasses of a generalization.

Domain {I,R82}
The domain in which the polymorphic event resides.

Data Type Refers to Transitioning Event.Domain

Model {I,R82}
The name of the state model to which the polymorphic event belongs.

Data Type Refers to Transitioning Event.Model

Event {I,R82}
The name of the polymorphic event.

Data Type Refers to Transitioning Event.Event

Implementation

```
<<micca configuration>>=
class LocalEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1

    reference R82 TransitioningEvent -link Domain -link Model -link Event
}
```

R83

- **NonLocal Event** is an **Inherited Event** or **Mapped Event**

Together, inherited and mapped events form a set of events that do not originate from the immediate local state model definition.

Implementation

```
<<micca configuration>>=
generalization R83 NonLocalEvent InheritedEvent MappedEvent
```

NonLocal Event

A non-local event is that type of event that arises from polymorphic events that are mapped through a generalization hierarchy.

Domain {I,R85}
The domain in which the polymorphic event resides.

Data Type Refers to Subclass.Domain

Model {I,R85}
The name of the state model to which the polymorphic event belongs.

Data Type Refers to Subclass.Class

Event {I}
The name of the polymorphic event.

Data Type string

Relationship {R85}
The generalization relationship along which the non-local event is applied.

Data Type Refers to Subclass.Relationship

Role {R85}
The role of the subclass to which the non-local event is applied.

Data Type Refers to Subclass.Role

Implementation

```
<<micca configuration>>=
class NonLocalEvent {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1
    attribute Relationship string
    attribute Role string

    reference R85 Subclass -link Domain -link {Model Class} -link Relationship\
        -link Role
}
```

R85

- **NonLocal Event** affects *exactly one* **Subclass**
- **Subclass** is affected by *zero or more* **NonLocal Event**

Regardless of the source of non-local events, they can only cause transition in the state model of a subclass. Classes that are not subclasses cannot respond to non-local events. Some subclasses will not have state models or polymorphic events defined in their superclass.

Implementation

```
<<micca configuration>>=
association R85 NonLocalEvent 0..*--1 Subclass
```

Chapter 9

Transition Subsystem

Introduction

The transition subsystem is concerned with the dispatch of state model events. The type of dispatch supported by XUML is the same as for a Moore type state machine, *i.e.* for a given state an event causes a transition into a new state (not necessarily distinct from the current state) and executes any activity associated with the state. In UML parlance, these are known as entry actions. No other form of action dispatch is supported as it is not needed.

One other complexity of state activity dispatch is that event arguments. Each state may define a signature of parameters that the state activity expects. It is a corollary of Moore state machines that any event that causes a transition *into* a state must carry arguments that match the formal parameter signature of the state.

The diagram below shows the transition subsystem classes.

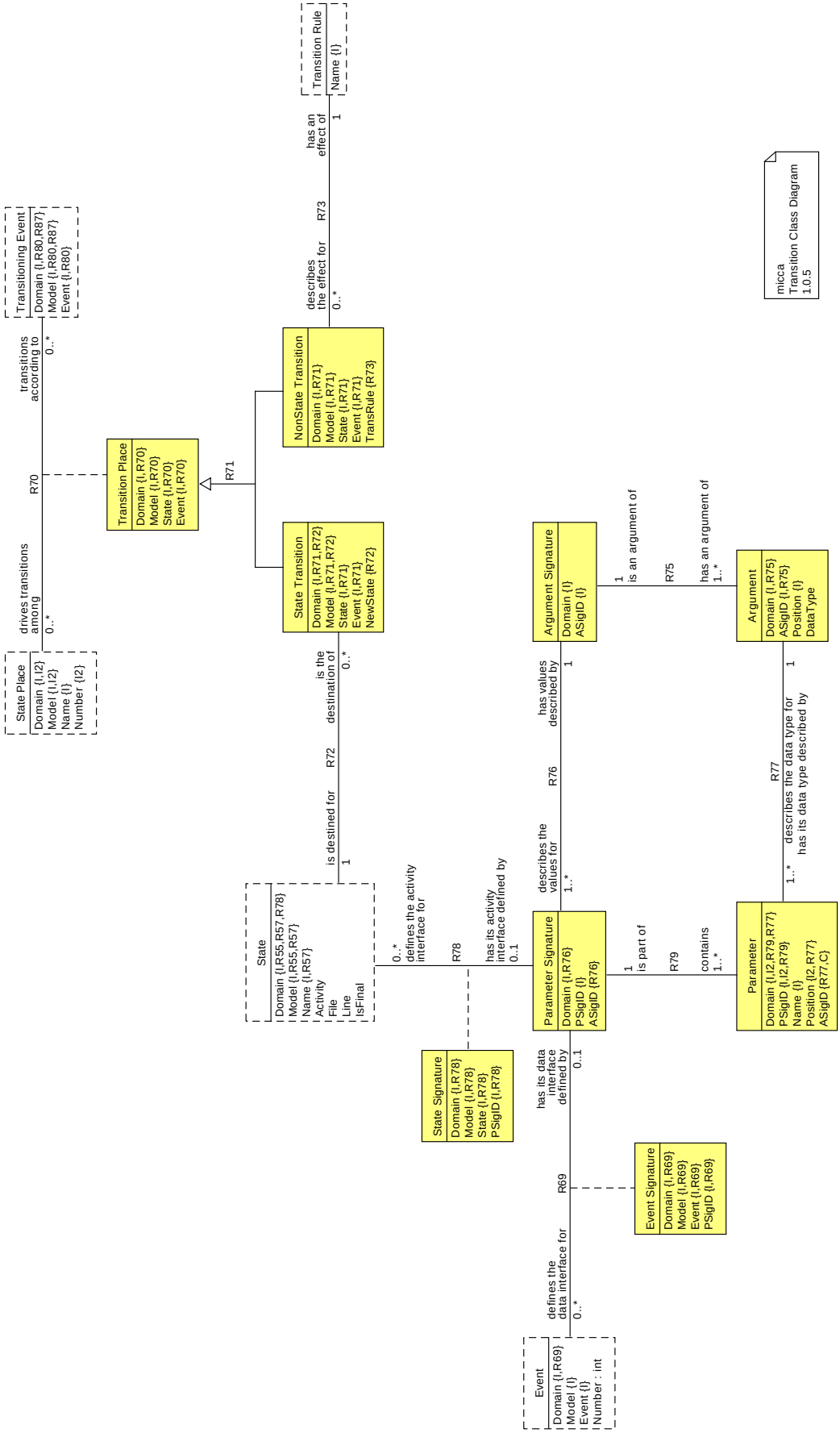


Figure 9.1: Transition Subsystem Class Diagram

All the **State Places** and **Transitioning Events** combine to form a transition matrix (**R70**) the cells of which are represented by **Transition Places**. There are two types of transitions that can occur in an state model (**R71**). A **State Transition** causes the state machine to enter a new state of the state model (**R72**). A **NonState Transition** does not cause a new state to be entered but has the side effect of either ignoring the event or causing a system error (**R73**).

When a state transition happens, the arguments carried by the event are transferred to the state activity according to an **Argument Signature** instance. An **Argument Signature** correponds to a set of positional parameters (**R75**) which have a fixed position in the argument list and a defined data type. Each **State** and **Event** may have an **Parameter Signature** (**R78** and **R69**) A **Parameter Signature** gives names to the positional parameters. The **Parameter Signature** for events causing a transition into a state must match the underlying **Argument Signature** of the **State Transition** (**R76**). A **State** or **Event** may name its parameters in any way it wishes (**R79**) but the names do not affect the underlying position or data type of a **Argument Signature** (**R77**). The only thing important about the arguments of a state activity are the position and data type, as is required by "C" when invoking a function. The parameter naming from the **State** and **Event** points of view need not be the same.

Transition Place

The Cartesian product of the state of a state model and the events of the model form a conceptual transition matrix. A transition place models the individual cells of the conceptual transition matrix.

Domain {I,R70}
The domain in which the transition place resides.

Data Type Refers to State Place.Domain and Transitioning Event.Domain

Model {I,R70}
The name of the state model to which the transition place belongs.

Data Type Refers to State Place.Model and Transitioning Event.Model

State {I,R70}
The name of the state to which the transition place refers.

Data Type Refers to State Place.Name

Event {I,R70}
The name of the event which will cause a transition out of the transition place.

Data Type Refers to Transitioning Event.Event

Implementation

```
<<micca configuration>>=
class TransitionPlace {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute State string -id 1
    attribute Event string -id 1

    reference R70 StatePlace -link Domain -link Model -link {State Name}
    reference R70 TransitioningEvent -link Domain -link Model -link Event
}
```

R70

- **Transition Place** is an instance of **State Place** transitions according to *zero or more* **Transitioning Event**
- **Transition Place** is an instance of **Transitioning Event** drives transition among *zero or more* **State Place**

A state model operates by having a function that maps states and events onto new states. A Transition Place is one of the enumerations of that mapping function.

Implementation

```
<<micca configuration>>=
association R70 StatePlace 0..*--0..* TransitioningEvent -associator TransitionPlace
```

R71

- **Transition Place** is a **State Transition** or **NonState Transition**

As a convenience to the analyst, events can be ignored or cause a system error. These rules simplify the state model and result in there being two different types of transitions.

Implementation

```
<<micca configuration>>=
generalization R71 TransitionPlace StateTransition NonStateTransition
```

State Transition

A State Transition is that type of Transition Place where the dispatch of an event causes the state machine to enter a new state.

Domain {I,R71,R72}
The domain in which the state transition resides.

Data Type Refers to Transition Place.Domain, State.Domain and Argument Signature.Domain

Model {I,R71,R72}
The name of the state model to which the state transition belongs.

Data Type Refers to Transition Place.Model and State.Model

State {I,R71}
The name of the state that is the starting state for the state transition.

Data Type Refers to Transition Place.Name

Event {I,R70}
The name of the event which will cause the state transition.

Data Type Refers to Transition Place.Event

NewState {R72}
The name of the state to which state transition arrives.

Data Type Refers to State.Name

Implementation

```
<<micca configuration>>=
class StateTransition {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute State string -id 1
    attribute Event string -id 1
    attribute NewState string

    reference R71 TransitionPlace -link Domain -link Model -link State\
        -link Event
    reference R72 State -link Domain -link Model -link {NewState Name}
}
```

R72

- **State Transition** is destined for *exactly one* **State**
- **State** is the destination of *at most one* **State Transition**

When an event is dispatched and it causes a transition, a new state becomes the current state. This must happen for every state transition. Conversely, not all states are the destination of a transition. Some state may only have outbound transitions.

Implementation

```
<<micca configuration>>=
association R72 StateTransition 0..*--1 State
```

NonState Transition

A NonState Transition is that type of Transition Place where the dispatch of an event does not cause a transition and the system takes a prescribed action.

Domain {I,R71}
The domain in which the non-state transition resides.

Data Type Refers to Transition Place.Domain

Model {I,R71}
The name of the state model to which the non-state transition belongs.

Data Type Refers to Transition Place.Model

State {I,R71}
The name of the state that is the starting state for the non-state transition.

Data Type Refers to Transition Place.Name

Event {I,R71}
The name of the event which is received by the non-state transition.

Data Type Refers to Transition Place.Event

TransRule {R73}
The system rule that is performed when the event is received in the state.

Data Type Refers to Transition Rule.Name

Implementation

```
<<micca configuration>>=
class NonStateTransition {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute State string -id 1
    attribute Event string -id 1
    attribute TransRule string

    reference R71 TransitionPlace -link Domain -link Model -link State\
        -link Event
    reference R73 TransitionRule -link {TransRule Name}
}
```

R73

- **NonState Transition** has the effect of _exactly one _*Transition Rule*
- **Transition Rule** describes the effect for _zero or more _ **NonState Transition**

All transitions that do not result in a new state being entered must take an action that is dictated by transitions rules that govern non-state transitions. Any given transition rule may not be applied to a non-state transition.

Implementation

```
<<micca configuration>>=
association R73 NonStateTransition 0..*--1 TransitionRule
```

Argument Signature

A State can define formal parameters for its activity. The set of those parameters form a signature for the state. Any event that causes a transition into a given state must carry argument values that match the formal parameters of the state, *i.e.* the event can be considered to have the same signature as the state into which it causes a transition.

Domain {I}
The domain in which the argument signature resides.

Data Type string

ASigID {I}
The identifier of the argument signature. Argument signatures are identified uniquely within a given domain. This choice of identification is somewhat arbitrary and this particular choice is an attempt to keep the signature information localized to a particular domain.

Data Type arbitrary identfier

Implementation

```
<<micca configuration>>=
class ArgumentSignature {
    attribute Domain string -id 1
    attribute ASigID string -id 1
}
```

Parameter Signature

An state activity may require a set of parameters when it is invoked. Those parameters may have arbitrary names, but for the implementation to operate correctly, the order of a parameter and its data type, not its name, govern whether or not a set of event arguments matches a set of state activity parameters. A Parameter Signature is a way for states and events to name parameters as they wish, but still have a way to determine if the arguments match the formal parameters.

Domain {I,R76}
The domain in which the argument signature resides.

Data Type Refers to Argument Signature.Domain

PSigID {I}
The identifier of the parameter signature. Parameter signatures are identified uniquely within a given domain. This choice of identification is somewhat arbitrary and this particular choice is an attempt to keep the signature information localized to a particular domain.

Data Type arbitrary identfier

ASigID {I,R76}
The identifier of the argument signature for the parameter signature.

Data Type	Refers to Argument Signature.ASigID

Implementation

```
<<micca configuration>>=
class ParameterSignature {
    attribute Domain string -id 1
    attribute PSigID string -id 1
    attribute ASigID string

    reference R76 ArgumentSignature -link Domain -link ASigID
}
```

R76

- **Parameter Signature** has values described by *exactly one* **Argument Signature**
- **Argument Signature** describes the values for *one or more* **Parameter Signature**

Every Parameter Signature must have an Argument Signature to describe the position and data types of the parameters. Argument Signatures do not exist unless they describe some set of parameters.

Implementation

```
<<micca configuration>>=
association R76 ParameterSignature 1..*--1 ArgumentSignature
```

R76 is *not* an independent relationship. It is constrained by the requirement that the Argument Signature associated with both the new state of a transition and the event that causes a transition must be the same. We insist that the following two relation traversals yields the same instance of ArgumentSignature

- StateTransition → R72 → R78 → R76
- StateTransition → R71 → R70 → R80 → R69 → R76

In other words, both the Argument Signatures of the New State of a State Transition and that of the Event that causes the transition into the state must be the same. To enforce that constraint, we write a TclRAL procedural constraint on the State Transition class.

Matching argument signatures for a state transition is one of the more complicated aspects of the platform model. We define the procedural constraint below that will insure that the two constaints are enforced. We must also print any error messages as part of the procedural constraint, because after the constraint fails, the configuration classes (relvars) will have been rolled back to their previous values.

Procedural constraints are not directly supported by `rosea`, so we have to fall back to TclRAL commands to implement it. The strategy is to compute the argument signatures for the event causing the transition and the new state of the transition and then compare that result with the argument signature of the state transition. If there is a mismatch, then we determine which transitions are causing the problem.

There are two complications in the enforcing this constraint. First, we are interested in the Argument Signatures of the state and event and *not* the Parameter Signatures. In other words, we only care about the positions and data types of the parameters and *not* the names given to them from the points of view of the event and state. Second, we have to deal with the fact that **R78** and **R69** are conditional, *i.e.* a State or Event may not have any parameters at all. This means that we cannot simply join across those relationships because we will miss the case where a State or Event does not have a Parameter Signature (since those tuples would not show up in the join). Here we will have to rely on the `rvajoin` to insure all the cases are considered.

The strategy for the query is to compute a relation value that has the Argument Signature ID for the State Transition, Event and State. Then we can find tuples where there is a mismatch.

```
<<micca constraints>>=
relvar procedural R76C StateTransition {
    set stateSig [ralutil pipe {
        ral relation semijoin $StateTransition\
            $State -using {Domain Domain Model Model NewState Name} |
        ral relation project ~ Domain Model Name |
        ral relation rename ~ Name State |
        ral relation join ~ $StateSignature $ParameterSignature $ArgumentSignature
    }]
    #puts [ral relformat $stateSig stateSig]

    set eventSig [ralutil pipe {
        ral relation semijoin $StateTransition\
            $TransitionPlace $TransitioningEvent $Event |
        ral relation project ~ Domain Model Event |
        ral relation join ~ $EventSignature $ParameterSignature $ArgumentSignature
    }]
    #puts [ral relformat $eventSig eventSig]

    set missing [pipe {
        ::ralutil::rvajoin $eventSig $stateSig Match Domain Model ASigID |
        ral relation restrict ~ mat_tup\
                {[ral relation isempty [ral tuple extract $mat_tup Match]]} |
        ral relation eliminate ~ Match |
        ::ralutil::rvajoin ~ $Argument Arguments |
        ral relation project ~ Domain Model Event Arguments
    }]
    #puts [ral relformat $missing missing]

    ral relation foreach miss $missing {
        ral relation assign $miss\
            {Domain domain} {Model model} {Event event} {Arguments arguments}
        puts stderr "for domain, $domain, class, $model, the signature for event,\
            $event\([join [ral relation list $arguments DataType -ascending Position] {,  ↩
                }])\
            does not match the signature of any state"
    }

    return [ral relation isempty $missing]
}
```

R78

- **State Signature** is an instance of **State** has its activity interface defined by *at most one* **Parameter Signature**
- **State Signature** is an instance of **Parameter Signature** defines the activity interface for *zero or more* **State**

A State may not have a Parameter Signature if it does not require any parameters for its state activity. A Parameter Signature may not describe the parameters of any state, *i.e.* it may describe the parameters of an Event.

Implementation

```
<<micca configuration>>=
class StateSignature {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute State string -id 1
    attribute PSigID string -id 1
```

```
    reference R78 State -link Domain -link Model -link {State Name}
    reference R78 ParameterSignature -link Domain -link PSigID
}

association R78 State 0..*--0..1 ParameterSignature -associator StateSignature
```

R69

- **Event Signature** is an instance of **Event** has its data interface defined by *at most one* **Parameter Signature**
- **Event Signature** is an instance of **Parameter Signature** defines the data interface for *zero or more* **Event**

An Event may not have a Parameter Signature if it does not carry any event arguments. A Parameter Signature may not describe the parameters of any event, *i.e.* it may describe the parameters of an State.

Note that R69 and R78 technically allow a Parameter Signature to exist that is not associated with any State or Event. We will not create such a situation in the `micca` configuration DSL code and no real harm would come if we did.

Implementation

```
<<micca configuration>>=
class EventSignature {
    attribute Domain string -id 1
    attribute Model string -id 1
    attribute Event string -id 1
    attribute PSigID string -id 1

    reference R69 Event -link Domain -link Model -link Event
    reference R69 ParameterSignature -link Domain -link PSigID
}

association R69 Event 0..*--0..1 ParameterSignature -associator EventSignature
```

Argument

An Argument is a specific data value of a given type in a fixed location in a set of arguments.

Domain {I,R75}
The domain in which the argument resides.

Data Type Refers to Argument Signature.Domain

ASigID {I,R75}
The argument signature ID for the argument.

Data Type Refers to Argument Signature.SigID

Position {I}
An ordinal number given to the argument to indicate its order in a set of arguments. In “C” arguments to functions are passed by position and we need to be able to create argument lists properly.

Data Type zero based sequential numeric

DataType
The data type of the activity argument.

Data Type c-typename

Implementation

```
<<micca configuration>>=
class Argument {
    attribute Domain string -id 1
    attribute ASigID string -id 1
    attribute Position int -id 1
    attribute DataType string\
        -check {[::micca::@Config@::Helpers::typeCheck verifyTypeName $DataType]}

    reference R75 ArgumentSignature -link Domain -link ASigID
}
```

R75

- **Argument** is an argument of *exactly one* **Argument Signature**
- **Argument Signature** has an argument of *one or more* **Argument**

When a State is entered on a transition of a dispatched event, its activity is executed. The event causing the transition may carry argument values for the state activity. An Argument is a description of the arguments that the state activity accepts. An Argument is always part of a Argument Signature and does not exist outside of the context of a single signature.

Implementation

```
<<micca configuration>>=
association R75 Argument 1..*--1 ArgumentSignature
```

Parameter

A Parameter is a formal parameter to the activity of a state or the data of an Event. When the state activity is executed, the transitioning event carrys the actual argument values for each parameter.

Domain {I,I2,R79,R77}
The domain in which the parameter resides.

Data Type Refers to Parameter Signature.Domain and Argument.Domain

PSigID {I,I2,R79}
The parameter signature ID for the parameter.

Data Type Refers to Parameter Signature.PSigID

Name {I}
The name of the parameter. This name may be used as a variable name in the code for the state activity.

Data Type c-identifier

Position {I2,R77}
A ordinal number given to the argument to indicate its order in a set of arguments for an activity.

Data Type Refers to Argument.Position

ASigID {R77,C}
The argument signature ID for the parameter. Note this attribute's value is further constrained to match the value of the ASigID of its Parameter Signature related across R79.

Data Type Refers Argument.ASigID.

Implementation

```
<<micca configuration>>=
class Parameter {
    attribute Domain string -id 1 -id 2
    attribute PSigID string -id 1 -id 2
    attribute Name string -id 1
    attribute Position int -id 2
    attribute ASigID string

    reference R77 Argument -link Domain -link ASigID -link Position
    reference R79 ParameterSignature -link Domain -link PSigID
}
```

R77

- **Parameter** has its data type described by *exactly one* **Argument**
- **Argument** describes the data type for *one or more* **Parameter**

All parameters simply give a name to a particular position and data type in a parameter set. Arguments always describe a parameter's position and data type.

Implementation

```
<<micca configuration>>=
association R77 Parameter 1..*--1 Argument
```

R79

- **Parameter** is part of *exactly one* **Parameter Signature**
- **Parameter Signature** contains *one or more* **Parameter**

Parameters always belong to some Parameter Signature and there are no empty Parameter Signatures.

Implementation

```
<<micca configuration>>=
association R79 Parameter 1..*--1 ParameterSignature
```

Chapter 10

Population Subsystem

Introduction

The Population subsystem is concerned with defining the initial data values that class instances will have when the domain starts running and to specifying the storage characteristics for classes. In a `micca` generated system, there is no dynamic memory allocation and all class storage is specified at compile time. Many domains have substantial initial instance populations and `micca` provides a way to specify those populations. When the code is generated, inital instances become initializers in the array where a class is stored. One goal of managing the initial instance population is to avoid having to write executable code that creates the instances. Such code is run only once, and for platforms targeted by `micca`, code space for code that is only run once needs to be minimized. We can take advantage of initialized variables in "C" to have data values placed in the proper variables when the system is started.

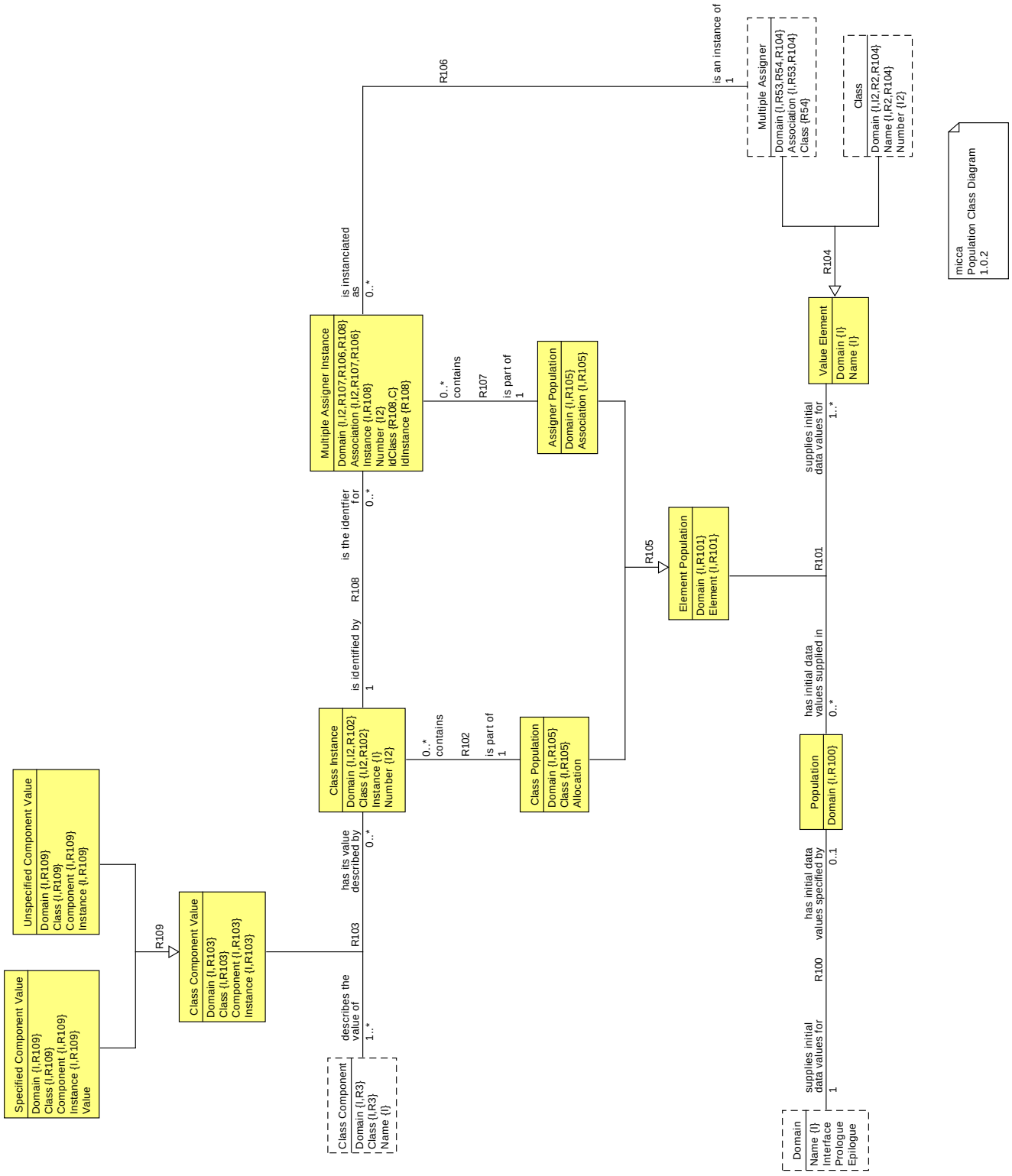


Figure 10.1: Population Subsystem Class Diagram

A **Population** is a collection of initial data values for a **Domain** (**R100**). Only certain **Value Element** components in a **Domain** can be supplied with initial values. A **Value Element** is one of two types (**R104**), either a **Class** or a **Multiple Assigner**. A

Population must consist of an **Element Population** for each **Value Element** (**R101**). Reflecting the two different types of **Value Elements**, there are two different types of **Element Populations** (**R105**). A **Class Population** contains **Class Instances** (**R102**) and an **Assigner Population** contains **Multiple Assigner Instances** (**R107**). A **Class Instance** defines values for **Class Components** (**R103**) and this will include values for both attributes and references. The **Class Component Value** are of two types (**R109**). The **Specified Component Value** are those either set by a user or determined for a generated **Class Component**. An **Unspecified Component Value** corresponds to a zero initialized attribute for which no other value was given. In this case the initialization to a zero value is left for the "C" compiler to perform. A **Multiple Assigner Instance** is always an instance for a **Multiple Assigner** (**R106**). A **Multiple Assigner Instance** is also always associated with the **Class Instance** of the identifying class for the multiple assigner (**R108**).

Population

A Population is a set of instance values for a Domain.

Domain {I,R100}
The domain for which the population defines values.

Data Type Refers to Domain.Name

Implementation

```
<<micca configuration>>=
class Population {
    attribute Domain string -id 1

    reference R100 Domain -link {Domain Name}
}
```

R100

- **Population** supplies initial data values for *exactly one* **Domain**
- **Domain** has initial data values specified by *at most one* **Population**

A Population always applies to a single domain. It is possible to define a Domain without giving a population. This allows domain definitions to be reused with many different populations for testing and deployment purposes. However, code generation insists that there be a population for the domain.

Implementation

```
<<micca configuration>>=
association R100 Population 0..1--1 Domain
```

Value Element

A Value Element is an entity in a domain that can be supplied population values.

Domain {I}
The domain to which the value element belongs.

Data Type string

Name {I}
The name of the value element.

Data Type c-identifier

Implementation

```
<<micca configuration>>=
class ValueElement {
    attribute Domain string -id 1
    attribute Name string -id 1
}
```

R104

- **Value Element** is a **Class** or **Multiple Assigner**

There are two types of entities in a domain that can be supplied inital values, namely a Class or a Multiple Assigner. All other entities of a domain do not have any value semantics.

Implementation

```
<<micca configuration>>=
generalization R104 ValueElement Class MultipleAssigner
```

Element Population

An Element Population defines the space allocated for storage of the Value Elements in a Population.

Domain {I,R101}
The domain to which the element population belongs

Data Type Refers to Population.Domain and Value Element.Domain

Element {I,R101}
The name of the value element for which the element population defines an allocation.

Data Type Refers to Value Element.Name

Implementation

```
<<micca configuration>>=
class ElementPopulation {
    attribute Domain string -id 1
    attribute Element string -id 1

    reference R101 Population -link Domain
    reference R101 ValueElement -link Domain -link {Element Name}
}
```

R101

- **Element Population** is an instance of **Population** supplies initial data values for *one or more* **Value Element**
- **Element Population** is an instance of **Value Element** has initial data value supplied in **Population**

A Population supples the values for Value Elements. All Value Elements should be supplied population values. A Value Element may exist independent of a Population so that a domain may be defined separately from any population.

Implementation

```
<<micca configuration>>=
association R101 Population 0..*--1..* ValueElement -associator ElementPopulation
```

R105

- **Element Popluation** is a **Class Population** or **Multiple Assigner Population**

There are two types of element populations that correspond to the two types of Value Elements.

Implementation

```
<<micca configuration>>=
generalization R105 ElementPopulation ClassPopulation AssignerPopulation
```

Class Population

A Class Population is a set of class instances.

Domain {I,R105}
The domain to which the class population belongs.

Data Type Refers to Element Population.Domain

Class {I,R102}
The name of the class to which the class population applies.

Data Type Refers to Element Population.Element

Allocation
The number instances of the class which is allocated beyond storage required for the instances specified by the population. The space for instance storage will consist of that specified in the initial instance population plus the value of the Allocation attribute. For classes where all instances are created at runtime, this attribute is set to the maximum number of the instances that can exist at the same time. For classes where all the instances are created at population time and no instances are created at run time, this attribute can be set to zero. For classes where some instances are specified at population time and others are created at run time, this attribute sets the initial space available for dynamic instance creation.

Data Type Non-negative numeric

Implementation

```
<<micca configuration>>=
class ClassPopulation {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Allocation int -check {$Allocation >= 0}

    reference R105 ElementPopulation -link Domain -link {Class Element}
}
```

Assigner Population

An Assigner Population is a set of multiple assigner instances.

Domain {I,R105}
The domain to which the assigner population belongs.

Data Type Refers to Element Population.Domain

Association {I,R102}
The name of the association to which the assigner population applies.

Data Type Refers to Element Population.Element

Implementation

```
<<micca configuration>>=
class AssignerPopulation {
    attribute Domain string -id 1
    attribute Association string -id 1

    reference R105 ElementPopulation -link Domain -link {Association Element}
}
```

Class Instance

A Class Instance represents the set of values that are used to populate the components of a particular instance of a class.

Domain {I,R102}
The domain to which the class instance belongs

Data Type Refers to Class Population.Domain

Class {I,R102}
The name of the class to which the class instance pertains.

Data Type Refers to Class Population.Class

Instance {I}
The name of the class instance. Instances may be given arbitrary names and those names are used to refer to the instance when populating relationship references to the instance. We insist that instance names be valid "C" identifiers since they will show up in the generated header file as preprocessor defines.

Data Type c-identifier

Number {I2}
A number associated with the class instance. The number will be used as an array index into the storage array associated with the class.

Data Type zero based sequential number

Implementation

```
<<micca configuration>>=
class ClassInstance {
    attribute Domain string -id 1 -id 2
    attribute Class string -id 1 -id 2
    attribute Instance string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Instance]}
    attribute Number int -id 2

    reference R102 ClassPopulation -link Domain -link Class
}
```

R102

- **Class Instance** is part of *exactly one* **Class Population**
- **Class Population** contains *one or more* **Class Instance**

Class instances are specific to a given population. It is possible not to specify any initial instances for those classes where all instances are created at run time.

Implementation

```
<<micca configuration>>=
association R102 ClassInstance 0..*--1 ClassPopulation
```

Class Component Value

A Class Component Value represent the value of an attribute or generated component of a Class Instance in the initial instance population.

Domain {I,R103}
The domain to which the class component value belongs.

Data Type Refers to Class Instance.Domain and Class Component.Domain

Class {I,R103}
The name of the class to which the class component value pertains.

Data Type Refers to Class Instance.Class and Class Component.Class

Component {I,R103}
The name of the class component to which the class component value applies.

Data Type Refers to Class Component.Name

Instance {I,R103}
The name of the class instance to which the class component value pertains.

Data Type Refers to Class Instance.Instance

Implementation

```
<<micca configuration>>=
class ClassComponentValue {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Component string -id 1
    attribute Instance string -id 1

    reference R103 ClassInstance -link Domain -link Class -link Instance
    reference R103 ClassComponent -link Domain -link Class -link {Component Name}
}
```

R103

- **Class Component Value** is an instance of **Class Instance** describes the value of *one or more* **Class Component**
- **Class Component Value** is an instance of **Class Component** has its value describe by *zero or more* **Class Instance**

A Class Instance gives the value of the Class Components. All Class Components need to be given a value. Class Components exist independent of any population value so that they may be defined separately from a population.

Implementation

```
<<micca configuration>>=
association R103 ClassInstance 0..*--1..* ClassComponent\
    -associator ClassComponentValue
```

R109

- **Class Component Value** is either a **Specified Component Value** or an **Unspecified Component Value**

A Class Component Value may be set either by explicitly specifying the value or it may be left to the "C" compiler to initialize. Most Class Components are specified, either by a user or because they are generated components. However, Zero Initialized Attributes may be unspecified and in that case they are set to zero by the usual rules of "C" variable initialization.

Implementation

```
<<micca configuration>>=
generalization R109 ClassComponentValue\
        SpecifiedComponentValue UnspecifiedComponentValue
```

Specified Component Value

Domain {I,R109}
The domain to which the class component value belongs.

Data Type Refers to Class Component Value.Domain

Class {I,R109}
The name of the class to which the class component value pertains.

Data Type Refers to Class Component Value.Class

Component {I,R109}
The name of the class component to which the class component value applies.

Data Type Refers to Class Component Value.Component

Instance {I,R109}
The name of the class instance to which the class component value pertains.

Data Type Refers to Class Component Value.Instance

Value
The value to be given the the class component.

Data Type string

Implementation

```
<<micca configuration>>=
class SpecifiedComponentValue {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Component string -id 1
    attribute Instance string -id 1
    attribute Value string

    reference R109 ClassComponentValue -link Domain -link Class\
            -link Component -link Instance
}
```

Unspecified Component Value

Domain {I,R109}
The domain to which the class component value belongs.

Data Type Refers to Class Component Value.Domain

Class {I,R109}
The name of the class to which the class component value pertains.

Data Type Refers to Class Component Value.Class

Component {I,R109}
The name of the class component to which the class component value applies.

Data Type Refers to Class Component Value.Component

Instance {I,R109}
The name of the class instance to which the class component value pertains.

Data Type Refers to Class Component Value.Instance

Implementation

```
<<micca configuration>>=
class UnspecifiedComponentValue {
    attribute Domain string -id 1
    attribute Class string -id 1
    attribute Component string -id 1
    attribute Instance string -id 1

    reference R109 ClassComponentValue -link Domain -link Class\
            -link Component -link Instance
}
```

Multiple Assigner Instance

A Multiple Assigner Instance represents an instanciation of a Multiple Assigner.

Domain {I,I2,R107,R106,R108}
The domain to which the multiple assigner instance belongs.

Data Type Refers to Assigner Population.Domain, Class Instance.Domain and Multiple Assigner.Domain

Association {I,I2,R107,R106}
The name of the association to which the multiple assigner instance pertains.

Data Type Refers to Assigner Population.Association and Multiple Assigner.Association

Instance {I}
The name of the multiple assigner instance.

Data Type string

Number {I2}
A number associated with the multiple assigner instance. The number will be used as an array index into the storage array associated with the class.

Data Type zero based sequential number

IdClass {R108,C}
The name of the class which identifies the multiple assigner instance. Note that this attribute is further constrained to match the value of Multiple Assigner.Class when traversed along the R106 relationship.

Data Type Refers to Class Instance.Class

IdInstance {R108}
The name of the class instance which identifies the multiple assigner instance.

Data Type Refers to Class Instance.Instance

Implementation

```
<<micca configuration>>=
class MultipleAssignerInstance {
    attribute Domain string -id 1 -id 2
    attribute Association string -id 1 -id 2
    attribute Instance string -id 1
    attribute Number int -id 2
    attribute IdClass string
    attribute IdInstance string

    reference R107 AssignerPopulation -link Domain -link Association
    reference R106 MultipleAssigner -link Domain -link Association
    reference R108 ClassInstance -link Domain -link {IdClass Class}\
        -link {IdInstance Instance}
}
```

R106

- **Multiple Assigner Instance** is an instance of *exactly one* **Multiple Assigner**
- **Multiple Assigner** is instanciated as *zero or more* **Multiple Assigner Instance**

Each Multiple Assigner Instance must be associated to a particular Multiple Assigner. A Multiple Assigner may be defined separately from any population.

Implementation

```
<<micca configuration>>=
association R106 MultipleAssignerInstance 0..*--1 MultipleAssigner
```

R107

- **Multiple Assigner Instance** is part of *exactly one* **Assigner Population**
- **Assigner Population** contains *one or more* **Multiple Assigner Instance**

Assigner instances are specific to a given population. It is possible not to specify any initial instances for those assigners where all instances are created at run time.

Implementation

```
<<micca configuration>>=
association R107 MultipleAssignerInstance 0..*--1 AssignerPopulation
```

R108

- **Multiple Assigner Instance** is identified by *exactly one* **Class Instance**
- **Class Instance** is the identifier for *zero or more* **Multiple Assigner Instance**

A Multiple Assigner is always associated with a Class and so the Multiple Assigner Instances must be associated with a Class Instance. A Class Instance may be used to identify more that one Multiple Assigner Instances if the Class is the identifying class for more that one Multiple Assigner. It is possible for a Class to not identify any Multiple Assigners.

Implementation

```
<<micca configuration>>=
association R108 MultipleAssignerInstance 0..*--1 ClassInstance
```

Part IV

Configuring a Domain

In the last part of the book, we showed the platform specific model that `micca` uses. To translate a domain it is necessary to populate the platform specific model with the specific characteristics of the domain to be translated. Since the platform model is implemented in `rosea`, we could populate it using the `rosea populate` command. This would be a difficult and tedious way to populate the platform model for a human. It might be perfectly acceptable to create a `rosea` population programmatically, but it requires detailed knowledge of the platform model. We will provide a much more convenient interface for humans by using a domain specific language (DSL).

The DSL in this case is also a Tcl script and we will build the DSL processing directly into the `micca` domain. We will use the Tcl interpreter to parse the DSL. The code that reads the DSL arranges for the script to be executed in a context where command names resolve to procedures that populate the `micca` platform specific model.

In this part, we describe the commands that make up the DSL used to populate the platform model.

Chapter 11

Configuration Commands

Configuring a domain in `micca` happens as a domain operation. When we parse the various portions of a domain description, we will use namespaces to confine the script execution and to expose the appropriate commands to the configuration scripts. This requires us to define child namespaces for the `micca` domain. We must use some care in this because a `rosea` domain already has some child namespaces defined on it. We will use a naming convention to avoid any possible naming collisions.

The configuration script is a Tcl script and the full resources of Tcl are available. In particular, the `source` command is useful for organizing configuration scripts into smaller pieces that can be included, using the `source` command, into a configuration.

In this part, we first present the commands that are used to configure a domain under `micca`. These will be domain operations of the `micca` domain. Afterwards, we will define further child namespaces and commands within them that accomplish gathering all the information needed to define the characteristics of a domain.

Configure

One of our top level `micca` commands is to configure a domain from a script.

`::micca configure` *script*

`script`
A Tcl script that is executed in an environment that will resolve *domain configuration* commands that may be used to define the characteristics of a domain to be translated by `micca`.

Implementation

```
<<micca configuration>>=
operation configure {script} {
    return [@Config@::miccaConfigure $script]
}
```

ConfigureFromChan

It is often convenient to obtain the `configure` script from an I/O channel. The `micca configureFromChan` command supports reading the configure script directly from a channel.

`::micca configureFromChan` *channel*

`channel`
A Tcl channel handle that has been opened for reading.

The `configureFromChan` command invokes `configure` on the script obtained by reading *channel* until end of file is reached.

Implementation

```
<<micca configuration>>=
operation configureFromChan {chan} {
    return [@Config@::miccaConfigure [::chan read -nonewline $chan]]
}
```

ConfigureFromFile

It is often convenient to hold the `configure` script in a file. The `micca configureFromFile` command supports reading the configure script directly from a file.

`::micca configureFromFile` *filename*

`filename`
The name of a file containing a domain configuration script.

The `configureFromFile` command invokes `configure` on the script obtained by reading the contents of the file named, *filename*.

Implementation

```
<<micca configuration>>=
operation configureFromFile {filename} {
    namespace upvar @Config@ configfile configfile
    set configfile $filename
    set chan [::open $filename r]
    try {
        return [@Config@::miccaConfigure [::chan read -nonewline $chan]]
    } finally {
        ::chan close $chan
        set configfile {}
    }
}
```

Clearing the Configuration

It is sometimes convenient to be able to clear out the population of the underlying micca model. For example, processing successive different domain configurations requires removing the previous population.

`::micca clear`

Implementation

```
<<micca configuration>>=
operation clear {} {
    return [@Config@::miccaClear]
}
```

Chapter 12

Configuration Namespace Layout

The `::micca::@Config@` namespace holds all the procedures and data that correspond to the DSL to populate the platform model. We will create a set of child namespaces that mirror the nesting of the configuration language statements. At the top level, the `domain` command is used to define each domain. The `domain` command accepts a script body that then defines the components of the domain. That script will be evaluated in a namespace where the commands that define the domain components, *e.g.* `class` and relationship commands, resolve appropriately. This lets us put commands for the body specifying a particular component into a namespace which prevents any problems of accidentally invoking commands that are inappropriate for that context.

We will use this technique for other child namespaces that correspond to the nesting of DSL statements. We layout the `::micca::@Config@` namespace as shown below.

```
<<configuration commands namespace>>=
namespace eval @Config@ {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    <<helper commands namespace>>

    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds} ; # ❶
    variable configfile {}
    <<config data>>
    <<config commands>>

    <<domain config namespace layout>>
    <<population config namespace layout>>
}
```

❶ We are evaluating the configuration DSL inside a child namespace of of the `micca` domain, but we are trying to populate `micca` classes. We find it convenient to resolve commands through the main `micca` namespace as well as others that help things along.

Evaluating Configuration Scripts

The top level domain operations of `micca` that configure domains all invoke `miccaConfigure` as the common entry point into the configuration DSL code.

Implementation

```
<<config commands>>=
proc miccaConfigure {script} {
    variable errcount
    set errcount 0

    variable configlineno
    set configlineno 1

    ConfigEvaluate [namespace current] $script

    if {$errcount > 0} {
        tailcall DeclError CONFIG_ERRORS $errcount
    }
    return $errcount
}
```

```
<<error code formats>>=
CONFIG_ERRORS     {%d configuration script error(s)}
```

You could evaluate the configuration script by simply gathering it together and passing it to the core `namespace eval` command. Unfortunately, the first error that is encountered will terminate the evaluation. This is decidedly inconvenient as you would prefer to continue on and come up with a list of errors for the evaluation much like a conventional language compiler does when compiling a source file. Discovering errors one at a time is tediously under-productive.

Of course, it is possible to continue evaluating after an error but that requires a bit more code. We will also want to be able to evaluate scripts in different namespace contexts as we described above. We will accomplish all this by taking advantage of the core `::apply` command and its ability to execute a lambda function in a given namespace. So given a script body and an namespace we can evaluate it as shown below.

```
<<config commands>>=
namespace export ConfigEvaluate

proc ConfigEvaluate {ns body} {
    variable evalLambda
    tailcall ::apply [concat $evalLambda [list $ns]] $body ; # ❶
}
```

❶ The `concat` command treats its arguments as a list. On the off chance that the `ns` argument contains embedded whitespace, we need to insure that it is treated as a single element list. Hence the need to include the invocation of the `list` command.

The lambda function that performs the command evaluation is given below. We hold the evaluation lambda as a piece of data.

```
<<config data>>=
variable evalLambda {{body} {
    upvar #0 ::micca::@Config@::errcount errcount ; # ❶
    upvar #0 ::micca::@Config@::configlineno configlineno
    set lineno $configlineno
    set command {}
    foreach line [split $body \n] { # ❷
        append command $line \n
        incr lineno ; # ❸
        if {[info complete $command]} {
            try {
                eval $command
            } on error {result} {
                set cleancmd [CleanUpCommand $command]
                log::error "line $configlineno: \"$cleancmd\":\n\"$result\""
                incr errcount
```

```
            }
            set command {} ; # ❹
            set configlineno $lineno
        }
    }
    if {$command ne {}} {    # ❺
        set cleancmd [CleanUpCommand $command]
        log::error "line [expr {$lineno - 1}]: end of script reached in the\
            middle of the command starting at line $configlineno: $cleancmd"
        incr errcount
    }
    return $errcount
}}
```

❶ We need to keep track of all the errors encountered and what line of the script we are currently dealing with.

❷ We split the body along lines and then reassemble the lines into a complete command. The `info complete` command tells us when we have something that has some chance of being a real command.

❸ We need to keep track of where we are in the `body`.

❹ After evaluating a command we begin to assemble another one and set our location counter to where that command starts.

❺ Check if there is any dangling partial command. This could happen by missing a closing brace at the end of the configuration script.

We clean up the command string so that we don't print an excessive amount of information.

```
<<helper commands>>=
proc CleanUpCommand {command} {
    set cleancmd [string trim $command]
    if {[string length $cleancmd] > 60} {
        set cleancmd "[string range $cleancmd 0 54] ..."
    } else {
        set cleancmd $cleancmd
    }
    return $cleancmd
}
```

Clearing the Micca Model Population

Implementation

```
<<config commands>>=
proc miccaClear {} {
    set preserve {
        ::micca::TransitionRule
    }
    relvar eval {
        foreach var [relvar names {::micca::[A-Z]*}] {
            if {$var ni $preserve} {
                relvar set $var [relation empty [relvar set $var]]
            }
        }
    }

    set config {
        ::micca::@Config@::Helpers::SeqNumbers
```

```
    }
    foreach var $config {
        relvar set $var [relation empty [relvar set $var]]
    }
}

return [@Config@::miccaClear]
```

Chapter 13

Defining a Domain

As we described above, the domain configuration commands will take the data in their arguments and store it away. After we have finished all the configuration scripts, then the `generate` command is used to produce the "C" code and header files.

When configuring a domain, we will evaluate the configuration script in the `DomainDef` child namespace. That namespace will define commands for all the components of the domain. We will use a similar arrangement for other nested aspects of domain definitions.

Domain

`domain` *name body*

name
: The name of the domain. A domain name must be a non-empty string. The domain is used as a component for the file name of the generated code files, so domain names must be compatible with file name path components for the operating platform where `micca` is run.

body
: A Tcl script containing invocation of the domain definition commands to specify the details of the domain configuration.

The `domain` command defines a domain named, *name*, and evaluates *body* in a context where commands in *body* may be used to define the characteristics of a domain.

Implementation

```
<<config commands>>=
proc domain {name body} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME $name domain
    }

    namespace upvar DomainDef DomainName DomainName ; # ❶
    set DomainName $name

    CheckDuplicate Domain domain Name $name ; # ❷
    Domain create Name $name

    try {
```

```
        ral relvar eval {
            set errs [ConfigEvaluate [namespace current]::DomainDef $body]

            # No reason to go on if we already have errors in the domain script
            if {$errs == 0} {
                upvar #0 ::micca::@Config@::errcount errcount

                # We must compute how polymorphic events are inherited down
                # generalization hierarchies.
                <<domain: propagate polymorphic events>>
                <<domain: union subclass semantics>>
            }
        }
    } on error {result opts} {
        # puts $::errorInfo
        HandleConfigError $result
    }
}
```

❶ We place the domain name into a variable in the child namespace where the context implies that all the components defined are to be part of this domain.

❷ We do not allow the `domain` command to be invoked more than once. To configure a domain properly we need to be able to see its entire definition at once.

```
<<error code formats>>=
BAD_NAME      {"%s" is not a valid name for a(n) %s}
```

After evaluating the domain configuration script, it is necessary to perform some additional clean up and evaluation. For now, we will say that there are semantic evaluations and checks that can only be done after the entire domain configuration script has been processed. The reason for this lies mainly with the way the DSL script commands were defined to make them more convenient to use when translating an XUML model. Polymorphic events must be dealt with after the domain configuration is in place. Again we will discuss this further below.

Chapter 14

Defining Domain Components

In this section we discuss the commands that are used to define the components of a domain. Following our established pattern, we define child namespaces for those domain component commands that have a nested structure.

We start with the namespace layout for the `DomainDef` namespace.

```
<<domain config namespace layout>>=
namespace eval DomainDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace import ::micca::@Config@::Helpers::DeclError
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}
    <<domain config commands>>

    # Child namespaces for domain commands with additional structure
    <<class config namespace layout>>
    <<assigner config namespace layout>>
    <<external entity config namespace layout>>
}
```

We see that there are three child namespaces defined with `DomainDef`. These correspond to commands that have script bodies associated with them.

Interface

It is sometimes useful to include additional output in the header file generated for a domain. For example, the domain may define operation parameters that are of a type already defined in a header file. That header file needs to be included in the generated header for the domain. The `interface` command provides a means to include arbitrary text into the header file generated for the domain.

`interface` *text*

`text`
Arbitrary text to be placed in the output of the generated interface file.

The `interface` command adds the string of the *text* argument to the interface file generated for the domain.

Implementation

```
<<domain config commands>>=
proc interface {text} {
    variable DomainName
    AppendToDomainAttribute $DomainName Interface $text
}
```

Prologue

It is also convenient to be able to include arbitrary text into the generated code file for a domain. The `prologue` command includes that text before any of the generated code.

`prologue` *text*

`text`
Arbitrary text to be placed in the output code file before any generated code.

The `prologue` command adds the string of the *text* argument to the generated code file for the domain. The `prologue` command may be invoked multiple times and the value of all the *text* arguments is concatenated in the order encountered. The accumulated prologue text is emitted into the generated code file before any `micca` generated code.

Implementation

```
<<domain config commands>>=
proc prologue {text} {
    variable DomainName
    AppendToDomainAttribute $DomainName Prologue $text
}
```

Epilogue

The `epilogue` command includes text into the domain code file after any generated code. This can be used for small functions defined to handle the particulars of user defined data types.

`epilogue` *text*

`text`
Arbitrary text to be placed in the output code file after any generated code.

The `epilogue` command adds the string of the *text* argument to the generated code file for the domain. The `epilogue` command may be invoked multiple times and the value of all the *text* arguments is concatenated in the order encountered. The accumulated epilogue text is emitted into the generated code file after any `micca` generated code.

Implementation

```
<<domain config commands>>=
proc epilogue {text} {
    variable DomainName
    AppendToDomainAttribute $DomainName Epilogue $text
}
```

Append To Domain Attribute

The operations for the `interface`, `prologue` and `epilogue` commands can be factored into common code.

Implementation

```
<<domain config commands>>=
proc AppendToDomainAttribute {domainname attrname text} {
    set domref [Domain findById Name $domainname]
    set value [readAttribute $domref $attrname]
    if {$value ne {} && [string index $value end] ne "\n"} {
        append value \n ; # ❶
    }
    append value $text
    updateAttribute $domref $attrname $value
    return
}
```

❶ We make sure that the text appended to the domain attribute is done as a line. In case the previous line didn't have a new line in it, we will add one. This will prevent any problems where continuing on the same line could be misinterpreted.

Class

The class command is used to define classes and specify their characteristics.

`class` *name* ?*script*?

`name`
The name of the class. A class name must be a valid "C" identifier.

`script`
An optional Tcl script containing invocations of the class definition commands to specify the details of the class configuration.

The `class` command defines a class named, *name*, and evaluates *script* to configure the class. Class definitions are open ended and extensible. If the `class` command is invoked on a previously defined class, then any additional configuration is added to the definition of the class.

Implementation

```
<<domain config commands>>=
proc class {name {body {}}} {
    variable DomainName ; # ❶

    set elemRef [DomainElement findById Domain $DomainName Name $name]
```

```
    if {[isEmptyRef $elemRef]} {# ❷
        DomainElement create Domain $DomainName Name $name
        Class create Domain $DomainName Name $name\
            Number [GenNumber $DomainName Class [list $DomainName]]
        ValueElement create Domain $DomainName Name $name
    } else {
        set classRef [findRelated $elemRef {~R2 Class}]
        if {[isEmptyRef $classRef]} {
            EntityError "Domain Element" DUPLICATE_ENTITY\
                [list Domain $DomainName Name $name]
        }
    }

    namespace upvar ClassDef ClassName ClassName ; # ❸
    set ClassName $name
    ConfigEvaluate [namespace current]::ClassDef $body

    return
}
```

❶ We have arranged for a namespace variable to hold the current domain context. This is a convenient way to pass the domain name to where it is needed by the configuration commands.

❷ We allow the `class` command to be invoked multiple times with the same name. This makes extending class definitions (*e.g.* adding a state model to a class) much easier.

❸ Provide the class name to the body of the class definition script.

Tests

Association

The `association` command is used to define both simple and class based associations depending upon arguments. In the `association` command we use some syntax conventions to mirror the UML graphical notation to make the clerical aspects of the translation easier.

`association` *name ?option value . . . ? source spec target* ?*script*?

name
The name of the association. Conventionally, relationships names are of the form **R<d>** where <*d*> is one or more decimal digits but any non-empty string which does not start with a tilde character (~) can be used.

option value ...
Options to the `association` command are given as argument option / value pairs: Valid options and their values are:

-associator *class*
The `-associator` option defines the association to be **class based** and specifies *class* as the associator class.

-multiple
For associations formalized with an associator class, the `-multiple` option specifies if the multiple instances of the associator class are allowed to reference the same participant instances. This option may not be set for simple associations and is needed to support many-to-many-to-many associations.

-static|-dynamic
The `-static` option declares that the instances of the association does not change over the run time of the domain. The `-dynamic` option declares that the instances of the association may change during the run time of the domain. If neither option is present, then the association is assumed to be `-dynamic`. If the options are present multiple times, then the last option given takes effect. Knowledge of the dynamic behavior of the association allows `micca` to choose the best data structures for holding association information.

--
Forces the end of options processing.

source
The name of the class that defines the starting class for a forward traversal of the relationship.

spec
The specifier of the relationship conditionality and multiplicity. The *spec* argument is one of the following strings. We allow two forms for some of the specifiers. The first form follows UML notation and the second form follows TclRAL notation which is patterned after regular expression notation.

1--1
The association is *one to one*.

0..1--1 or ?--1
The association is *at most one to one*.

0..1--0..1 or ?--?
The association is *at most one to at most one*.

1..*--1 or +--1
The association is *one or more to one*.

0..*--1 or *--1
The association is *zero or more to one*.

1..*--0..1 or +--?
The association is *one or more to at most one*.

0..*--0..1 or *--?
The association is *zero or more to at most one*.

1..*--1..* or +--+
The association is *one or more to one or more*.

0..*--1..* or *--+
The association is *zero or more to one or more*.

1..*--0..* or +--*
The association is *one or more to zero or more*.

0..*--0..* or *--*
The association is *zero or more to zero or more*.

target
The name of the class that defines the ending class for a forward traversal of the relationship.

script
An optional *script* that is evaluated to define an assigner on the association.

The `association` command defines an association relationship between *source* class and *target* class. The forward direction of navigating the relationship is from *source* to *target*. The conditionality and multiplicity of the association is given by the *spec* argument. A class based association is indicated by the `-associator` option. The association command may have an optional *script* argument that is used to define an assigner on the association.

In the command, we are using the *spec* string to encode several distinct pieces of information. Each different association specifier implies the conditionality, multiplicity and other properties of the association. We encode those properties in data whose identifier is the *spec* string itself.

Since we allow two different techniques to identify the association specifics, we will factor away the specification data from its name.

```
<<config data>>=
relvar create Config_AssocSpec {
    SpecName    string
    SpecID      string
} SpecName
```

The following relvar contains the properties of an association. The `NeedsAssociator` attribute determines where the must be an association class defined for the association. Since we do not allow `NULL` attribute values, some associations will require association classes. The `ReflexiveAllowed` attribute determine is the association may be reflexive, *i.e.* if association participants can be the same class. The `MultipleAllowed` attibute determines if a many-to-many-to-many association is allowed. In that case, there may be multiple instances of the association class that reference the same participant class instances.

```
<<config data>>=
relvar create Config_SpecDetail {
    SpecID              string
    NeedsAssociator     boolean
    ReflexiveAllowed    boolean
    MultipleAllowed     boolean
    ReferringCond       boolean
    ReferringMult       boolean
    ReferencedCond      boolean
    ReferencedMult      boolean
} SpecID
```

The following is the population of the association specifications for the 10 distinct combinations of multiplicity and conditionality.

```
<<config data>>=
relvar association C1\
    Config_AssocSpec SpecID +\
    Config_SpecDetail SpecID 1

relvar eval {
    relvar insert Config_AssocSpec {
        SpecName    1--1
        SpecID      sp0
    } {
        SpecName    0..1--1
        SpecID      sp1
    } {
        SpecName    ?--1
        SpecID      sp1
```

```
    } {
        SpecName    0..1--0..1
        SpecID      sp2
    } {
        SpecName    ?--?
        SpecID      sp2
    } {
        SpecName    1..*--1
        SpecID      sp3
    } {
        SpecName    +--1
        SpecID      sp3
    } {
        SpecName    0..*--1
        SpecID      sp4
    } {
        SpecName    *--1
        SpecID      sp4
    } {
        SpecName    1..*--0..1
        SpecID      sp5
    } {
        SpecName    +--?
        SpecID      sp5
    } {
        SpecName    0..*--0..1
        SpecID      sp6
    } {
        SpecName    *--?
        SpecID      sp6
    } {
        SpecName    1..*--1..*
        SpecID      sp7
    } {
        SpecName    +--+
        SpecID      sp7
    } {
        SpecName    0..*--1..*
        SpecID      sp8
    } {
        SpecName    *--+
        SpecID      sp8
    } {
        SpecName    1..*--0..*
        SpecID      sp9
    } {
        SpecName    +--*
        SpecID      sp9
    } {
        SpecName    0..*--0..*
        SpecID      sp10
    } {
        SpecName    *--*
        SpecID      sp10
    }

    relvar insert Config_SpecDetail {
        SpecID sp0  NeedsAssociator false ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond false ReferringMult false
            ReferencedCond false ReferencedMult false
    } {
```

```
        SpecID sp1 NeedsAssociator false ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond true ReferringMult false
            ReferencedCond false ReferencedMult false
    } {
        SpecID sp2 NeedsAssociator true ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond true ReferringMult false
            ReferencedCond true ReferencedMult false
    } {
        SpecID sp3 NeedsAssociator false ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond false ReferringMult true
            ReferencedCond false ReferencedMult false
    } {
        SpecID sp4 NeedsAssociator false ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond true ReferringMult true
            ReferencedCond false ReferencedMult false
    } {
        SpecID sp5 NeedsAssociator true ReflexiveAllowed false
            MultipleAllowed false
            ReferringCond false ReferringMult true
            ReferencedCond true ReferencedMult false
    } {
        SpecID sp6 NeedsAssociator true ReflexiveAllowed true
            MultipleAllowed false
            ReferringCond true ReferringMult true
            ReferencedCond true ReferencedMult false
    } {
        SpecID sp7 NeedsAssociator true ReflexiveAllowed true
            MultipleAllowed true
            ReferringCond false ReferringMult true
            ReferencedCond false ReferencedMult true
    } {
        SpecID sp8 NeedsAssociator true ReflexiveAllowed false
            MultipleAllowed true
            ReferringCond true ReferringMult true
            ReferencedCond false ReferencedMult true
    } {
        SpecID sp9 NeedsAssociator true ReflexiveAllowed false
            MultipleAllowed true
            ReferringCond false ReferringMult true
            ReferencedCond true ReferencedMult true
    } {
        SpecID sp10 NeedsAssociator true ReflexiveAllowed true
            MultipleAllowed true
            ReferringCond true ReferringMult true
            ReferencedCond true ReferencedMult true
    }
}
```

Implementation

Like most of the commands in the configuration language, the bulk of the code is involved with populating the relvars holding the essential data provided in the command arguments. For the `association` command, there is some argument parsing to determine whether the association is simple or class based. Then it is a matter of filling in the correct information based on the type of the association.

```
<<domain config commands>>=
```

```
proc association {name args} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME [list $name] association
    }
    if {[string index $name 0] eq "~"} {
        tailcall DeclError TILDE_NAME $name
    }

    if {[llength $args] < 3} {
        tailcall DeclError ASSOC_OPTIONS $args
    }

    set allargs $args
    set associator {}
    set isstatic false
    set ismultiple false
    while {1} {
        set args [lassign $args arg]

        if {$arg eq "-associator"} {
            set args [lassign $args associator]
        } elseif {$arg eq "-multiple"} {
            set ismultiple true
        } elseif {$arg eq "-static"} {
            set isstatic true
        } elseif {$arg eq "-dynamic"} {
            set isstatic false
        } elseif {$arg eq "--"} {
            set args [lassign $args source]
            break
        } elseif {[string index $arg 0] eq "-"} {
            tailcall DeclError ASSOC_OPTIONS $allargs
        } else {
            set source $arg
            break
        }
    }
    if {[llength $args] < 2 || [llength $args] > 3} {
        tailcall DeclError ASSOC_OPTIONS $allargs
    }
    lassign $args spec target script

    # Obtain references to the domain name
    variable DomainName

    <<association: check arguments>>

    # Many relvars have tuples with the same heading, so we construct it
    # once here.
    set reltuple [list\
        Domain  $DomainName\
        Name $name\
    ]
    CheckDuplicate DomainElement association {*}$reltuple
    DomainElement create {*}$reltuple

    # Populate the data for a Relationship and Association since that
    # is what this command defines.
    Relationship create {*}$reltuple\
        Number [GenNumber $DomainName Relationship [list $DomainName]]
    Association create {*}$reltuple IsStatic $isstatic
```

```
    # Set up tuple values for later use.
    set sourcetuple [list\
        Domain          $DomainName\
        Class           $source\
        Relationship    $name\
        Role            source\
    ]
    set targettuple [list\
        Domain          $DomainName\
        Class           $target\
        Relationship    $name\
        Role            target\
    ]

    # Populate the type of association we are dealing with.
    if {$associator eq {}} {
        <<association: populate simple association>>
    } else {
        <<association: populate class based association>>
    }

    if {$script ne {}} {
        namespace upvar ClassDef ClassName ClassName
        set ClassName $name

        namespace upvar AssignerDef IdClassName IdClassName
        set IdClassName {}

        ConfigEvaluate [namespace current]::AssignerDef $script

        if {$IdClassName eq {}} {
            SingleAssigner create\
                Domain          $DomainName\
                Association     $name
        } else {
            MultipleAssigner create\
                Domain          $DomainName\
                Association     $name\
                Class           $IdClassName
            ValueElement create Domain $DomainName Name $name
        }
    }

    return
}
```

```
<<error code formats>>=
TILDE_NAME {names beginning with the tilde character are not allowed\
    in this context, "%s"}
ASSOC_OPTIONS {association options error, expected\
    "?-associator <class> | -multiple | -static | -dynamic | --?\
    source spec target ?script?", got "%s"}
```

Here we use the data that we supplied above to make sure the `spec` association specifier and the supplied arguments make sense together. Certain forms of reflexiveness are not allowed and certain specifiers imply that you must define a class based association. Such rules are easier to express in data rather than long sequences of checking code.

```
<<association: check arguments>>=
if {$source eq {}} {
    tailcall DeclError BAD_NAME [list $source] class
}
```

```
if {$target eq {}} {
    tailcall DeclError BAD_NAME [list $target] class
}

set csd [pipe {
    relvar restrictone ::micca::@Config@::Config_AssocSpec SpecName $spec |
    relation semijoin ~ [relvar set ::micca::@Config@::Config_SpecDetail]
}]
if {[relation isempty $csd]} {
    tailcall DeclError BAD_RELATIONSHIP_SPEC $spec
}
relation assign $csd
if {$NeedsAssociator && $associator eq {}} {
    tailcall DeclError NEED_ASSOCIATOR $spec
}
if {!($source ne $target || $ReflexiveAllowed)} {
    tailcall DeclError REFLEXIVE_NOT_ALLOWED $spec
}
if {$ismultiple && !$MultipleAllowed} {
    tailcall DeclError BAD_MULTIPLE_OPT $name
}
```

```
<<error code formats>>=
BAD_RELATIONSHIP_SPEC  {bad association specifier, "%s"}
NEED_ASSOCIATOR {relationship of type, "%s", requires an associative class}
REFLEXIVE_NOT_ALLOWED   {associations of type, "%s", cannot be reflexive}
BAD_MULTIPLE_OPT  {for relationship "%s", multiple option is not allowed}
```

For simple associations, the `association` command arguments supply what we need and we perform a series of inserts into the platform data model.

```
<<association: populate simple association>>=

SimpleAssociation create {*}$reltuple

SimpleReferringClass create {*}$sourcetuple\
        Conditionality  $ReferringCond\
        Multiplicity    $ReferringMult
ClassRole create {*}$sourcetuple

set poptuple [list\
    Domain          $DomainName\
    Class           $source\
    Name            $name\
]
AssociationReference create {*}$poptuple
Reference create {*}$poptuple
PopulatedComponent create {*}$poptuple
ClassComponent create {*}$poptuple

SimpleReferencedClass create {*}$targettuple
DestinationClass create {*}$targettuple
ClassRole create {*}$targettuple

set gentuple [list\
    Domain          $DomainName\
    Class           $target\
    Name            ${name}__BACK\
]
ComplementaryReference create {*}$gentuple
BackwardReference create {*}$gentuple Relationship $name
if {!$ReferringMult} {
```

```
    SingularReference create {*}$gentuple
} elseif {$isstatic} {
    ArrayReference create {*}$gentuple
} else {
    LinkReference create {*}$gentuple
    set conttuple [list\
        Domain      $DomainName\
        Class       $source\
        Name        ${name}__BLINKS\
    ]
    LinkContainer create {*}$conttuple LinkClass $target LinkComp ${name}__BACK
    GeneratedComponent create {*}$conttuple
    ClassComponent create {*}$conttuple
}
GeneratedComponent create {*}$gentuple
ClassComponent create {*}$gentuple
```

For class based associations we have a few more class instances to create as we populate that branch of the `micca` platform model.

```
<<association: populate class based association>>=
ClassBasedAssociation create {*}$reltuple

SourceClass create {*}$sourcetuple\
        Conditionality  $ReferringCond\
        Multiplicity    $ReferringMult
ClassRole create {*}$sourcetuple

set srccomp [list\
    Domain          $DomainName\
    Class           $source\
    Name            ${name}__FORW\
]
ComplementaryReference create {*}$srccomp
ForwardReference create {*}$srccomp Relationship $name
if {!$ReferencedMult} {
    SingularReference create {*}$srccomp
} elseif {$isstatic} {
    ArrayReference create {*}$srccomp
} else {
    LinkReference create {*}$srccomp
    set conttuple [list\
        Domain      $DomainName\
        Class       $associator\
        Name        ${name}__FLINKS\
    ]
    LinkContainer create {*}$conttuple LinkClass $source LinkComp ${name}__FORW
    GeneratedComponent create {*}$conttuple
    ClassComponent create {*}$conttuple
}
GeneratedComponent create {*}$srccomp
ClassComponent create {*}$srccomp

TargetClass create {*}$targettuple\
    Conditionality  $ReferencedCond\
    Multiplicity    $ReferencedMult
DestinationClass create {*}$targettuple
ClassRole create {*}$targettuple

set trgcomp [list\
    Domain          $DomainName\
    Class           $target\
```

```
    Name            ${name}__BACK\
]
ComplementaryReference create {*}$trgcomp
BackwardReference create {*}$trgcomp Relationship $name
if {!$ReferringMult} {
    SingularReference create {*}$trgcomp
} elseif {$isstatic} {
    ArrayReference create {*}$trgcomp
} else {
    LinkReference create {*}$trgcomp
    set conttuple [list\
        Domain      $DomainName\
        Class       $associator\
        Name        ${name}__BLINKS\
    ]
    LinkContainer create {*}$conttuple LinkClass $target LinkComp ${name}__BACK
    GeneratedComponent create {*}$conttuple
    ClassComponent create {*}$conttuple
}
GeneratedComponent create {*}$trgcomp
ClassComponent create {*}$trgcomp

set assoctuple [list\
    Domain          $DomainName\
    Class           $associator\
    Relationship    $name\
    Role            associator\
]
set assoccomp [list\
    Domain          $DomainName\
    Class           $associator\
    Name            $name\
]

AssociatorClass create {*}$assoctuple Multiplicity $ismultiple
ClassRole create {*}$assoctuple
AssociatorReference create {*}$assoccomp
Reference create {*}$assoccomp
PopulatedComponent create {*}$assoccomp
ClassComponent create {*}$assoccomp
```

Generalization

Defining a generalization relationship is quite a bit simpler than the effort we just saw in defining associations. For generalizations, it is necessary to specify the superclass and the set of subclasses. The only complexity is that `micca` allows two choices for how to store subclasses of a generalization. The simplest is to use references to store the relationship pointers in much the same manner as used for associations. Because of the nature of a generalization relationship, the other possibility is to store the subclass as a discriminated union. The alternatives are specified as options to the generalization command.

`generalization` *name ?-union | -reference? super sub1 sub2 ?... ?*

name
The name of the generalization. Conventionally, relationships names are of the form **R<d>** where *<d>* is one or more decimal digits but any non-empty string can be used.

-union | -reference
An option to specify if subclasses of the generalization are to be held as a discriminated union in the same structure as the superclass or held in separate storage having the relationship navigation implemented using reference pointers. If no option is given, the `-reference` is assumed.

super
The name of the class that serves as the superclass of the generalization.

subN
The names of the classes that serve as the subclasses of the generalization. You must specify at least two subclasses to define a generalization.

Implementation

```
<<domain config commands>>=
proc generalization {name args} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME [list $name] generalization
    }
    if {[string index $name 0] eq "~"} {
        tailcall DeclError TILDE_NAME $name
    }

    set allargs $args
    set type reference
    while {1} {
        set args [lassign $args arg]

        if {$arg eq "-reference"} {
            set type reference
        } elseif {$arg eq "-union"} {
            set type union
        } elseif {$arg eq "--"} {
            set args [lassign $args super]
            break
        } elseif {[string index $arg 0] eq "-"} {
            tailcall DeclError GEN_OPTIONS $allargs
        } else {
            set super $arg
            break
        }
    }

    if {[llength $args] < 2} {
        tailcall DeclError TOO_FEW_SUBCLASSES [llength $args]
    }
    if {$super in $args} {
        tailcall DeclError SUPER_AS_SUBCLASS $super [join $args {, }]
    }
    if {[llength [lsort -unique $args]] != [llength $args]} {
        tailcall DeclError DUPLICATE_SUBCLASS $args
    }
```

```
variable DomainName

set reltuple [list\
    Domain  $DomainName\
    Name $name\
]

CheckDuplicate DomainElement generalization {*}$reltuple
DomainElement create {*}$reltuple
Relationship create {*}$reltuple\
    Number [GenNumber $DomainName Relationship [list $DomainName]]
Generalization create {*}$reltuple

set supertuple [list\
    Domain          $DomainName\
    Class           $super\
    Relationship    $name\
    Role            target\
]
set subtuple [dict create\
    Domain          $DomainName\
    Relationship    $name\
    Role            source\
]
set reftuple [list\
    Domain          $DomainName\
    Class           $super\
    Name            $name
]
if {$type eq "reference"} {
    ReferenceGeneralization create {*}$reltuple
    ReferencedSuperclass create {*}$supertuple
    SubclassReference create {*}$reftuple
    GeneratedComponent create {*}$reftuple
    ClassComponent create {*}$reftuple

    foreach sub $args {
        dict set subtuple Class $sub
        ReferringSubclass create {*}$subtuple
    }
} elseif {$type eq "union"} {
    UnionGeneralization create {*}$reltuple
    UnionSuperclass create {*}$supertuple
    SubclassContainer create {*}$reftuple
    GeneratedComponent create {*}$reftuple
    ClassComponent create {*}$reftuple

    foreach sub $args {
        set usubRef [UnionSubclass findWhere {$Domain eq $DomainName &&\
            $Class eq $sub}] ; # ❶
        if {[isNotEmptyRef $usubRef]} {
            tailcall DeclError UNION_SUBCLASS_EXISTS $sub\
                [readAttribute $usubRef Relationship]
        }
        dict set subtuple Class $sub
        UnionSubclass create {*}$subtuple
    }
}
Superclass create {*}$supertuple
ClassRole create {*}$supertuple
```

```
    foreach sub $args {
        dict set subtuple Class $sub
        Subclass create {*}$subtuple
        ClassRole create {*}$subtuple
        dict set reftuple Class $sub
        SuperclassReference create {*}$reftuple
        Reference create {*}$reftuple
        PopulatedComponent create {*}$reftuple
        ClassComponent create {*}$reftuple
    }

    return
}
```

❶ A union subclass cannot be a subclass of two independent generalizations. This is just another way of saying that a component of a "C" structure cannot be a component of a different "C" structure at the same time. If a subclass is to be a subclass in multiple generalizations, only one can be a union subclass and the others must be reference subclasses.

```
<<error code formats>>=
GEN_OPTIONS     {generalization options error, expected\
    "?-union | -reference | --? superclass subclass1 subclass2 ...", got "%s"}
TOO_FEW_SUBCLASSES  {at least 2 subclasses must be specified, got %d}
SUPER_AS_SUBCLASS   {super class, "%s", cannot be included in subclasses, "%s"}
DUPLICATE_SUBCLASS  {subclass set contains a duplicate subclass name, "%s"}
UNION_SUBCLASS_EXISTS {subclass, "%s", is already a union subclass for\
                    relationship, "%s"}
```

External Entity

We define external entities as a collection of operations that serve as a proxy for services delegated by the domain.

`eentity` *name* ?*script*?

name
The name of the external entity. An external entity name must be a valid "C" identifier.

script
An optional Tcl script containing invocations of the external entity definition commands to specify the operations of the external entity configuration.

The `eentity` command defines a external entity named, *name*, and evaluates *script* to configure the external entity. External entity definitions are open ended and extensible. If the `eentity` command is invoked on a previously defined eentity, then any additional configuration is added to the definition of the eentity.

Implementation

```
<<domain config commands>>=
proc eentity {name {body {}}} {
    variable DomainName ; # ❶

    set elemRef [DomainElement findById Domain $DomainName Name $name]

    if {[isEmptyRef $elemRef]} {# ❷
```

```
        DomainElement create Domain $DomainName Name $name
        ExternalEntity create Domain $DomainName Name $name
    } else {
        set eeRef [findRelated $elemRef {~R2 ExternalEntity}]
        if {[isEmptyRef $eeRef]} {
            EntityError "Domain Element" DUPLICATE_ENTITY\
                [list Domain $DomainName Name $name]
        }
    }

    namespace upvar EEntityDef EntityName EntityName ; # ❸
    set EntityName $name
    ConfigEvaluate [namespace current]::EEntityDef $body

    return
}
```

❶ We have arranged for a namespace variable to hold the current domain context. This is a convenient way to pass the domain name to where it is needed by the configuration commands.

❷ We allow the `eentity` command to be invoked multiple times with the same name. This makes extending external entity definitions much easier.

❸ Provide the external entity name to the body of the external entity definition script.

Tests

Type Alias

It is useful to be able to define domain specific data types in "C" implementation terms. This allows for using the domain type name in an attribute definition while obtaining the proper "C" data type in the implementation.

`typealias` *name definition*

`name`
The name of the type alias.

`definition`
A "C" typename.

The `typealias` command defines an alias *name* for a "C" typename given by, *definition*. Type alias helps map model level type information to "C" `typedef` statements that are included in the generated code.

Implementation

```
<<domain config commands>>=
proc typealias {aliasname typename} {
    variable DomainName
    CheckDuplicate TypeAlias typealias Domain $DomainName TypeName $aliasname
    TypeAlias create Domain $DomainName TypeName $aliasname\
        TypeDefinition [string trim $typename]

    return
}
```

Domain Operation

The `domainop` command is used to define domain operations. The set of domain operations defined for a domain constitute the external callable interface to the domain. Typically, domain operations are defined to allow other domains to access major services of the domain.

`domainop` *rettype name parameters body* ?*comment*?

`rettype`
A "C" typename that gives the type of the value returned from the domain operation.

`name`
The name of the domain operation. `Name` must be a valid "C" identifier.

`parameters`
A list of parameter name / parameter type pairs giving the names of the domain operation parameters and their corresponding "C" data type names.

`body`
A string containing the code that is executed when the domain operation is invoked. This string is presumed to be "C" code optionally interspersed with architecture macros.

`comment`
An optional string that is passed along by the code generated to the generated header file.

Implementation

```
<<domain config commands>>=
proc domainop {rettype name parameters body {comment {}}} {
    variable DomainName
    upvar #0\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno

    CheckDuplicate DomainOperation {domain operation}\
            Domain $DomainName Name $name
    DomainOperation create\
        Domain $DomainName\
        Name $name\
        Body $body\
        ReturnDataType $rettype\
        Comment [string trim $comment]\
        File $configfile\
        Line $configlineno

    set paramtuple [dict create Domain $DomainName Operation $name]
    dict for {paramname paramtype} $parameters {
        dict set paramtuple Name $paramname
        dict set paramtuple DataType $paramtype
        dict set paramtuple Number\
                [GenNumber $DomainName DomainOperationParameter\
                [list $DomainName $name]]
        DomainOperationParameter create {*}$paramtuple
    }

    return
}
```

Chapter 15

Defining Class Components

Since there are several aspects of classes, the `class` command takes a *script* argument which should invoke the commands we discuss in this section. Following our pattern, we define a namespace where the class body script is evaluated.

```
<<class config namespace layout>>=
namespace eval ClassDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<class config commands>>

    <<state model config namespace layout>>
}
```

Attribute

Within a class definition, the `attribute` command specifies the attributes of the class. In this context, attributes are a slightly different concept than the attributes seen on a class diagram. As part of the translation process, some attributes found on the XUML class diagram are elided from the implementation. For example, attributes that are used solely for referential or identifying purposes need not be included as attributes in the implementation. This is because the translation is using the address of a class instance as an architecturally supplied identifier and in implementing relationship traversal. So the attributes defined here are descriptive in nature and define the logical parameters of the class rather than the structural aspects of the class.

`attribute` *name type* ? *-option value* ... ?

? `-default` *value* `-dependent` *formula*?

`name`
The name of the attribute. Attributes names may not be the empty string.

`type`
The type of the attribute. The *type* may be any valid "C" type name.

-option / value
Valid attribute command options are:

`-default` *value*
If specified, the `-default` option specifies a default value for the attribute. If no value is supplied when a class instance is created or populated, then the default value is used.

`-dependent` *formula*
If the `-dependent` option is specified, then the attribute is considered to be mathematically dependent. Such attributes may only be read and the value of the attribute will be the return value of the "C" code contained in *formula*. The *formula* argument is a string containing "C" code that is invoked when the attribute is read. The *formula* code is placed in a function whose interface is three parameters. The parameters are `self`, a pointer to the result whose name is the same as the attribute name and an unsigned `size` parameter giving the number of bytes pointed to by the result pointer. The formula code is expected to compute the result and assign it via the pointer parameter. It is not allowed to specify the `-dependent` option and the `-default` option for the same attribute.

The implementation of the `attribute` command simply creates tuples in the relvars to hold the attribute characteristics.

Implementation

```
<<class config commands>>=
proc attribute {name type args} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME [list $name] attribute
    }

    upvar #0\
            [namespace parent]::DomainName DomainName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno
    variable ClassName

    set options [dict create]
    while {[llength $args] != 0} {
        set args [lassign $args opt]
        switch -exact -- $opt {
            -default {
                if {[llength $args] != 0} {
                    set args [lassign $args optValue]
                    dict set options $opt $optValue
                } else {
                    tailcall DeclError ARG_FORMAT $opt
                }
            }
            -dependent {
                if {[llength $args] != 0} {
                    set args [lassign $args optValue]
                    dict set options $opt $optValue
```

```
                } else {
                    tailcall DeclError ARG_FORMAT $opt
                }
            }
            -zeroinit {
                dict set options $opt true
            }
            default {
                tailcall DeclError ATTR_OPTIONS $opt
            }
        }
    }

    # The attribute options are mutually exclusive. Make sure more than
    # did not get set.
    if {[dict size $options] > 1} {
        tailcall DeclError ATTR_OPTIONS $options
    }

    set attrtuple [list\
        Domain      $DomainName\
        Class       $ClassName\
        Name        $name\
    ]
    CheckDuplicate Attribute attribute {*}$attrtuple
    Attribute create {*}$attrtuple DataType $type
    PopulatedComponent create {*}$attrtuple
    ClassComponent create {*}$attrtuple
    if {[dict exists $options -dependent]} {
        DependentAttribute create {*}$attrtuple\
            Formula [dict get $options -dependent]\
            File $configfile\
            Line $configlineno
    } else {
        IndependentAttribute create {*}$attrtuple
        if {[dict exists $options -zeroinit]} {
            ZeroInitializedAttribute create {*}$attrtuple
        } else {
            ValueInitializedAttribute create {*}$attrtuple
            if {[dict exists $options -default]} {
                DefaultValue create\
                    Domain      $DomainName\
                    Class       $ClassName\
                    Attribute   $name\
                    Value       [dict get $options -default]
            }
        }
    }
}
```

```
<<error code formats>>=
ARG_FORMAT      {options and values must come in pairs, got "%s"}
ATTR_OPTIONS    {attribute options error, expected\
    "-default <value> or -dependent <formula> or -zeroinit", got "%s"}
```

Polymorphic Events

Polymorphic events are defined in a superclass and when dispatched to the superclass are mapped into events in the subclass state models. The `polymorphic` commands defines a polymorphic event and optionally the signature of event parameters that the

event carries.

`polymorphic` *event ?argname argtype ...?*

`event`
The name of an event that will be deemed polymorphic across any generalization relationships in which the class participates.

`argname argtype ...`
The argument signature of the event. Arguments must be given in name / type pairs. Argument names must be "C" identifiers and argument types must be "C" type names.

The `polymorphic` command defines *event* as being polymorphic with optional event arguments given by *argname* / *argtype* pairs. Note that the signature of event arguments of a polymorphic event is inherited down the generalization hierarchy. If when the polymorphic event is mapped onto a local state machine any event parameters required by the state must have been defined using this command. Unlike a non-polymorphic event which will assume the parameter signature of a state unless otherwise defined, polymorphic event parameters must be defined in this command.

Implementation

For a polymorphic event, we simply create instances of the PolymorphicEvent class and its related classes. At the end of a domain configuration script we will find it necessary to rationalize the polymorphic events and the transition events defined locally in a state model.

```
<<class config commands>>=
proc polymorphic {name args} {
    upvar #0 [namespace parent]::DomainName DomainName
    variable ClassName

    set eventtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        Event   $name\
    ]
    CheckDuplicate Event {polymorphic event} {*}$eventtuple
    Event create {*}$eventtuple Number -1
    set psigid [FindParameterSignature $args]
    if {$psigid ne {}} {
        EventSignature create {*}$eventtuple PSigID $psigid
    }
    DeferredEvent create {*}$eventtuple
    PolymorphicEvent create {*}$eventtuple

    return
}
```

Propagating Polymorphic Events

Previously, we indicated that polymorphic event rules require that we perform some other processing once the configuration script for a domain has been executed. Now we are prepared to show that processing. It will be helpful to refer to the Event subsystem of the platform model above to follow the description.

The `polymorphic` command above inserts the argument event names into the model classes simply as a deferred event that is polymorphic. We must resolve two issues.

1. For classes that are super classes for generalizations, we must have corresponding instances of **Deferral Path** for each of the generalization hierarchies.

2. For classes that are sub classes, they either consume the polymorphic event in a state model or it continues to be inherited by any sub classes further down the generalization hierarchy.

To solve the first issue we know that relationship, **R86**, specifies how a polymorphic event may be propagated along multiple generalizations when it is signaled to a given superclass. After the configuration script has been executed, we will have all the **Superclass** instances and all the **Deferred Event** instances and can now make up the correlation between them.

Starting with the **Polymorphic Event** class we can navigate **R81** to find the corresponding **Deferred Event** and then join to the **Superclass**. This gives the set of paths along which the polymorphic event must be propagated.

```
<<domain: propagate polymorphic events>>=
# Create Deferral Path instances corresponding to polymorphic events

set paths [pipe {
    PolymorphicEvent findWhere {$Domain eq $name} |
    findRelated ~ R81 |
    deRef ~ Domain Model Event |
    relation join ~ [deRef [Superclass findAll]]\
        -using {Domain Domain Model Class}
}]

foreach path [relation body $paths] {
    DeferralPath create {*}$path
}
```

To solve the second issue, we must walk the generalization hierarchy and determine if events are being consummed or inherited down the hierarchy. We want to start the walk only on the ultimate super classes, *i.e.* those super classes that are *not* themselves the sub class of some other generalization.

```
<<domain: propagate polymorphic events>>=
forAllRefs super [FindUltimateSuperclasses $name] {
    PropagatePolyEvents $super
}
```

After all the polymorphic events have been rearranged, we can now number the events. There is one continuous set of event numbers for any class, but we want them separated into two groups. Each group will be sequentially ordered so that the event number can be used as an index by the run time code. We group Transitioning Events first followed by Deferred Events.

```
<<domain: propagate polymorphic events>>=
set trevents [pipe {
    TransitioningEvent findAll |
    deRef ~ |
    relation tag ~ Number -ascending Event -within {Domain Model}
}]
# puts [relformat $trevents trevents]

set trcounts [relation summarizeby $trevents {Domain Model} evts\
        EventCount int {[relation cardinality $evts]}]
# puts [relformat $trcounts trcounts]

set dfevents [pipe {
    DeferredEvent findAll |
    deRef ~ |
    relation tag ~ DfNumber -ascending Event -within {Domain Model} |
    ralutil::rvajoin ~ $trcounts TEvents |
    relation extend ~ dftup Number int {
        [relation isempty [tuple extract $dftup TEvents]] ?\
        [tuple extract $dftup DfNumber] :\
        [tuple extract $dftup DfNumber] +\
        [relation extract [tuple extract $dftup TEvents] EventCount]} |
    relation eliminate ~ DfNumber TEvents
}]
```

```
# puts [relformat $dfevents dfevents]

Event update [relation union $trevents $dfevents]
# puts [relformat $::micca::Event Event]
```

So we find all **Superclass** instances that have no corresponding **Subclass** role and propagate any polymorphic events down the generalization hierarchy.

```
<<helper commands>>=
proc FindUltimateSuperclasses {domain} {
    set subs [deRef [Subclass findWhere {$Domain eq $domain}]]
    set supers [deRef [Superclass findWhere {$Domain eq $domain}]]
    return [pipe {
        relation semiminus $subs $supers -using {Domain Domain Class Class} |
        ::rosea::Helpers::ToRef ::micca::Superclass ~
    }] ; # ❶
}
```

❶ This is where a little relational algebra can go a long way. The `semiminus` command finds all the tuples that are **not** related. In this case we find the Superclass instances that do **not** have a correspondence to a Subclass instance. The `semiminus` operation is done only across the same domain and class, *i.e.* without respect to any relationship. This yields the Superclass instances that are **not** Subclass instances of any relationship.

The problem we are trying to solve in the propagation of polymorphic events arises from the way in which we attempt to minimize the amount of input from the user when specifying the events. Recall that the `polymorphic` command simply inserts instances into the **Polymorphic Event** class (and corresponding instances in **Deferred Event** and **Event** classes). Events found when defining a state model are simply created as instances of the **Local Event** class (and again the corresponding **Transitioning Event** and **Event** instances). At the end of the configuration process we can now deduce which events were inherited down the generalization hierarchy and which were truly local events.

For sub classes that are leafs of generalization hierarchy, polymorphic events inherited from the super class are migrated to be **Mapped Events**. For sub classes that are intermediate in the hierarchy, polymorhphic events are migrated to instances of **Inherited Event**. Both **Inherited Event** and **Mapped Event** are types of **NonLocal Event** and **R85** insures that they affect only **Subclass** instances.

There are two other twists to be accounted. It is possible for a mid-level class to consume an inherited polymorphic event in a state transition. In that case, the event is no longer available to any subclasses of the consuming class. We want to detect if a user mistakenly used an event by the same name in a subclass state model. Second, inherited polymorphic events may not be explicitly consumed in a leaf subclass. In this case, the event is subject to the default transition rule for the state model. However, we want to warn the user of such events as it may indicate a problem in specifying the state model (*e.g.* a polymorphic event was added to a superclass and not accounted for in the subclasses).

The `PropagatePolyEvents` procedure below accomplishes this reclassification operation. By doing it this way, we do not burden the user with all the subtle characteristics of polymorphic events. We need only specify which events are polymorphic and which events cause a state model transition and then we can deduce the intent to inherit down the generalization hierarchy. Of course, what is unburdened from the user will be placed upon the code and in this case there are some subtle twists and turns.

The `PropagatePolyEvents` procedure takes a singular instance reference to a **Superclass** and tracks the polymorphic event inheritance down the hierarchy.

```
<<config commands>>=
proc PropagatePolyEvents {super} {
    variable errcount
    # Starting at the superclass, find all the subclasses along the
    # generalization.
    set subs [instop $super findSubclasses]

    # Find the events that are deferred by the superclass. These could be
    # either polymorphic events defined in the superclass or inherited events
    # from another generalization.
```

```
    set defrdevents [findRelated $super R86 R80]

    # Set up some variables with the superclass attribute values.
    assignAttribute $super {Domain supDomain} {Class supClass}\
        {Relationship supRelationship}

    # Iterate over each subclass of the generalization.
    forAllRefs sub $subs {
        assignAttribute $sub {Domain subDomain} {Class subClass}\
            {Relationship subRelationship} {Role subRole}

        # We need to know if this subclass is also a superclass for another
        # generalization. We find that out by querying the ClassRole for all
        # the other relationships the subclass participates in and then
        # filtering those where it serves as a Superclass.
        set multigens [Superclass findWhere {$Domain eq $subDomain &&\
                $Class eq $subClass && $Relationship ne $subRelationship}]

        # Iterate over the deferred events
        forAllRefs defrdevent $defrdevents {
            # We need the event name and argument signature
            assignAttribute $defrdevent {Event event}
            set evtsig [findRelated $defrdevent R69]
            set psigid [expr {[isNotEmptyRef $evtsig] ?\
                    [readAttribute $evtsig PSigID] : {}}]

            # Check if the event already exists
            set evt [Event findById Domain $subDomain Model $subClass\
                    Event $event]
            if {[isEmptyRef $evt]} {
                <<PropagatePolyEvents: inherit missing deferred event>>
            } else {
                <<PropagatePolyEvents: resolve found deferred event>>
            }
        }
        # Recursively descend any generalization hierarchies repeating the
        # process for them.
        forAllRefs multigen $multigens {
            PropagatePolyEvents $multigen
        }
    }
}
```

If an event has been deferred to a subclass, then we need to know if a given subclass is a leaf subclass in the generalization hierarchy or if it is part of a repeated generalization. A leaf subclass has no generalizations for which it is the superclass. This tells us if we must consume any inherited events or if they will be allowed to be inherited further down the hierarchy. Subclasses that are part of a repeated generation will have instances of **Superclass** to show their role in that generalization.

```
<<PropagatePolyEvents: inherit missing deferred event>>=
if {[isEmptyRef $multigens]} {
    <<PropagatePolyEvents: leaf missing deferred event>>
} else {
    <<PropagatePolyEvents: non-leaf missing deferred event>>
}
```

If an event has been deferred to a leaf subclass and there is no corresponding defined event for the leaf subclass, then it must be that the event was never found in a transition statement or declared in the subclass. We create a new Mapped Event for it, but also issue a warning.

From the Events subsystem of the platform model, we can see that creating a Mapped Event requires creating a number of other related class instances.

```
<<PropagatePolyEvents: leaf missing deferred event>>=
Event create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    Number -1
TransitioningEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event
MappedEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    ParentModel $supClass
NonLocalEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    Relationship $subRelationship\
    Role $subRole
if {$psigid ne {}} {
    EventSignature create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        PSigID $psigid
}

<<PropagatePolyEvents: warn of missing deferred event>>
```

When we warn about the fact that a deferred event was not explicitly consumed in a state model, we look up the default transition so that we can say what will happen if the event is ever dispatched to the subclass. The analyst will then have to decide if that is acceptable behavior.

```
<<PropagatePolyEvents: warn of missing deferred event>>=
set trule [findRelated $sub R40 R41 ~R51 R50 R59]
# Guard against there being no defined state model.
if {[isNotEmptyRef $trule]} {
    switch -exact -- [readAttribute $trule Name] {
        IG {
            set phrase "ignoring the event"
        }
        CH {
            set phrase "a system error"
        }
        default {
            set phrase "unknown behavior"
        }
    }

    log::warn "In domain, \"$subDomain\", class, \"$subClass\", event,\
        \"$event\" is polymorphic across, \"$subRelationship\",\
        and was not consumed in the state model: signalling $event to\
        $subClass or its related superclasses will result in $phrase"
}
```

When an event is deferred to a subclass that is subject to repeated generalization, then the event is inherited down the generalization hierarchy. Again, creating a Deferred Event requires creating a number of other class instances.

```
<<PropagatePolyEvents: non-leaf missing deferred event>>=
```

```
Event create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    Number -1
DeferredEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event
InheritedEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event
NonLocalEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    Relationship $subRelationship\
    Role $subRole
if {$psigid ne {}} {
    EventSignature create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        PSigID $psigid
}

# Note that we need to provide a Deferral Path for all the generalizations that
# may stem from this subclass.  So we iterate over all the superclass
# relationships this subclass participates in.
forAllRefs multigen $multigens {
    DeferralPath create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        Relationship [readAttribute $multigen Relationship]\
        Role target
}
```

If the deferred event is found to already have been defined, then there are several cases that must be dealt with.

1. The event could be a Local Event that was defined as part of a State Model.
2. The event could be inherited from a superclass.
3. The event could be polymorphic. This would be the case if a user redefined an inherited polymorphic event in one of the subclasses of the generalization. That is not necessary to do (that is what this code is actually doing), but neither is it wrong.

We handle these cases, starting with the Local Event.

```
<<PropagatePolyEvents: resolve found deferred event>>=
set local [findRelated $evt {~R80 TransitioningEvent} {~R82 LocalEvent}]
set localsig [findRelated $evt {R69 EventSignature}]
if {[isNotEmptyRef $local]} {
    <<PropagatePolyEvents: migrate local event to mapped event>>
} else {
    <<PropagatePolyEvents: migrate inherited event to mapped event>>
}
```

Migrating a Local Event to a Mapped Event implies deleting the Local Event and creating a new instance of Mapped Event. **R83** implies that we need to create a new instance of NonLocal Event.

One further concern here is whether this subclass is a leaf subclass. If it is, then this is just the normal consumption of an inherited event as a transition in the state model of the subclass. However, if the subclass is subject to repeated generalization, then the event may no longer be inherited by the lower level generalizations. It is an easy mistake for a user to make, so we do some checking.

```
<<PropagatePolyEvents: migrate local event to mapped event>>=
delete $local
MappedEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    ParentModel $supClass
NonLocalEvent create\
    Domain $subDomain\
    Model $subClass\
    Event $event\
    Relationship $subRelationship\
    Role $subRole
if {[isNotEmptyRef $localsig]} {
    delete $localsig
    EventSignature create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        PSigID $psigid
}

<<PropagatePolyEvents: check for non-leaf consumption>>
```

We deem it an error to consume an event at a mid-level subclass and then have lower level subclasses also handle the event as if it were inherited to them. It doesn't make any sense. The test fundamentally depends upon whether there are any superclass roles played by this subclass. We have already computed that and need to test it here. The added complexity is that we need to find all the subclasses down all the generalizations that may stem from this subclass. This is effectively the transitive closure of the generalization hierarchy.

```
<<PropagatePolyEvents: check for non-leaf consumption>>=
if {[isNotEmptyRef $multigens]} {
    <<PropagatePolyEvents: compute subclasses using event>>

    if {[llength $usingevent] != 0} {
        log::error "In domain, \"$subDomain\", class, \"$subClass\",\
            event, \"$event\", is inherited across, \"$subRelationship\"\
            and is consumed in the $subClass state model:\
            yet class(es), \"[join $usingevent {, }]\", assume \"$event\"\
            is deferred to them"
        incr errcount
    }
}
```

To compute the entire set of subclasses that may be contained in generalizations rooted at the current subclass, we will build a binary relation that can be used to compute the transitive closure. We define a helper procedure that can be called recursively to descend the generalization hierarchy. The strategy is to build a relation that is the Cartesian product of a relation that holds the names of superclasses and one that holds the names of the subclasses on the next level. Then, we find superclasses that are immediately related to the subclasses and recursively compute the binary relation again. By taking the union of this binary relation at each level in the generalization, we have the entire hierarchy and can pass that off to compute the transitive closure.

```
<<helper commands>>=
proc findSubclassesOf {supers} {
    if {[isEmptyRef $supers]} {# ❶
        return [relation create {Super string Sub string}]
    }
```

```
    set supnames [pipe {
        deRef $supers |
        relation project ~ Class |
        relation rename ~ Class Super
    }] ; # ❷
    set subclasses [instop $supers findSubclasses] ; # ❸
    set subnames [pipe {
        deRef $subclasses |
        relation project ~ Class |
        relation rename ~ Class Sub
    }] ; # ❹

    set uses [relation times $supnames $subnames]

    set nextsupers [pipe {
        deRef $subclasses |
        relation semijoin ~ [deRef [ClassRole findAll]]\
            -using {Domain Domain Class Class} |
        relation semijoin ~ [deRef [Superclass findAll]] |
        ::rosea::Helpers::ToRef ::micca::Superclass ~
    }] ; # ❺

    return [relation union $uses [findSubclassesOf $nextsupers]] ; # ❻
}
```

❶ Stop if there are no further generalizations.

❷ Superclass names in a relation with one attribute, `Super`.

❸ Find the immediate subclasses.

❹ Subclass names in a relation with one attribute, `Sub`.

❺ Step down the hierarchy to those superclasses that have the same name as the subclasses.

❻ Recursively descend to the next level and union the results from there.

Given a relation value that has the Superclasses and Subclasses at each level in a generalization hierarchy, we can compute the transitive closure of this relation. The `relation tclose` command computes all possible paths and we we restrict the result to being just paths from our current subclass.

```
<<PropagatePolyEvents: compute subclasses using event>>=
set subsusing [pipe {
    findSubclassesOf $multigens|
    relation tclose ~ |
    relation restrictwith ~ {$Super eq $subClass} |
    relation project ~ Sub |
    relation list ~ Sub
}] ; # ❶
set usingevent [pipe {
    Event findWhere {$Domain eq $subDomain && $Model in $subsusing &&\
            $Event == $event} |
    deRef ~ |
    relation project ~ Model |
    relation list ~ Model
}] ; # ❷
```

❶ We are only interested in the names of the subclasses that are ultimately related to the current subclass. That is sufficient for the next query.

❷ We get the names of the subclasses that use the event by querying the Events to see which ones have Model attributes present in our list of using subtypes.

If the deferred event was not local, then two other possiblities exist. First, it could be inherited. If so, then for leaf subclasses we want to migrate the event to be a Mapped Event. If we aren't at a leaf, then the event passed further down. The second possibility is that the event is classified as a Polymorphic Event. This can happen when a user redeclares an event as Polymorphic at a mid-level subclass. It is not necessary to do but it's not wrong.

```
<<PropagatePolyEvents: migrate inherited event to mapped event>>=
set inherit [findRelated $evt {~R80 DeferredEvent} {~R81 InheritedEvent}]
set inheritsig [findRelated $evt {R69 EventSignature}]
if {[isNotEmptyRef $inherit]} {
    <<PropagatePolyEvents: create mapped event at leaf>>
} else {
    <<PropagatePolyEvents: handle superfluous polymorphic event>>
}
```

If we have found an inherited event at a leaf subclass, we want to migrate it to be a mapped event.

```
<<PropagatePolyEvents: create mapped event at leaf>>
if {[isEmptyRef $multigens]} {
    delete $inherit
    MappedEvent create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        ParentModel $supClass
    if {[isNotEmptyRef $inheritsig]} {
        delete $inheritsig
        EventSignature create\
            Domain $subDomain\
            Model $subClass\
            Event $event\
            PSigID $psigid
    }
}
```

If we find a superfluous polymorphic event declaration, we patch things up by simply migrating it to be an Inherited Event as it would have been if the user had not inserted the extra polymorphic statement in the configuration. Also note, that the case where we define a polymorphic event in a leaf subclass does not arise though this code path. Because a leaf subclass never plays a Superclass role, then no Deferral Path instance will be created that is associated with the Polymorphic Event defined by the leaf subclass. In other words, the error is detected because referential integrity checks will fail.

```
<<PropagatePolyEvents: handle superfluous polymorphic event>>=
set poly [findRelated $evt {~R80 DeferredEvent} {~R81 PolymorphicEvent}]
set polysig [findRelated $evt {R69 EventSignature}]
if {[isNotEmptyRef $poly] && [isNotEmptyRef $multigens]} {
    delete $poly
    InheritedEvent create\
        Domain $subDomain\
        Model $subClass\
        Event $event
    NonLocalEvent create\
        Domain $subDomain\
        Model $subClass\
        Event $event\
        Relationship $subRelationship\
        Role $subRole
    if {[isNotEmptyRef $polysig]} {
        delete $polysig
        EventSignature create\
```

```
            Domain $subDomain\
            Model $subClass\
            Event $event\
            PSigID $psigid
    }
}
```

Class Operation

A class operation is a body of code that associated to a particular class. Class operations do not arise at the model level, but when translating they are often useful. For example, if you wish to implement some search or sort algorithm on the instances of a class that could be defined as a class operation. Also operations that summarize information across multiple class instances are often implemented easiestly using a class operation.

`classop` *rettype name parameters body*

`rettype`
A "C" typename that gives the type of the value returned from the class operation.

`name`
The name of the class operation. `Name` must be a valid "C" identifier.

`parameters`
A list of parameter name / parameter type pairs giving the names of the domain operation parameters and their corresponding "C" data type names.

`body`
A string containing the code that is executed when the class operation is invoked. This string is presumed to be "C" code optionally interspersed with architecture macros.

Implementation

```
<<class config commands>>=
proc classop {rettype name parameters body} {
    DefineOperation false $rettype $name $parameters $body
}
```

Instance Operation

An instance operation is a body of code executed on a particular class instance.

`instop` *rettype name parameters body*

`rettype`
A "C" typename that gives the type of the value returned from the instance operation.

`name`
The name of the instance operation. `Name` must be a valid "C" identifier.

`parameters`
A list of parameter name / parameter type pairs giving the names of the domain operation parameters and their corresponding "C" data type names.

`body`
A string containing the code that is executed when the instance operation is invoked. This string is presumed to be "C" code optionally interspersed with architecture macros. The code in *body* may refer to a variable named `self` which holds a reference to the class instance on which the operation was invoked.

Implementation

```
<<class config commands>>=
proc instop {rettype name parameters body} {
    DefineOperation true $rettype $name $parameters $body
}
```

Define Operation

Defining class and instance operations is easily factored into common code.

Implementation

```
<<class config commands>>=
proc DefineOperation {isinst rettype name parameters body} {
    upvar #0\
            [namespace parent]::DomainName DomainName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno
    variable ClassName

    set entity "[expr {$isinst ? "instance" : "class"}] operation"
    CheckDuplicate Operation $entity Domain $DomainName Class $ClassName\
            Name $name
    Operation create\
        Domain $DomainName\
        Class $ClassName\
        Name $name\
        Body $body\
        ReturnDataType $rettype\
        IsInstance $isinst\
        File $configfile\
        Line $configlineno

    set paramtuple [dict create Domain $DomainName Class $ClassName\
            Operation $name]

    if {$isinst} {
```

```
        dict set paramtuple Name self
        dict set paramtuple DataType "struct $ClassName *const"
        dict set paramtuple Number\
                [GenNumber $DomainName OperationParameter\
                [list $DomainName $ClassName $name]]
        OperationParameter create {*}$paramtuple
    }

    dict for {paramname paramtype} $parameters {
        dict set paramtuple Name $paramname
        dict set paramtuple DataType $paramtype
        dict set paramtuple Number\
                [GenNumber $DomainName OperationParameter\
                [list $DomainName $ClassName $name]]
        OperationParameter create {*}$paramtuple
    }

    return
}
```

Constructor

A constructor for a class is another concept that is occasionally useful in an implementation but not needed at the model level. When we create class instances, we will insist that a value for every attribute be supplied. This will insure that there are no uninitialized attributes. Sometimes, we will find it convenient to have attribute data types that are user defined and encapsulated. It may require a constructor to initialize such encapsulated data types. Note that constructors do not take arguments and so are not particularly useful for setting attribute values.

`constructor` *body*

body
A string containing the code that is executed when the class operation is invoked. This string is presumed to be "C" code optionally interspersed with architecture macros.

The `constructor` command defines a *body* of code that is executed when an instance of a class is created.

Implementation

```
<<class config commands>>=
proc constructor {body} {
    upvar #0\
            [namespace parent]::DomainName DomainName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno
    variable ClassName

    CheckDuplicate Constructor constructor Domain $DomainName Class $ClassName
    Constructor create\
        Domain $DomainName\
        Class $ClassName\
        Body $body\
        File $configfile\
        Line $configlineno
}
```

Destructor

By analogy to a constructor, a class may have a destructor that is called when an instance is deleted.

`destructor` *body*

body
A string containing the code that is executed when the class operation is invoked. This string is presumed to be "C" code optionally interspersed with architecture macros.

The `destructor` command defines a *body* of code that is executed when an instance of a class is deleted.

Implementation

```
<<class config commands>>=
proc destructor {body} {
    upvar #0\
            [namespace parent]::DomainName DomainName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno
    variable ClassName

    CheckDuplicate Destructor destructor Domain $DomainName Class $ClassName
    Destructor create\
        Domain $DomainName\
        Class $ClassName\
        Body $body\
        File $configfile\
        Line $configlineno
}
```

Chapter 16

Defining Class State Models

State models in XUML are used to specify the sequence of computations associated with the life cycle of class instances. All instances of a class have the same behavior, yet each instance has it own notion of current state and so may progress through its life cycle independently of any other instance.

Traditionally, Moore type state models are used in XUML to define the life cycle behavior. The other alternative is a Mealy type state model. They are mathematically equivalent in the sense that any problem that can be solved by one formulation can also be solved by the other. Individuals have their preferences over which formulation is better and we will not indulge in that discussion here. We only point out that hierarchical state models are *not* supported in the this translation scheme. Hierarchical state models are an unnecessary abomination.

As usual, we need a namespace in which to hold the state model definition commands.

```
<<state model config namespace layout>>=
namespace eval StateModelDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<state model config commands>>
}
```

Statemodel

The `statemodel` command is used to define a state model for a class or an assigner. Both use the same command and support the same commands in the script body for defining states, events and transitions.

`statemodel` *body*

body
A Tcl script that is evaluated in a context to allow the definition of the properties of the class state model.

Implementation

The implementation of the `statemodel` command follows the usual pattern. We evaluate `body` in the proper context and then insert the argument data into the proper relvars that are used to collect the state model specifications.

```
<<class config commands>>=
proc statemodel {body} {
    upvar #0 [namespace parent]::DomainName DomainName
    variable ClassName

    namespace upvar StateModelDef\
        InitialState InitialState\
        DefaultTrans DefaultTrans\
        Finals Finals

    set InitialState {}
    set DefaultTrans {}
    set Finals [list]

    ConfigEvaluate [namespace current]::StateModelDef $body

    if {$DefaultTrans eq {}} {
        set DefaultTrans CH
    }

    CheckDuplicate StateModel {state model} Domain $DomainName Model $ClassName
    StateModel create\
        Domain          $DomainName\
        Model           $ClassName\
        InitialState    $InitialState\
        DefaultTrans    $DefaultTrans
    InstanceStateModel create\
        Domain          $DomainName\
        Class           $ClassName

    # Mark any final states that were defined.
    set fstates [State findWhere {$Domain eq $DomainName &&\
            $Model eq $ClassName && $Name in $Finals}]
    forAllRefs fstate $fstates {
        updateAttribute $fstate IsFinal true
        struct::set subtract Finals [readAttribute $fstate Name]
    }
    if {![struct::set empty $Finals]} {
        tailcall DeclError BAD_FINAL [join $Finals {, }]
    }

    # It is an error to define an outgoing transition for a final state.
    set finalouts [pipe {
        deRef $fstates |
        relation semijoin ~ $::micca::StateTransition\
            -using {Domain Domain Model Model Name State} |
        relation list ~ State
    }]
    if {[llength $finalouts] != 0} {
        tailcall DeclError FINAL_OUTTRANS [join $finalouts {, }]
    }

    # Check for isolated states. These are states with no inbound or
    # outbound transitions and which are not the default initial state.
    # Such states are unreachable unless the instance is explicitly
    # created in the state.
    set states [pipe {
        State findWhere {$Domain eq $DomainName && $Model eq $ClassName} |
        deRef ~
    }]
    # puts [relformat $states states]
```

```
    set trans [pipe {
        StateTransition findWhere\
                {$Domain eq $DomainName && $Model eq $ClassName} |
        deRef ~
    }]
    # puts [relformat $trans trans]
    set initialState [pipe {
        StateModel findWhere {$Domain eq $DomainName && $Model eq $ClassName} |
        deRef ~ |
        relation project ~ InitialState |
        relation rename ~ InitialState Name
    }]
    # puts [relformat $initialState initialState]

    set noins [pipe {
        relation semiminus $trans $states\
            -using {Domain Domain Model Model NewState Name} |
        relation project ~ Name
    }]
    # puts [relformat $noins noins]
    set noouts [pipe {
        relation semiminus $trans $states\
            -using {Domain Domain Model Model State Name} |
        relation project ~ Name
    }]
    # puts [relformat $noouts noouts]
    set isolated [pipe {
        relation intersect $noins $noouts |
        relation minus ~ $initialState |
        relation list ~
    }]
    if {[llength $isolated] != 0} {
        log::warn "In domain, \"$DomainName\", class, \"$ClassName\",\
            state(s), \"[join $isolated {, }]\", have no inbound or outbound\
            transitions and is(are) not the default initial state; the state(s)\
            are unreachable unless an instance is explicitly created in\
            the state"
    }
}
```

```
<<error code formats>>=
FINAL_OUTTRANS      {state, "%s", are final states, but have outgoing\
                     transitions defined for them}
BAD_FINAL           {states, "%s", are defined as final states,\
                     but do not exist}
```

State

The `state` command defines code that is executed when the state is entered and any parameters that are passed in from the event that caused the transition.

`state` *name parameters body*

name
The name of the state. Name must not be the empty string or one of the reserved names of **@**, **CH** or **IG**.

parameters
A list of parameter name / parameter type pairs giving the names of the domain operation parameters and their corresponding "C" data type names. A list of the formal parameters of the state. Parameters are specified in the same manner as for the `::proc` command.

body
A string containing the code that is executed when the state is entered. This string is presumed to be "C" code optionally interspersed with architecture macros.

Implementation

```
<<state model config commands>>=
proc state {name params body} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME [list $name] state
    }
    if {$name eq "@" || [isNotEmptyRef [TransitionRule findById Name $name]]} {
        tailcall DeclError BAD_STATE_NAME $name
    }

    upvar #0\
            ::micca::@Config@::DomainDef::DomainName DomainName\
            ::micca::@Config@::DomainDef::ClassDef::ClassName ClassName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno

    variable InitialState

    if {$InitialState eq {}} {
        set InitialState $name
    }

    CheckDuplicate State state Domain $DomainName Model $ClassName Name $name
    set stateref [State create\
        Domain          $DomainName\
        Model           $ClassName\
        Name            $name\
        Activity        $body\
        File            $configfile\
        Line            $configlineno\
        IsFinal         false\
    ]
    StatePlace create\
        Domain          $DomainName\
        Model           $ClassName\
        Name            $name\
        Number          [GenNumber $DomainName StatePlace\
                            [list $DomainName $ClassName]]

    set psigid [FindParameterSignature $params] ;           # ❶
    if {$psigid ne {}} {
        # Set up the State Signature for the newly created state
        StateSignature create\
```

```
            Domain          $DomainName\
            Model           $ClassName\
            State           $name\
            PSigID          $psigid
        #puts [relformat $::micca::StateSignature StateSignature]

if 0 {
        # Check if any State Transitions have been defined where
        # the newly created state is the New State.
        # Insert Action Signatures for those transitions.
        set asig [findRelated $stateref R78 R76]
        #puts [relformat [deRef $asig] asig]
        set actsig [pipe {
            findRelated $stateref ~R72 |
            deRef % |
            relation project % Domain Model NewState Event |
            relation rename % NewState State |
            relation join % [deRef $asig]
        } {} |%]

        #puts [relformat $actsig actsig]
        relvar insert ::micca::ActionSignature {*}[relation body $actsig]
}

        # Check if any Event which causes a State Transitions to this state
        # is lacking an Event Signature. If so, give it the Event Signature
        # of the newly created state.
        set events_needing_sig [pipe {
            findRelated $stateref ~R72 R71 R70 R80 |
            deRef % |
            relation semiminus $::micca::EventSignature %
        } {} |%]
        #puts [relformat $events_needing_sig events_needing_sig]

        set event_sigs [pipe {
            relation project $events_needing_sig Domain Model Event |
            relation extend ~ es_tup PSigID string {$psigid}
        }]
        #puts [relformat $event_sigs event_sigs]
        relvar insert ::micca::EventSignature {*}[relation body $event_sigs]
    }
    #puts [relformat $::micca::StateTransition StateTransition]
    #puts [relformat $::micca::ActionSignature ActionSignature]

    return
}
```

❶ If the state has parameters, then we find / create the signature of the state. This signature is then used to set up the Action Signature for those state transitions where this state is the destination.

```
<<error code formats>>=
BAD_STATE_NAME    {"%s" is not a valid state name}
```

State Parameters

In XUML, a state may define a set of parameters that are passed to the activity of the state when the activity is executed. Most of the time, states do not define any parameters and so there is nothing else to understand about how the parameter passing works.

If a state does define parameters, then it is a corollary of the Moore machine formulation that any event that causes a transition into a state with parameters must carry a matching set of arguments whose values were determined when the event was signalled. So if an event causes a transition to different states, then each state must have the same parameter signature. Conversely, if different event cause a transition into the same state, the events must have the same parameter signature.

What determines the matching of event and state parameters is the position and data type of the parameters. Specifically, the parameter names don't matter when we are trying to decide if the parameters of an event and the parameters of the state into which the event causes a transition match. For example an event may view its parameters as `int size, float incr` and a corresponding state may view its parameters as `int count, float prec`. In this case the signature of position and type is the same despite the difference in the parameter naming. Seen another way, states and events may have a different semantic view of their parameters as given by the parameter names, but have matching syntax of parameter position and data type.

In our attempt so minimize the amount of specification of information in the configuration DSL, the declaration of events is optional. Here are the rules:

1. An event and its parameters may be declared with the `event` command and that is the signature it is given.
2. A polymorphic event that has parameters must declare them as part of the `polymorphic` command.
3. If there is no event declaration, the event name and signature is assumed from the state into which the event causes a transition. Its parameter naming, position and type information is assumed from that of the state.

The rules outlined in this discussion are not the only possiblity. It is the one that has been chosen for `micca` as it matches the function invocation rules of "C" while still allowing flexiblity in parameter name specification. We deem this approach more consistent with "C" implementation concepts and more flexible than one that is based strictly on parameter names and the need to declare event parameter signatures.

The helper procedures in this section do the work of setting up parameter signatures and the underlying argument signatures. Here is where signature equality is determined.

FindParameterSignature

The `FindParameterSignature` procedure searches for an existing parameter signature that matches the parameter list argument and returns the parameter signature ID. If none exists that matches, it creates a new one and returns the ID of the new one.

Implementation

The difficult part in determining the equality of two parameter signatures is that we want to find that Parameter Signature where all the related Parameter instances have the correct values for the parameter name, position and data type. The information about position and data type is held in the Argument Signature and we will cover that next.

Again this is a situation where some relational algebra can come to the rescue. In this case we will create a relation value that holds the parameter name, position and data type. We then use the `relation group` command to create a relation valued attribute that groups together the parameter name, position and data type for the existing instances of Parameter. Equality can then be determined by the equality of two relations.

```
<<helper commands>>=
proc FindParameterSignature {params} {
    if {[dict size $params] == 0} {
        return
    }

    set asigid [FindArgumentSignature $params] ; # ❶
    if {$asigid eq {}} {
        error "panic: did not create an argument signature for \"$params\""
    }

    set poscounter -1
```

```
    set parampos [relation create {Name string Position int ASigID string}]
    dict for {pname ptype} $params {
        set parampos [relation insert $parampos [list\
            Name $pname\
            Position [incr poscounter]\
            ASigID $asigid\
        ]]
    } ; # ❷
    set psigref [pipe {
        Parameter findAll | deRef ~ |
        relation group ~ ParamNames Name Position ASigID |
        relation restrictwith ~ {[relation is $ParamNames == $parampos]} |
        relation semijoin ~ [deRef [ParameterSignature findAll]]
        ::rosea::Helpers::ToRef ::micca::ParameterSignature ~
    }] ; # ❸

    if {[isEmptyRef $psigref]} {# ❹
        namespace upvar ::micca::@Config@::DomainDef DomainName DomainName

        set psigid psig[GenNumber $DomainName ParameterSignature\
                [list $DomainName]]
        ParameterSignature create\
            Domain $DomainName\
            PSigID $psigid\
            ASigID $asigid
        set poscounter -1
        dict for {pname ptype} $params {
            Parameter create\
                Domain $DomainName\
                PSigID $psigid\
                Name $pname\
                Position [incr poscounter]\
                ASigID $asigid
        }
    } else {
        set psigid [readAttribute $psigref PSigID]
    }

    return $psigid
}
```

❶ The argument signature encodes the position and data type information.

❷ The heading of this relation matches the attribute names in the Parameter class.

❸ Step by step:

1. Get all the Parameter instances as a relation value.
2. Group together the Name, Position and ASigID attributes. This leaves us with a relation value with the attributes Domain, PSigID and ParamNames, where ParamName is relation valued.
3. Find the one matching the information in the parameters argument.
4. Find the corresponding Parameter Signature.
5. Turn it into an instance reference for later use.

❹ If we fail to find a match, then create the Parameter Signature and Parameter instances. Otherwise, we can return the PSigID value.

FindArgumentSignature

To separate parameter names from their position and data type, we introduce the concept of an Argument Signature. Here equality is based solely on positions` and data types of the arguments. This concept is useful when considering how argument values carried by an event are transferred to the formal parameter of a state activity. In that situation, the names don't matter in "C", only order and data type.

Implementation

The implementation here follows the same strategy used for `FindParameterSignature`. We create a relation value that contains the Position and DataType information that corresponds to the parameters. The grouping and comparing is similar.

```
<<helper commands>>=
proc FindArgumentSignature {params} {
    if {[llength $params] == 0} {
        return
    }

    set poscounter -1
    set parampos [dict create]
    dict for {pname ptype} $params {
        dict set parampos [incr poscounter] $ptype
    }
    set argpos [relation fromdict $parampos Position int DataType string] ; # ❶
    set asigref [pipe {
        Argument findAll | deRef ~ |
        relation group ~ ArgPositions Position DataType |
        relation restrictwith ~ {[relation is $ArgPositions == $argpos]} |
        relation semijoin ~ [deRef [ArgumentSignature findAll]] |
        ::rosea::Helpers::ToRef ::micca::ArgumentSignature ~
    }]

    if {[isEmptyRef $asigref]} {
        namespace upvar ::micca::@Config@::DomainDef DomainName DomainName

        set asigid asig[GenNumber $DomainName ArgumentSignature\
                [list $DomainName]]
        ArgumentSignature create\
            Domain  $DomainName\
            ASigID   $asigid
        set poscounter -1
        dict for {pname ptype} $params {
            Argument create\
                Domain      $DomainName\
                ASigID      $asigid\
                Position    [incr poscounter]\
                DataType    $ptype
        }
    } else {
        set asigid [readAttribute $asigref ASigID]
    }

    return $asigid
}
```

❶ Since there are only two attributes in the relation value of position and data type information, we can use a Tcl `dict` to accumulate the information and then TclRAL has convenience methods to move between relation values and Tcl dictionaries.

Event

Event definitions are optional, but in the case where an event has parameters, it may be clearer to define the parameter set by declaring the event.

`event` *name* ?*argname argtype* . . . ?

name
The name of the event.

argname argtype ...
The argument signature of the event. Arguments must be given in name / type pairs. Argument names must be "C" identifiers and argument types must be "C" type names.

The `event` command defines *name* as being an event with optional event arguments given by *argname* / *argtype* pairs.

Implementation

The implementation does the usual platform class instance creation to store away the data in the command arguments. However, since events may be defined simply by their appearance in a `transition` command, we need to check if we have already seen the event. If we have, then we want to update the parameter signature to match what was defined in this command. Further, if this event causes any state transition where the argument signature has not already been determined, then we need to update the StateTransition instances to reflect the argument signature implied by this event. This processing is done here to make the order of event and transition specification arbitrary.

```
<<state model config commands>>=
proc event {name args} {
    namespace upvar ::micca::@Config@::DomainDef DomainName DomainName
    namespace upvar ::micca::@Config@::DomainDef::ClassDef ClassName ClassName

    set eventtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        Event   $name\
    ]
    set evtref [Event findById {*}$eventtuple]
    if {[isEmptyRef $evtref]} {
        set evtref [Event create {*}$eventtuple Number -1]
        TransitioningEvent create {*}$eventtuple
        LocalEvent create {*}$eventtuple
    }

    set evtsig [findRelated $evtref {R69 EventSignature}]
    delete $evtsig
    set psigid [FindParameterSignature $args] ;             # ❶
    if {$psigid ne {}} {
        EventSignature create {*}$eventtuple PSigID $psigid
        #puts [relformat $::micca::EventSignature EventSignature]
    }
}
```

❶ If we have a parameter signature, then create an instance of Event Signature . It is necessary to delete any existing Event Signature which might have been assigned to the event in a `transition` command. An explicit `event` command takes precedence.

Transition

For Moore type state models, one can conceive of the mapping of states to events as a transition matrix with a row for each state and a column for each event. Each matrix cell has a value of the new state for the transition. The `transition` command is used to define the values of cells in the conceptual transition matrix.

`transition` *source* – *event* –> *target*

`source`
The name of a state in the state model being defined or the special reserved name, `@`.

`event`
The name of an event that causes the transition.

`target`
The name of a state in the state model being defined or one of the special non-transitioning states, `IG` or `CH`.

The `transition` command defines the transition that is to happen when *event* is dispatched to a state machine when the current state is *source*, causing the new state to be *target*.

Implementation

The implementation of the `transition` command consists mainly of inserting tuples into the event classes using the data from the command arguments. Note however, that all the events defined by invoking `transition` are deemed to be a **LocalEvent**. This certainly may not be true for classes that are leaf subclass of a generalization hierarchy. When the domain configuration is completed we will propagate the polymorphic events down the generalization hierarchies and in that process recategorize any inherited events properly.

One other minor concern is dealing with the initial pseudo-state, @. This state is where an instance resides if it has been created asynchronously. One may not define the `@` state in a `state` command (it cannot have an activity) and the only valid place it can appear is as the `source` state in a `transition` command.

We also have to handle the argument signature of the transition. If the new state or the event already has an argument signature we will use it here when the State Transition is created.

```
<<state model config commands>>=
proc transition {source - event -> target} {
    if {$event eq {}} {
        tailcall DeclError BAD_NAME [list $event] event
    }
    if {[isNotEmptyRef [TransitionRule findById Name $source]]} {# ❶
        tailcall DeclError BAD_STATE_NAME $name
    }
    namespace upvar ::micca::@Config@::DomainDef DomainName DomainName
    namespace upvar ::micca::@Config@::DomainDef::ClassDef ClassName ClassName

    if {$source eq "@"} {
        if {[isNotEmptyRef [TransitionRule findById Name $target]]} {
            tailcall DeclError BAD_CREATION_TARGET $target
        }
        set cstuple [list\
            Domain          $DomainName\
            Model           $ClassName\
            Name            @\
        ]
        # We have to conditionally create the CreationState instance since we
        # can have multiple transition commands that use @ as the source state.
        if {[isEmptyRef [CreationState findById {*}$cstuple]]} {
```

```
            CreationState create {*}$cstuple
        }
        if {[isEmptyRef [StatePlace findById {*}$cstuple]]} {
            StatePlace create {*}$cstuple\
                Number [GenNumber $DomainName StatePlace\
                        [list $DomainName $ClassName]]
        }
    }

    set transtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        State   $source\
        Event   $event\
    ]
    CheckDuplicate TransitionPlace transition {*}$transtuple
    TransitionPlace create {*}$transtuple

    set eventtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        Event   $event\
    ]
    set eventref [Event findById {*}$eventtuple]
    if {[isEmptyRef $eventref]} {
        Event create {*}$eventtuple Number -1
        TransitioningEvent create {*}$eventtuple
        LocalEvent create {*}$eventtuple

        # When we create a new event, check if the target state has
        # a Parameter Signature. If so, then use that signature for
        # the event.
        set stateref [State findWhere {$Domain eq $DomainName &&\
            $Model eq $ClassName && $Name eq $target}]
        #puts [relformat [deRef $stateref] stateref]
        if {[isNotEmptyRef $stateref]} {
            set statesig [findRelated $stateref {R78 StateSignature}]
            if {[isNotEmptyRef $statesig]} {
                EventSignature create {*}$eventtuple\
                    PSigID [readAttribute $statesig PSigID]
                #puts [relformat $::micca::EventSignature EventSignature]
            }
        }
    }

    set trref [TransitionRule findById Name $target]
    if {[isNotEmptyRef $trref]} {
        NonStateTransition create {*}$transtuple TransRule $target
    } else {
        # Create the state transition instance. If the NewState of the
        # State Transition corresponds to a State which has a State Signature,
        # then create an instance of Action Signature to match.

        set stref [StateTransition create {*}$transtuple NewState $target]
        #puts [relformat [deRef $stref] StateTransition]

if 0 {
        set statesig [pipe {
            State findById Domain $DomainName Model $ClassName Name $target |
            findRelated ~ R78
        }]
        #puts [relformat [deRef $statesig] statesig]
```

```
        if {[isNotEmptyRef $statesig]} {
            ActionSignature create {*}$eventtuple\
                State $target ASigID [readAttribute $statesig ASigID]
            #puts [relformat $::micca::ActionSignature ActionSignature]
        }
}
    }

    return
}
```

❶ Ignoring a transition out of the pseudo-initial state is not allowed and any transition out of the pseudo-initial state not mentioned in a `transition` command will be set to CH anyway. So just don't let transition out of the pseudo-initial state go anywhere except a real state.

```
<<error code formats>>=
BAD_CREATION_TARGET {the target of a creation event must be a state,\
        got "%s"}
```

Initialstate

By default, instances created from classes that have a state model are placed in the first state that was defined for the model. The `initialstate` command is used to specify explicitly the inital state for newly created instances.

`initialstate` *name*

name
The name of at state in the state model being defined. Instances of the class that are created synchronously using the `create` class command will be placed in this state.

The `initialstate` command sets the default state that an instance will be created in. If no `initialstate` command is encountered in the definition of a state model, then the first state defined to the state model is used as the initial state.

Implementation

```
<<state model config commands>>=
proc initialstate {name} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME [list $name] initialstate
    }
    # The initial state may not be the pseudo-initial state name
    # nor any of the other transition rule pseudo states (i.e. IG or CH).
    if {$name eq "@" || [isNotEmptyRef [TransitionRule findById Name $name]]} {
        tailcall DeclError BAD_STATE_NAME $name
    }
    variable InitialState $name
    return
}
```

Defaulttrans

It is customary to write `transition` commands only for outgoing transition that appear on the state model graphic. For any entries in transition matrix that are not set by a `transition` command, a default value is supplied. That default is either `IG` or `CH` depending upon the argument to the `defaulttrans` command. If no `defaulttrans` command is invoked when a state model is defined, then the default transition will be `CH`.

`defaulttrans` *trans*

trans
Either the string **IG** or **CH**. For all transitions not explicitly mentioned in a `transition` command, the default transition is defined as *trans*. If `defaulttrans` is not invoked during a state model definition then the default transition is **CH**.

Implementation

```
<<state model config commands>>=
proc defaulttrans {name} {
    if {[isEmptyRef [TransitionRule findById Name $name]]} {
        tailcall DeclError EXPECTED_NONTRANS_STATE $name
    }
    variable DefaultTrans $name
    return
}
```

```
<<error code formats>>=
EXPECTED_NONTRANS_STATE    {expected CH or IG, got "%s"}
```

Final

Final states are those where the class instance is deleted after the state activity is executed. This allows for asynchronous deletion of class instances.

`final` *?state ... ?*

state
The name of a state in the state model that will be marked as a final state. If a state machine transitions into a final state, the associated instance is deleted after the state activity is executed.

Implementation

```
<<state model config commands>>=
proc final {args} {
    variable Finals
    if {"@" in $args} {
        tailcall DeclError BAD_STATE_NAME @
    }
    struct::set add Finals $args
    return
}
```

Chapter 17

Defining Assigners

The assigner concept is not frequently used in XUML models but is an essential concept in modeling competitive relationships. Some associations model competitive behavior such as allocating resources. For such situations, instances of the relationships must be created and deleted, according to some protocol and in some fashion that serializes the relationship lifecycle between potentially competing classes. When two classes are competing for the same resource, the cannot simply create on their own instances of an association involving the resource. If they did, then the essential parallel behavior of the model would potentially end up with multiple allocations of a resource. Thus the association involving the resource needs a lifecycle of its own. Lifecycle behavior of associations is accomplished in XUML models by associating a state model to the association and having the state model activities insure the proper creation of association instances. Such state models are known as *assigners*.

Note that we associate the assigner with the association relationship. We do **not** require factoring the relationships into a class based association and then attach the assigner to the associator class. Many associations that have assigners are class based associations, but the assigner is *not* the state model of the associator class. Indeed, the associator class may have its own state model if it exhibits lifecycle behavior. In `micca`, the relationship may or may not be class based and if it is the associator class may or may not have a state model. The assigner state model is independent of any class and its state model. It is, after all, a state model to deal with the application semantics of forming and dissolving the relationship.

Also note that only associative relationships may have an assigner. Assigners are not meaningful for generalization relationships given the disjoint union nature of a generalization.

There is a further complication. Usually, there is a single state machine associated with the association assigner. However, some assigners have more complicated competitive protocols. In those cases the association may have multiple assigners that are partitioned by another class. The archetypal example is that of assigning department store clerks to customers. If any clerk can service any customer, then a single assigner is used to sequence properly the assignment of available clerks to waiting customers. If the store policy is that a customer can only be served by a clerk working in a particular department, then there will be as many assigners as there are departments and the identity of the department class serves to identify the assigner instances needed to insure that clerks, within the department, are not over allocated and that customers, visiting the department, are not over served by clerks from the wrong department. This is the essential concept behind a multiple assigner.

Assigners and multi-assigners fall into the XUML semantics category of not-that-common but essential to express the proper execution semantics. As we will see, the specifications required for assigners is almost the same as that for ordinary class state models.

Assigners are defined by a script specified as part of an `association` command. In this section we show the command that may be executed in that script. We will need to define a namespace in which the script can be evaluated.

```
<<assigner config namespace layout>>=
namespace eval AssignerDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<assigner config commands>>
}
```

Assigner State Model

The `statemodel` command for assigners takes the same commands as that for classes. Indeed, much of the same code is in common.

`statemodel` *body*

body
A Tcl script that is evaluated in a context to allow the definition of the properties of the assigner state model.

Implementation

The difference between assigner state model and class state models is only in a few class instances and relationships, notably, R50 and R53.

```
<<assigner config commands>>=
proc statemodel {body} {
    namespace upvar ::micca::@Config@::DomainDef DomainName DomainName
    namespace upvar ::micca::@Config@::DomainDef::ClassDef ClassName AssocName

    namespace upvar ::micca::@Config@::DomainDef::ClassDef::StateModelDef\
        InitialState InitialState\
        DefaultTrans DefaultTrans\
        Finals Finals

    set InitialState {}
    set DefaultTrans {}
    set Finals [list]

    ConfigEvaluate ::micca::@Config@::DomainDef::ClassDef::StateModelDef $body ; # ❶

    if {$DefaultTrans eq {}} {
        set DefaultTrans CH
    }
    CheckDuplicate StateModel {assigner state model}\
            Domain $DomainName Model $AssocName
    StateModel create\
        Domain          $DomainName\
        Model           $AssocName\
        InitialState    $InitialState\
        DefaultTrans    $DefaultTrans

    set assoctuple [list\
        Domain          $DomainName\
        Association     $AssocName
    ]
    AssignerStateModel create {*}$assoctuple

    foreach final $Finals {
        set sref [State findWhere Domain $DomainName Model $ClassName\
                Name $final]
        updateAttribute $sref IsFinal true
    }
}
```

❶ We evaluate the assigner state model script in the same namespace as for the class state models. Other than setting up a few variables differently, the configuration data for assigners is the same as for classes.

Identify By Class

For multiple assigners we need to specify which class serves to partition the assigners into multiple instances. Defining such a class signifies that the assigner is to be multiple rather than singular.

`identifyby` *class*

`class`
The name of a class that serves to partition the instances of a multiple assigner.

The `identifyby` command defines an assigner to be a multiple assigner and specifies *class* as the partitioning class.

Implementation

```
<<assigner config commands>>=
proc identifyby {class} {
    variable IdClassName $class
}
```

Chapter 18

Defining External Entities

External entities are a collection of operations that define the services delegated by the domain to be satisfied elsewhere. The `eentity` command takes a *script* argument which should invoke the commands we discuss in this section. Following our pattern, we define a namespace where the class body script is evaluated.

```
<<external entity config namespace layout>>=
namespace eval EEntityDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<external entity config commands>>
}
```

External Operation

The `operation` command is used in a `eentity` script defines an operation of the external entity. External operations define the invocable interface for functionality the domain has delegated to an entity outside of the domain itself.

`operation` *rettype name parameters body* ?*comment*?

`rettype`
A "C" typename that gives the type of the value returned from the external operation.

`name`
The name of the external operation. `Name` must be a valid "C" identifier.

`parameters`
A list of parameter name / parameter type pairs giving the names of the external operation parameters and their corresponding "C" data type names.

`body`
A string containing code. Normally the external operation code is *not* included in the generated output for the domain. However, the code generate can be requested to generate the external operation code. This simplifies certain test situation and makes it easier to generate an executable for the domain by resolving the external operations via the code generateor. This string is presumed to be "C" code optionally interspersed with architecture macros.

`comment`
An optional string that is passed along by the code generated to the generated header file.

Implementation

```
<<external entity config commands>>=
proc operation {rettype name parameters {body {}} {comment {}}} {
    variable EntityName
    upvar #0\
            [namespace parent]::DomainName DomainName\
            ::micca::@Config@::configfile configfile\
            ::micca::@Config@::configlineno configlineno

    CheckDuplicate ExternalOperation {external operation}\
            Domain $DomainName Entity $EntityName Name $name
    ExternalOperation create\
        Domain $DomainName\
        Entity $EntityName\
        Name $name\
        Body $body\
        ReturnDataType $rettype\
        Comment [string trim $comment]\
        File $configfile\
        Line $configlineno

    set paramtuple [dict create Domain $DomainName Entity $EntityName\
            Operation $name]
    dict for {paramname paramtype} $parameters {
        dict set paramtuple Name $paramname
        dict set paramtuple DataType $paramtype
        dict set paramtuple Number\
                [GenNumber $DomainName ExternalOperationParameter\
                [list $DomainName $name]]
        ExternalOperationParameter create {*}$paramtuple
    }

    return
}
```

Chapter 19

Populating a Domain

At this point we have covered all the commands that are used to define the configuration of a domain. We now turn our attention to specifying an initial instance population. The initial instances of a domain are those class (and assigner) instances that will exist when the domain start running.

For many systems, particularly small embedded systems, much of the class instance data can be specified as part of the initial instance population. This has a substantial benefit to the implementation. Since class data is held in "C" arrays, initial instances can be specified as initializers for those class storage arrays. The "C" language then guarantees that initializer values are in place before `main` begins execution. This usually requires much less memory space than having initialiation code to create the initial instances and having that code executed only once. So, we will specify the initial instance population in data rather than procedurally.

We will insist that there be a population for a domain before code is generated for it. In practice, development of a domain will usually entail creating multiple populations for both testing and deployment. Populations can be managed by keeping them in separate files and then assembling the domain using Tcl `source` commands.

Following our usual pattern, we define a namespace in which to evaluate the population commands.

```
<<population config namespace layout>>=
namespace eval PopDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<population config commands>>

    <<class instance config namespace layout>>
    <<assigner instance config namespace layout>>
}
```

Population

The `population` command gives values to the class components of the domain. The complete domain population must be given by the invocation of `population`.

`population` *domain script*

domain
The name of the domain to which the population applies.

script
A Tcl script that is evaluated in a context to allow the definition of initial instance values for a domain.

The `population` command is used to define initial instance values for the classes and assigners in *domain*.

Implementation

The implementation of the `population` procedure follows our general pattern. The supplied script is evaluated in a namespace where the population commands will resolve without further qualification.

```
<<config commands>>=
proc population {domain script} {
    variable errcount

    if {$errcount != 0} {
        error "cannot populate with configuration errors"
    }

    namespace upvar PopDef DomainName DomainName
    set DomainName $domain

    set popref [Population create Domain $domain]

    try {
        ral relvar transaction begin
        try {
            ConfigEvaluate [namespace current]::PopDef $script

            # At this point the population is complete and we can begin the
            # processing to validate that the population is correct.

            <<population: verify population integrity>>

            if {$errcount != 0} {
                ral relvar transaction rollback
                return
            }

            # Once we like the populated values supplied by the user we can use
            # them to set the values of the generated class components.

            <<population: set generated component values>>
            <<population: update union subclass numbering>>
        } on error {result opts} {
            ral relvar transaction rollback
            return -options $opts $result
        }
        ral relvar transaction end
    } on error {result} {
        # puts $::errorInfo
        HandleConfigError $result
        return
    }
```

```
}
```

There are two important other steps required for the initial instance population.

1. We must check that the specified population is correct and complete.
2. There are a number of class components whose values are derived from those specified in the population commands and we must calculate those values in preparation for code generation.

These two steps are described later after all the population commands have been considered.

Class

The `class` command (in the context of a population) is used to specify the initial instances of a class.

`class` *name script*

name
The name of the class.

script
A Tcl script that is evaluated in a context to allow the definition of class component values for a class.

The `class` command is used to define initial instances for a class.

Populating a class entails specifying attribute and reference values as well as setting space allocation and storage characteristics of the class. As we said before, the `micca` architecture allocates a fixed amount of space for each class. This determines the maximum number of instances that may exist simultaneously. A class population can consist of initial instances as well as those created at run time. Initial instances may be deleted at run time and their space is then available for newly created instances. To control the amount of space allocated, the `allocation` command can be used to set the additional number of instances beyond the initial instances that are allocated. The total number of simultaneous instances is then the sum of the allocation amount and the initial instances.

Classes can also be allocated to read-only memory. If there is no dynamic activity on a class, such as state models or attribute updates, then the attributes and references can be placed in constant memory. For small embedded system this can be a significant saving of read/write memory which is usually in shorter supply.

To accommodate the class population commands we need a namespace in which to evaluate the script.

```
<<class instance config namespace layout>>=
namespace eval ClassInstDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<class instance commands>>
}
```

Implementation

```
<<population config commands>>=
proc class {name script} {
    if {$name eq {}} {
        tailcall DeclError BAD_NAME $name class
```

```
    }

    variable DomainName

    namespace upvar ClassInstDef ClassName ClassName Allocation Allocation
    set ClassName $name
    set Allocation 0

    CheckExists Class class Domain $DomainName Name $ClassName

    ConfigEvaluate [namespace current]::ClassInstDef $script

    set epRef [ElementPopulation findById Domain $DomainName Element $name]
    if {[isEmptyRef $epRef]} {
        ElementPopulation create Domain $DomainName Element $name
    }

    set cpRef [ClassPopulation findById Domain $DomainName Class $name]
    if {[isEmptyRef $cpRef]} {
        ClassPopulation create Domain $DomainName Class $name\
                Allocation $Allocation
    } else {
        assignAttribute $cpRef {Allocation prevAllocation}
        updateAttribute $cpRef Allocation\
                [expr {max($Allocation, $prevAllocation)}]
    }

    return
}
```

Class Storage Allocation

`allocate` ?*size*?

`size`
The number of additional class instances beyond those defined by the initial instances for which storage is allocated.

The `allocate` command is used declare that a class population is to have *size* additional storage slots allocated to it beyond those used by the initial instance population. If no `allocate` command is invoked for a class of if the *size* argument is missing, then the class will have zero additional instances allocated.

Implementation

```
<<class instance commands>>=
proc allocate {{size 0}} {
    if {![string is integer -strict $size]} {
        tailcall DeclError EXPECTED_INT size $size
    }
    if {$size < 0} {
        tailcall DeclError EXPECTED_NONNEG size $size
    }

    variable Allocation $size

    return
```

```
}
```

```
<<error code formats>>=
EXPECTED_INT    {expected integer "%s": got "%s"}
EXPECTED_NONNEG {expected non-negative "%s": got "%s"}
```

Specifying Instance Values

Instances are populated according to their class components. Class components are of several types but primarily are either attributes or references.

Attribute values are specified using strings in "C" syntax. The attribute value will be passed along to the generated code file and must be valid "C" initializers. For example the value for an `int` attribute can be specified as 0x20 or 32 and the "C" compiler will handle either.

Reference components are more varied in how they are specified but correspond to the way that referential attributes in the analysis model refer to identifying attributes. Since fundamentally references will resolve down to object addresses, initializing references uses instance names as the place holder for the class instance address. The code generator then maps reference names to instance address expressions. Different reference types require some additional structure.

- Simple associations require a single instance name. The instance name must be one that is defined for the class to which the reference refers. It is specified using the name of the association followed by the name of the referenced instance. For example, an association defined as:

  `association R27 X 1..*--1 Y`

  would require that `X` instances specify a value for `R27`:

  `instance x1 R27 y1`

  which creates an instance named, `x1`, that references an instance of class, `Y`, named, `y1`, across the `R27` association.

- Class based associations linkages is specified in the instances of the associator class. Each associator class instance represents an instance of the association itself and refers to both participants in the association. For example, a class based association defined as:

  `association R28 -associator A X 1..*--1..* Y`

  would require that `A` instances specify a value for `R28` referring to both an `X` instance and a `Y` instance:

  `instance a1 R27 {X x1 Y y1}`

  which creates an instance named, `a1`, that references an instance of class, `X`, named `x1` and an instance of class, `Y`, named, `y1`, across the `R28` association. The order of specifying the participants does not matter.

- For class based associations that are reflexive, *i.e.* the same class serves as both the source and target of the association, we cannot use the class names to distinguish references. In this case we use the words `forward` and `backward` to denote which instances are reached by navigating the association in the forward and backward directions. For example, a reflexive class based association defined as:

  `association R29 -associator A X 0..*--0..* X`

  would require that `A` instances specify a value for `R28` referring to which instance of `X` is reached when navigating forward and which is reached navigating backward:

  `instance a1 R27 {backward x1 forward x2}`

  which creates an instance named, `a1`, that references an instance of class, `X`, named `x2` when navigating forward and an instance of class, `X`, named, `x1`, when navigating backward. The order of specifying the forward and backward participants does not matter.

- For generalization relationships, a reference to the superclass instance is specified in the subclass instance. For example, a generalization specified as:

  `generalization R30 S A B C`

would require that `A`, `B`, or `C` instances specify a value for `R30`:

```
instance c1 R30 s1
```

which creates an instance named, `c1`, that references an instance of the superclass, `S`, named, `s1`, across the `R30` generalization. The specification of subclass to superclass references is the same regardless of whether the generalization is a reference type or a union type.

The default value for an attribute is specified by the literal string, "-". It is an error to specify a default value for an attribute for which no default was defined during the domain configuration. Note there is no default value for a relationship reference. All relationship references must be specified as described above. If an attribute value needs to be set to the literal "-" string, then it must be escaped and written as, "\-".

Setting Instance Values

The `instance` command defines the class component values for an instance and gives the instance a name. It is a convenient way to specify a small number of class instances.

`instance` *name comp1 value1 comp2 value2 ...*

name
The name of the instance. Instance names must be unique within a class.

compN valueN
The remaining arguments are in the form of component name / component value pairs. All components of a class must be present except for those which have defined default values.

The `instance` command defines a class instance named, *name*, and gives the class components given by the *compN* arguments the corresponding *valueN* value.

Implementation

We want the `instance` command to set the values for every class component that can be set, *i.e.* we don't want any underdefined instances. To do that, we will determine if the component names provided are sufficient. If they are, then we can create the Class Instance and the set of Class Component Value instances. The latter fall into two types, those provided as arguments and those which are missing from the arguments but for which a default value can be deduced.

```
<<class instance commands>>=
proc instance {name args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }

    upvar #0 [namespace parent]::DomainName DomainName
    variable ClassName

    CheckPopAttrs $DomainName $ClassName [dict keys $args]

    set insttuple [list\
        Domain $DomainName\
        Class $ClassName\
        Instance $name\
    ]
    CheckDuplicate ClassInstance {class instance} {*}$insttuple
    ClassInstance create {*}$insttuple\
        Number [GenNumber $DomainName ClassInstance [list $DomainName $ClassName]]
```

```
    CreateRequiredValues $DomainName $ClassName $name $args
    CreateDefaultedValues $DomainName $ClassName $name $args
    CreateZeroInitValues $DomainName $ClassName $name $args

    return
}
```

To determine if we have the correct components as arguments, we need to find those components that must be set, *i.e.* they have no reasonable default. That set component names must be a subset of those provided as arguments.

All class attributes must be set to a value *except* attributes that have a default value or are zero initialized. Dependent attributes cannot be given an initial value.

```
<<helper commands>>=
proc CheckPopAttrs {domainName className attrNames} {
    set reqcomps [FindRequiredPopComps $domainName $className]
    set missing [struct::set difference $reqcomps $attrNames]
    if {![struct::set empty $missing]} {
        tailcall DeclError MISSING_VALUES $domainName $className\
                [join $missing {, }]
    }

    set noinitattrs [FindNoInitializeAttrs $domainName $className]
    set cantinit [struct::set intersect $noinitattrs $attrNames]
    if {![struct::set empty $cantinit]} {
        tailcall DeclError NOINIT_VALUES $domainName $className\
                [join $cantinit {, }]
    }

    set allcomps [FindAllPopComps $domainName $className]
    set extra [struct::set difference $attrNames $allcomps]
    if {![struct::set empty $extra]} {
        tailcall DeclError UNKNOWN_INIT $domainName $className\
                [join $extra {, }]
    }

    return $reqcomps
}
```

```
<<error code formats>>=
MISSING_VALUES  {for domain population, %s, class, %s,\
        values for class attribute,\
        "%s", are not provided and have no default values}
NOINIT_VALUES  {for domain population, %s, class, %s,\
        values for class attribute,\
        "%s", may not have initial values specified}
UNKNOWN_INIT  {for domain population, %s, class, %s,\
        "%s" are not attributes of the class}
```

Create a Class Component Value instance for all the provided component name / component values provided as arguments to the `instance` command.

```
<<helper commands>>=
proc CreateRequiredValues {domainName className instName namedValues} {
    set comptuple [dict create\
        Domain $domainName\
        Class $className\
        Instance $instName\
    ]
    foreach attrname [FindRequiredPopComps $domainName $className] {
        dict set comptuple Component $attrname
```

```
        set attrvalue [dict get $namedValues $attrname]
        if {$attrvalue eq "-"} {    # ❶
            set attrvalue [ResolveInitialValue $domainName $className\
                    $attrname $attrvalue]
        }
        ClassComponentValue create {*}$comptuple
        SpecifiedComponentValue create {*}$comptuple Value $attrvalue
    }
}
```

❶ If the required value is given as a dash, then we have to resolve it to its default value.

Attributes with default values may be overriden with a value, may be specified as “-”, or have their value set as specified when the attributes was created.

```
<<helper commands>>=
proc CreateDefaultedValues {domainName className instName namedValues} {
    set comptuple [dict create\
        Domain $domainName\
        Class $className\
        Instance $instName\
    ]
    foreach attrname [FindDefaultableAttrs $domainName $className] {
        dict set comptuple Component $attrname
        if {[dict exists $namedValues $attrname]} {
            set argvalue [dict get $namedValues $attrname]
        } else {
            set argvalue -
        }
        set attrvalue [ResolveInitialValue $domainName $className\
                $attrname $argvalue]
        ClassComponentValue create {*}$comptuple
        SpecifiedComponentValue create {*}$comptuple Value $attrvalue
    }
}
```

Zero initialized attributes may be left unspecified, may be specified with an explicit value or may be specified as "-". In the last case, we want to treat it as if it were unspecified, *i.e.* we want to allow the compiler to do the initialization.

```
<<helper commands>>=
proc CreateZeroInitValues {domainName className instName namedValues} {
    set comptuple [dict create\
        Domain $domainName\
        Class $className\
        Instance $instName\
    ]
    foreach attrname [FindZeroInitAttrs $domainName $className] {
        dict set comptuple Component $attrname
        if {[dict exists $namedValues $attrname] &&\
                [dict get $namedValues $attrname] ne "-"} {
            set attrvalue [dict get $namedValues $attrname]
            SpecifiedComponentValue create {*}$comptuple Value $attrvalue
        } else {
            UnspecifiedComponentValue create {*}$comptuple
        }
        ClassComponentValue create {*}$comptuple
    }
}
```

The projection of the Name attribute from the Populated Component class gives the set of all components that must be populated.

```
<<helper commands>>=
proc FindAllPopComps {domain class} {
    return [pipe {
        PopulatedComponent findWhere {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Name
    }]
}
```

The components that are required to be populated when they are created are those whose values cannot otherwise be determined. These are the Reference components and those attributes without a default value.

```
<<helper commands>>=
proc FindRequiredPopComps {domain class} {
    # The attributes having NO default values must be specified
    set defattrs [pipe {
        DefaultValue findWhere {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Attribute
    }]
    set valueattrs [pipe {
        ValueInitializedAttribute findWhere\
                {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Name
    }]
    set popattrs [struct::set difference $valueattrs $defattrs]
    # All Reference components must be specified
    set refattrs [pipe {
        Reference findWhere {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Name
    }]
    return [struct::set union $popattrs $refattrs]
}
```

The set of attributes with a default value is obtained from the Default Value class.

```
<<helper commands>>=
proc FindDefaultableAttrs {domain class} {
    # The attributes having a default value may be left unspecified or
    # may be specified as "-".
    return [pipe {
        DefaultValue findWhere {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Attribute
    }]
}
```

The set of attributes that are zero initialized are obtained from the Zero Initialized Attribute class.

```
<<helper commands>>=
proc FindZeroInitAttrs {domain class} {
    # The zero initialized attributes may be specified explicitly,
    # may be specfied as "-", or may be left unspecified.
    return [pipe {
        ZeroInitializedAttribute findWhere\
                {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Name
    }]
}
```

The set of attributes for which a value may not be specified are the Dependent Attributes which are read only because their value is computed when accessed.

```
<<helper commands>>=
proc FindNoInitializeAttrs {domain class} {
    # The dependent attributes may be not be initialized at all.
    return [pipe {
        DependentAttribute findWhere\
                {$Domain eq $domain && $Class eq $class} |
        deRef ~ |
        relation list ~ Name
    }]
}
```

Resolving Initial Values

Because we introduced the "-" syntax for specifying default values, we need to look up any defaults at instance population time. This procedure does the work of insuring that the component exists and if the default value was specified, makes sure such a default was defined for the attribute or is the *nil* reference value.

```
<<helper commands>>=
proc ResolveInitialValue {domain class component value} {
    set compref [PopulatedComponent findById\
        Domain $domain\
        Class $class\
        Name $component]
    if {[isEmptyRef $compref]} {
        error "component, \"$component\", is not a component that can be\
            populated"
    }
    if {$value eq "-"} {
        # Check for default attribute value
        set defref [findRelated $compref {~R21 Attribute}\
                {~R29 IndependentAttribute} {~R19 ValueInitializedAttribute}\
                ~R22]
        if {[isNotEmptyRef $defref]} {
            return [readAttribute $defref Value]
        }
        error "attribute $component does not have a default value"
    } else {
        set value [subst -nocommands -novariables $value]
    }

    return $value
}
```

Tables of Instance Values

The `instance` command is useful for defining a small number of initial instances, but can become tedious when larger number of instances are involved. For that case, the `table` command is provided which allows class component values to be specified in a more tabular layout without having to repeat the class component names.

`table` *heading name1 values1 name2 values2 . . .*

`heading`
A list of class component names that are to be specified in the remaining arguments

`nameN valuesN`
A list of instance names / instance value set pairs. Each *nameN* argument gives the name of an instance to be initialized. Each corresponding *valueN* argument is a list of values to be given the the instance. The number of values must match the number of component names in the *heading* argument. Values are assigned to class components from the *valueN* argument in the same order as the component names in the *heading*.

The `table` command is used to define initial instance values for the class components. The components to be initialized are given as a list in the *heading* argument. Subsequent arguments then give a name to the instance and a list of class component values that correspond to *heading*. Appropriately formatted, the `table` command can show instance values in a more orderly tablular fashion than the `instance` command and saves repeating class component names.

Implementation

Our implementation of the `table` command follows from that of the `instance` command. The difference is that there is another iteration that allows multiple instances to be created.

```
<<class instance commands>>=
proc table {heading args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }

    upvar #0 [namespace parent]::DomainName DomainName
    variable ClassName

    CheckPopAttrs $DomainName $ClassName $heading

    set insttuple [list\
        Domain $DomainName\
        Class $ClassName\
    ]

    dict for {instname compvalues} $args {
        if {[llength $heading] != [llength $compvalues]} {# ❶
            tailcall DeclError COMPONENT_MISMATCH $DomainName $ClassName\
                [llength $compvalues] [llength $heading]
        }
        dict set insttuple Instance $instname ; # ❷
        CheckDuplicate ClassInstance {class instance} {*}$insttuple
        ClassInstance create {*}$insttuple\
            Number [GenNumber $DomainName ClassInstance\
            [list $DomainName $ClassName]]

        set namedValues [dict create] ; # ❸
        foreach h $heading v $compvalues {
            dict set namedValues $h $v
        }

        CreateRequiredValues $DomainName $ClassName $instname $namedValues
        CreateDefaultedValues $DomainName $ClassName $instname $namedValues
        CreateZeroInitValues $DomainName $ClassName $instname $namedValues
    }
```

```
    return
}
```

1. The number values associated with each instance have to match the heading.
2. Each time through the loop, set up a new instance name.
3. We weave together the heading and component values to get a dictionary keyed by attribute name containing the component value. This lets us reuse the procedures used by the `instance` command.

```
<<error code formats>>=
COMPONENT_MISMATCH  {for domain population, %s, class, %s,\
                     number of component values does not match heading:\
                     got %d values, expected %d values}
```

Populating Multiple Assigners

For multiple assigners, it is necessary to specify which instances of the identifying class is associated to the assigner. The only information needed to define an assigner is its name and the name of the instance of the identifying class corresponding to the assigner.

We start with the now familiar namespace definition.

```
<<assigner instance config namespace layout>>=
namespace eval AssignerInstDef {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::micca::@Config@::ConfigEvaluate
    namespace path {::micca ::micca::@Config@::Helpers ::rosea::InstCmds}

    <<assigner instance commands>>
}
```

Assigner

The `assigner` command (in the context of a population) is used to specify the initial instances of an assigner

`assigner` *association script*

association
: The name of the assocition which has a multiple assigner.

script
: A Tcl script that is evaluated in a context to allow the definition of assigner values.

The `assigner` command is used to define initial instances for an assigner.

Implementation

```
<<population config commands>>=
proc assigner {assoc script} {
    variable DomainName

    set maref [MultipleAssigner findById Domain $DomainName Association $assoc]
    if {[isEmptyRef $maref]} {# ❶
        tailcall DeclError UNKNOWN_MULT_ASSIGNER $assoc
    }

    namespace upvar AssignerInstDef AssocName AssocName IdClass IdClass
    set AssocName $assoc
    set IdClass [readAttribute $maref Class]

    CheckExists AssignerStateModel assigner Domain $DomainName\
            Association $AssocName

    ConfigEvaluate [namespace current]::AssignerInstDef $script

    if {[isEmptyRef [ElementPopulation findById Domain $DomainName\
            Element $assoc]]} {
        ElementPopulation create Domain $DomainName Element $assoc
        AssignerPopulation create Domain $DomainName Association $assoc
    }

    return
}
```

❶ We need to know the identifying class and insist it is define by this point.

Populating An Assigner Instance

The `instance` command (in the context of an assigner) is used to specify the initial instance of the identifying class assocated with the assigner instance.

`instance` *name idinstance*

name
The name of the assigner instance.

idinstance
The name of the identifying instance associated with the assigner. The *idinstance* must be an instance defined for the identifying class associated with the multiple assigner.

The `instance` command is used to define initial instances for an assigner.

Implementation

```
<<assigner instance commands>>=
proc instance {name idinstance} {

    upvar #0 [namespace parent]::DomainName DomainName
    variable AssocName
    variable IdClass
```

```
    CheckDuplicate MultipleAssignerInstance {assigner instance}\
            Domain $DomainName Association $AssocName Instance $name
    MultipleAssignerInstance create\
        Domain $DomainName\
        Association $AssocName\
        Instance $name\
        Number [GenNumber $DomainName MultipleAssignerInstance\
            [list $DomainName $AssocName]]\
        IdClass $IdClass\
        IdInstance $idinstance

    return
}
```

Verifying Population Integrity

Once a population has been gather by the `population` command, we perform some verification on the population to make sure it is correct and we will be able to proceed to code generations. We inspect several factors:

- We insist that there be some population for every domain element that can have values.
- All classes that are not union subclasses must have either a set of initial instances or an allocation greater than 0 or both. Since union subclasses are stored within their superclass, they do not have a separate storage pool.
- The number of instances of a multiple assigner is required to be the same as the number of instances of its partitioning class. Multiple assigner population commands do not support the `allocate` command. So, multiple assigner population must consist of initial instances with any remaining count of instances in the storage pool being automatically allocated so that the total is the same as the number of partitioning class instances. Because of **R108** it is not possible to allocate more Multiple Assigner Instances than instances of the partitioning class. However, it is the case that an initial instance population for a multiple assigner need not be provided.
- We know which associations are static and can make the final decision about reference storage.
- All initial instances that make references to other class instances must have their references specified so we can make sure that the references themselves are correct and that there is no referential integrity problems with the initial instance population.

There is a substantial amount of code required to perform this verification. This is the result of the different types of references that are supported and the fact that the value of class component is taken as a simple string. The format and content of an initial value string depends upon the context of the initialization and so each has to be examined. The simple string nature of initial values means we cannot take advantage of referential integrity and other declarative means to insure correctness. Hence, there will be a lot of code.

We verify the population in the following steps.

```
<<population: verify population integrity>>=
<<population: check completeness>>
<<population: check class populations>>
<<population: check reference values>>
```

Checking Completeness

Value Elements in the platform model are the entities that can have initial instance population values. We want to query all the Value Elements that have no corresponding Element Population in the Population. These are the missing elements that were not included in the population. Two types of values elements can remain unpopulated:

- A union subclass has no separate storage pool since its values are stored as part of its related superclass instance. This is true despite the fact that union subclass instances are defined in the same manner as any other class instance.
- A multiple assigner initial instance population need not be specified since the number of instances of a multiple assigner is bound by the number of instances of its identifying class.

Note that instances of union superclasses must have related union subclasses, so we will insist that referential integrity is preserved. But if all the related instances of a union superclass happen to exclude a particular union subclass, then that is not an error. The case for reference subclasses is different. Since they have memory allocated for them, we will insist that there be space for at least one instance or else it would not be possible for a reference superclass to be related to that particular reference subclass.

```
<<population: check completeness>>=
set usubs [pipe {
    UnionSubclass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation project ~ Domain Class |
    relation rename ~ Class Name
}] ; # ❶
set massigners [pipe {
    MultipleAssigner findWhere {$Domain eq $domain} |
    deRef ~ |
    relation project ~ Domain Association |
    relation rename ~ Association Name
}]
set missing [pipe {
    findRelated $popref {R101 ElementPopulation} |
    findUnrelated ~ R101 |
    deRef ~ |
    relation minus ~ $usubs |
    relation minus ~ $massigners
}] ; # ❷
if {[relation isnotempty $missing]} {
    tailcall DeclError MISSING_POPULATION $domain\
        [join [relation list $missing Name] {, }]
}
```

❶ Arrange for the union subclasses and multiple assigners to have a heading that is compatible with the Value Element heading. We will want to do a subtraction operation in the next step and need a compatible heading.

❷ Here we are looking for the Value Elements that are *unrelated* to the population. They are the ones for which no values were specified. We subtract off the union subclasses and multiple assigners so they are not diagnosed as missing.

```
<<error code formats>>=
MISSING_POPULATION    {for domain population, "%s", the following elements\
                       are missing any population values: "%s"}
```

Checking Class Allocation

For class populations, the maximum number of simultaneous instances is the sum of the initial instance population plus the number of additional storage locations allocated in the `allocate` command. If both these quantities sum to zero, then it is not possible to ever have an instance of the class and the "C" compiler will not let us declare an array variable with 0 elements.

```
<<population: check class populations>>=
# First, we need the set of class component values and class
# instances that are part of the population of this domain.
set popinstrefs [ClassInstance findWhere {$Domain eq $domain}]
set popinsts [deRef $popinstrefs]
```

```
# puts [relformat $popinsts popinsts]
set popvalues [pipe {
    findRelated $popinstrefs {R103 ClassComponentValue}\
            {~R109 SpecifiedComponentValue} |
    deRef %
} {} |%]

# puts [relformat $popvalues popvalues]
set usubClasses [pipe {
    UnionSubclass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation project ~ Class |
    relation list ~
}]
set noinsts [pipe {
    ClassPopulation findWhere {
            $Domain eq $domain && $Class ni $usubClasses && $Allocation == 0} |
    deRef ~ |
    ralutil::rvajoin ~ $popinsts Instances |
    relation restrictwith ~ {[relation isempty $Instances]}
}] ; # ❶

if {[relation isnotempty $noinsts]} {
    tailcall DclError NO_INSTANCES $domain\
        [join [relation list $noinsts Class] {, }]
}
```

❶ The `rvajoin` command performs the relational equivalent of an *outer join* but with no `NUL` values. The result has a relation valued attribute (hence the name `rvajoin`) whose cardinality matches the number of matched tuples in the join. In this case, we are looking for where the relation valued attribute (`Instances` in this case) is empty, indicating that there were no matching instances in the population. We will see this construct often in the population validation code as we frequently are looking for when there are *no* matches within a set.

```
<<error code formats>>=
NO_INSTANCES    {for domain population, "%s", the following classes have no\
                defined instances and a zero allocation: "%s"}
```

```
<<population: check class populations>>=
# Next we want to issue warnings for union subclasses that have a non-zero
# Allocation value. This value is ignored as the storage for union
# subclass instances is always tied to the ultimate superclass of the
# generalization. We don't call it an error so that generalizations
# can be easily switched from reference to union types of storage.
set allocsubs [pipe {
    UnionSubclass findWhere {$Domain eq $domain} |
    findRelatedWhere % {R47 R40 R41 R104 {~R101 ElementPopulation}\
            {~R105 ClassPopulation}} {$Allocation != 0}
} {} |%]

forAllRefs alloced $allocsubs {
    assignAttribute $alloced {Class className} {Allocation allocValue}
    log::warn "In domain, \"$domain\", union-based subclass, \"$className\",\
            has a non-zero \"allocate\" value, \"$allocValue\":\
            this value is ignored since union-based subclasses do not have\
            storage independent of the superclass of the generalization"
}
```

Checking Reference Values

The values given to reference components for a population are simply strings that must refer to instances names of the proper class. We need to go though all the values for the various types of references and make sure they were specified correctly. There are three distinct types of references in the platform model.

```
<<population: check reference values>>=
<<population: check association references>>
<<population: check associator references>>
<<population: check subclass references>>
```

Checking Association Reference Values

When a class is the referring class in a simple association, the value corresponding to the class component for the association is the name of an instance of the referenced class.

The first step in the check is to assemble a relation value whose heading has all the information we need. This information will include the classes that participate in the association as well as the value of the reference from the referring class to the referenced class.

```
<<population: check association references>>=
set simprefs [pipe {
    SimpleReferringClass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation eliminate ~ Role |
    relation rename ~ Class ReferringClass\
        Conditionality ReferringCond Multiplicity ReferringMult|
    relation join ~ $::micca::SimpleReferencedClass |
    relation eliminate ~ Role |
    relation rename ~ Class ReferencedClass Relationship Component |
    relation join ~ $popvalues -using\
        {Domain Domain ReferringClass Class Component Component} |
    relation rename ~ Instance ReferringInstance Value ReferencedInstance
}]
# puts [relformat $simprefs simprefs]
```

Checking Referenced Instances

First we check for references to instances that do not exist.

```
<<population: check association references>>=
set badrefs [pipe {
    relation rename $simprefs ReferencedClass Class ReferencedInstance Instance |
    ralutil::rvajoin ~ $popinsts Instances |
    relation restrictwith ~ {[relation isempty $Instances]}
}] ; # ❶
# puts [relformat $badrefs badrefs]
relation foreach badref $badrefs {
    relation assign $badref
    log::error "for domain population, $Domain,\
        instance, $ReferringClass.$ReferringInstance, refers to\
        instance, $Class.$Instance across,\
        $Component, but $Class.$Instance does not exist"
    incr errcount
}
```

❶ We join against the instances to determine the matches for the `ReferencedClass` and `ReferencedInstance`. If there is no match, then the relation valued attribute will have a cardinality of zero and we have found a reference to an instance that does not exist.

Removing Bad References

We subtract out the bad references so that we can continue the checking. To do the subtraction, we need a compatible heading and that takes a bit of manipulation of names and the effects of the `rvajoin`.

```
<<population: check association references>>=
set simprefs [pipe {
    relation eliminate $badrefs Instances |
    relation rename ~ Class ReferencedClass Instance ReferencedInstance |
    relation minus $simprefs ~
}]
# puts [relformat $simprefs simprefs]
```

Checking Referenced Multiplicity

For a simple association, the referring class always refers to either *exactly one* instance or *at most one* instance. The conditionality of the reference is handled in the population code by insisting that a value be given for the class component that represents the association reference if the association is indeed unconditional.

It remains to check the multiplicity and conditionality of the referring class in the association. For the multiplicity, we need to count the number of times an instance of the referenced class shows up as a reference by the referring class. If that number is greater than one, the we insist the association be multiple on the referring side.

```
<<population: check association references>>=
set refedmults [pipe {
    relation eliminate $simprefs ReferringCond |
    relation group ~ ReferringInstances ReferringInstance |
    relation restrictwith ~ {!$ReferringMult &&\
        [relation cardinality $ReferringInstances] > 1}
}] ; # ❶
# puts [relformat $refedmults refedmults]
relation foreach refedmult $refedmults {
    relation assign $refedmult
    log::error "for domain population, $Domain,\
        relationship, $Component, is singular from\
        $ReferringClass to $ReferencedClass, but instance, $ReferencedInstance,\
        is referred to by multiple instances:\
        [join [relation list $ReferringInstances] {, }]"
    incr errcount
}
```

❶ By grouping the `ReferringInstance` attribute, we are computing the set of instances of the referring class that reference the same instance of the referenced class. This is allowed to be greater than one only if multiplicity is true on the referring side.

Checking Referenced Conditionality

Checking the conditionality of the references requires a slightly different strategy. Here we are interested in determining if all the instances of the referenced class in the association are actually referred to by some instance of the referring class. If some of the instances of the referenced class do not show up, then the association must be conditional on the referring side.

The strategy for this query is to group the referenced instances as a relation valued attribute. Then we can perform an `rvajoin` against the instances in the population to determine the total set of referenced instances. The relation valued attributes from these two operation must be equal if the association is unconditional on the referring side, *i.e.* unconditionality implies that all possible instances of the referenced class appear as an actual reference by the referring class instances. There is a minor complication in the query since to compare the two relation values they must have the same heading, so some juggling of the attribute names is required.

```
<<population: check association references>>=
set refingconds [pipe {
    relation project $simprefs Domain Component ReferringClass ReferringCond\
        ReferencedClass ReferencedInstance |
    relation rename ~ ReferencedClass Class ReferencedInstance Instance |
    relation group ~ ReferencedInstances Instance |
    ralutil::rvajoin ~ [relation eliminate $popinsts Number] Instances |
    relation restrictwith ~\
        {!$ReferringCond && [relation is $ReferencedInstances != $Instances]}
}]
# puts [relformat $refingconds refingconds]

relation foreach refingcond $refingconds {
    relation assign $refingcond
    set notrefed [relation minus $Instances $ReferencedInstances]
    log::error "for domain population, $Domain,\
        relationship, $Component, is unconditional from\
        $Class to $ReferringClass, but the\
        \"[join [relation list $notrefed] {, }]\" instance(s) of $Class\
        is(are) not referred to by any instance of $ReferringClass"
    incr errcount
}
```

Checking Associator Reference Values

The associator class in a class based association always makes unconditional references to each of the participants in the association. Since there is a one-to-one correspondence between the associator class instances and instances of the association itself, the references by an associator class cannot be `nil`. There are a couple of complications when dealing with associator references. First, we are dealing with references to two classes and do not want to impose any particular order on the mention of classes in the instance value. We are expecting a four element list of the form:

```
{<oneclass> <oneinstance> <otherclass> <otherinstance>}
```

where <oneclass> and <otherclass> are class names of the participants in the association and <oneinstance> and <otherinstance> are instance names of the corresponding classes.

The second complication arises with reflexive associations. In this case, the references are both to the same class. We resolve the directional ambiguity by expecting an initial value of the form:

```
{forward <forwinstance> backward <backinstance>}
```

where <forwinstance> is the instance reached when traversing the association in its forward direction and <backinstance> is that reached when traversing in the backward direction (reflexivity means we cannot determine direction from the class names alone). The order of the `forward` and `backward` pairs is arbitrary.

We start with a query that is analogous to the one for simple associations. We gather all the information together in one relation value and we will use this base relation value as the starting point for other queries below. In the case of class based association, we have information for the associator class and both participants.

```
<<population: check associator references>>=
set assocs [pipe {
    AssociatorClass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation project ~ Domain Class Relationship |
    relation rename ~ Class AssocClass |
    relation join ~ $::micca::SourceClass |
    relation eliminate ~ Role |
    relation rename ~ Class SourceClass Conditionality SourceCond\
            Multiplicity SourceMult |
```

```
    relation join ~ $::micca::TargetClass |
    relation eliminate ~ Role |
    relation rename ~ Class TargetClass Conditionality TargetCond\
            Multiplicity TargetMult Relationship Component |
    relation join ~ $popvalues -using\
        {Domain Domain AssocClass Class Component Component} |
    relation rename ~ Instance AssocInstance
}]
# puts [relformat $assocs assocs]
```

The query starts with the Associator Class and then joins on the relation values for the participants. Attributes that are not needed, *e.g.* `Role`, are removed and attributes must be renamed to avoid naming conflicts as the relation value is built up. Finally, we can join in the values of the association class instances which have the references that we are attempting to validate.

We first check that the class component value is a proper four element list as is required for associative references.

```
<<population: check associator references>>=
set badvalues [relation restrictwith $assocs {[llength $Value] != 4}]
relation foreach badvalue $badvalues {
    relation assign $badvalue
    log::error "for domain population, $Domain,\
        instance, $AssocClass.$AssocInstance, has a bad format for the\
        value of the associative reference for, $Component:\
        expected a four element list, got: \"$Value\""
    incr errcount
}
```

To continue checking, we subtract out the references with badly formatted values. Subsequent steps in the validation assume that the values have the correct form.

```
<<population: check associator references>>=
set assocs [relation minus $assocs $badvalues]
```

Next we split out the reflexive and non-reflexive associations. These must be treated differently.

```
<<population: check associator references>>=
set nrassocs [relation restrictwith $assocs {$SourceClass ne $TargetClass}]
set rassocs [relation minus $assocs $nrassocs]
```

For the non-reflexive case, the value forms a four element list consisting of class name/instance name pairs. The class names must match the names of the association participants.

```
<<population: check associator references>>=
set nonparts [relation restrictwith $nrassocs {![struct::set equal\
        [list [lindex $Value 0] [lindex $Value 2]]\
        [list $SourceClass $TargetClass]]}]
# puts [relformat $nonparts nonparts]
relation foreach nonpart $nonparts {
    relation assign $nonpart
    lassign $Value oneclass oneinst otherclass otherinst
    if {$oneclass eq $otherclass} {
        log::error "for domain population, $Domain,\
            associator instance, $AssocClass.$AssocInstance,
            the reference value, $Value, duplicates the reference\
            to class, $oneclass"
        incr errcount
    } else {
        set refedclasses [list $SourceClass $TargetClass]
        foreach {class instance} $Value {
            if {![struct::set contains $refedclasses $class]} {
                log::error "for domain population, $Domain,\
                    associator instance, $AssocClass.$AssocInstance,\
```

```
                    references class instance, $class.$instance, across,\
                    $Component, but $class does not participate in $Component"
                incr errcount
            }
        }
    }
}
```

Now we expand the `Value` attribute into separate attributes for the two references. We will do the same for reflexive associations later and this will make it easier both to join the code flow back together and to examine the references in both directions.

```
<<population: check associator references>>=
set nrassocs [pipe {
    relation minus $nrassocs $nonparts |
    relation extend ~ nrtup\
        TargetInstance string {[dict get [tuple extract $nrtup Value]\
            [tuple extract $nrtup TargetClass]]}\
        SourceInstance string {[dict get [tuple extract $nrtup Value]\
            [tuple extract $nrtup SourceClass]]} |
    relation eliminate ~ Value
}] ; # ❶
```

❶ In this query, we treat the `Value` attribute as a dictionary. We are allowed to do this since the `Value` string is of the form of a proper dictionary.

Now we do the corresponding checks for reflexive associations.

```
<<population: check associator references>>=
set baddirs [relation restrictwith $rassocs\
    {![struct::set equal [dict keys $Value] {forward backward}]}]
relation foreach baddir $baddirs {
    relation assign $baddir
    log::error "for domain population, $Domain,\
        instance, $AssocClass.$AssocInstance, has a bad format for the\
        value of component, $Component: expected a dictionary\
        with \"forward\" and \"backward\" keys, got: \"$Value\""
    incr errcount
}

set rassocs [relation minus $rassocs $baddirs]
```

We can now expand the reflexive value into separate attributes. Notice that the `forward` direction is a reference to an instance of the `TargetClass`.

```
<<population: check associator references>>=
set rassocs [pipe {
    relation extend $rassocs rtup\
        TargetInstance string {[dict get [tuple extract $rtup Value] forward]}\
        SourceInstance string {[dict get [tuple extract $rtup Value] backward]} |
    relation eliminate ~ Value
}]
```

After expanding the `Value` attribute appropriately, the headings for the non-reflexive and reflexive cases are the same and so we can combine them back together for the remainder of the processing.

```
<<population: check associator references>>=
set assocs [relation union $nrassocs $rassocs]
# puts [relformat $assocs assocs]
```

Checking Associator References

At this point we know the associator component values are of the proper format and have split the value into separate attributes. This makes it easier to check whether the associator references are to actual instances.

```
<<population: check associator references>>=
set badsrcrefs [pipe {
    relation rename $assocs SourceClass Class SourceInstance Instance |
    relation project ~ Domain Class Component Instance AssocClass AssocInstance |
    ralutil::rvajoin ~ $popinsts Instances |
    relation restrictwith ~ {[relation isempty $Instances]}
}]
set badtrgrefs [pipe {
    relation rename $assocs TargetClass Class TargetInstance Instance |
    relation project ~ Domain Class Component Instance AssocClass AssocInstance |
    ralutil::rvajoin ~ $popinsts Instances |
    relation restrictwith ~ {[relation isempty $Instances]}
}] ; # ❶

set badrefs [relation union $badsrcrefs $badtrgrefs]
# puts [relformat $badrefs badrefs]
relation foreach badref $badrefs {
    relation assign $badref
    log::error "for domain population, $Domain,\
        instance, $AssocClass.$AssocInstance, refers to\
        instance, $Class.$Instance across,\
        $Component, but $Class.$Instance does not exist"
    incr errcount
}
```

❶ You might be tempted to factor the `rvajoin` statement in the `badsrcrefs` and `badtrgrefs` expressions into the common code below. However, such a refactoring will fail for the reflexive association case. Although we have changed the attribute names to make the two relation union compatible, what the attributes represent is quite different between the two statements and the `relation union` operation ignores that difference. You might also be tempted to refactor the `relation restrictwith` operation. However, the `badsrcrefs` and `badtrgrefs` values are used below to remove those bad references.

Removing Bad References

To continue checking, we will remove any bad references. Since we have modified the heading in the `badsrcrefs` and `badtrgrefs` relations, we will use the `semiminus` operation to find the good references.

```
<<population: check associator references>>=
set assocs [pipe {
    relation semiminus $badsrcrefs $assocs -using\
        {Domain Domain Component Component AssocClass AssocClass\
        AssocInstance AssocInstance Class SourceClass Instance SourceInstance} |
    relation semiminus $badtrgrefs ~ -using\
        {Domain Domain Component Component AssocClass AssocClass\
        AssocInstance AssocInstance Class TargetClass Instance TargetInstance}
}]
# puts [relformat $assocs assocs]
```

Checking Duplicate Associator References

The references made by an instance of an associator class must form a set, *i.e.* you cannot have two associator class instances that refer to the same two participating class instances.

```
<<population: check associator references>>=
set duprefs [pipe {
    relation project $assocs Domain Component AssocClass AssocInstance\
        SourceClass SourceInstance TargetClass TargetInstance |
    relation group ~ References AssocInstance |
    relation restrictwith ~ {[relation cardinality $References] > 1}
}]
# puts [relformat $duprefs duprefs]
relation foreach dupref $duprefs {
    relation assign $dupref
    log::error "for domain population, $Domain,\
        associative class, $AssocClass, instances,\
        [join [relation list $References] {, }],\
        all refer to both $SourceClass.$SourceInstance and\
        $TargetClass.$TargetInstance across, $Component:\
        associative class reference pairs must be unique"
    incr errcount
}
# Remove duplicate references
set assocs [pipe {
    relation ungroup $duprefs References |
    relation semiminus ~ $assocs -using\
        {Domain Domain Component Component AssocClass AssocClass\
        AssocInstance AssocInstance}
}]
```

Checking Associative Multiplicity

We can now check the referential integrity of the class based association references. The pattern of the query is analogous to the one we used for simple associations. We want to count the number of times an instance of a Source Class or Target Class is referenced by the Associator Class. We then compare that to the multiplicity specified for the association. Note, the inversion of the multiplicity from source to target. We have seen this before with class based association. If you count the number of associator instances that refer to a particular source class instance it is the target side multiplicity that determines the allowed values for that count, and *vice versa*.

```
<<population: check associator references>>=
set srcreftos [pipe {
    relation project $assocs Domain Component AssocClass AssocInstance\
        SourceClass SourceInstance TargetMult |
    relation rename ~ SourceClass Class SourceInstance Instance\
        TargetMult Multiplicity |
    relation group ~ AssocInstances AssocInstance
}]
# puts [relformat $srcreftos srcreftos]

set trgreftos [pipe {
    relation project $assocs Domain Component AssocClass AssocInstance\
        TargetClass TargetInstance SourceMult |
    relation rename ~ TargetClass Class TargetInstance Instance\
        SourceMult Multiplicity |
    relation group ~ AssocInstances AssocInstance
}]
# puts [relformat $trgreftos trgreftos]

set reftos [pipe {
    relation union $srcreftos $trgreftos |
    relation restrictwith ~ {!$Multiplicity &&\
        [relation cardinality $AssocInstances] > 1}
}]
```

```
# puts [relformat $reftos reftos]
relation foreach refto $reftos {
    relation assign $refto
    log::error "for domain population, $Domain,\
        class based association, $Component, is singular to, $Class,\
        but instance, $Instance,\
        is referred to multiple times by associative class, $AssocClass,\
        instances: [join [relation list $AssocInstances] {, }]"
    incr errcount
}
```

Checking Associative Conditionality

Again, we use a query that is a counterpart to the one used for simple associations. We wish to find all the source and target instances that are referenced by an associator class instance and then compare that to the total set of instances of the participating class. If the association is unconditional, then the two set must be the same. Once again note the swap of source and target conditionality values when performing the check.

```
<<population: check associator references>>=
set srcconds [pipe {
    relation project $assocs Domain Component AssocClass\
            SourceClass SourceInstance TargetCond |
    relation rename ~ SourceClass Class SourceInstance Instance\
            TargetCond Conditionality |
    relation group ~ ReferencedInstances Instance |
    ralutil::rvajoin ~ [relation eliminate $popinsts Number] Instances
}]
# puts [relformat $srcconds srcconds]

set trgconds [pipe {
    relation project $assocs Domain Component AssocClass\
            TargetClass TargetInstance SourceCond |
    relation rename ~ TargetClass Class TargetInstance Instance\
            SourceCond Conditionality |
    relation group ~ ReferencedInstances Instance |
    ralutil::rvajoin ~ [relation eliminate $popinsts Number] Instances
}]
# puts [relformat $trgconds trgconds]

set badconds [pipe {
    relation union $srcconds $trgconds |
    relation restrictwith ~ {!$Conditionality &&\
            [relation is $ReferencedInstances != $Instances]}
}]
# puts [relformat $badconds badconds]
relation foreach badcond $badconds {
    relation assign $badcond
    set notrefed [relation minus $Instances $ReferencedInstances]
    log::error "for domain population, $Domain,\
        class based association, $Component, requires all instances of $Class\
        to be referenced, but the \"[join [relation list $notrefed] {, }]\"\
        instance(s) of $Class is(are) not referenced by any instance\
        of $AssocClass"
    incr errcount
}
```

Checking Subclass Reference Values

The generalization relationship represents a disjoint union of the subclasses. The implications of this are:

- Each subclass instance must unconditionally refer to exactly one instance of the superclass of the generalization.
- Each superclass instance must be referenced by exactly one subclass instance from among all the subclass instances in the generalization.

We follow the same query stategy as we used above. First we construct a relation value that contains all the information about the superclasses and subclasses in the generalization.

```
<<population: check subclass references>>=
set subrefs [pipe {
    Subclass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation eliminate ~ Role |
    relation rename ~ Class SubClass |
    relation join ~ $::micca::Superclass |
    relation eliminate ~ Role |
    relation rename ~ Class SuperClass Relationship Component |
    relation join ~ $popvalues -using\
        {Domain Domain SubClass Class Component Component} |
    relation rename ~ Instance SubClassInstance Value SuperClassInstance
}]
# puts [relformat $subrefs subrefs]
```

We can now check to references to superclasses that don't exist.

```
<<population: check subclass references>>=
set badrefs [pipe {
    relation rename $subrefs SuperClass Class SuperClassInstance Instance |
    ralutil::rvajoin ~ $popinsts Instances |
    relation restrictwith ~ {[relation isempty $Instances]}
}]
# puts [relformat $badrefs badrefs]

relation foreach badref $badrefs {
    relation assign $badref
    log::error "for domain population, $Domain, generalization, $Component,\
        subclass instance, $SubClass.$SubClassInstance, references\
        superclass instance, $Class.$Instance, but $Instance does not exist"
    incr errcount
}
```

We subtract out the bad references so that we can proceed with the next checks.

```
<<population: check subclass references>>=
set subrefs [pipe {
    relation eliminate $badrefs Instances |
    relation rename ~ Class SuperClass Instance SuperClassInstance |
    relation minus $subrefs ~
}]
# puts [relformat $subrefs subrefs]
```

Checking Subclass Reference Multiplicity

No superclass instance may be referenced multiple times. Here we compute the multiplcity of the references. This query is analogous to the ones we did for associative relationships except here we know that it must be singular.

```
<<population: check subclass references>>=
set multirefs [pipe {
    relation group $subrefs SubClassReferences SubClass SubClassInstance |
    relation restrictwith ~ {[relation cardinality $SubClassReferences] > 1}
```

```
}]
# puts [relformat $multirefs multirefs]
relation foreach multiref $multirefs {
    relation assign $multiref
    relation foreach sref $SubClassReferences {
        relation assign $sref
        lappend screfs $SubClass.$SubClassInstance
    }
    log::error "for domain population, $Domain, generalization, $Component,\
        superclass instance, $SuperClass.$SuperClassInstance, is referenced by\
        multiple subclass instances, \"[join $screfs {, }]\":\
        this violates the referential integrity of generalizations"
    incr errcount
}
```

Checking Subclass Reference Conditionality

All superclass instances must be referred to by some subclass instance. In that sense the generalization is unconditional.

```
<<population: check subclass references>>=
set unrefeds [pipe {
    relation rename $subrefs SuperClass Class SuperClassInstance Instance |
    relation eliminate ~ SubClass SubClassInstance |
    relation group ~ ReferencedSupers Instance |
    ralutil::rvajoin ~ [relation eliminate $popinsts Number] Instances |
    relation restrictwith ~ {[relation is $ReferencedSupers != $Instances]}
}]
# puts [relformat $unrefeds unrefeds]

relation foreach unrefed $unrefeds {
    relation assign $unrefed
    set notrefeds [pipe {
        relation minus $Instances $ReferencedSupers |
        relation list ~
    }]
    foreach notrefed $notrefeds {
        log::error "for domain population, $Domain, generalization, $Component,\
            superclass instance, $Class.$notrefed, is not referenced by any\
            subclass: this violates the referential integrity of\
            generalizations"
        incr errcount
    }
}
```

Calculating Generated Component Values

The platform model has a number of classes that are subclasses of **Generated Component**. These classes all have the same characteristic that their values can be calculated from the values of the class components specified in the initial instance population. All of these calculated component values are in some sense redundant. They are all involved in generating the data required to navigate relationships and enforce referential integrity.

Recall that the way `micca` deals with relationships is to store pointer values in the instances of all participants in a relationship so that the relationship can be navigated directly in either direction. This is a space/speed tradeoff that makes it much easier to both navigate relationships and to enforce referential integrity at the end of a data transaction. In the `population` command, relationship references are specified only in their most fundamental form, *i.e.* as references from referring classes to named instances. That is sufficient information to compute the references in the opposite direction. This is what the **Generated Component** instances hold. A further complication relates to two other factors:

1. An association may be specified as static, in which case we choose a more space efficient arrangement to store the reference pointer values.
2. A generalization may be held as a union, in which case its related instance is stored directly in the memory slot allocated to the superclass instance.

Examining the platform model, we see that there are four primary generated components whose values we must calculate. Of the four, values for the **Link Container** are computed when the values for the **Link Reference** are computed since the two components form a doubly linked list.

```
<<population: set generated component values>>=
set gencomps [GeneratedComponent findWhere {$Domain eq $domain}]

<<population: set complementary reference values>>
<<population: set subclass reference values>>
<<population: set subclass container values>>
```

Calculating Complementary Reference Values

A **Complementary Reference** is a pointer reference from a referred to class instance back to the referring class instance. For simple associations, it is the reference that enables navigating the association in the *backward* direction directly by pointer indirection. There are three different types of complementary references:

1. Singular references are held as a single pointer value and can be used when the multiplicity of the relationships is one.
2. Array references are held as a counted array of pointer values and are used for static associations when the multiplicity is greater than one.
3. Link references are held as a doubly-linked list and are used for dynamic associations when the multiplicity is greater than one.

In addition, a **Complementary Reference** is either a backward reference, *i.e.* used to navigate the association in *backward* direction, or a foward reference used to navigate in the *forward* direction.

```
<<population: set complementary reference values>>=
<<population: set singular reference values>>
<<population: set array reference values>>
<<population: set link reference values>>
```

Since Complementary References are references that arise out of associative relationships, there are three classes where they appear:

1. Simple Referenced Classes have a reference to their corresponding Simple Referring Class
2. Target Classes refer to the Associator class in class based associations.
3. Source Classes also refer to the Associator class in class based associations and are used to navigate the class based association in the forward direction.

So for each of the three ways of storing reference pointers there are also three different classes that need to have references generated for them. Below we take up each case individually.

Singular Reference Values

```
<<population: set singular reference values>>=
set singlerefs [pipe {
    SingularReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 BackwardReference} ~R94
} {} |%]
# puts [relformat [deRef $singlerefs] singlerefs]

<<population: set singular referenced class values>>
<<population: set singular target class values>>
<<population: set singular source class values>>
```

Singular Backward Reference Values

Backward complementary references arise in two situations:

1. A Simple Referenced Class refers to a Simple Referring Class when navigating an association in the *backward* direction.
2. A Target Class refers to the Associator Class when navigating a class based association in the *backward* direction.

```
<<population: set singular referenced class values>>=
set simprefedclass [findRelated $singlerefs {~R38 SimpleReferencedClass}]
# Compute the referenced made by the refering class instances.
set refingrefs [pipe {
    deRef $simprefedclass |
    relation eliminate ~ Role |
    relation rename ~ Class ReferencedClass |
    relation join ~ $::micca::SimpleReferringClass |
    relation rename ~ Class ReferringClass Relationship Component |
    relation eliminate ~ Role Conditionality Multiplicity |
    relation join ~ $popvalues -using\
        {Domain Domain Component Component ReferringClass Class} |
    relation update ~ sbtup {1} {
        tuple update $sbtup Component "[tuple extract $sbtup Component]__BACK"}
}]
# puts [relformat $refingrefs refingrefs]

# Now we have to split out the nil references
# Invert referring and referenced formating a new value to show
# which instances referenced by the referring class.
set backrefs [pipe {
    relation restrictwith $refingrefs {$Value ne "@nil@"} |
    relation extend ~ sbtup NewValue string {[list\
        [tuple extract $sbtup ReferringClass]\
        [tuple extract $sbtup Instance]]} |
    relation eliminate ~ ReferringClass Instance |
    relation rename ~ ReferencedClass Class Value Instance NewValue Value
}]
# puts [relformat $backrefs backrefs]

foreach backref [relation body $backrefs] {
    SpecifiedComponentValue create {*}$backref
    dict unset backref Value
    ClassComponentValue create {*}$backref
}

CreateNilDestBackRefs $simprefedclass $backrefs
```

We can factor out some common code for creating the back references for conditional associations where we have `nil` reference values.

```
<<helper commands>>=
proc CreateNilDestBackRefs {refedclasses backrefs} {
    set nilbackrefs [FindNilDestBackRefs $refedclasses $backrefs]
    # puts [relformat $nilbackrefs nilbackrefs]
    foreach nilbackref [relation body $nilbackrefs] {
        SpecifiedComponentValue create {*}$nilbackref Value @nil@
        ClassComponentValue create {*}$nilbackref
    }
}
```

```
<<helper commands>>=
proc FindNilDestBackRefs {refedclasses backrefs} {
    # Find the components for the Referenced Classes regardless of value.
    # The difference between this set and those components where a value
    # was specified must be the set of components whose value are nil.
    set valueinsts [relation project $backrefs Domain Class Component Instance]
    set nilrefs [pipe {
        findRelated $refedclasses R38 R94 |
        deRef ~ |
        relation project ~ Domain Class Name |
        relation rename ~ Name Component |
        relation join ~ $::micca::ClassInstance |
        relation eliminate ~ Number |
        relation minus ~ $valueinsts
    }]
    # puts [relformat $nilrefs nilrefs]
    return $nilrefs
}
```

Because a Target Class is part of a class based association, its backward references will be to the associator class itself.

```
<<population: set singular target class values>>=
set trgclass [findRelated $singlerefs {~R38 TargetClass}]
set trgrefing [pipe {
    deRef $trgclass |
    relation eliminate ~ Role Conditionality Multiplicity |
    relation rename ~ Class TargetClass |
    relation join ~ $::micca::AssociatorClass |
    relation rename ~ Class AssociatorClass |
    relation eliminate ~ Role Multiplicity |
    relation join ~ $::micca::SourceClass |
    relation rename ~ Class SourceClass Relationship Component |
    relation eliminate ~ Role Conditionality Multiplicity |
    relation rename ~ AssociatorClass Class |
    ralutil::rvajoin ~ $popvalues Values |
    relation update ~ sbtup {1} {
        tuple update $sbtup Component "[tuple extract $sbtup Component]__BACK"}
}]
# puts [relformat $trgrefing trgrefing]

set trgbackrefs [pipe {
    relation restrictwith $trgrefing {[relation isnotempty $Values]} |
    relation ungroup ~ Values |
    relation rename ~ Class AssociatorClass
}]
# puts [relformat $trgbackrefs trgbackrefs]
```

At this point we split out the reflexive versus the non-reflexive case. As usual with class based associations, reflexive association require special handling.

For the non-reflexive case, we extract the reference to the Target Class.

```
<<population: set singular target class values>>=
set nrtrgbackrefs [pipe {
    relation restrictwith $trgbackrefs {$SourceClass ne $TargetClass} |
    relation extend ~ tbtup TargetInstance string {[dict get\
        [tuple extract $tbtup Value] [tuple extract $tbtup TargetClass]]}
}]
# puts [relformat $nrtrgbackrefs nrtrgbackrefs]
```

For the reflexive case, we extract the reference for the **forward** direction. It is the forward direction reference for which we need to create a back reference.

```
<<population: set singular target class values>>=
set rtrgbackrefs [pipe {
    relation restrictwith $trgbackrefs {$SourceClass eq $TargetClass} |
    relation extend ~ tbtup TargetInstance string {[dict get\
        [tuple extract $tbtup Value] forward]}
}]
# puts [relformat $rtrgbackrefs rtrgbackrefs]
```

Once we have extracted the correct reference value, then we can combine the processing going on. The goal is to create a relation value whose heading matches the attributes of the **Class Component Value** class in the platform model.

```
<<population: set singular target class values>>=
set backrefs [pipe {
    relation union $nrtrgbackrefs $rtrgbackrefs |
    relation extend ~ brtup NewValue string {[list\
        [tuple extract $brtup AssociatorClass]\
        [tuple extract $brtup Instance]]} |
    relation eliminate ~ AssociatorClass SourceClass Instance Value |
    relation rename ~ TargetClass Class TargetInstance Instance NewValue Value
}]
# puts [relformat $backrefs backrefs]
```

Note that the value computed for the back reference contains both the class name and instance name of the reference. This is to make code generation easier so that we don't have to look up the association to determine to which class the reference is intended.

With the relation value in hand, it is a simple matter to create the necessary component values.

```
<<population: set singular target class values>>=
foreach backref [relation body $backrefs] {
    SpecifiedComponentValue create {*}$backref
    dict unset backref Value
    ClassComponentValue create {*}$backref
}

CreateNilDestBackRefs $trgclass $backrefs
```

Singular Forward Reference Values

The Complementary References for a Source Class are used to navigate the association in the forward direction. The query for Source Classes is much the same as for Target Classes with the two attribute names inverted.

```
<<population: set singular source class values>>=
set srcclass [pipe {
    SingularReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 ForwardReference} ~R95
} {} |%]
set srcrefing [pipe {
    deRef $srcclass |
```

```
    relation eliminate ~ Role Conditionality Multiplicity |
    relation rename ~ Class SourceClass |
    relation join ~ $::micca::AssociatorClass |
    relation rename ~ Class AssociatorClass |
    relation eliminate ~ Role Multiplicity |
    relation join ~ $::micca::TargetClass |
    relation rename ~ Class TargetClass Relationship Component |
    relation eliminate ~ Role Conditionality Multiplicity |
    relation rename ~ AssociatorClass Class |
    ralutil::rvajoin ~ $popvalues Values |
    relation update ~ sbtup {1} {
        tuple update $sbtup Component "[tuple extract $sbtup Component]__FORW"}
}]
# puts [relformat $srcforwrefs srcforwrefs]

set srcforwrefs [pipe {
    relation restrictwith $srcrefing {[relation isnotempty $Values]} |
    relation ungroup ~ Values |
    relation rename ~ Class AssociatorClass
}]
```

Again we must deal with the reflexive and non-reflexive cases separately so that we can interpret the value of the reference properly.

```
<<population: set singular source class values>>=
set nrsrcforwrefs [pipe {
    relation restrictwith $srcforwrefs {$SourceClass ne $TargetClass} |
    relation extend ~ tbtup SourceInstance string {[dict get\
        [tuple extract $tbtup Value] [tuple extract $tbtup SourceClass]]}
}]
# puts [relformat $nrsrcforwrefs nrsrcforwrefs]

set rsrcforwrefs [pipe {
    relation restrictwith $srcforwrefs {$SourceClass eq $TargetClass} |
    relation extend ~ tbtup SourceInstance string {[dict get\
        [tuple extract $tbtup Value] backward]}
}]
# puts [relformat $rsrcforwrefs rsrcforwrefs]
```

Combining the two cases back, we compute the component values.

```
<<population: set singular source class values>>=
set forwrefs [pipe {
    relation union $nrsrcforwrefs $rsrcforwrefs |
    relation extend ~ frtup NewValue string {[list\
        [tuple extract $frtup AssociatorClass]\
        [tuple extract $frtup Instance]]} |
    relation eliminate ~ AssociatorClass TargetClass Instance Value |
    relation rename ~ SourceClass Class SourceInstance Instance NewValue Value
}]
# puts [relformat $forwrefs forwrefs]
```

Finally, the forward reference values are created.

```
<<population: set singular source class values>>=
foreach forwref [relation body $forwrefs] {
    SpecifiedComponentValue create {*}$forwref
    dict unset forwref Value
    ClassComponentValue create {*}$forwref
}

CreateNilSourceForwRefs $srcclass $forwrefs
```

The case of `nil` references from Source Classes can also be factored away. The processing is very similar to that for destination back references.

```
<<helper commands>>=
proc CreateNilSourceForwRefs {srcclasses forwrefs} {
    set nilforwrefs [FindNilSourceForwRefs $srcclasses $forwrefs]
    # puts [relformat $nilforwrefs nilforwrefs]
    foreach nilforwref [relation body $nilforwrefs] {
        ClassComponentValue create {*}$nilforwref
        SpecifiedComponentValue create {*}$nilforwref Value @nil@
    }
}
```

```
<<helper commands>>=
proc FindNilSourceForwRefs {refedclasses forwrefs} {
    # Find the components for the Referenced Classes regardless of value.
    # The difference between this set and those components where a value
    # was specified must be the set of components whose value are nil.
    set valueinsts [relation project $forwrefs Domain Class Component Instance]
    set nilrefs [pipe {
        findRelated $refedclasses R95 |
        deRef ~ |
        relation project ~ Domain Class Name |
        relation rename ~ Name Component |
        relation join ~ $::micca::ClassInstance |
        relation eliminate ~ Number |
        relation minus ~ $valueinsts
    }]
    # puts [relformat $nilrefs nilrefs]
    return $nilrefs
}
```

Array Reference Values

When the multiplicity of an association is greater than one and the association has been declared static, then we can use a more convenient means to store the required pointer values, namely, a counted array. Again, we will consider each of the three cases separately.

```
<<population: set array reference values>>=
set arrayrefs [pipe {
    ArrayReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 BackwardReference} ~R94
} {} |%]
# puts [relformat [deRef $arrayrefs] arrayrefs]

<<population: set array referenced class values>>
<<population: set array target class values>>
<<population: set array source class values>>
```

First we examine the simple referenced classes.

```
<<population: set array referenced class values>>=
set arrayrefedclass [findRelated $arrayrefs {~R38 SimpleReferencedClass}]
set refingrefs [FindSimpleReferringRefs $arrayrefedclass $popvalues]
# puts [relformat $refingrefs refingrefs]
set backrefs [pipe {
    FindSimpleBackRefs $refingrefs |
    relation eliminate ~ Relationship
}]
# puts [relformat $backrefs backrefs]
```

```
foreach backref [relation body $backrefs] {
    SpecifiedComponentValue create {*}$backref
    dict unset backref Value
    ClassComponentValue create {*}$backref
}

CreateNilDestBackRefs $arrayrefedclass $backrefs
```

As it turns out, the query needed for array type pointer storage is the same as that needed for linked list storage (they are both multiple in nature). The only difference between the two is how the queried information is generated into the code. So we can factor out the query for simple backward references. We will also factor out the other cases below.

```
<<helper commands>>=
proc FindSimpleReferringRefs {refedclasses popvalues} {
    return [pipe {
        deRef $refedclasses |
        relation eliminate ~ Role |
        relation rename ~ Class ReferencedClass |
        relation join ~ $::micca::SimpleReferringClass |
        relation rename ~ Class ReferringClass |
        relation eliminate ~ Role Conditionality Multiplicity |
        relation join ~ $popvalues -using\
            {Domain Domain Relationship Component ReferringClass Class} |
        relation extend ~ abtup\
            Component string {"[tuple extract $abtup Relationship]__BACK"}
    }]
}

proc FindSimpleBackRefs {refingrefs} {
    return [pipe {
        relation group $refingrefs Instances Instance |
        relation extend ~ abtup\
            NewValue string {[list [tuple extract $abtup ReferringClass]\
                [relation list [tuple extract $abtup Instances]]]} |
        relation eliminate ~ ReferringClass Instances |
        relation rename ~ ReferencedClass Class Value Instance NewValue Value
    }]
}
```

For target class references we have the following.

```
<<population: set array target class values>>=
set trgclass [findRelated $arrayrefs {~R38 TargetClass}]
set refingrefs [FindTargetReferringRefs $trgclass $popvalues]
# puts [relformat $refingrefs refingrefs]
set backrefs [pipe {
    FindTargetBackRefs $refingrefs |
    relation eliminate ~ Relationship
}]
# puts [relformat $backrefs backrefs]

foreach backref [relation body $backrefs] {
    SpecifiedComponentValue create {*}$backref
    dict unset backref Value
    ClassComponentValue create {*}$backref
}

CreateNilDestBackRefs $trgclass $backrefs
```

Again, finding the target back references can be shared with the linked list references below.

```
<<helper commands>>=
proc FindTargetReferringRefs {refedclasses popvalues} {
    set backrefs [pipe {
        deRef $refedclasses |
        relation eliminate ~ Role Conditionality Multiplicity |
        relation rename ~ Class TargetClass |
        relation join ~ $::micca::AssociatorClass |
        relation rename ~ Class AssociatorClass |
        relation eliminate ~ Role Multiplicity |
        relation join ~ $::micca::SourceClass |
        relation rename ~ Class SourceClass |
        relation eliminate ~ Role Conditionality Multiplicity |
        relation join ~ $popvalues -using\
            {Domain Domain Relationship Component AssociatorClass Class}
    }]
    # puts [relformat $backrefs backrefs]

    # Non-reflexive
    set nrtrgbackrefs [pipe {
        relation restrictwith $backrefs {$SourceClass ne $TargetClass} |
        relation extend ~ tbtup TargetInstance string {[dict get\
            [tuple extract $tbtup Value] [tuple extract $tbtup TargetClass]]}
    }]
    # puts [relformat $nrtrgbackrefs nrtrgbackrefs]

    # Reflexive
    set rtrgbackrefs [pipe {
        relation restrictwith $backrefs {$SourceClass eq $TargetClass} |
        relation extend ~ tbtup TargetInstance string {[dict get\
            [tuple extract $tbtup Value] forward]}
    }]
    # puts [relformat $rtrgbackrefs rtrgbackrefs]
    return [pipe {
        relation union $nrtrgbackrefs $rtrgbackrefs |
        relation eliminate ~ SourceClass Value |
        relation extend ~ brtup\
            Component string {"[tuple extract $brtup Relationship]__BACK"}
    }]
}

proc FindTargetBackRefs {refingrefs} {
    return [pipe {
        relation group $refingrefs Instances Instance |
        relation extend ~ brtup\
            NewValue string {[list\
                [tuple extract $brtup AssociatorClass]\
                [relation list [tuple extract $brtup Instances]]]} |
        relation eliminate ~ AssociatorClass Instances |
        relation rename ~ TargetClass Class TargetInstance Instance NewValue Value
    }]
}
```

An lastly, the forward references made by source classes.

```
<<population: set array source class values>>=
set srcclass [pipe {
    ArrayReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 ForwardReference} ~R95
} {} |%]

set refingrefs [FindSourceReferringRefs $srcclass $popvalues]
```

```
# puts [relformat $refingrefs refingrefs]
set forwrefs [pipe {
    FindSourceForwRefs $refingrefs |
    relation eliminate ~ Relationship
}]
# puts [relformat $forwrefs forwrefs]

foreach forwref [relation body $forwrefs] {
    SpecifiedComponentValue create {*}$forwref
    dict unset forwref Value
    ClassComponentValue create {*}$forwref
}

CreateNilSourceForwRefs $srcclass $forwrefs
```

And we factor out the finding of these references for later use.

```
<<helper commands>>=
proc FindSourceReferringRefs {refedclasses popvalues} {
    set forwrefs [pipe {
        deRef $refedclasses |
        relation eliminate ~ Role Conditionality Multiplicity |
        relation rename ~ Class SourceClass |
        relation join ~ $::micca::AssociatorClass |
        relation rename ~ Class AssociatorClass |
        relation eliminate ~ Role Multiplicity |
        relation join ~ $::micca::TargetClass |
        relation rename ~ Class TargetClass |
        relation eliminate ~ Role Conditionality Multiplicity |
        relation join ~ $popvalues -using\
            {Domain Domain Relationship Component AssociatorClass Class}
    }]
    # puts [relformat $forwrefs forwrefs]

    # Non-reflexive
    set nrsrcforwrefs [pipe {
        relation restrictwith $forwrefs {$SourceClass ne $TargetClass} |
        relation extend ~ tbtup SourceInstance string {[dict get\
            [tuple extract $tbtup Value] [tuple extract $tbtup SourceClass]]}
    }]
    # puts [relformat $nrsrcforwrefs nrsrcforwrefs]

    # Reflexive
    set rsrcforwrefs [pipe {
        relation restrictwith $forwrefs {$SourceClass eq $TargetClass} |
        relation extend ~ tbtup SourceInstance string {[dict get\
            [tuple extract $tbtup Value] backward]}
    }]
    # puts [relformat $rsrcforwrefs rsrcforwrefs]
    return [pipe {
        relation union $nrsrcforwrefs $rsrcforwrefs |
        relation eliminate ~ TargetClass Value |
        relation extend ~ frtup\
            Component string {"[tuple extract $frtup Relationship]__FORW"}
    }]
}

proc FindSourceForwRefs {refingrefs} {
    return [pipe {
        relation group $refingrefs Instances Instance |
        relation extend ~ frtup\
            NewValue string {[list\
```

```
                [tuple extract $frtup AssociatorClass]\
                [relation list [tuple extract $frtup Instances]]]} |
        relation eliminate ~ AssociatorClass Instances |
        relation rename ~ SourceClass Class SourceInstance Instance NewValue Value
    }]
}
```

```
<<population: set link reference values>>=
set linkbackrefs [pipe {
    LinkReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 BackwardReference} ~R94
} {} |%]

<<population: set link referenced class values>>
<<population: set link target class values>>
<<population: set link source class values>>
```

The third manner in which references are stored is as linked lists. For dynamic association, inserting and removing instances from linked lists is much easier than dealing with an array based storage. Of course, the linkage pointers require more space and that is why both alternatives are supported.

For simple backward references, we use the same means to find them, but use different code to create the linked references.

```
<<population: set link referenced class values>>=
set linkrefedclass [findRelated $linkbackrefs {~R38 SimpleReferencedClass}]

set refingrefs [FindSimpleReferringRefs $linkrefedclass $popvalues]
# puts [relformat $refingrefs refingrefs]
set backrefs [FindSimpleBackRefs $refingrefs]
# puts [relformat $backrefs backrefs]

CreateLinkedRefs $backrefs BLINKS

CreateNilLinkedBackRefs $linkrefedclass $backrefs
```

Because the related class instances are threaded onto linked lists, we have a list terminus in the referring class and link containers in the other participant classes. We must have a way that any given instance may be threaded onto multiple lists and it is quite possible for any given class to participate in multiple associations. The function below creates both the Link Reference values as well as the Link Container values in the referenced class instances. It does this by traversing the list of related instances and generating values corresponding to the class, instance and component of the reference.

```
<<helper commands>>=
proc CreateLinkedRefs {backrefs compname} {
    foreach backref [relation body $backrefs] {
        lassign [dict get $backref Value] refing insts
        set relship [dict get $backref Relationship]
        set next [list $refing [lindex $insts 0] ${relship}__$compname]
        set prev [list $refing [lindex $insts end] ${relship}__$compname]
        ClassComponentValue create\
            Domain [dict get $backref Domain]\
            Class [dict get $backref Class]\
            Instance [dict get $backref Instance]\
            Component [dict get $backref Component]
        SpecifiedComponentValue create\
            Domain [dict get $backref Domain]\
            Class [dict get $backref Class]\
            Instance [dict get $backref Instance]\
            Component [dict get $backref Component]\
            Value [dict create next $next prev $prev]

        set instindex 0
```

```
        set terminus [list\
            [dict get $backref Class]\
            [dict get $backref Instance]\
            [dict get $backref Component]\
        ]
        set prev $terminus
        for {set value [lindex $insts $instindex]} {$value ne {}}\
                {set value [lindex $insts [incr instindex]]} {
            if {$instindex < [llength $insts] - 1} {
                set next [list\
                    $refing\
                    [lindex $insts $instindex+1]\
                    ${relship}__$compname\
                ]
            } else {
                set next $terminus
            }
            ClassComponentValue create\
                Domain [dict get $backref Domain]\
                Class $refing\
                Instance $value\
                Component ${relship}__$compname
            SpecifiedComponentValue create\
                Domain [dict get $backref Domain]\
                Class $refing\
                Instance $value\
                Component ${relship}__$compname\
                Value [dict create next $next prev $prev]
            set prev [list\
                $refing\
                $value\
                ${relship}__$compname\
            ]
        }
    }
}
```

```
<<helper commands>>=
proc CreateNilLinkedBackRefs {refedclasses backrefs} {
    set nilbackrefs [FindNilDestBackRefs $refedclasses $backrefs]
    # puts [relformat $nilbackrefs nilbackrefs]
    CreateNilLinkedRefs $nilbackrefs
}
```

```
<<helper commands>>=
proc CreateNilLinkedRefs {nilbackrefs} {
    relation foreach nilbackref $nilbackrefs {
        relation assign $nilbackref
        ClassComponentValue create\
            Domain $Domain\
            Class $Class\
            Instance $Instance\
            Component $Component
        SpecifiedComponentValue create\
            Domain $Domain\
            Class $Class\
            Instance $Instance\
            Component $Component\
            Value [dict create\
                next [list $Class $Instance $Component]\
                prev [list $Class $Instance $Component]\
```

```
            ]
    }
}
```

Target classes references are handled similar to the simple referenced classes.

```
<<population: set link target class values>>=
set trgclass [findRelated $linkbackrefs {~R38 TargetClass}]
set refingrefs [FindTargetReferringRefs $trgclass $popvalues]
# puts [relformat $refingrefs refingrefs]
set backrefs [FindTargetBackRefs $refingrefs]
# puts [relformat $backrefs backrefs]

CreateLinkedRefs $backrefs BLINKS

CreateNilLinkedBackRefs $trgclass $backrefs
```

And finally the forward references from source classes.

```
<<population: set link source class values>>=
set srcclass [pipe {
    LinkReference findWhere {$Domain eq $domain} |
    findRelated % R26 {~R28 ForwardReference} ~R95
} {} |%]

set refingrefs [FindSourceReferringRefs $srcclass $popvalues]
set forwrefs [FindSourceForwRefs $refingrefs]
CreateLinkedRefs $forwrefs FLINKS

CreateNilLinkedForwRefs $srcclass $forwrefs
```

```
<<helper commands>>=
proc CreateNilLinkedForwRefs {srcclasses forwrefs} {
    set nilbackrefs [FindNilSourceForwRefs $srcclasses $forwrefs]
    # puts [relformat $nilbackrefs nilbackrefs]
    CreateNilLinkedRefs $nilbackrefs
}
```

Subclass references are made by reference superclasses. We must look for superclass references made by reference subclasses to find the reference values needed to compute the subclass reference made by the superclass.

```
<<population: set subclass reference values>>=
set rsupercomps [pipe {
    ReferringSubclass findWhere {$Domain eq $domain} |
    findRelated % R47 R40 R41 ~R20 {~R25 PopulatedComponent} {~R21 Reference}\
        {~R23 SuperclassReference} R23 R21 R25 {~R103 ClassComponentValue}\
        {~R109 SpecifiedComponentValue} |
    deRef % |
    relation join %\
        $::micca::ReferenceGeneralization -using {Domain Domain Component Name}\
        [relation rename $::micca::ReferencedSuperclass Class Superclass]\
        -using {Domain Domain Component Relationship} |
    relation extend % rstup NewValue string {
        [list [tuple extract $rstup Class] [tuple extract $rstup Instance]]} |
    relation eliminate % Class Instance Role |
    relation rename % Superclass Class Value Instance NewValue Value
} {} |%]
# puts [relformat $rsupercomps rsupercomps]

foreach subref [relation body $rsupercomps] {
    SpecifiedComponentValue create {*}$subref
```

```
    dict unset subref Value
    ClassComponentValue create {*}$subref
}
```

Subclass containers are used by union superclasses and are the union superclass counterpart of a subclass reference by a referenced superclass. We use similar query.

```
<<population: set subclass container values>>=
set usupercomps [pipe {
    UnionSubclass findWhere {$Domain eq $domain} |
    findRelated % R47 R40 R41 ~R20 {~R25 PopulatedComponent} {~R21 Reference}\
        {~R23 SuperclassReference} R23 R21 R25 {~R103 ClassComponentValue}\
        {~R109 SpecifiedComponentValue} |
    deRef % |
    relation join %\
        $::micca::UnionGeneralization -using {Domain Domain Component Name}\
        [relation rename $::micca::UnionSuperclass Class Superclass]\
        -using {Domain Domain Component Relationship} |
    relation extend % rstup NewValue string {
        [list [tuple extract $rstup Class] [tuple extract $rstup Instance]]} |
    relation eliminate % Class Instance Role |
    relation rename % Superclass Class Value Instance NewValue Value
} {} |%]
# puts [relformat $usupercomps usupercomps]

foreach subcont [relation body $usupercomps] {
    SpecifiedComponentValue create {*}$subcont
    dict unset subcont Value
    ClassComponentValue create {*}$subcont
}
```

Union Subclass Numbering

Since subclass instances stored as a union do not have a separate storage pool, the instance numbering of union subclass instances is not independent. We modify the instance numbers of union subclass instances in the initial instance population to match those of the containing superclass instances.

```
<<population: update union subclass numbering>>=
set usupers [pipe {
    FindUltimateSuperclasses $domain |
    findRelated % {~R48 UnionSuperclass}
} {} |%]
# puts [relformat [deRef $usupers] usupers]

while {[isNotEmptyRef $usupers]} {
    set subrefs [pipe {
        deRef $usupers |
        relation eliminate ~ Role |
        relation rename ~ Relationship Component |
        relation join ~ $::micca::SpecifiedComponentValue\
                $::micca::ClassInstance |
        relation rename ~ Class Superclass Instance SuperInstance |
        relation extend ~ srtup\
            Class string {[lindex [tuple extract $srtup Value] 0]}\
            Instance string {[lindex [tuple extract $srtup Value] 1]} |
        relation project ~ Domain Class Instance Number
    }]
    # puts [relformat $subrefs subrefs]

    # relvar updateper does not update identifying attributes and
```

```
    # Number is a secondary identifier.
    # So we have to do it the hard way.
    # And because Number is part of a secondary identifier, we must remove the
    # old tuples and the add in the new ones or we risk a duplicate identifier
    # situation.
    relvar minus ::micca::ClassInstance [pipe {
        relation eliminate $subrefs Number |
        relation join ~ $::micca::ClassInstance -using\
            {Domain Domain Class Class Instance Instance}
    }]
    relvar union ::micca::ClassInstance $subrefs

    set usupers [pipe {
        findRelated $usupers R44 ~R45 |
        deRef % |
        relation semijoin % $::micca::UnionSuperclass\
            -using {Domain Domain Class Class} |
        ::rosea::Helpers::ToRef ::micca::UnionSuperclass %
    } {} |%]
}
```

Chapter 20

Handling Configuration Errors

One of the design elements of the configuration DSL is the use of the `micca` platform model to hold the configuration information. This design approach allows us to encode the rules for what is a valid domain configuration into the data constraints. Any attempt to define domain elements that violate those rules will be caught at the end of the relvar transaction that end the `configure` command.

The problem with this approach is that the error messages the are returned refer to the classes of the platform model not those of the domain we were attempting to define.

We will mitigate this problem by capturing the result returned by TclRAL and turning it into error messages that are more meaningful to the task of configuring a domain. Unfortunately, the messages produced by TclRAL are intended to be human readable, so we will have to parse them into a form that is easier to deal with programmatically. Fortunately, the messages are very regular in structure so the parsing code need not be very sophisticated.

Our strategy is to extract only the essential information from the error message and then map the information onto a more meaningful error message. Part of what we want to include in the error message is the values from the tuples failing the constraint. These values contain information the user entered rather than the abstractions of the platform model.

The essential information mapping is shown below. Naturally enough, we hold it in a relvar.

```
<<config data>>=
relvar create Config_DataError {
    Relationship        string
    RefClass            string
    RefType             string
    Format              string
} {Relationship RefClass RefType}
```

The **Relationship** attribute is the name of the relationship in the platform model that failed the constraint check. The **RefClass** attribute is the platform model class that has the tuples that failed the constraint check. The **RefType** attribute defines the way in which the constraint failed. The **Format** attribute is a string that contains the text of the error message that we want to display. The text in the **Format** attribute may also contain variable references to attributes in the tuple of **RefClass** that failed. The variable references in the **Format** will be substituted with the values from the failing tuples in order to give specifics of the error back to the user.

Implementation

The design of the `HandleConfigError` procedure is shown below. It is a nested iteration over the lines of the error result, pulling off the constraint violation details and then iterating over the tuples that were found in the violation.

```
<<config commands>>=
proc HandleConfigError {result} {
    set lines [split [string trimright $result] \n]
    set nlines [llength $lines]
```

```
    set lineno 0
    upvar #0 ::micca::@Config@::errcount errcount
    while {$lineno < $nlines} {
        <<HandleConfigError: examine one failure>>
    }

    return
}
```

The information in the TclRAL error message is simple enough and well structured enough that a regular expression can be used to match and extract the interesting parts.

```
<<HandleConfigError: examine one failure>>=
set line [lindex $lines $lineno]
incr lineno
if {[regexp {^for[^:]+([^(]+)\(.+\), in relvar (.+)$} $line\
        match rnum refclass]} {
    set rnum [namespace tail $rnum]
    set refclass [namespace tail $refclass]

    # Now iterate over the "tuple" lines that follow the constraint message.
    while {$lineno < $nlines} {
        set tupline [lindex $lines $lineno]
        if {[regexp {^tuple {(.+)} (.+)$} $tupline match tuple phrase]} {
            incr lineno
            incr errcount
            <<HandleConfigError: examine one tuple>>
            <<HandleConfigError: format error message>>
        } else {
            break
        }
    }
} elseif {[regexp {procedural contraint, "([^"]+)", failed} $line match\
        constraint]} {
    # There is only one procedural constraint, R74C.
    # If an error is detected in the procedural constraint script,
    # messages will be printed there.
    log::error $result
    incr errcount
} else {
    log::error $result
    incr errcount
}
```

We match the phrase in the tuple message to create an enumeration of the types of constraint violations. These are just a bit simpler to handle and look up.

```
<<HandleConfigError: examine one tuple>>=
if {[string match {is not referenced*} $phrase]} {
    set reftype notrefed
} elseif {[string match {references no*} $phrase]} {
    set reftype refnone
} elseif {[string match {is referenced by multiple*} $phrase]} {
    set reftype refedmult
} elseif {[string match {*to by multiple*} $phrase]} {
    set reftype multrefed
} elseif {[string match {is not referred to*} $phrase]} {
    set reftype notrefto
} else {
    log::error "unknown constraint phrasing, \"$phrase\""
    continue
}
```

Finally, we look up the format information and generate an error message. Using the `dict with` command allows us to take the tuple value from the error message, treat it like a dictionary and get the values into a Tcl variable. The `subst` command then will perform the variable substitutions in the format string.

```
<<HandleConfigError: format error message>>=
set cde [relvar restrictone Config_DataError Relationship $rnum RefClass\
        $refclass RefType $reftype]
if {[relation isnotempty $cde]} {
    dict with tuple {
        log::error [subst -nocommands [relation extract $cde Format]]
    }
} else {
    log::error "$line\n$tupline"
}
```

We now need to enumerate all the platform model constraints that can be violated and supply messages that provide less abstract and more useful user error messages. Because of the way the configuration data is stored into the platform model classes, not all violations are even possible.

```
<<config data>>=
relvar insert Config_DataError {
    Relationship        R20
    RefClass            Class
    RefType             notrefed
    Format              {in domain, $Domain, class, $Name,\
                        has no class components}
} {
    Relationship        R20
    RefClass            ClassComponent
    RefType             refnone
    Format              {in domain, $Domain, class, $Class, does not exist}
} {
    Relationship        R21
    RefClass            ClassComponent
    RefType             notrefto
    Format              {in domain, $Domain, class, $Class,\
                         component, $Name, failed to be completed created}
} {
    Relationship        R41
    RefClass            Relationship
    RefType             notrefed
    Format              {in domain, $Domain, relationship, $Name,\
                         failed to be completed created}
} {
    Relationship        R41
    RefClass            ClassRole
    RefType             refnone
    Format              {in domain, $Domain, relationship, $Relationship,\
                         $Class, participates in the relationship as a $Role,\
                         but the class was not defined}
} {
    Relationship        R54
    RefClass            MultipleAssigner
    RefType             refnone
    Format              {in domain, $Domain, relationship, $Association,\
                         the multi-assigner identifying class, $Class,\
                         does not exist}
} {
    Relationship        R58
    RefClass            StateModel
    RefType             refnone
    Format              {in domain, $Domain, the state model for, $Model,\
```

```
                         defines, $InitialState, the default initial state,\
                         but no such state exists}
} {
    Relationship         R70
    RefClass             TransitionPlace
    RefType              refnone
    Format               {in domain, $Domain, state model,\
                         $Model, event, $Event, causes a transition\
                         out of state, $State, but state $State does not\
                         exist}
} {
    Relationship         R72
    RefClass             StateTransition
    RefType              refnone
    Format               {in domain, $Domain, in the state model for,\
                         $Model, for the transition,\
                         $State - $Event -> $NewState, state $NewState does\
                         not exist}
} {
    Relationship         R86
    RefClass             DeferredEvent
    RefType              notrefed
    Format               {in domain, $Domain, class, $Model,\
                         defines event, $Event as polymorphic, yet\
                         $Model is not a superclass}
} {
    Relationship         R87
    RefClass             StateModel
    RefType              notrefed
    Format               {in domain, $Domain, class, $Model,\
                         there are no events defined for the state model}
} {
    Relationship         R87
    RefClass             TransitioningEvent
    RefType              refnone
    Format               {in domain, $Domain, class, $Model,\
                         event, $Event, is defined, but the class has no\
                         state model: possible inherited polymorphic event}
} {
    Relationship         R101
    RefClass             ElementPopulation
    RefType              refnone
    Format               {for domain population, $Domain, class or assigner,\
                         $Element, does not exist}
} {
    Relationship         R101
    RefClass             Population
    RefType              notrefed
    Format               {for domain population, $Domain, no values for the\
                         domain elements are specified}
} {
    Relationship         R103
    RefClass             ClassComponentValue
    RefType              refnone
    Format               {for domain population, $Domain, class, $Class,\
                         instance, $Instance, sets the value of\
                         $Component to $Value,\
                         but $Component is not a known component of $Class"}
} {
    Relationship         R103
    RefClass             ClassInstance
    RefType              notrefed
```

```
    Format              {for domain population, $Domain, class, $Class,\
                         instance, $Instance, does not have any values for\
                         its class components}
}
```

Chapter 21

Helper Commands

In this section we present a set of commands that factor common processing used in a number of areas. These commands are placed in a separate namespace which is imported into most of the other package namespaces.

```
<<helper commands namespace>>=
namespace eval Helpers {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path {::micca ::rosea::InstCmds}

    <<helper data>>
    <<helper commands>>
}
```

DeclError

All error notification in the package is consolidated in the `DeclError` procedure.

Implementation

The `DeclError` procedure locates a format string based on the error code and applies its arguments to it. We use the `::throw` command to raise the error to insure that we have consistent error code information for the package.

```
<<helper commands>>=
namespace export DeclError

proc DeclError {errcode args} {
    variable errFormats
    set errmsg [format [dict get $errFormats $errcode] {*}$args]
    tailcall throw [list MICCA $errcode {*}$args $errmsg] $errmsg
}
```

The mapping of `errcode` values to format strings is held as package data in the helper namespace of the package.

```
<<helper data>>=
variable errFormats
set errFormats [dict create {*}{
    <<error code formats>>
}]
```

Generate Numbers

There are a number of classes that have attributes that are zero based sequential numbers. We want a convenient way to generate these ordinal numbers. Many times we want the number to start at zero within a set of attributes.

Implementation

```
<<helper data>>=
relvar create SeqNumbers {
    Domain string
    ClassName string
    Attrs list
    Number int
} {Domain ClassName Attrs}
```

```
<<helper commands>>=
proc GenNumber {domain class args} {
    set num [relvar restrictone SeqNumbers Domain $domain ClassName $class\
        Attrs $args]
    if {[relation isempty $num]} {
        relvar insert SeqNumbers [list\
            Domain $domain\
            ClassName $class\
            Attrs $args\
            Number 0\
        ]
        return 0
    } else {
        set result [expr {[relation extract $num Number] + 1}]
        relvar updateone SeqNumbers sn [list Domain $domain ClassName $class\
            Attrs $args] {
            tuple update $sn Number $result
        }
        return $result
    }
}
```

Parse C Type Names

There a number of places where we require a “C” type name. We have adopted the strategy to separate out “C” identifiers from type names when parameters and other such things are required. In “C” itself, parameter and variables follow a declaration type syntax that is meant to be mnemonic of the way a variable is used in an assignment statement. We find it easier to keep identifiers and types separate, but it means we need to be able to verify that a given type name is syntactically correct and to compose a type name and identfier into a variable or parameter declaration.

Since this is strictly a parsing problem, we will rely on the parser tools that are part of `tcllib`. The parser tools take a PEG specification and generate a parser in Tcl. We will not discuss the grammar and other details of this parsing. Suffice it to say that we have formulated a PEG for parsing standard “C” type names. The output of the parser tools is a Tcl OO class that can parse a text string and return an AST of the type name. We will use the Tcl OO class as a superclass and provide two methods in our derived class that perform the required type name validation and generate a variable declaration from a type name and identifier.

Because of the `typedef` construct, “C” type names are ambiguous. A `typedef` introduces a new type name alias in “C”. We have no way to know what the compiler sees, so we have to make up some rules to disambiguate the situation.

1. Standard “C” types are recognized. This includes all the types in `stdint.h`, such as `int32_t`.
2. All types name of the `micca` run time are recognized. These types all start with `MRT_`.

3. Any type that starts with an upper case alphabetic character and includes any other alphanumeric characters and is followed by `_t` as a suffix is recognized as a valid type (N.B. no underscores are allowed in the name except the trailing `_t`).
4. As a last resort, the name can be asserted to be a "C" type name by using, `typename(<name>)`, where `<name>` is replaced by an valid "C" identifier.

Implementation

```
<<required packages>>=
package require oo::util
package require pt::util
```

```
<<helper commands>>=
if {[info exists ::iswrapped] && $iswrapped} {
    source [file join $::appdir typename typeparser.tcl]
} else {
    set tpfile [file join [file dirname [info script]]\
            typename typeparser.tcl]
    if {[file readable $tpfile]} {
        source $tpfile
    }
}
package require typeparser

oo::class create typeverifier {
    superclass typeparser

    method verifyTypeName {typename} {
        try {
            my parset $typename
            return true
        } on error {result} {
            puts [::pt::util error2readable $result $typename]
            return false
        }
    }

    method composeDeclaration {typename identifier} {
        my variable location

        set ast [my parset $typename]
        set location [lindex $ast 2]
        ::pt::ast topdown [mymethod TopWalker] $ast

        set decl [string cat\
            [string range $typename 0 $location]\
            " $identifier"\
            [string range $typename $location+1 end]\
        ]

        return [regsub -all -- {typename\s*\(\s*([[:alnum:]]\w*)\s*\)}\
            $decl {\1}]
    }

    method assignmentType {typename} {
        set ast [my parset $typename]

        set asgnType [dict create]
        set adnode [my SearchAST array_declarator $ast]
        if {[llength $adnode] != 0} {
            set adchildren [lassign $adnode type start end]
```

```
            if {[llength $adchildren] < 3} {
                error "data type, \"$typename\", has no dimension"
            }
            set dimnode [lindex $adchildren 1]
            lassign $dimnode dimtype dimstart dimend
            if {$dimtype eq "STAR"} {
                error "data type, \"$typename\", has indeterminate dimension"
            }
            dict set asgnType dimension\
                    [string range $typename $dimstart $dimend]

            set sqlnode [my SearchAST specifier_qualifier_list $ast]
            set tsnode [my SearchAST type_specifier $sqlnode]
            lassign $tsnode node_spec spec_begin spec_end
            if {$node_spec ne "type_specifier"} {
                error "expected \"type_specifier\" in AST node: got $node_spec"
            }
            set dtname [string trim\
                    [string range $typename $spec_begin $spec_end]]
            # puts "base type for \"$typename\" = \"$dtname\""
            dict set asgnType base $dtname
            dict set asgnType type\
                    [expr {$dtname eq "char" ? "string" : "array"}]
        } else {
            dict set asgnType type scalar
        }

        return $asgnType
    }

    method TopWalker {ast} {
        my variable location

        lassign $ast nodetype start end
        if {$nodetype eq "pointer"} {
            set location $end
        } elseif {$nodetype eq "array_declarator"} {
            set location [expr {$start - 1}]
        }
        return $ast
    }

    method SearchAST {nodeName ast} {
        set children [lassign $ast type start end]
        # puts "node = $type, children = [llength $children]"
        if {$type eq $nodeName} {
            return $ast
        }
        set result [list]
        foreach child $children {
            set result [my SearchAST $nodeName $child]
            if {[llength $result] != 0} {
                break
            }
        }
        return $result
    }
}

typeverifier create typeCheck
namespace export typeCheck
```

Checking for Duplicates

```
<<helper commands>>=
proc CheckDuplicate {class entity args} {
    set ref [$class findById {*}$args]
    if {[isNotEmptyRef $ref]} {
        EntityError $entity DUPLICATE_ENTITY $args
    }
    return
}
```

```
<<error code formats>>=
DUPLICATE_ENTITY    {%s, identified by, "%s", already exists}
```

```
<<helper commands>>=
proc EntityError {entity error kvs} {
    foreach {key value} $kvs {
        append ids "$key = \"$value\", "
    }
    set ids [string trimright $ids {, }]
    set lastcomma [string last , $ids]
    if {$lastcomma != -1} {
        set ids [string replace $ids $lastcomma $lastcomma { and}]
    }
    tailcall DeclError $error $entity $ids
}
```

Checking for Existance

```
<<helper commands>>=
proc CheckExists {class entity args} {
    set ref [$class findById {*}$args]
    if {[isEmptyRef $ref]} {
        EntityError $entity MISSING_ENTITY $args
    }
    return
}
```

```
<<error code formats>>=
MISSING_ENTITY    {%s, identified by, "%s", does not exist}
```

Part V

Runtime Support

In the previous part of the book, we described the domain specific language (DSL) that is used to describe a domain to `micca`. From that description, code is generated and the generated code targets the run-time environment that is explained in this part of the book.

When translating an executable model, parts of the model can be mapped directly onto the implementation language. For example in a `micca` translation, class instances are referred to using a "C" pointer to the instance memory. This allows attribute access by simple indirection on the pointer, *e.g.* for a state activity, `self->Temp = 27`, can be used to update the `Temp` attribute of the instance.

Other aspects of mapping model execution to the implementation are not directly supported in the implementation language. Consider the execution of a state model. There is no intrinsic support for state machines in the "C" language. To dispatch events to state machines requires that we write additional "C" code to implement the execution rules of Moore state machines. We will gather that code into "C" functions and the functions will require data that must be structured in a particular way.

This part of the book is primarily concerned with showing the functions and data structures required to map model execution rules onto the implementation. Taken together, they define the details of how execution happens in a `micca` translated domain. We have purposely chosen a simple scheme of execution, namely, single threaded and event driven with callbacks to handle the domain specific computations. This choice enables this translation scheme to be used in small, embedded, "bare metal" computing technology. We will also produce a flavor of the run-time that can execute in a POSIX environment. This will allow us to target POSIX, if that is appropriate to the application, and also to use POSIX as a simulation environment for testing an embedded system. This turns out to be useful since debugging and other facilities tend to be much richer in the POSIX world than in the bare metal embedded world.

Once we know how the functions and data structures of the run-time code operate, we can show the code generation that takes the platform model data and converts it to "C" code. That is the subject of the next part of the book.

Chapter 22

Introduction to the Micca Run Time

To start, we enumerate the characteristics of the computing technology targeted by the `micca` run-time.

- The implementation language is "C".
- All data is held in primary memory and no automatic persistence of data to external storage is provided.
- All classes and other resources are allocated fixed size memory pools whose size is known at compile time. The run time code does not perform any dynamic memory allocation from a system heap (*e.g.* no calls to `malloc`).
- The address in memory of a class instance serves as an architecturally generated identifier for the instance and we use that identifier exclusively in the interfaces of the run-time code.
- No other constraints on the identification of class instances is provided, particularly, no enforcement of uniqueness based on attribute values is provided by the run-time. Translations are strongly encouraged to provide such enforcement, especially when the identifying attribute values are obtained from outside of a domain.
- The only support for asynchronous execution is that of an interrupt. Facilities are provided to allow interrupt service code to synchronize to the running background. There is no notion of semaphores, condition variables or any other mutual exclusion locking scheme other than that required to prevent interrupts from executing during small critical sections of code involving a single data structure.
- Background execution is single threaded. There is no notion of *task* or *process* or *thread* as those terms are usually defined for computer operating systems. There is no notion of processor context, context switching or process preemption. All execution is run-to-completion except as preempted by interrupts.
- The execution pattern is event driven with callbacks that run to completion and perform the required computation. Given the single threaded nature of sequencing execution, callbacks that run for longer than the desired response latency time of the system can be problematic. This scheme is targeted at applications that are *reactive* in nature, sensing external stimulus and responding to that stimulus, and *not* long-running computationally intensive transformations. The execution scheme is similar to that used for most graphical user interface applications.
- There is no notion of event priority. Events are dispatched in the order they are signaled[1].
- Events are dispatched within the context of a *thread of control*. Threads of control have an direct corollation with transactions on the domain data.
- Referential integrity is enforced at the end of each data transaction. This implies that any changes in the stored data for relationships is verified against the constraints implied by the relationship conditionality and multiplicity. Referential integrity is evaluated whenever there has been any change in class instances or relationship instances during a thread of control.
- Some of the constructs for the run-time depend upon the target processor. We will show three implementations, namely, the ARM® V7-M architecture, the TI® MSP430 architecture and the POSIX architecture.

[1] With the exception that self directed events are dispatched before non-self directed ones. However, this is one of the prescribed execution rules of state models.

As we have said before, this execution scheme is *not* intended for all applications. The computational demands of some applications are such that the single threaded, event driven, callback nature is simply inappropriate and another execution scheme should be used. However, for a large class of applications, this scheme works well and has many advantages in terms of simplicity of the implementation and the lack of shared information between concurrent execution threads.

Limitation of the Run Time

Although `micca` supplies everything needed to sequence execution and manage model data, there are several other components of a target platform that are **not** supplied by `micca`. Notably:

- Exception handling is system specific. The details of how processor exceptions are handled must be supplied by the project. The run-time provides the means to synchronize between exceptions and the background processing but nothing else.
- The run-time requires access to some type of timing resource. The interface to the timing resource is well defined but projects will have to supply code to manipulate the timing resource. Typically in an embedded context, a hardware timer is used. For a POSIX environment, code is included to time delayed event dispatch using `SIGALRM`.
- No support is provided to obtain the current time of day. Many embedded platforms have no timing facilities. If time of day is important to your project, then hardware should provide some battery backed up time of day clock hardware or you will have to provide some timing resource from which the time of day can be deduced.
- Access to hardware can require significant effort to implement. Projects are strongly encouraged **not** to access hardware directly from domain models. Direct hardware access limits the ability of the model to be run in a different environment for simulation and debugging. Rather, a well defined hardware access layer should be put into place and external operations defined by the domain models expressing their hardware access dependencies. Then bridging code may be created to map from the external operations of the domain to the hardware access layer. This scheme is **not** supplied by `micca` and projects will have to create the hardware access to provide a complete target platform.
- The `micca` run-time provides simple linear iteration techniques for access to class instances and to navigate relationships. These simple techniques work well in many circumstances, but when the number of class instances is large other types of searching will be desired. `Micca` does not supply any techniques for hashing, binary searches, creating and maintaining indices or any of the many other ways that a class instance may be located. If these techniques are required for a particular domain, projects will have to supply their own code to accomplish it. Such code is not difficult to use in the `micca` environment since you have direct access to the code for the activities, but it is not supplied. `Micca` does guarantee the ordering of initial instances in the class storage pool, so often a binary search using the standard "C" library function, `bsearch`, will suffice if initial instances are defined in a sorted order.

Conditional Compilation

The run-time support the following "C" preprocessor symbols:

Important
It is important the code files for the run-time and all the domains that constitute an application be compiled with the same set of preprocessor symbol definitions.

`NDEBUG`
The run-time uses the standard `assert` macro and the assertions may be removed by defining this symbol.

`__ARM_ARCH`
Defining this symbol to be `7` compiles in the ARM Version 7 architecture specific code for executing sync requests from the foreground. This symbol is defined automatically by most compilers when compiling for armv7 architectures.

MRT_NO_NAMES
If defined, this symbol will exclude naming information about classes, relationships and other domain entities from being compiled in. For small memory systems, strings can consume a considerable amount of space and are usually only used during debugging.

MRT_NO_TRACE
Defining this symbol removes code from the run-time that traces event dispatch. Event dispatch tracing is important during testing and debugging but may be removed from the delivered system.

MRT_NO_STDIO
Defining this symbol insures that `stdio.h` is not included and no references are made to functions in the standard I/O library. This is useful for smaller embedded systems that cannot support the memory required by the standard I/O library.

MRT_TRANSACTION_SIZE
The value of this symbol sets the maximum number of relationships that can be modified during a data transaction. The default value is 64.

MRT_EVENT_POOL_SIZE
The value of this symbol sets the number event control blocks which are used for signaling events. The default value is 32. This number represents the maximum number of signaled events that may be *in flight* at the same time.

MRT_ECB_PARAM_SIZE
This value of this symbol set the maximum number of bytes that can be occupied by event parameters or sync function parameters. The default value is 32.

MRT_SYNC_QUEUE_SIZE
The value of this symbol defines the maximum number of synchronization requests from the foreground processing that may be outstanding at the same time. The default value is 10. This symbol represent the number of interrupts that may occur during the execution of a state activity.

MRT_INSTANCE_SET_SIZE
The value of this symbol is the maximum number of instance references that may be held in an instance reference set. The default value is 128.

MRT_INSTRUMENT
Defining this symbol includes code to print the function name, file name and line number for all functions generated by the code generator. This information forms a trace of executed functions.

MRT_INSTRUMENT_ENTRY
Defining this symbol can override the code that is placed at the beginning of each generated function for instrumentation. By default when `MRT_INSTRUMENT` is defined the following code is placed at the entry of each generated function:

```
printf("%s: %s %d\n", __func__, __FILE__, __LINE__) ;
```

MRT_DEBUG(...)
The `MRT_DEBUG` macro has the same invocation interface as `printf()`. If `MRT_INSTRUMENT` is defined, then `MRT_DEBUG` invocations will include the implied `printf` invocations. Otherwise, the implied `printf` invocations are removed from the code (*i.e.* `MRT_DEBUG` is defined as empty). If `MRT_INSTRUMENT` is defined, then the expansion of `MRT_DEBUG` may be overridden.

Chapter 23

The Main Program

In “C”, execution begins with the function called, `main`. `Micca` **does not provide** this function. Each application will have to construct a custom `main` function. The goals of `main` are to perform any required initialization and then to enter the event loop. Below we show an outline of what a `main` function would do.

```
int
main(
    int argc,
    char *argv[])
{
    /*
     * Hardware and other low level system initialization is usually done
     * first.
     */

    /*
     * Initialize the run-time code itself.
     */
    mrt_Initialize() ;

    /*
     * Initialize domains, bridges and any other code that might require access
     * to the facilities of the run-time code. Typically, each domain in the
     * system would have an "init()" domain operation and these can be invoked
     * here. Sometimes domain interactions are such that a second round of
     * initialization is required.  Bridges between domains may also require
     * that the initialization for a domain be done before the bridge can be
     * initialized. Once mrt_Initialize() has been invoked, domains may
     * generate events and do other model level activities.  Regardless of how
     * the initialization is accomplished, it is system specific and,
     * unfortunately, only temporally cohesive.
     */

    /*
     * Entering the event loop causes the system to run.
     */
    mrt_EventLoop() ;

    /*
     * It is possible that domain activities can cause the main loop to exit.
     * Here we consider that successful. Other actions are possible and
     * especially if the event loop is exited as a result of some unrecoverable
     * system error.
     */
    return EXIT_SUCCESS ;
```

```
}
```

The only hard and fast requirements are that `mrt_Initialize` must be called before any facilities of the run-time are used.

```
<<mrt external interfaces>>=
extern void
mrt_Initialize(void) ;
```

The `mrt_Initialize` function initializes the micca run-time. This function should be called early in the initialization of a system and must be invoked before using any run-time facilities such as signaling an event.

The implementation of `mrt_Initialize` is target platform specific. So we postpone showing its code until later when we discuss how the run-time is tailored to specific target computing platforms.

Runtime Use Cases

The `micca` runtime supports three use cases for causing the system to execute.

1. An infinite event loop.

   This is the primary intended mode of operation. Once the system is initialized, the event loop is entered and execution remains there forever.

2. Single thread of control.

   This use is intended to help integration of `micca` based code into an existing execution control scheme. This case comes up, for example, when a system is transitioned to being `micca` generated but there is existing code which controls execution sequencing. An existing event loop can be modified to periodically execute `micca` threads of control to coordinate the different execution schemes. This case is also useful for debuggers and testing. Executing a thread of control corresponds, roughly, to a *next* command in a conventional code debugger. A thread of control also roughly corresponds to a *unit* for testing purposes. Executing a complete thread of control insures that a data transaction on the domain is completed and the domain population is in a consistent state. The `bosal` test harness generated uses this mode extensively.

3. Single event.

   This use case is intended almost exclusively for debugging and simulation. Executing a single event is analogous to a *step* command in a conventional code debugger. Executing a single event may leave a domain population in an inconsistent state in the case where a thread of control consists of dispatching multiple events.

There is a fourth situation for controlling execution which arises in conjunction with error handling. As described in the following error handling section, it is possible to install an application specific error handler which is executed when an error occurs. An application specific error handler may use `setjmp/longjmp` to jump back to `main` when an error occurs. By default, all runtime errors are *fatal* in that they end up calling `abort()`. Such default behavior is not desired in many situations and the use of `setjmp/longjmp` is standard mechanism to handle errors detected at lower levels in the call tree. The runtime supports this usage by insuring that resuming event dispatch can continue if the error which the problem was corrected. This use case is also important for both integrating with other event loops and for testing purposes.

The following sections discuss the external functions provided to support the three use cases.

Event Loop

The figure below shows a simplified diagram of the flow of control in the event loop.

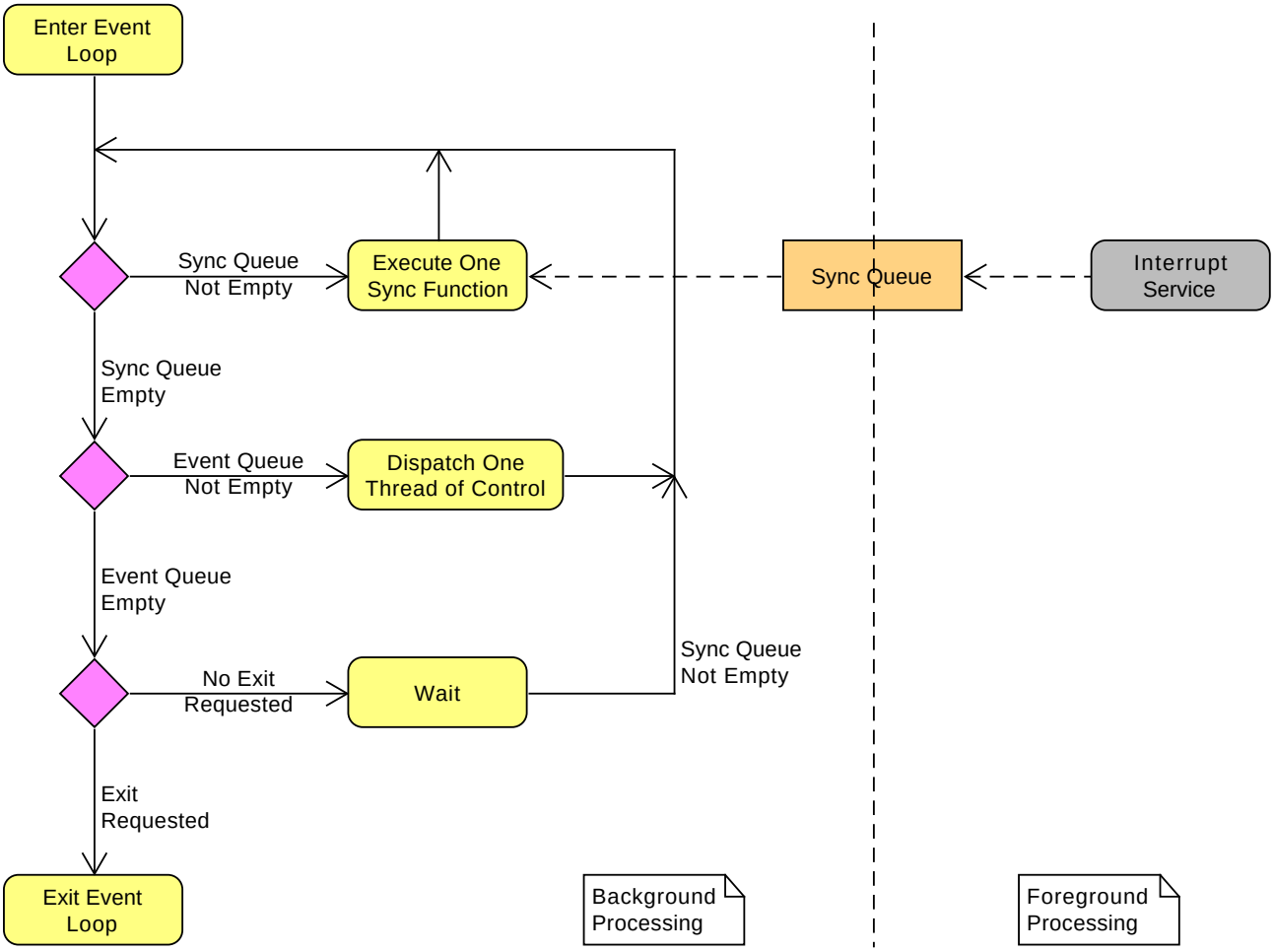


Figure 23.1: Micca Run Time Event Loop

Processing is divided between the background and foreground. Foreground processing happens as a result of an interrupt[1]. A queue is used to buffer and synchronize between the two execution realms. The background processing is single threaded. We will discuss the internal logic of processing an event later. For now, we make these points.

- If interrupts have occurred, we deal with their background synchronization first and completely. The exact manner used to synchronize between interrupts and the background is processor architecture dependent.
- Any event signaled by a sync function starts a thread of control. Only one thread of control event is dispatched after synchronizing with interrupts. Because we invoke the synchronization functions requested by all the interrupts before dispatching any event, interrupt synchronization has an effective higher priority than event dispatch.
- If the event queue becomes empty, we check if any activity has requested that the event loop terminate. If so, then flow continues to the remainder of the `main` program.
- If the event loop is not exited, the run-time waits until there is further work to be done. Note that any further work must arise from interrupt service and its synchronization to the background. Waiting is platform specific, but usually, for embedded systems, involves going into a sleep or low power mode. For the POSIX version of the run-time, the `pselect(2)` system call is used to suspend the process awaiting either a signal or a change in status of a file descriptor.

[1] In the POSIX environment, signals and changes in state of file descriptors serve the same role as an interrupt

As we shall see below, the above description of the event loop flow is somewhat simplified. There is one additional concept for which we must account. As stated previously, events originating outside of an instance context executing an activity start a new *thread of control*. This includes events that originate from a domain operation or are delayed events. Delayed events (even if self directed) are considered to be delivered by the run-time and hence outside of a state activity.

The thread of control terminates when the event that started it and all the other events that arose from the starting event have been dispatched. The event that starts a thread of control causes a transition that runs a state activity. That activity may signal other events, causing other transitions which may signal yet more events. At some point, the tree of state activity invocations does not produce any further events. When the entire set of events is dispatched, the thread of control ends. When a thread of control ends, the referential integrity of the data is checked. There is a close association between a transaction on the domain data and the bounds of the thread of control. At the boundary of threads of control, run-time evaluates requests to exit the event loop. The event loop will not normally be exited until the end of any ongoing thread of control

To keep the event dispatches that start a thread of control separate from those that run during a thread of control, the run-time keeps two queues. All events generated outside of a state machine activity are queued to the thread-of-control queued while those generated by state machine activities are queued to an immediate dispatch event queue. We will see those queues in the code below,

```
<<mrt external interfaces>>=
extern void
mrt_EventLoop(void) ;
```

The `mrt_EventLoop` function enters the micca run-time event loop and dispatches events to cause the system to run. Control remains in the event loop unless specifically requested to exit via the `mrt_SyncToEventLoop` function.

Important
The `mrt_EventLoop` function must not be called recursively, *i.e.* `mrt_EventLoop` is not to be invoked as part of an activity. The system must remain in the event loop until properly exited. Only after the event loop has been exited may it be re-entered.

To a first approximation, the event loop is an infinite loop. Most applications will enter the event loop after their initialization phase and remain there forever.

```
<<mrt external functions>>=
void
mrt_EventLoop(void)
{
    mrtExitEventLoop = false ;

    for (;;) {
        bool didtoc = mrtRunThreadOfControl() ; // ❶

        if (mrtExitEventLoop) {                 // ❷
            break ;
        } else if (!didtoc) {                   // ❸
            mrtWait() ;
        }
        /*
         * Else we ran a thread of control and weren't requested to exit
         * the event loop.
         */
    }
}
```

❶ Run one thread of control, recording whether there was any work done.

❷ Check if we have been requested to exit the event loop. We do this first because it is possible that something took us out of `mrtWait()` and requested to exit the event loop without actually signaling an event (and so no thread of control actually ran). For example, this happens when domains are run under a test harness.

❸ If no thread of control actually ran, then wait for something in the outside world to wake us up.

As we see, the flow of control loops, processing one thread of control at a time as long as it has not been requested to exit the loop.

Testing and other considerations mean that we may want to exit the event loop to gain control of the program. Exiting the event loop is controlled by a boolean variable.

```
<<mrt static data>>=
static bool mrtExitEventLoop ;
```

Rather than expose the variable directly, we provide a function to set it.

```
<<mrt external interfaces>>=
extern bool
mrt_SyncToEventLoop(void) ;
```

The `mrt_SyncToEventLoop` function will cause the event loop to exit after completing any ongoing thread of control. The function returns `false` if the event loop would not have exited and `true` otherwise, *i.e.* the function returns the exit control value before it was modified.

One use case for this function is in test code where the main test application will enter the event loop and some state activity will execute `mrt_SyncToEventLoop()` to return control back to the test application. There are other uses. For example, if the system detects a catastrophic situation, it may want to exit the event loop and allow the subsequent code to gracefully bring down the system or force the system to reset. The exact details are system specific, but this provides the means to gain control of the run-time execution for system specific circumstances.

```
<<mrt external functions>>=
bool
mrt_SyncToEventLoop(void)
{
    bool exitControl = mrtExitEventLoop ;
    mrtExitEventLoop = true ;
    return exitControl ;
}
```

Dispatching a Thread of Control

The second mechanism provided for controlling execution is to dispatch a single thread of control. Testing and some other situations are most easily handled if we can dispatch all the events in a single thread of control, waiting for the starting event to come along if necessary.

```
<<mrt external interfaces>>=
extern bool
mrt_DispatchThreadOfControl(
    bool wait) ;
```

wait
: If `wait` is `true`, and there is no thread of control event to dispatch, then the function blocks until one comes along and then runs the thread of control.

The `mrt_DispatchThreadOfControl` function attempts to run a single thread of control and then returns to the caller a boolean value to indicate if a thread of control was actually executed. Any ongoing thread of control is finished and is not considered in the return value. If there are no queued events that would start a thread of control, then the function waits until one is queued if the `wait` argument is `true` and then runs the thread of control that results. If `wait` is `false` and there is no thread of control ready to run, then the function returns `false` immediately. Since this function will wait for a thread of control event to arrive, it can be useful to wait for a delayed event or an event that arises from the external environment. It is possible that a foreground / background synchronization could occur that wakes up the processor but did not start a thread of control. In that case as well as when the `wait` argument is `false`, the function could return `false`.

Important
The `mrt_DispatchThreadOfControl` function must not be called recursively, *i.e.* `mrt_DispatchThreadOfControl` is not to be invoked as part of an activity. The system must remain in the event loop until properly exited. Only after the event loop has been exited may it be re-entered.

```
<<mrt external functions>>=
bool
mrt_DispatchThreadOfControl(
    bool wait)
{
    bool rantoc = mrtRunThreadOfControl() ;
    if (!rantoc && wait) {
        mrtWait() ;
        rantoc = mrtRunThreadOfControl() ;
    }
    return rantoc ;
}
```

We also provide an interface to dispatch one event and then return to the caller.

```
<<mrt external interfaces>>=
extern bool
mrt_DispatchSingleEvent(void) ;
```

Note
The `mrt_DispatchSingleEvent` function is deprecated and will be removed in a future release. The proper granularity for event dispatch is the *thread of control*.

The `mrt_DispatchSingleEvent` function may be invoked to process a single event and then returns to the caller. The return value for `mrt_DispatchSingleEvent` indicates if an event was actually dispatched. A return value of `false` indicates that the event queue is empty. This function ignores thread of control boundaries and so will dispatch the first available event even if that event starts a thread of control.

Important
The function `mrt_DispatchSingleEvent` is not to be invoked while in the event loop, *i.e.* `mrt_DispatchSingleEvent` is not to be invoked by domain activities. It is only valid to invoke this function if the program is not currently running under the control of the event loop.

This function can be used to control the dispatch of each event. This is *not* the recommended mechanism of running a `micca` based application, but sometimes it is necessary. Again, testing is sometimes best accomplished by letting a test program control event-by-event dispatch. Another use case involves situations where `micca` generated domains must be integrated with legacy code. The legacy code will probably have its own execution loop and we will have to dispatch `micca` events under control of the legacy execution scheme. This is not an ideal situation, but when migrating an older system to a `micca` translated system a transition period is unavoidable. We provide the ability to micro-manage the event dispatch with the full realization that the capablity will be abused by some projects.

```
<<mrt external functions>>=
bool
mrt_DispatchSingleEvent(void)
{
    bool dispatched = mrtEventQueueEmpty(&eventQueue) ?
            mrtDispatchTOCEvent() :
            mrtDispatchEventFromQueue(&eventQueue) ;

    if (mrtEventQueueEmpty(&eventQueue)) {                  // ❶
        mrtEndTransaction() ;
    }

    return dispatched ;
}
```

❶ We must make sure to enforce referential integrity whenever the thread of control is over as indicated by an empty event queue.

Running a Thread of Control

To run a thread of control, we take one event off the thread of control event queue and process it. Then, we simply finish dispatching any events off of the imminent event queue.

However, because an error handler may have transferred control outside of the run time, we must keep track of the state data transaction. We call the state of the system where the domain populations are consistent to be the *black* state (analogous to the usage of "black box" in testing terminology). When the domain populations are in flux, during a thread of control, we call this the *white* state (again, analogous to the usage in testing terminology). We keep track of the state using a boolean value.

```
<<mrt static data>>=
static bool mrtInWhiteState ;
```

In the case of executing threads of control, if we find ourselves in the **white** state, we must first get back to the **black** state before starting a new thread of control.

```
<<mrt static functions>>=
static bool
mrtRunThreadOfControl(void)
{
    if (mrtInWhiteState) {                          // ❶
        mrtFinishThreadOfControl() ;
    }

    bool dispatched = mrtDispatchTOCEvent() ;
    if (dispatched) {
        mrtFinishThreadOfControl() ;
    }

    return dispatched ;
}
```

❶ *N.B.* we must check if previous transaction finished before starting a new one.

A thread of control is finished by dispatching all the events on the imminent event queue and then ending the transaction.

```
<<mrt forward references>>=
static void mrtFinishThreadOfControl(void) ;
```

```
<<mrt static functions>>=
static void
mrtFinishThreadOfControl(void)
{
    while (mrtDispatchEventFromQueue(&eventQueue)) {
        // N.B. empty loop body
    }
    mrtEndTransaction() ;
}
```

Dispatching a Thread of Control Event

Before each event which starts a thread of control is processed, all the synchronization functions must be invoked. This allows the foreground to inject new data and events into the system when it has a consistent population.

```
<<mrt forward references>>=
static bool mrtDispatchTOCEvent(void) ;
```

```
<<mrt static functions>>=
static bool
mrtDispatchTOCEvent(void)
{
    while (mrtInvokeOneSyncFunction()) {
        ; /* empty loop body */
    }

    bool dispatched ;
```

```
    if (!mrtEventQueueEmpty(&tocEventQueue)) {
        mrtInWhiteState = true ;
        dispatched = mrtDispatchEventFromQueue(&tocEventQueue) ;
    } else {
        dispatched = false ;
    }
    return dispatched ;
}
```

The code for `mrtDispatchEventFromQueue` is shown later when we discuss event dispatch.

Chapter 24

Managing Data

Before we discuss the details of how execution is sequenced, we will show how data is managed by the run-time. Execution sequencing is directed at class instances and it will be helpful to understand how instances are stored before we get to discussing how they are operated upon.

The run-time provides functions to support the basic lifetime of instances. We need to be able to:

- Create instances synchronously as part of an activity.
- Create instances asynchronously as part of a event dispatch.
- Delete instances synchronously as part of an activity.
- Delete instances asynchronously when the instance enters a final state.

Instance Data

For every class (and assigner), the code generator will declare a "C" structure whose members contain all the attributes and other elements of the class. The structure of each class is, in general, different. Each class can have different attributes and relationships and this is reflected in members of the "C" structure that is used for each class. From the point of view of the run-time and the operations provided by the run-time, instances can be treated the same. The view of a class instance by the run-time is shown below.

```
<<mrt internal aggregate types>>=
typedef struct mrtinstance {
    struct mrtclass const *classDesc ;
    MRT_AllocStatus alloc ;
    MRT_StateCode currentState ;
    MRT_RefCount refCount ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Instance ;
```

classDesc
: A pointer to a class data structure. The class data structure contains information that is common to all instances of the class.

alloc
: A value that shows the allocation status of the memory for the instance. A value of 0, means the memory slot is not in use and can be allocated to a new instance. A negative value means the memory has been reserved but is not in active use as an instance. A positive value means the memory has been reserved and the instance is in active use.

`currentState`
: For those classes that have a state model, the `currentState` member holds a small integer number indicating the current state in which the instance resides. A value of `MRT_StateCode_IG` (-1) is used to show that the class does not have a state model.

`refCount`
: A counter value used to enforce referential integrity. At the end of a data transaction, this member is used as an accumulator of the number of times the instance is referenced in a relationship.

`name`
: An instance may have a name that it was given during the intial instance population. For instances not defined as part of the initial instance population, this member is set to `NULL`.

The `micca` code generator will facilitate this view of an arbitrary instance by inserting this structure as the first element of the "C" structure that is generated for each class. In this manner, a pointer to an arbitrary class instance can be cast to a pointer to a `MRT_Instance` with impunity.

Allocation Status Data Type

```
<<mrt internal simple types>>=
typedef int16_t MRT_AllocStatus ;
```

A class instance is nothing more than an element of a "C" array. The size of the array, and consequently the maximum number of instances of the class, is fixed at compile time. The `alloc` member of the instance structure is used to keep track of the status of the array elements as instances are created and deleted at run-time. We will also use this member to track *event-in-flight* errors. We will discuss this more below, but we need a way to insure that events that have been signaled are not delivered to instances that have been deleted.

Current State Data Type

```
<<mrt interface simple types>>=
typedef int8_t MRT_StateCode ;
```

The data type for the `currentState` member is just a small integer. By specifying 8 bits we limit the number of states of a state model to 127. That is an enormous number of states for a class state model. We use negative state numbers to indicate the non-transitioning actions that may occur when an event is dispatched.

```
<<mrt interface constants>>=
#define MRT_StateCode_IG    (-1)
<<mrt internal constants>>=
#define MRT_StateCode_CH    (-2)
```

We will also have need for a counter used in the enforcement of referential integrity.

Reference Count Data Type

```
<<mrt internal simple types>>=
typedef uint8_t MRT_RefCount ;
```

Class Data

All the behavior of data management and execution sequencing is completely determined by the values contained in the data structures supplied to the various functions of the run-time. This is distinct from some software architecture mechanisms that use generated code from a model compiler to implement some capabilities. Being completely data driven and separately compiled is a design goal of the run-time.

Since the behavior of all instances of a given class is the same, each class has a data structure that contains all the class invariant information.

First, we consider class attributes. We will have need to obtain attribute values in a generic way. From the platform model, we see that there are two types of attributes, independent and dependent. We will need to encode that difference.

Attribute Type Data Type

```
<<mrt internal simple types>>=
typedef enum {
    mrtIndependentAttr,
    mrtDependentAttr
} MRT_AttrType ;
```

Dependent attributes are those computed by some formula. The code generator will enclose the formula in a function that we can invoke to obtain the attribute value.

Attribute Formula Data Type

```
<<mrt internal aggregate types>>=
typedef void MRT_AttrFormula(
    void const *const self,
    void *const pvalue,
    MRT_AttrSize vsize) ;
```

`self`
A pointer to the instance upon which the formula function is invoked.

`pvalue`
A pointer to the memory where the formula result is to be placed.

`vsize`
The number of bytes pointed to by `pvalue`.

Formula functions will be invoked with a pointer to the instance, a pointer to where the result is to be stored and the size of the result area in bytes.

The description of an attribute is then a discriminated union.

```
<<mrt internal aggregate types>>=
typedef struct mrtattribute {
    MRT_AttrSize size ;
    MRT_AttrType type ;
    union {
        MRT_AttrOffset offset ;
        MRT_AttrFormula *formula ;
    } access ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Attribute ;
```

`size`
The number of bytes of memory occupied by the attribute value.

`type`
The type of the attribute, independent or dependent.

`access`
A union whose value depends upon the value of the `type` member.

`offset`
For `mrtIndependentAttr` types, the `offset` member contains the offset in bytes from the beginning of the instance memory to the attribute.

formula
For `mrtDependentAttr` types, the `formula` member contains a pointer to a function that computes the value of the attribute.

name
The name of the attribute.

To manage all the aspects of class instances, we need information on memory allocation, event dispatch, relationships in which the instances participate and many other pieces of information.

```
<<mrt internal aggregate types>>=
typedef struct mrtclass {
    struct mrtinstallocblock *iab ;
    unsigned eventCount ;
    struct mrteventdispatchblock const *edb ;
    struct mrtpolydispatchblock const *pdb ;
    unsigned relCount ;
    struct mrtrelationship const * const *classRels ;
    unsigned attrCount ;
    MRT_Attribute const *classAttrs ;
    unsigned instCount ;
    struct mrtsuperclassrole const *containment ;

#       ifndef MRT_NO_NAMES
    char const *name ;
    char const *const *eventNames ;
#       endif /* MRT_NO_NAMES */
} MRT_Class ;
```

iab
A pointer to an instance allocation block (IAB). The IAB is used to dynamically allocation class instances.

eventCount
The total number of transitioning and polymorphic events to which the class responds.

edb
A pointer to an event dispatch block (EDB). The EDB is used to dispatch events to a state machine. For classes that do not have a state model, this value is set to `NULL`.

pdb
A pointer to a polymorphic dispatch block (PDB). The PDB is used to dispatch polymorphic events. For classes that have no polymorphic events, this value is set to `NULL`.

relCount
The number of relationships in which the class participates.

classRels
A pointer an array of relationship description pointers describing the relationships in which the class participates. The array has `relCount` elements.

attrCount
The number of attributes the class contains.

classAttrs
A pointer an array of descriptions for the attributes the class contains. The array has `attrCount` elements.

instCount
The number of instances of the class. This number represents the number of elements in the array that is used to store the class instances.

containment
: For classes that are union subclasses, this member points to a descriptor for the immediate superclass in the generalization. For other classes, the value is `NULL`.

name
: A pointer to a `NUL` terminated string containing the name of the class.

eventNames
: A pointer to an array of character pointers to the names of the class events. This information is used in tracing event dispatch.

When generically describing classes, the subclasses of union based generalizations pose a special situation. All other class instances are stored in an array. Union subclasses are stored in the structure of their related superclass. The essential information for a union subclass is the class of its ultimate superclass and the offset from the beginning of the superclass to where the instance is located. Note that a union subclass can be subject to repeated generalization of another union subclass. The ultimate superclass is the class at the top of the generalization hierarchy.

In the next section, the instance allocation block is described as we continue to define how data is managed by the run-time. Later, we describe the event dispatch block and polymorphic dispatch block when we discuss event dispatch.

Instance Allocation

There are three ways to create an instance:

1. Create an instance as part of an initial instance population.
2. Create an instance synchronously to the execution of some activity by invoking a function.
3. Create an instance asynchronously to the execution of some activity by sending an event.

Creating initial instances is handled during code generation using the population data specified when the domain is configured. The code generator arranges for initial instance values to be inserted as initializers of the class storage array. In this section, we are going to discuss synchronous instance creation. This is instance creation by a direct function invocation and when the function returns the instance is ready and available. Later we will discuss asynchronous instance creation which is instance creation by signaling an event.

Instance Allocation Block

The instances of a class are contained in a single array variable which serves as the memory pool for the instances. To support managing a pool of class instances, an **Instance Allocation Block**, or **IAB** for short, data structure is used to keep track of the memory pool. What we need is a data structure that describes the properties of the class instance memory pool.

```
<<mrt internal aggregate types>>=
typedef void (*MRT_InstCtor)(void *const) ;
typedef void (*MRT_InstDtor)(void *const) ;
```

```
<<mrt internal aggregate types>>=
typedef struct mrtinstallocblock {
    void *storageStart ;
    void *storageFinish ;
    void *storageLast ;
    MRT_AllocStatus alloc ;
    size_t instanceSize ;
    MRT_InstCtor construct ;
    MRT_InstDtor destruct ;
    unsigned linkCount ;
    MRT_AttrOffset const *linkOffsets ;
} MRT_iab ;
```

storageStart
A pointer to the beginning of the memory where the instance storage pool is located. This is the array allocated to hold the instances of a class.

storageFinish
A pointer to one element beyond the end of the instance storage pool for the class. This pointer may not be dereferenced, of course, but provides the boundary marker for the end of the pool.

storageLast
A pointer to the instance that was last allocated. This is used as the starting point for allocating the next instance.

alloc
The next value of the allocation counter to be assigned to a newly allocated instance. This member is used to give a unique number (modulo the maximum value that can be held in the data type) to each allocation of the array element where an instance is stored. The number is used to diagnose run-time analysis errors.

instanceSize
The number of bytes of memory occupied by an instance.

construct
A pointer to a constructor function. If there is no constructor defined for the class, then the value of this member may be set to `NULL`.

destruct
A pointer to a destructor function. If there is no destructor defined for the class, then the value of this member may be set to `NULL`.

linkCount
The number of link pointer containers in the instance.

linkOffsets
A pointer to an array of offsets to the link pointer containers in an instance. The array contains `linkCount` elements.

Instance creation and deletion also supports a very simplified notion of construction and destruction for the instances. This is no where near as complicated or full featured as something in C++. Constructors and destructors take no arguments but are implicitly supplied a pointer to the instance when invoked.

Constructors and destructors are primarily useful for when the instance has a more complicated data structure as an attribute, as might be the case if the attribute data type is user defined. If you need to do complicated construction of instances, the preferred method is to do that with an instance based operation or as part of a state activity for an asynchronously created instance.

Finding Instance Memory

Before we can create an instance, we need to find memory for it.

```
<<mrt forward references>>=
static MRT_Instance *
mrtFindInstSlot(
    MRT_iab *iab) ;
```

iab
A pointer to the instance allocation block for the class for which instance memory is to be allocated.

`mrtFindInstSlot` searchs for unused instance memory in the memory pool described by `iab`. It returns a pointer to the allocated memory if successful and `NULL` if no memory is available in the class pool.

The allocation algorithm is a simple sequential search starting at the last location that was allocated.

```
<<mrt static functions>>=
static MRT_Instance *
mrtFindInstSlot(
    MRT_iab *iab)
{
    assert(iab != NULL) ;
    assert(iab->storageLast < iab->storageFinish) ;
    /*
     * Search for an empty slot in the pool. Start at the next location after
     * where we last allocated an instance.
     */
    MRT_Instance *inst ;
    for (inst = mrtNextInstSlot(iab, iab->storageLast) ;
            inst->alloc != 0 && inst != iab->storageLast ;
            inst = mrtNextInstSlot(iab, inst)) {
        /* Empty Body */
    }
    /*
     * Check if we ended up on a slot that is free.
     */
    return inst->alloc == 0 ? inst : NULL ; // ❶

}
```

❶ If we wrap all the way around to where we started and still did not find an instance storage element whose `alloc` member was zero, then we have run out of space! That condition is indicated by returning `NULL`.

Finding the next element in the instance storage array involves performing the pointer arithmetic modulo the size of the array. Since the pool is allocated in a contiguous block of memory, we must wrap around the iterator when it passes the end of the storage pool. That is accomplished with the `mrtNextInstSlot()` function.

```
<<mrt forward references>>=
static inline void *mrtNextInstSlot(MRT_iab *iab, void *ptr) ;
```

The `mrtNextInstSlot` function returns the memory address of the next instance after the one pointed to by `ptr` in the class instance pool described by `iab`.

```
<<mrt static functions>>=
static inline void *
mrtNextInstSlot(
    MRT_iab *iab,
    void *ptr)
{
    ptr = (void *)((uintptr_t)ptr + iab->instanceSize) ; // ❶
    if (ptr >= iab->storageFinish) { // ❷
        ptr = iab->storageStart ;
    }
    return ptr ;
}
```

❶ Since the size of instance varies from class to class, we must take over scaling the pointer arithmetic by the size of the instance.

❷ Perform the wrap around if we cross over the boundary of the storage array.

Instance Containment

For most classes, the Instance Allocation Block describes everything needed about how instances are stored. However, classes that are subclasses of a union generalization do not have their own storage pool. Union subclass instances are stored as part of the instance structure of the containing superclass. The containment may be several levels deep as a union subclass can be the superclass for another union generalization. The `MRT_Class` data structure contains the necessary information to deal with this. The following function forms the basis for treating the union subclasses the same as other classes.

```
<<mrt forward references>>=
static MRT_iab *
mrtGetStorageProperties(
    MRT_Class const *const classDesc,
    size_t *offsetptr) ;
```

`classDesc`
A pointer to the class data for which storage properties are computed.

`offsetptr`
A pointer to a location where the instance offset is returned. If not `NULL`, then a value is returned via the pointer that is the offset in bytes from the beginning of an instance to where the class storage begins.

The `mrtGetStorageProperties` function returns a pointer to the IAB that describes the storage containment for the class. For non-union subclasses, this is simply the IAB that describes the class storage pool. For union subclasses, the returned IAB will be for the ultimate superclass that contains the class instances.

```
<<mrt static functions>>=
static MRT_iab *
mrtGetStorageProperties(
    MRT_Class const *const classDesc,
    MRT_AttrOffset *offsetptr)
{
    assert(classDesc != NULL) ;
    MRT_iab *iab = classDesc->iab ;
    MRT_AttrOffset instanceOffset = 0 ; // ❶
    for (struct mrtsuperclassrole const *container = classDesc->containment ;
            container != NULL ;
            container = container->classDesc->containment) {// ❷
        instanceOffset += container->storageOffset ;
        iab = container->classDesc->iab ;
    }
    if (offsetptr) {// ❸
        *offsetptr = instanceOffset ;
    }

    return iab ;
}
```

❶ For non-union subclasses, the offset to the beginning of the instance data is always zero. Only union subclasses are stored with their superclass and will have a non-zero offset.

❷ We iterate up the generalization hierarchy to find the ultimate superclass that contains the class instances. At each step of the iteration, we accumulate the relative offset of each contained instance.

❸ The offset is returned only if requested.

We will also find occasion to want to compute the index into the storage pool for a particular instance. Storage pool indices make convenient identifiers of a class instance outside of a domain. Again we must allow that the instance may be a union subclass instance.

```
<<mrt internal external interfaces>>=
extern MRT_InstId
mrt_InstanceIndex(
    void const *instance) ;
```

instance
A pointer to a class instance.

The `mrt_InstanceIndex` function returns the array index value of the instance in its storage pool.

```
<<mrt external functions>>=
MRT_InstId
mrt_InstanceIndex(
    void const *instance)
{
    MRT_Instance const *instref = instance ;
    assert(instref != NULL) ;
    assert(instref->classDesc != NULL) ;

    MRT_AttrOffset offset ;
    MRT_iab *iab = mrtGetStorageProperties(instref->classDesc, &offset) ;
    assert(instance >= iab->storageStart && instance < iab->storageFinish) ;
    MRT_InstId index = (((uintptr_t)instance - offset) -
            (uintptr_t)iab->storageStart) / iab->instanceSize ;     // ❶
    return index ;
}
```

❶ The `instanceSize` member of the IAB is the number of bytes occupied by an instance. For union subclasses, the IAB returned by `mrtGetStorageProperties` will be for the enclosing supertype. So we have to make sure and subtract off the offset to the subclass before computing the index. For non-union subclass instances, `offset` here will be zero. Such are the complexities when taking over the pointer arithmetic and it is necessary to be generic.

The inverse of finding the instance index is to convert such an index into a reference to an instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstanceReference(
    MRT_Class const *const classDesc,
    MRT_InstId index) ;
```

classDesc
A pointer to the class description for the reference to be computed.

index
An array index into the class storage pool. If `index` out of bounds, then a fatal system error occurs.

The `mrt_InstanceReference` function returns a pointer to the instance in the class storage pool for the class described by `classDesc` and indexed by the value of `index`.

```
<<mrt external functions>>=
void *
mrt_InstanceReference(
    MRT_Class const *const classDesc,
    MRT_InstId index)
```

```
{
    void *instance = mrtIndexToInstance(classDesc, index) ;      // ❶
    if (instance == NULL) {
        mrtUnallocSlotError(index, classDesc) ;
    }

    return instance ;
}
```

❶ First, compute the instance from the index. Then we check if the instance is in use. For external callers, computing a reference that is not in use is a fatal error.

The common code used internally to compute an instance reference from an index doesn't test to see if the instance is in use. Internal callers take on that responsibility and most frequently it is not necessarily an error.

```
<<mrt static functions>>=
static void *
mrtIndexToInstance(
    MRT_Class const *const classDesc,
    MRT_InstId index)
{
    assert(classDesc != NULL) ;
    MRT_AttrOffset offset ;
    MRT_iab *iab = mrtGetStorageProperties(classDesc, &offset) ;
    void *instance = (void *)((uintptr_t)iab->storageStart +
            (index * iab->instanceSize) + offset) ;
    if (instance >= iab->storageFinish) {                        // ❶
        mrtNoInstSlotError(classDesc) ;
    }

    return (((MRT_Instance *)instance)->alloc <= 0) ? NULL : instance ;
}
```

❶ It is a fatal error to index outside the bounds of the class storage pool.

Creating an Instance

Creating an instance involves allocating memory to hold the instance. Normally, each class has its own pool of memory for its instance and the IAB and functions shown previously are used to manage that memory. However, instances of union-based subclasses have their memory allocated internal to the superclass instance. This means finding memory for a union subclass instance is different than finding memory for other classes. For the union subclass case, we must locate the correct place inside the instance memory for its related superclass instance. This difference is dealt with by using a separate function to create union subclass instances. Note there are two functions for each of the various types of creation operations.

We first start with a description of synchronously creating a normal, non-union-based instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState) ;
```

classDesc
A pointer to the class data for the instance to be created.

initialState
The state number into which the instance will be placed. For classes that do not have an associated state model this argument is ignored. For classes that do have an associated state model, the class will be created in the state given by `initialState`. If this argument is `MRT_StateCode_IG` or if the value given for `initialState` is not a valid state number for the class, then the instance is created in its default initial state.

`mrt_CreateInstance` allocates memory for an instance of the class described by `classDesc` and places the instance in the state given by `initialState`. No state activity is run as part of synchronous instance creation.

```
<<mrt external functions>>=
void *
mrt_CreateInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState)
{
    assert(classDesc != NULL) ;

    /*
     * Search for an empty slot in the pool.
     */
    MRT_iab *iab = classDesc->iab ;
    MRT_Instance *inst = mrtFindInstSlot(iab) ;
    if (inst == NULL) {
        mrtNoInstSlotError(classDesc) ;
    }
    /*
     * Record where we left off for the next allocation attempt.
     */
    iab->storageLast = inst ;
    /*
     * Initialize the memory for the instance.
     */
    mrtInitializeInstance(inst, classDesc, initialState) ;

    return inst ;
}
```

Union based subclasses pose special cases since the memory used to hold an instance is *not* allocated from a pool but rather is embedded in the instance of a superclass. A different function is required and additional parameters are used to determine the memory location of the subclass instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateUnionInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState,
    MRT_Relationship const *const genRel,
    void *super) ;
```

classDesc
A pointer to the class data for the instance to be created. The class described by `classDesc` must be a subclass of the relationship described by `genRel`.

initialState
The state number into which the instance will be placed. For classes that do not have an associated state model this argument is ignored. For classes that do have an associated state model, the class will be created in the state given by `initialState`. If this argument is `MRT_StateCode_IG` or if the value given for `initialState` is not a valid state number for the class, then the instance is created in its default initial state.

genRel
A pointer to a relationship description for the generalization in which the instance is a union-based subclass. The `relType` field of `genRel` must be `mrtUnionGeneralization`.

super
A pointer to the superclass instance where the subclass instance is to be created. `super` must be an instance of the superclass described by `genRel`.

`mrt_CreateUnionInstance` allocates memory for an instance of the union-based subclass described by `classDesc` and places the instance in the state given by `initialState`. No state activity is run as part of synchronous instance creation. The instance is created in place in the memory pointed to by `super`.

The two additional parameters required by `mrt_CreateUnionInstance` are need to provide the memory of the created instance (`super`) and to determine the offset into that memory where the instance storage is located (`genRel`).

```
<<mrt external functions>>=
void *
mrt_CreateUnionInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState,
    MRT_Relationship const *const genRel,
    void *super)
{
    assert(classDesc != NULL) ;
    assert(genRel != NULL) ;
    assert(super != NULL) ;
    assert(genRel->relType == mrtUnionGeneralization) ;

    if (genRel->relType != mrtUnionGeneralization) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    MRT_UnionGeneralization const *const gen =
            &genRel->relInfo.unionGeneralization ;
    /*
     * Verify that the new subclass is actually a subclass of the
     * generalization.
     */
    mrtFindUnionGenSubclassCode(classDesc, gen->subclasses,
            gen->subclassCount) ;
    /*
     * Verify the super class instance is truly of the super class.
```

```
     */
    MRT_Instance *superInst = super ;
    if (superInst->classDesc != gen->superclass.classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    /*
     * Compute the location of the union subclass instance
     * within the superclass instance.
     */
    void *inst = (void *)((uintptr_t)super + gen->superclass.storageOffset) ;
    /*
     * Initialize the memory for the instance.
     */
    mrtInitializeInstance(inst, classDesc, initialState) ;

    return inst ;
}
```

Initializing Instance Memory

Once we have identified the memory for an instance, we can make is ready to use. This involves zeroing out the memory and then initialzing the members of the `MRT_Instance` structure. Any link list references must be initialized to empty and the constructor is run if there is one.

```
<<mrt static functions>>=
static void
mrtInitializeInstance(
    MRT_Instance *inst,
    MRT_Class const *const classDesc,
    MRT_StateCode initialState)
{
    assert(inst != NULL) ;
    assert(classDesc != NULL) ;

    MRT_iab *iab = classDesc->iab ;
    assert(iab != NULL) ;
    /*
     * Start with a zeroed out memory space.
     */
    memset(inst, 0, iab->instanceSize) ;                                    // ❶
    inst->classDesc = classDesc ;
    /*
     * Mark the slot as in use.
     */
    inst->alloc = mrtIncrAllocCounter(iab) ;
    if (classDesc->edb != NULL) {
        assert(initialState < classDesc->edb->stateCount) ;

        inst->currentState = (initialState == MRT_StateCode_IG ||
            initialState >= classDesc->edb->stateCount) ?
                classDesc->edb->initialState : initialState ;               // ❷
    } else {
        inst->currentState = MRT_StateCode_IG ;                             // ❸
    }

    MRT_AttrOffset const *offsets = iab->linkOffsets ;                      // ❹
    for (unsigned count = iab->linkCount ; count != 0 ; count--) {
        MRT_LinkRef *link = (MRT_LinkRef *)((uintptr_t) inst + *offsets++) ;
        mrtLinkRefInit(link) ;
    }
```

```
    /*
     * Run the constructor if there is one.
     */
    if (iab->construct) {
        iab->construct(inst) ;
    }
    mrtMarkRelationship(classDesc->classRels, classDesc->relCount) ; // ❺
}
```

❶ Setting the memory of the instance to zero is very important. This insures that backward referencing pointers that are used in relationships are `NULL` and we depend upon that fact in the relationship linkage code.

❷ We use `MRT_StateCode_IG` as a special value to indicate that we want the instance created in its default initial state. We also protect against illegal values of the initial state and treat them the same.

❸ If a class does not have a state model, then we ignore the value passed in to the function and set the current state to ignored. This is a convenient value for such a situation.

❹ If a class contains any linked list terminus used for relationship navigation, they must be initialized to show that the linked list is empty.

❺ Creating an instance means it must be evaluated for referential integrity at the end of the data transaction. This is discussed in the next chapter.

One other important point here. There is a counter in the IAB that is incremented each time an instance is allocated and this value is used in the `alloc` member of the instance. This is another part of the strategy to detect an *event-in-flight* error. This is described further below. The effect of running this counter is that every instance gets a different `alloc` member value (modulo the size of the counter variable) The increment has one little catch. The counter is signed and the positive and negative values are used differently. Here, we make sure the value remains positive.

```
<<mrt forward references>>=
static inline MRT_AllocStatus mrtIncrAllocCounter(MRT_iab *iab) ;
```

```
<<mrt static functions>>=
static inline MRT_AllocStatus
mrtIncrAllocCounter(
    MRT_iab *iab)
{
    /*
     * Catch any overflow
     */
    iab->alloc = (iab->alloc == INT16_MAX ? 1 : iab->alloc + 1) ;
    return iab->alloc ;
}
```

Deleting an Instance

The function, `mrt_DeleteInstance`, is used to synchronously destroy an instance. Note that union subclass instances are treated the same as other class instances. Returning the memory for an instance is accomplished by setting the value of its `alloc` member to zero and this is the same in both cases, regardless of the fact that union subclass instance do not have their own memory pool.

```
<<mrt internal external interfaces>>=
extern void
mrt_DeleteInstance(
    void *instref) ;
```

instref
A pointer to an instance to be destroyed.

The `mrt_DeleteInstance` function deletes the instance given by `instref`.

Just as there was a distinction between synchronous and asynchronous instance creation, there is a similar distinction for destruction. Asynchronous destruction happens as a result of an instance entering an *final* state and that is discussed further below. Here we are dealing with synchronous destruction of an instance.

Deleting an instance also implies that when the class of the instance has references to other instance to link relationships, these linkages must also be deleted.

```
<<mrt external functions>>=
void
mrt_DeleteInstance(
    void *instref)
{
    MRT_Instance *inst = instref ;
    assert(inst != NULL) ;
    if (inst == NULL || inst->alloc <= 0) {          // ❶
        return ;
    }
    MRT_Class const *const classDesc = inst->classDesc ;
    assert(classDesc != NULL) ;
    MRT_iab *iab = classDesc->iab ;
    assert(iab != NULL) ;
    /*
     * Unlink the instance from its relationships.
     */
    mrtDeleteLinks(classDesc->classRels, classDesc->relCount, instref) ;
    /*
     * Run the destructor, if there is one.
     */
    if (iab->destruct) {
        iab->destruct(inst) ;
    }
    /*
     * Mark the slot as free.
     */
    inst->alloc = 0 ;
}
```

❶ Don't delete any instance that is unallocated or is awaiting the delivery of a creation event. In the later case, we would cause an event in flight error. Since `mrt_CreateInstanceAsync` returns a reference to the created instance, it is possible that something foolish like a request to immediately delete it could happen. Won't happen when using the embedded commands, but ...

Deleting an instance is a simple matter. First, unlink the instance from its relationships. If there is a destructor, it is run. The slot is free when its `alloc` member has a value of 0. But beware, for designs that have complicated relationships among the classes, instance deletion can be very complicated, requiring much care that the interdependencies among classes are properly preserved. That work is not done here! It is the responsibility of the analysis model to take any actions necessary to preserve data integrity when deleting an instance.

Iterating Over Class Instances

It is a common operation to iterate over the instances of a class. This is done both in the internals of the run-time as well as by domain activity code. For example, domain code has the need to search for instances meeting certain criteria such as the value of an attribute. For small instance populations, that search can be implemented by iterating over the instances of the class and making the comparison to the criteria. Consequently, the run-time provides a general means to iterate over the instances of a class.

The interface to instance iteration follows familiar patterns of Start, More, Next and Get. We have designed the interface in this way to avoid having to return `NULL` as a special value to indicate the end of the iteration. First, we need a data structure to hold the information we need for iteration.

```
<<mrt interface aggregate types>>=
struct mrtinstanceiterator ;
typedef struct mrtinstanceiterator MRT_InstIterator ;

<<mrt internal aggregate types>>=
struct mrtinstanceiterator {
    void *instance ;
    MRT_Class const *classDesc ;
    MRT_iab *iab ;
} ;
```

instance
A pointer to the memory of the next instance in the iteration.

classDesc
A pointer to the class descriptor across which the iteration occurs.

iab
A pointer to the Instance Allocation Block for the class across which the iteration is to be performed.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstIteratorStart(
    MRT_InstIterator *iter,
    MRT_Class const *const classDesc) ;
```

iter
A pointer to a class instance iterator that is used to record the state of the iteration.

classDesc
A pointer to the class description for the class across which the iteration will happen.

The `mrt_InstIteratorStart` function is called to initialize an instance iterator to iterate over instances of the class described by `classDesc`.

```
<<mrt external functions>>=
void
mrt_InstIteratorStart(
    MRT_InstIterator *iter,
    MRT_Class const *const classDesc)
{
    assert(iter != NULL) ;
    assert(classDesc != NULL) ;
    iter->classDesc = classDesc ;
```

```
    MRT_AttrOffset instanceOffset ;
    MRT_iab *iab = mrtGetStorageProperties(classDesc, &instanceOffset) ;
    assert(iab != NULL) ;

    iter->iab = iab ;
    iter->instance = (void *)((uintptr_t)iab->storageStart + instanceOffset) ;
    MRT_Instance *instref = iter->instance ;
    if (instref->alloc <= 0 || instref->classDesc != iter->classDesc) { // ❶
        mrt_InstIteratorNext(iter) ;
    }
    return ;
}
```

❶ The iterator is designed to check if the instance is actually in use. Here we check the first instance and if it is not being used advance the iterator onward.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstIteratorMore(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorMore` function returns a boolean value indicating if there are addition class instances which have not been visited. It returns `true` to indicate that the iterator references a valid class instance and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstIteratorMore(
    MRT_InstIterator *iter)
{
    assert(iter != NULL) ;
    return iter->instance < iter->iab->storageFinish ;
}
```

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstIteratorGet(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorGet` function returns a pointer to the class instance that is currently being visited in the iteration. It is not valid to invoke `mrt_InstIteratorGet` after the `mrt_InstIteratorMore` function has returned `false`. This function is analogous to dereferencing a pointer.

```
<<mrt external functions>>=
void *
mrt_InstIteratorGet(
    MRT_InstIterator *iter)
```

```
{
    assert(iter != NULL) ;
    return iter->instance ;
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstIteratorNext(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorNext` function advances the class instance iterator to the next valid class instance. To be a valid class instance, the instance memory slot must be allocated and in active use.

```
<<mrt external functions>>=
void
mrt_InstIteratorNext(
    MRT_InstIterator *iter)
{
    assert(iter != NULL) ;

    MRT_iab *iab = iter->iab ;
    assert(iab != NULL) ;

    while (iter->instance < iab->storageFinish) { // ❶
        iter->instance = (void *)((uintptr_t)iter->instance + iab->instanceSize) ;
        MRT_Instance *instref = iter->instance ;
        if (instref->alloc > 0 && instref->classDesc == iter->classDesc) {// ❷
            break ;
        }
    }
    return ;
}
```

❶ Advancing the instance pointer to past the storage pool for the class indicates that we have visited all the instances.

❷ It may seem strange to test that the class descriptor for the instance matches that of the class we are iterating across. However, we must handle the case of union based subtypes which can be reclassified. In that case, we are iterating across all potential subclass and only want to stop on the ones that match our original intent.

Instance Sets

Another common model level operation is to accumulate a set of class instances. Most action languages are based on the concept of *selecting* a set of class instances based on some criteria (*e.g.* matching some value of an attribute) and then iterating over that set to perform an operation on each instance. When the operation is simple, this construct can be translated into an interation over the class instances performing the criteria test and operation together. This is to say that just because the action language statement implies determining the contents of an instance set does not mean that it must necessarily be implemented that way. For simple situations involving searching for a criteria and then immediately performing some operation, it is not necessary to accumulate a set first and then iterate over the set.

However, there are times when obtaining a set is crucial and cannot be replaced by simple iteration over the class instances. For example, iterating across a many-to-many association may result in visiting the same destination instance multiple times. If we

are applying a non-idempotent operation to the destination instance, the what we really need is the set of related instances so as not to have duplicates. For those cases the run-time provides an instance set concept.

The design of the instance set has the goal of making it reasonable to allocate the sets as automatic variables on the stack. One could design the instance set to be a list of instance reference pointers. Unfortunately, such sets would occupy a significant amount of space. We want to be able to allocate instance sets as automatic variables to avoid the complication of managing the lifetime of set variables. Since our target environment is usually quite limited, large automatic variables are to be avoided.

The chosen design uses a bit vector. Since class instances are contained in an array, the index of a class instance in the storage pool array can be used as an index into a bit vector. The bit vector design trades off more processing to build and access the set, but makes many other set operations much easier. For example, since instance sets do not contain duplicates, the ability to arithmetically `OR` in a bit means we do not have to make any explicit tests to avoid inserting duplicate instances in the set.

The following is the data structure for an instance set.

```
<<mrt internal simple types>>=
typedef uint32_t MRT_SetWord ;

<<mrt interface aggregate types>>=
struct mrtinstanceset ;
typedef struct mrtinstanceset MRT_InstSet ;

<<mrt internal aggregate types>>=
struct mrtinstanceset {
    MRT_Class const *classDesc ;                                    // ❶
    MRT_SetWord instvector[(MRT_INSTANCE_SET_SIZE + MRT_SETWORD_BITS - 1) /
            MRT_SETWORD_BITS] ;                                     // ❷
} ;

<<mrt internal constants>>=
#define MRT_SETWORD_BITS    (sizeof(MRT_SetWord) * 8)
```

❶ An instance set records instances of one particular class. In other words, instance sets are typed to the class of instances they hold.

❷ The bit vector is allocated in `MRT_SetWord` words and we round up to insure we have enough space for all the instances.

Like all the other memory allocations in the run-time, we have to fix the maximum number of instance references that can be held in the set. By default we set that number to 128.

```
<<mrt interface constants>>=
#ifndef MRT_INSTANCE_SET_SIZE
#   define MRT_INSTANCE_SET_SIZE 128
#endif /* MRT_INSTANCE_SET_SIZE */
#if __STDC_VERSION__ >= 201112L
static_assert(MRT_INSTANCE_SET_SIZE > 0, "Instance set size must be > 0") ;
#endif /* __STDC_VERSION__ >= 201112L */
```

A function is provided to properly initalize an instance set variable.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetInitialize(
    MRT_InstSet *set,
    MRT_Class const *const classDesc) ;
```

set
A pointer to an instance set.

classDesc
A pointer to a class descriptor. The instance set will contain only instances of this class.

The `mrt_InstSetInitialize` function initialized the instance set data structure pointed to by `set` to prepare it to accept instances of the class described by `classDesc`.

```
<<mrt external functions>>=
void
mrt_InstSetInitialize(
    MRT_InstSet *set,
    MRT_Class const *const classDesc)
{
    assert(classDesc != NULL) ;
    set->classDesc = classDesc ;
    memset(set->instvector, 0, sizeof(set->instvector)) ;
}
```

The primary operation on instance sets is to add an instance.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetAddInstance(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be inserted into the set.

The `mrt_InstSetAddInstance` function inserts `instance` into the instance set pointed to by `set`. Attempts to add an instance already in the set are silently ignored. Attempts to add an instance that is not of the same class for which the set was initialized is also silently ignored.

```
<<mrt external functions>>=
void
mrt_InstSetAddInstance(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return ;
    }
    MRT_Instance *instref = instance ;
```

```
    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {          // ❶
        return ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
    }
    set->instvector[instid / MRT_SETWORD_BITS] |=
            (1 << (instid % MRT_SETWORD_BITS)) ;         // ❷
}
```

❶ Make sure we do not add instances of the wrong class. Instance indices are only unique within a given class.

❷ This bit twiddling selects the correct bit in the correct word of the bit vector. The correct word is given by the quotient of the index and the number of bits in a set vector word. The correct bit offset within a word is given by the modulus. We are using divide and modulus operations with the full expectation that the compiler will recognize, at some level of optimization, that `MRT_SETWORD_BITS` is a power of two and transform the divide and modulus into bitwise operations.

We also need a means to remove an instance from the set.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetRemoveInstance(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be inserted into the set.

The `mrt_InstSetRemoveInstance` function removes `instance` from the instance set pointed to by `set`. Attempts to remove an instance that is not in the set are silently ignored.

```
<<mrt external functions>>=
void
mrt_InstSetRemoveInstance(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return ;
    }
    MRT_Instance *instref = instance ;

    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {
        return ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
```

```
    }
    set->instvector[instid / MRT_SETWORD_BITS] &=
            ~(1 << (instid % MRT_SETWORD_BITS)) ; // ❶
}
```

❶ More bit twiddling. The bitwise AND of the one's complement of a mask clears in the result any bit that was set in the mask.

We also provide a test for membership in an instance set.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetMember(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be tested for set membership.

The `mrt_InstSetMember` function determines if `instance` is a member of the instance set pointed to by `set`. The instance must be of the same class as that associated with `set`. The function returns `true` is `instance` is in the set and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstSetMember(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return false ;
    }
    MRT_Instance *instref = instance ;

    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {
        return false ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
    }
    MRT_SetWord w = set->instvector[instid / MRT_SETWORD_BITS] ;
    MRT_SetWord mask = (1 << (instid % MRT_SETWORD_BITS)) ;

    return (w & mask) != 0 ;
}
```

We also supply set operations, starting with determining if the instance set is empty.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetEmpty(
    MRT_InstSet *set) ;
```

set
A pointer to an instance set.

The `mrt_InstSetEmpty` returns `true` if the instance set pointed to by `set` contains no elements and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstSetEmpty(
    MRT_InstSet *set)
{
    assert(set != NULL) ;
    MRT_SetWord *pvect = set->instvector ;
    while (pvect < set->instvector + COUNTOF(set->instvector)) {
        if (*pvect++ != 0) {
            return false ;
        }
    }

    return true ;
}
```

A function to determine the number of members of the instance set is provided

```
<<mrt internal external interfaces>>=
extern unsigned
mrt_InstSetCardinality(
    MRT_InstSet *set) ;
```

set
A pointer to an instance set.

The `mrt_InstSetCardinality` returns `true` if the instance set pointed to by `set` contains no elements and `false` otherwise.

```
<<mrt external functions>>=
unsigned
mrt_InstSetCardinality(
    MRT_InstSet *set)
{
    assert(set != NULL) ;
    unsigned card = 0 ;
    for (MRT_SetWord *pvect = set->instvector ;
            pvect < set->instvector + COUNTOF(set->instvector) ; pvect++) {
        MRT_SetWord w = *pvect ;
        MRT_SetWord mask = 1 ;
        for (unsigned bit = MRT_SETWORD_BITS ; w != 0 && bit != 0 ; bit--) {
            if ((w & mask) != 0) {
                card++ ;
                w &= ~mask ; // ❶
            }
            mask <<= 1 ;
```

```
        }
    }

    return card ;
}
```

❶ Once we count the bit, we clear it. This allows us to short circuit the loop if no other bits in the word are set.

A test for set equality is also supplied.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetEqual(
    MRT_InstSet *set1,
    MRT_InstSet *set2) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

The `mrt_InstSetEqual` returns true if instance set `set1` is equal to instance set `set2` and false otherwise. If `set1` and `set2` refer to different instance set, then false is returned.

```
<<mrt external functions>>=
bool
mrt_InstSetEqual(
    MRT_InstSet *set1,
    MRT_InstSet *set2)
{
    assert(set1 != NULL) ;
    assert(set2 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2->classDesc != NULL) ;

    if (set1->classDesc != set2->classDesc) {
        return false ;
    }

    MRT_SetWord *src1 = set1->instvector ;
    MRT_SetWord *src2 = set2->instvector ;
    while (src1 < set1->instvector + COUNTOF(set1->instvector)) {
        if (*src1++ != *src2++) {
            return false ;
        }
    }

    return true ;
}
```

Basic set operations are also provided. We start with the set union operation.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetUnion(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetUnion` function computes the set union of `set1` and `set2` storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the result set is the same set as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetUnion(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ | *src2++ ;
        }
    } else {
        memcpy(result->instvector, set1->instvector, sizeof(result->instvector)) ;
    }
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIntersect(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetIntersect` function computes the set intersection of `set1` and `set2` storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the returned set is the empty set of the same class as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetIntersect(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ & *src2++ ;
        }
    } else {
        memset(result->instvector, 0, sizeof(result->instvector)) ;
    }
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetMinus(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetMinus` function computes the set difference of `set1` minus `set2` (order is significant in this case) storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the result set is the same as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetMinus(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ & ~*src2++ ;
        }
    } else {
        memcpy(result->instvector, set1->instvector, sizeof(result->instvector)) ;
    }
}
```

Iterating Over Instance Sets

Once we have computed an instance set, we need some way to iterate over the resulting set. Since we have chosen a bit vector representation of the set, the iteration code is slightly more complicated as we must keep track of which word and bit offset we are currently referring to. Iterating over instance sets has one additional complication. It is possible that a class instance referenced in an instance set has been deleted in the time between when the instance set was accumulated and the time when the iteration across the set occurs. So, iterating over instance set must also guard against an instance that, although it may be a member of the set, has been deleted between the time the set was constructed and the time that the iteration occurs.

First, we start with the iterator data structure. This holds the state information we need to determine our place in the set.

```
<<mrt interface aggregate types>>=
struct mrtinstsetiterator ;
typedef struct mrtinstsetiterator MRT_InstSetIterator ;

<<mrt internal aggregate types>>=
struct mrtinstsetiterator {
    MRT_InstSet *set ;
    void *instance ;
    MRT_SetWord *vectorloc ;
    unsigned bitoffset ;
} ;
```

set
A pointer to the instance set across which the iteration will occur.

instance
A pointer to the current instance of the iteration.

vectorloc
A pointer to the word within the instance set where the current instance is located.

bitoffset
The bit offset into the word pointed to by `vectorloc` that represents the current instance of the iteration.

The operations on the iteration follow our usual pattern of Begin, More, Next and Get.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIterBegin(
    MRT_InstSet *set,
    MRT_InstSetIterator *iter) ;
```

set
A pointer to an instance set across which the iteration will occur.

iter
A pointer to an instance set iterator which will hold the state of the iteration.

The `mrt_InstSetIterBegin` initializes the set iterator pointed to by `iter` in order to iterate across the instance set pointed to by `set`. This function must be called first to prepare an iteration across an instance set.

```
<<mrt external functions>>=
void
mrt_InstSetIterBegin(
    MRT_InstSet *set,
    MRT_InstSetIterator *iter)
{
    assert(set != NULL) ;
    assert(iter != NULL) ;

    iter->set = set ;
    iter->vectorloc = set->instvector ;
    iter->bitoffset = 0 ;
    if ((*iter->vectorloc & 1) == 0) { // ❶
        mrt_InstSetIterNext(iter) ;
    } else {
        iter->instance = mrtIndexToInstance(iter->set->classDesc, 0) ;
```

```
        if (iter->instance == NULL) { // ❷
            mrt_InstSetIterNext(iter) ;
        }
    }
}
```

❶ Check if the first instance belongs to the set. If not, we want to advance the iterator until we have a valid set member.

❷ Check if the set instance is still actively being used. If not, we want to advance the iterator to one that is.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetIterMore(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterMore` function returns `true` if there are more instances to visit in the instance set referenced by `iter` and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstSetIterMore(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;
    return iter->instance != NULL ;
}
```

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstSetIterGet(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterGet` function obtains the current instance from the instance set referenced by `iter`. It is not valid to invoke this function after `mrt_InstSetIterMore` returns `false`.

```
<<mrt external functions>>=
void *
mrt_InstSetIterGet(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;
    assert(iter->instance != NULL) ;
    return iter->instance ;
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIterNext(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterNext` function advances the set iterator pointed to by `iter` to the next element of the set.

Advancing the iterator is a more complicated operation. We must account for the word boundaries in the bit vector as well as the number of bits in each of the vector words. Finally, we must also make sure the next element of the set is still a valid, allocated instance of the class.

```
<<mrt external functions>>=
void
mrt_InstSetIterNext(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;

    iter->bitoffset++ ; //❶
    if (iter->bitoffset >= MRT_SETWORD_BITS) {
        iter->vectorloc++ ;
        iter->bitoffset = 0 ;
    }
    while (iter->vectorloc <
            iter->set->instvector + COUNTOF(iter->set->instvector)) {
        if (*iter->vectorloc != 0) { // ❷
            MRT_SetWord mask = 1 << iter->bitoffset ;
            for ( ; iter->bitoffset < MRT_SETWORD_BITS ; iter->bitoffset++) {
                if ((*iter->vectorloc & mask) != 0) { // ❸
                    unsigned instindex =
                        (iter->vectorloc - iter->set->instvector) *
                        MRT_SETWORD_BITS + iter->bitoffset ; // ❹
                    void *instance = mrtIndexToInstance(iter->set->classDesc,
                            instindex) ;
                    if (instance != NULL) { // ❺
                        iter->instance = instance ;
                        return ;
                    }
                }
                mask <<= 1 ;
            }
        }
        iter->vectorloc++ ; // ❻
        iter->bitoffset = 0 ;
    }

    iter->instance = NULL ;
}
```

❶ Advance the iterator by one bit in the bit vector, accounting for running off the end of the vector word.

❷ A simple test determines if we can skip the entire word of bits.

❸ Check if we have found a set bit in the bit vector. This indicates that the corresponding instance is a member of the set.

4. Compute the index of the instance in bit vector. This is the same to the index of the instance in the storage array.
5. Verify that the instance was not deleted between the time the set was build and the time the iteration across the set happens.
6. Advance to the next word of the bit vector.

Chapter 25

Managing Referential Integrity

When the target translation platform is based on a Relational Database Management System (RDMS) or some other data architecture that supports the relational model of data (*e.g.* `rosea` or TclRAL), referential integrity checking comes with the underlying data management facilities. Such data management facilities constrain the values of data and move much of the data integrity validation away from the runtime code. Such is the motivation to use these facilities since they replace application code with declarative specifications of validity.

When targeting static-typed languages such as "C", translation platforms frequently do not offer any help in insuring referential integrity of the data as the system runs. Sometimes this limitation is understandable. It takes more code and data space to deal with referential integrity. For applications that do little dynamic instance creation or relationship changes, the risk that some code path results in a violation of referential integrity is much less. Nonetheless, execution platforms that do not provide any referential integrity checks impose a significant burden on the programmer and result in systems that are potentially less reliable.

In `micca`, we will support detection of referential integrity violations. As we will see, there is considerable complexity in achieving this goal. To be clear, this is what `micca` supports.

- Integrity constraints implied by the associations and generalizations supplied in the configuration DSL will be checked at run-time.
- Violations of referential integrity will result in a fatal system error. There is no support for rolling back data values to a known good state or ignoring the error and proceeding anyway.
- Integrity constraints are checked at the end of transactions on the data model. The duration of the transaction is not under direct program control, but there is an ongoing transaction when a thread of control is executing. A thread of control is defined to be the duration of the dispatch of the set of events initiated by an event signaled from outside of a state activity or by the delivery of a delayed event. We discuss the thread of control concept in more detail below when we deal with managing execution.

Because of our design choice to use the address of a class instance as an identifier, to manage referential integrity we need:

- Functions to create, delete and update reference pointers that implement relationships.
- Data structures and values that encode the specifications of the relationships from the platform model.
- Run-time code that, using the relationship specifications, evaluates whether referential integrity was maintained.

In keeping with our goal that the run-time be completely data driven, the code generator will supply the necessary information about all the associations and generalizations in the domain. This will allow us to manipulate the pointer values used in the relationships and to check that the result does not violate any of the multiplicity and conditionality constraints specified for the relationship.

We will show how this is accomplished by:

1. Showing the data structures used to describe relationships.
2. Explaining how transactions work and how referential integrity is checked.
3. Giving code to relate and unrelate instances in a relationship that operates with the integrity check.

Describing Relationships

In this section we present the data structures that are supplied by the `micca` code generator to describe the properties of a relationship. You can think of these data structures as an implementation version of the various classes in the platform model that deal with class relationships. The code generator takes data from the platform model population that was created during domain configuration and generates initialized "C" variables of the types described below.

From the platform model, we know that there are four different types of relationships.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtSimpleAssoc,
    mrtClassAssoc,
    mrtRefGeneralization,
    mrtUnionGeneralization
} MRT_RelType ;
```

The combination of the conditionality and multiplicity of a relationship can be encoded in four values.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtAtMostOne = 0,
    mrtExactlyOne,
    mrtZeroOrMore,
    mrtOneOrMore
} MRT_Cardinality ;
```

There are also three different ways that pointer references are stored.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtSingular,
    mrtArray,
    mrtLinkedList
} MRT_RefStorageType ;
```

For singular references, a single pointer member is allocated in the class structure.

For array reference storage, the class structure member is a counted array of the type shown below:

```
<<mrt internal aggregate types>>=
typedef struct mrtarrayref {
    MRT_Instance * const *links ;
    unsigned count ;
} MRT_ArrayRef ;
```

For linked list reference storage, the referring class has a set of link pointers that serve as the terminus for the linked list. In addition, the referenced class has a set of link pointers that enable it to be linked into the list of instances headed by the referring class.

```
<<mrt internal aggregate types>>=
typedef struct mrtlinkref {
    struct mrtlinkref *next ;
    struct mrtlinkref *prev ;
} MRT_LinkRef ;
```

Association Participants

To characterize the role a participant class in an association plays, the following data is needed.

```
<<mrt internal aggregate types>>=
typedef struct mrtassociationrole {
    MRT_Class const *classDesc ;
    MRT_Cardinality cardinality ;
    MRT_RefStorageType storageType ;
    MRT_AttrOffset storageOffset ;
    MRT_AttrOffset linkOffset ;
} MRT_AssociationRole ;
```

classDesc
A pointer to the class data for the participant.

cardinality
The encoding of the conditionality and multiplicity of the relationship. This encoding is from point of view of how many references to the participant that are made by the other participant in the relationship.

storageType
An encoding of the structure of how the reference pointers are stored.

storageOffset
The offset in bytes from the beginning of an instance to where the reference pointers for the relationship are stored.

linkOffset
For participants whose value of `storageType` is `mrtLinkedList`, this member gives the number of bytes from the beginning of the target participant instance where the linked list pointers are stored. For other values of `storageType` the value is ignored.

Simple Associations

Simple associations are made up of a source and target. The forward direction of a simple association is from the source instance to the target instance.

```
<<mrt internal aggregate types>>=
typedef struct mrtsimpleassociation {
    MRT_AssociationRole source ;
    MRT_AssociationRole target ;
} MRT_SimpleAssociation ;
```

Class Based Associations

For class based associations, there is another role played by the associator class. Associator classes always have two singular references to the participant classes.

```
<<mrt internal aggregate types>>=
typedef struct mrtassociatorrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset forwardOffset ;
    MRT_AttrOffset backwardOffset ;
    bool multiple ;
} MRT_AssociatorRole ;
```

classDesc
A pointer to the class data for the participant.

forwardOffset
The offset in bytes from the begining of an instance of the associator class to the forward reference pointer. The forward reference pointer refers to a class instance that serves as the target of the association.

backwardOffset
The offset in bytes from the begining of an instance of the associator class to the backward reference pointer. The backward reference pointer refers to a class instance that serves as the source of the association.

multiple
A boolean value that specifies if the same instances are allowed to be referenced by multiple association classes.

A class based association has descriptive information on all three class roles.

```
<<mrt internal aggregate types>>=
typedef struct mrtclassassociation {
    MRT_AssociationRole source ;
    MRT_AssociationRole target ;
    MRT_AssociatorRole associator ;
} MRT_ClassAssociation ;
```

Superclasses in a Generalization

The information needed to describe a superclass in a generalization is the same for both reference and union based generalizations.

```
<<mrt internal aggregate types>>=
typedef struct mrtsuperclassrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset storageOffset ;
} MRT_SuperClassRole ;
```

classDesc
A pointer to the class data for the participant.

storageOffset
The offset in bytes from the beginning of a superclass instance to where subclass information is stored. The information stored at `storageOffset` with either be a reference to the subclass instance (for reference based generalizations) or the union of the subclass structures in the generalization (for union based generalizations).

Reference Generalizations

For generalization relationships implemented by pointer references, the superclass implements its reference to a subclass by holding a simple pointer to the subclass instance. It is possible for the superclass to determine the type of the related subclass by simply following the reference to the subclass and then examining the `classDesc` member of the instance. data structure.

The information needed to describe the role of the subclass in a reference generalization is:

```
<<mrt internal aggregate types>>=
typedef struct mrtrefsubclassrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset storageOffset ;
} MRT_RefSubClassRole ;
```

classDesc
A pointer to the class data for the participant.

`storageOffset`
: The offset in bytes from the beginning of a reference subclass instance to where the reference to its related superclass is stored. The structure member found at `storageOffset` from the beginning of the instance is a pointer to the superclass instance.

A reference generalization relationship can be described by its superclass role information and the set of subclass roles for the subclasses that participate in the generalization. We store the subclass role set as a count and pointer to a corresponding array.

```
<<mrt internal aggregate types>>=
typedef struct mrtrefgeneralization {
    MRT_SuperClassRole superclass ;
    unsigned subclassCount ;
    MRT_RefSubClassRole const *subclasses ;
} MRT_RefGeneralization ;
```

Union Generalizations

A union generalization results in subclass instances being held as a union in the superclass instance structure. From the superclass perspective, the information needed to describe where the subclass is stored is the same as for a reference supertype. The only difference is that `storageOffset` is an offset to the union holding the subclass rather than a reference pointer to a subclass.

Subclass instances held in union generalizations do not have a pointer back to their related superclasses. The related superclass can be determined by pointer arithmetic (*i.e.* subtracting the superclass `storageOffset` from the subclass instance pointer). So the role information for a union subclass is nothing more than a pointer to its class descriptor

Similar to the reference generalization, the union generalization information consists of that for the superclass an a set of subclass role information values.

```
<<mrt internal aggregate types>>=
typedef struct mrtuniongeneralization {
    MRT_SuperClassRole superclass ;
    unsigned subclassCount ;
    MRT_Class const * const *subclasses ;
} MRT_UnionGeneralization ;
```

Relationship Properties

Finally, the relationship data is held as a discriminated union. There is a type field and a union of four structures for the specific information required to describe each type of relationship.

```
<<mrt internal aggregate types>>=
typedef struct mrtrelationship {
    MRT_RelType relType ;
    union {
        MRT_SimpleAssociation simpleAssociation ;
        MRT_ClassAssociation classAssociation ;
        MRT_RefGeneralization refGeneralization ;
        MRT_UnionGeneralization unionGeneralization ;
    } relInfo ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Relationship ;
```

`relType`
: A value to indicate the type of the relationship.

relInfo
A union of the various types of relationship information structures that describes the details of the relationship. The union must be interpreted according to the value of the `relType` member.

name
The name of the relationship.

Data Transactions

To check referential integrity, it is necessary to introduce the concept of a transaction on the domain data model. The transaction concept is necessary because we must be able to defer the integrity check until such time as the domain activities can insure that the integrity constraints are met. For example, consider two classes in a one-to-one unconditional association. If an instance of one class is created we do not want to check referential integrity until the code has had an opportunity to create a corresponding instance of the associated class.

There are two situations where we must enforce referential integrity:

1. After completion of a *thread of control*.
2. After completion of a *domain operation*.

We will describe how threads of control work later. For now, we are concerned with data transactions as a means of deciding when data integrity is to be verified.

For execution sequencing by event dispatch, the data integrity rules require each state activity to leave the domain data in a consistent state or to signal one or more events that, when dispatched, will ultimately bring the domain data into a consistent state. The later phrase of this rule implies that there a time where the domain data is allowed to be inconsistent while execution continues to bring it into a consistent state. For us, this implies that a thread of control is allowed to complete before the domain data is evaluated for consistency.

A domain operation is a form of synchronous service supplied by a domain. Like state activities, synchronous services exhibit run to completion semantics. Domain operations have full access to the underlying domain data and may also signal events which eventually initiate a new thread of control. Thus, domain operations must leave the domain data in a consistent state. However, there is a complication associated with domain operations.

- While executing a thread of control, an activity may invoke a domain operation. At that time, the domain data may be inconsistent. Note that threads of control do not *nest*, *i.e.* there is only ever one thread of control executing at any time because only one thread of execution is supported.
- Domain operations are not so restricted. Considering external entity operations, it is not possible to predict the chain of invocations of domain operations.
- Finally, the `micca` run time does not have any concept of a domain. Surprising as that may seem, from the point of view of the `micca` run time, the world is big flat and governed only by the addresses of descriptive data.

The implications of these considerations imply that data transactions must be nestable. Domain operations must start a new nested transaction and end it when they are done. Threads of control need not bother since they are not nestable in the single threaded `micca` run time. The code generator for `micca` does handle the nesting requirement, so there is no additional information required for the translation. It is just a matter of putting all domain operations within a nested data transaction and the code generation accomplishes that.

Synchronous Service Support

Two additional external functions are supplied by the run time to support synchronous operations on the domain, e.g. domain operations.

1. `mrt_BeginSyncService` marks the beginning of the synchronous service.

2. `mrt_EndSyncService` marks the end of the synchronous service and is the indication to the run time code that the referential integrity of the domain data is to be checked.

Normally, translated models do not have to use these functions. `Micca` generates code to surround automatically each domain operation with a start / end pair. However, bridge operations may choose to use them, particularly in conjunction with the `micca` portal operations. Portal operations do not enforce referential integrity and so bridges which use the portal operations that can potentially affect the data integrity must use these functions to mark a data transaction boundary.

```
<<mrt external interfaces>>=
extern void mrt_BeginSyncService(void) ;
```

The `mrt_BeginSyncService` function informs the run time that synchronous operations are starting on the domain.

The implementation of the `mrt_BeginSyncService` just increments a variable that contains the current transaction level.

```
<<mrt external functions>>=
void
mrt_BeginSyncService(void)
{
    mrtIncrTransLevel() ;
    mrtInWhiteState = true ;
}
```

```
<<mrt external interfaces>>=
extern void mrt_EndSyncService(void) ;
```

The `mrt_EndSyncService` function informs the run time that synchronous operations are ended on the domain. The run time uses this indication to enforce integrity constraints on the domain data.

The implementation of `mrt_EndSyncService` invokes `mrtEndTransaction()` to verify the referential integrity of the domain data. The transaction level is decremented to indicate that the nested transaction is complete.

```
<<mrt external functions>>=
void
mrt_EndSyncService(void)
{
    mrtEndTransaction() ;
}
```

We keep track of the transaction level as a simple unsigned counter.

```
<<mrt implementation simple types>>=
typedef uint16_t MRT_LevelCount ;
```

```
<<mrt implementation constants>>=
#define MRT_MAX_TRANS_LEVEL_COUNT     UINT16_MAX
```

A file static variable holds the current transaction level.

```
<<mrt static data>>=
static MRT_LevelCount mrtTransLevel ;
```

```
<<mrt implementation static inlines>>=
static inline
void
mrtIncrTransLevel(void)
{
```

```
    if (mrtTransLevel >= MRT_MAX_TRANS_LEVEL_COUNT) {
        mrtFatalError(mrtTransOverflow) ;       // ❶
    }
    mrtTransLevel += 1 ;
}
```

❶ Overflowing the transaction level counter indicates a serious problem.

```
<<mrt implementation static inlines>>=
static inline
void
mrtDecrTransLevel(void)
{
    if (mrtTransLevel != 0) {
        mrtTransLevel -= 1 ;        // ❶
    }
}
```

❶ We allow for unbalanced decrements. This is important to the transaction level algorithm since threads of control don't adjust the transaction nesting level. Most of the time a TOC will run at level 0. At the end it will attempt to decrement the level. We must make sure the transaction level does not underflow.

Recording Transaction Details

Each time a relationship is manipulated, by:

a. creating instances that participate in a relationship

b. deleting instances that participate in a relationship

c. updating references between participating instances

d. reclassifying subclass instances in a generalization

the relationship pointer references must be verified. As described below, the run-time provides functions to perform the relationship pointer operations. When a transaction is ongoing, we save a reference to the relationship information along with the current transaction level for each relationship that is manipulated. The affected relationships are stored in the following structure.

```
<<mrt implementation aggregate types>>=
typedef struct mrtTransEntry {
    struct mrtTransEntry *next ;
    MRT_LevelCount level ;
    MRT_Relationship const *relationship ;
} MRT_TransEntry ;
```

The individual transaction entries are managed on a linked list. We'll follow our usual pattern and have a free list and an in-use list.

```
<<mrt static data>>=
static MRT_TransEntry *mrtFreeTransEntries ;
static MRT_TransEntry *mrtTransEntries ;
```

Storage for the transaction entries is allocated as an array. In this case, we need transaction entries for the maximum nesting depth of transactions.

```
<<mrt static data>>=
static MRT_TransEntry mrtTransStorage[MRT_TRANSACTION_SIZE] ;
```

```
<<mrt implementation constants>>=
#ifndef MRT_TRANSACTION_SIZE
#   define MRT_TRANSACTION_SIZE 16
#endif /* MRT_TRANSACTION_SIZE */
```

To manage the transaction entries, we need functions to initialize the various lists and preform the usual list manipulations.

```
<<mrt implementation static inlines>>=
static inline
void
mrtInsertTrans(
    MRT_TransEntry **list,
    MRT_TransEntry *entry)
{
    assert(list != NULL) ;
    assert(entry != NULL) ;

    entry->next = *list ;
    *list = entry ;
}
```

```
<<mrt implementation static inlines>>=
static inline
MRT_TransEntry *
mrtRemoveTrans(
    MRT_TransEntry **list)
{
    assert(list != NULL) ;

    MRT_TransEntry *entry = *list ;
    if (entry != NULL) {
        *list = entry->next ;
        entry->next = NULL ;                // ❶
    }

    return entry ;
}
```

❶ Not strictly necessary, but useful during debugging.

After checking the referential integrity of an entry, it is discarded by removing it from the set of check entries and queue the entry back to the free queue for later use.

```
<<mrt implementation static inlines>>=
static inline
void
mrtDiscardTrans(void)
{
    MRT_TransEntry *entry = mrtRemoveTrans(&mrtTransEntries) ;
    assert(entry != NULL) ;                                    // ❶
    if (entry != NULL) {
        mrtInsertTrans(&mrtFreeTransEntries, entry) ;
    }
}
```

❶ We shouldn't be *over* discarding, but will tolerate it in a release build.

```
<<mrt static functions>>=
static void
mrtTransactionsInit(void)
{
    mrtFreeTransEntries = NULL ;
    mrtTransEntries = NULL ;
    mrtTransLevel = 0 ;

    MRT_TransEntry const *end = mrtTransStorage + MRT_TRANSACTION_SIZE ;
    for (MRT_TransEntry *iter = mrtTransStorage ;
            iter < end ; iter++) {
        mrtInsertTrans(&mrtFreeTransEntries, iter) ;
    }
}
```

Adding a relationship to the set to be checked is done by obtaining a free entry and inserting it onto the list of transaction entries to check.

```
<<mrt static functions>>=
static void
mrtAddRelToCheck(
    MRT_Relationship const *rel)
{
    MRT_TransEntry *entry = mrtRemoveTrans(&mrtFreeTransEntries) ;
    if (entry == NULL) {
        mrtFatalError(mrtTransOverflow) ;
    }

    entry->level = mrtTransLevel ;
    entry->relationship = rel ;
    mrtInsertTrans(&mrtTransEntries, entry) ;
}
```

During a transaction, run-time functions that either modify the relationship pointer storage or change the number of active instances invoke `mrtMarkRelationship` to save away the relationships that will need to be checked at the end of the transaction. This prevents us from having to check every relationship at the end of every transaction.

The only complication in the marking algorithm is that we wish to maintain the marked relationship descriptions as a set with no duplicates. Modifying instances of relationships multiple times during the transaction does not mean we need to evaluate the referential integrity more than once. We wish to save that computation and consequently will search the marked relationships to eliminate any duplicate mark.

```
<<mrt static functions>>=
static bool
mrtFindRelEntry(
    MRT_Relationship const *const rel)
{
    for (MRT_TransEntry *iter = mrtTransEntries ;
            iter != NULL && iter->level >= mrtTransLevel ;  // ❶
            iter = iter->next) {
        if (iter->relationship == rel) {
            return true ;
        }
        // N.B. no "else"
    }

    return false ;
}
```

❶ The search can be stopped when we cross over a transaction boundary. It is possible to have the same relationship modified

in two different transactions, but there is no harm. We will just end up doing an extraneous check. It is more important to discovery any integrity problems in the current ongoing transaction.

```
<<mrt forward references>>=
static void
mrtMarkRelationship(
    MRT_Relationship const *const *rel,
    unsigned relCount) ;
```

Marking a relationship to be checked means searching the current set of marked relationships and, if we have not already added it, then it is inserted into our set to check.

```
<<mrt static functions>>=
static void
mrtMarkRelationship(
    MRT_Relationship const * const *rel,
    unsigned relCount)
{
    for ( ; relCount != 0 ; relCount--, rel++) {
        bool found = mrtFindRelEntry(*rel) ;
        if (!found) {
            mrtAddRelToCheck(*rel) ;
        }
    }
}
```

At the end of a transaction, we successively examine entries in our list and determine if they pass referential integrity. We perform the examination on the head of the list. If that succeeds, we discard the entry and continue looking by examining the entry left at the head. We are also only going to examine entries that are part of the current transaction.

The first failing check stops the whole process with a fatal error. The entry containing the relationship in error is left on the list. This is done because it is possible for applications to supply fatal error handlers which `longjmp()` out of the event loop[1]. The integrity problem still exists and could be repaired. In any case, we want that relationship to be evaluated again if we reenter the event loop. It is also useful for debugging since the faulting relationship is always at the head of the list.

```
<<mrt forward references>>=
static void mrtEndTransaction(void) ;

<<mrt static functions>>=
static void
mrtEndTransaction(void)
{
    for (MRT_TransEntry *rentry = mrtTransEntries ;
            rentry != NULL && rentry->level >= mrtTransLevel ;  // ❶
            rentry = mrtTransEntries) {
        if (mrtCheckRelationship(rentry->relationship)) {
            mrtDiscardTrans() ;
        } else {
            mrtDecrTransLevel() ;                               // ❷
            mrtRefIntegrityError(rentry->relationship) ;
        }
    }
    mrtDecrTransLevel() ;                                       // ❸
    if (mrtTransLevel == 0) {
        mrtInWhiteState = false ;
    }
}
```

❶ Again, we stop processing the entries in the list when we cross the boundary of a transaction.

[1] Test programs benefit from this arrangement

❷ Since the integrity failure is a fatal error, we want to decrement the transaction level to show this evaluation is over. Some bridges, for testing specifically, will want to fix the integrity problem and be able to proceed on. This implies that failed checks become part of the next nested transaction or are caught when a thread of control terminates.

❸ We finish successfully and can mark the end of the transaction by decrementing the level.

Verifying Referential Integrity

At the end of a transaction on the data model, `mrtEndTransaction` iterates across the set of saved relationship descriptors and checks the relationship to insure the data model is in a consistent state. The design approach is to use the `refCount` member of each instance as a place to store how many times the instance is referred to by the other participant in the relationship. Knowing the cardinality of the relationship, we can evaluate the `refCount` values to determine if referential integrity has been maintained.

The overall steps are:

1. Zero out the `refCount` member of all instances of both participating classes.
2. Using the relationship description information, find the reference pointers in the class instance that refer to the other participating class instances and increment the `refCount` member in the other participating instance if its pointer is found. This is done for both participants in the relationship.
3. Evaluate the `refCount` value against what the cardinality of the relationship requires.

This algorithm must be generic and capable of operating on any class instance. This will mean that there will be a lot of pointer arithmetic using offsets to structure members and other type unsafe operations[2] happening in the code. Sometimes "C" as a language is accused of being little more than a high level assembler and it is those type unsafe facilities that we must employ here to obtain a single body of code that can operate on an arbitrary class instance.

```
<<mrt forward references>>=
static bool mrtCheckRelationship(MRT_Relationship const *rel) ;
```

The `mrtCheckRelationship` function returns a boolean value indicating if the relationship described by `rel` has correct referential integrity.

Since there are four types of relationships in terms of how the reference pointers are stored, the function considers each type separately.

```
<<mrt static functions>>=
static bool
mrtCheckRelationship(
    MRT_Relationship const *rel)
{
    bool result = false ;

    switch (rel->relType) {
    case mrtSimpleAssoc: {
        <<mrtCheckRelationship: simple associations>>
    }
        break ;

    case mrtClassAssoc: {
        <<mrtCheckRelationship: class based associations>>
    }
        break ;

    case mrtRefGeneralization: {
        <<mrtCheckRelationship: reference generalizations>>
```

[2] By *type unsafe* we mean that the operations cannot be checked by the compiler to insure the type system isn't violated. The code itself is still standard "C" with well defined behavior.

```
    }
        break ;

    case mrtUnionGeneralization: {
        <<mrtCheckRelationship: union generalizations>>
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }

    return result ;
}
```

For simple associations, we zero out the `refCount` member. The references between the class instances are tracked and then the values of the `refCount` members are compared against what the relationship requires. We do this in two steps, first tracking from source to target and then tracking from target to source. The reason for dividing it this way is to avoid stomping on the `refCount` in the case of a reflexive association. Since there is only one `refCount` member, we have to zero it, count the references and check the counts completely for one side before doing the same for the other. In a reflexive association, each side is the same class and so the `refCount` member is actually the same memory location for both sides.

```
<<mrtCheckRelationship: simple associations>>=
MRT_SimpleAssociation const *assoc = &rel->relInfo.simpleAssociation ;

MRT_Class const *const targetClass = assoc->target.classDesc ;
mrtZeroRefCounts(targetClass) ;                                    // ❶
mrtCountAssocRefs(&assoc->source, targetClass) ;                   // ❷

result = mrtCheckRefCounts(targetClass, assoc->target.cardinality) ;
if (!result) {
    break ;
}

MRT_Class const *const sourceClass = assoc->source.classDesc ;  // ❸
mrtZeroRefCounts(sourceClass) ;
mrtCountAssocRefs(&assoc->target, sourceClass) ;
result = mrtCheckRefCounts(sourceClass, assoc->source.cardinality) ;
```

❶ Start with the class that is the target of the association navigation.

❷ Count in the target side for each source side that references it.

❸ Now flip it around and do the target to source traversal.

Class based associations are more complicated. Recall that class based associations are treated in a decomposed manner where each side is related to the associator class and the associator class is unconditionally and singularly related to each participant. So we will consider each side separately.

```
<<mrtCheckRelationship: class based associations>>=
MRT_ClassAssociation const *assoc = &rel->relInfo.classAssociation ;

result = mrtCheckAssociatorRefs(&assoc->associator) ; // ❶
if (!result) {
    break ;
}
/*
 * On the first side, we evaluate the references from the source class
 * to the associator class.
```

```
 */
MRT_Class const *const assocClass = assoc->associator.classDesc ;
MRT_Class const *const sourceClass = assoc->source.classDesc ;

mrtZeroRefCounts(sourceClass) ;
mrtCountSingularRefs(assocClass, assoc->associator.backwardOffset,
        sourceClass) ;

result = mrtCheckRefCounts(sourceClass, assoc->source.cardinality) ;
if (!result) {
    break ;
}

mrtZeroRefCounts(assocClass) ;
mrtCountClassAssocRefs(&assoc->source, assocClass) ;

result = mrtCheckRefCounts(assocClass, mrtExactlyOne) ;
if (!result) {
    break ;
}
/*
* If the first side is okay, then we can evaluate the references from
* the target class to the associator class.
*/
MRT_Class const *const targetClass = assoc->target.classDesc ;

mrtZeroRefCounts(targetClass) ;
mrtCountSingularRefs(assocClass, assoc->associator.forwardOffset,
        targetClass) ;

result = mrtCheckRefCounts(targetClass, assoc->target.cardinality) ;
if (!result) {
    break ;
}

mrtZeroRefCounts(assocClass) ;
mrtCountClassAssocRefs(&assoc->target, assocClass) ;
result = mrtCheckRefCounts(assocClass, mrtExactlyOne) ;
```

❶ We start by checking the singular references from the associator class to the two participants.

Since a generalization represents a disjoint union between between the superclass and the set of subclasses, each superclass instance must be referenced exactly once from among all the subclasses of the generalization. Conversely, each subclass must reference exactly one superclass instance. This means we will have to iterate over the subclasses as we zero out the `refCount` member and count the references.

```
<<mrtCheckRelationship: reference generalizations>>=
MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;

mrtZeroRefCounts(gen->superclass.classDesc) ;

MRT_RefSubClassRole const *subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
        subcount--, subclass++) {
    mrtZeroRefCounts(subclass->classDesc) ;
}

mrtCountGenRefs(gen) ;

result = mrtCheckRefCounts(gen->superclass.classDesc, mrtExactlyOne) ;
```

```
subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; result && subcount != 0 ;
        subcount--, subclass++) {
    result = mrtCheckRefCounts(subclass->classDesc, mrtExactlyOne) ;
}
```

When a generalization is stored in a union, there are no reference pointers. So we don't have to follow any pointers to get to the subclass. Rather, the subclass in part of the superclass storage and pointer arithmetic computes the address of the subclass instance. However, the process is much the same as for reference generalizations. What is different is the way the references are counted.

```
<<mrtCheckRelationship: union generalizations>>=
MRT_UnionGeneralization const *gen = &rel->relInfo.unionGeneralization ;

mrtZeroRefCounts(gen->superclass.classDesc) ;

MRT_Class const *const *subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
        subcount--, subclass++) {
    mrtZeroRefCounts(*subclass) ;
}

mrtCountUnionRefs(gen) ;                                                    // ❶

result = mrtCheckRefCounts(gen->superclass.classDesc, mrtExactlyOne) ;

subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; result && subcount != 0 ;
        subcount--, subclass++) {
    result = mrtCheckRefCounts(*subclass, mrtExactlyOne) ;
}
```

❶ We need a different counting strategy to account for the union storage.

Zero Reference Counts

The first of the generic operations that must be performed on class instances is to set the `refCount` member of the instance structure to zero.

```
<<mrt forward references>>=
static void mrtZeroRefCounts(MRT_Class const *const classDesc) ;
```

The algorithm is simple. We iterate over all the class instances and set the `refCount` member to zero.

```
<<mrt static functions>>=
static void
mrtZeroRefCounts(
    MRT_Class const *const classDesc)
{
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, classDesc) ; mrt_InstIteratorMore(&iter) ;
            mrt_InstIteratorNext(&iter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        instref->refCount = 0 ;
    }
}
```

Checking Reference Counts

The other operation that depends only upon accessing the `refCount` member of an instance is to evaluate the value of `refCount` against what the cardinality of the relationship. There are four possible values of the cardinality. One of the values, `mrtZeroOrMore`, implies that any value of `refCount` is satisfactory since `refCount` is defined as an unsigned quantity and must, necessarily, be greater than or equal to zero.

For the other three values of cardinality we define functions to perform the comparison.

```
<<mrt static functions>>=
static bool
mrtCompareAtMostOne(
    unsigned refCount)
{
    return refCount <= 1 ;
}

static bool
mrtCompareExactlyOne(
    unsigned refCount)
{
    return refCount == 1 ;
}

static bool
mrtCompareOneOrMore(
    unsigned refCount)
{
    return refCount >= 1 ;
}
```

The reference counts are checked by iterating over the instances and invoking the proper cardinality comparison function. The first failure means we can stop.

```
<<mrt forward references>>=
static bool
mrtCheckRefCounts(MRT_Class const *const classDesc, MRT_Cardinality cardinality) ;
```

```
<<mrt static functions>>=
static bool
mrtCheckRefCounts(
    MRT_Class const *const classDesc,
    MRT_Cardinality cardinality)
{
    if (cardinality == mrtZeroOrMore) { // ❶
        return true ;
    }

    static bool (*const compareFuncs[])(unsigned) = {
        [mrtAtMostOne] = mrtCompareAtMostOne,
        [mrtExactlyOne] = mrtCompareExactlyOne,
        [mrtZeroOrMore] = NULL, // ❷
        [mrtOneOrMore] = mrtCompareOneOrMore
    } ;

    assert(cardinality <= mrtOneOrMore) ;
    bool (*const compareCardinality)(unsigned) = compareFuncs[cardinality] ; // ❸

    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, classDesc) ; mrt_InstIteratorMore(&iter) ;
            mrt_InstIteratorNext(&iter)) {
```

```
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        if (!compareCardinality(instref->refCount)) {
            return false ; // ❹
        }
    }

    return true ;
}
```

❶ Dispense with the always true case first. There is no reason to iterate through the instances when the result is always true.

❷ We could omit this and it would be set to zero like any other uninitialized static variable. We include it to make clear that that we have accounted for all the cases and have factored out the `mrtZeroOrMore` case above.

❸ Select the comparison function based on the encoded cardinality. A small table of pointers to the comparison functions is handy to do this.

❹ We need not go past the first failure.

The singular references made by associator classes are special in the sense that they should never be `NULL`. Unrelating class based associations will set those references to `NULL` and we need to make sure that either the associator instance was reused by relating to other instances or it was deleted.

```
<<mrt forward references>>=
static bool
mrtCheckAssociatorRefs(MRT_AssociatorRole const *associator) ;
```

```
<<mrt static functions>>=
static bool
mrtCheckAssociatorRefs(
    MRT_AssociatorRole const *associator)
{
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, associator->classDesc) ;
            mrt_InstIteratorMore(&iter) ; mrt_InstIteratorNext(&iter)) {
        void *inst = mrt_InstIteratorGet(&iter) ;
        MRT_Instance *ref = *(MRT_Instance **)
                ((uintptr_t)inst + associator->forwardOffset) ;
        if (ref == NULL || ref->alloc <= 0) { // ❶
            return false ;
        }
        ref = *(MRT_Instance **)((uintptr_t)inst + associator->backwardOffset) ;
        if (ref == NULL || ref->alloc <= 0) {
            return false ;
        }
    }

    return true ;
}
```

❶ Reference pointers in an associator class must be non-NULL and must point to an allocated instance.

Counting References

Of the four types of relationships, three of them actually contain pointer values that refer to class instances. The differences in the manner in which the three pointer references are organized leads us have separate functions for each type.

We use some common code to count the references made by an instance.

```
<<mrt implementation static inlines>>=
static inline
uint8_t
mrtIncrRefCount(
    uint8_t count)
{
    return (count == UINT8_MAX) ? 2 : count + 1 ;       // ❶
}
```

❶ We want to avoid the possibility of overflow of the reference counter. It is only an unsigned 8-bit quantity. Should it overflow, we could misinterpret the count. If we are at its max value, then we set it back to 2. Why 2? Zero and one are significant in this case and we know the values has already passed two, so we resume our count from there. Any value greater than 1 will yield the same conclusion about the referential integrity.

First, we consider counting the references in a simple association.

```
<<mrt forward references>>=
static void
mrtCountAssocRefs(
    MRT_AssociationRole const *source,
    MRT_Class const *const targetClass) ;
```

The `mrtCountAssocRefs` function counts the number of times each active instance of `source` refers to active instances of `targetClass`.

Counting references implies understanding how the reference pointers are stored. There are three ways that reference pointers are stored: single pointer, counted arrays and linked lists. We use the knowledge of offsets in the instance to where the pointer values are stored to access target instances.

```
<<mrt static functions>>=
static void
mrtCountAssocRefs(
    MRT_AssociationRole const *source,
    MRT_Class const *const targetClass)
{
    switch (source->storageType) {
    case mrtSingular:
        mrtCountSingularRefs(source->classDesc, source->storageOffset,
            targetClass) ;
        break ;

    case mrtArray:
        mrtCountArrayRefs(source->classDesc, source->storageOffset,
            targetClass) ;
        break ;

    case mrtLinkedList:
        mrtCountLinkedListRefs(source->classDesc, source->storageOffset,
                targetClass, source->linkOffset) ;
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

We will now show how the references stored in the three types are counted. For singular references, the reference is a pointer to the related class instance.

```
<<mrt forward references>>=
static void
mrtCountSingularRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass) ;
```

sourceClass
A pointer to the class description block for the class instances that make a singular reference to target instances.

offset
The offset in bytes from the beginning of source instances to where the pointer to the target instance is located.

targetClass
A pointer to the class description for the class instances to which the source instances refer.

The `mrtCountSingularRefs` functions iterates across all the active `source` instances, accesses the pointer in the source instance and increments the `refCount` member in the referenced target instance.

```
<<mrt static functions>>=
static void
mrtCountSingularRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;

    MRT_InstIterator srciter ;
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&srciter) ;
        MRT_Instance *targetInst =
                *(MRT_Instance **)((uintptr_t)instref + offset) ;
        if ((void *)targetInst >= targetiab->storageStart &&
                (void *)targetInst < targetiab->storageFinish &&
                targetInst->alloc > 0) { // ❶
            targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
        }
    }
}
```

❶ Validate that the reference pointer value actually points into target instance storage. This will eliminate any NULL values also. Note we also insist that the target instance be active.

For array references, the references are an array of pointers to the related class instance.

```
<<mrt forward references>>=
static void
mrtCountArrayRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass) ;
```

sourceClass
A pointer to the class description block for the class instances that make an array reference to target instances.

offset
The offset in bytes from the beginning of source instances to where the pointer array to the target instance is located.

targetClass
A pointer to the class description for the class instances to which the source instances refer.

The `mrtCountArrayRefs` functions iterates across all the active `source` instances, accesses the pointer in the source instance and increments the `refCount` member in the referenced target instance.

```
<<mrt static functions>>=
static void
mrtCountArrayRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;

    MRT_InstIterator srciter ;
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&srciter) ;
        MRT_ArrayRef *srcrefs =
                (MRT_ArrayRef *)((uintptr_t)instref + offset) ;
        MRT_Instance *const *iter = srcrefs->links ;
        for (unsigned count = srcrefs->count ; count != 0 ; count--, iter++) {
            MRT_Instance *targetInst = *iter ;
            if ((void *)targetInst >= targetiab->storageStart &&
                    (void *)targetInst < targetiab->storageFinish &&
                    targetInst->alloc > 0) {
                targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
            }
        }
    }
}
```

The structure of the linked list pointers makes counting them a bit more complicated. We must know the offset within the source instance where the link list terminus is located. Since the link pointers in the target are embedded somewhere in the target instance structure, we need to know the offset into the target instance that the links point to in order recover a pointer to the beginning of the target instance..

```
<<mrt forward references>>=
static void
mrtCountLinkedListRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset refOffset,
    MRT_Class const *const targetClass,
    MRT_AttrOffset linkOffset) ;
```

sourceClass
A pointer to the class description for the class instances that make a linked reference to target instances.

refOffset
The offset in bytes from the beginning of source instances to where the linked list terminus is located.

targetClass
A pointer to the class description for the class instances to which the source instances refer.

linkOffset
The offset in bytes from the beginning of target instances to where the linked list pointers are located.

The `mrtCountLinkedListRefs` function iterates through the linked list contained in the source instances to reference the target instances. When a target instance is found, it's `refCount` member is incremented.

```
<<mrt static functions>>=
static void
mrtCountLinkedListRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset refOffset,
    MRT_Class const *const targetClass,
    MRT_AttrOffset linkOffset)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;
    MRT_InstIterator srciter ;
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *srcInst = mrt_InstIteratorGet(&srciter) ;
        MRT_LinkRef *ref = (MRT_LinkRef *)((uintptr_t)srcInst + refOffset) ;
        for (MRT_LinkRef *trgIter = mrtLinkRefBegin(ref) ;
                !(trgIter == NULL || trgIter == mrtLinkRefEnd(ref)) ;    // ❶
                trgIter = trgIter->next) {
            MRT_Instance *targetInst =
                (MRT_Instance *)((uintptr_t)trgIter - linkOffset) ;      // ❷
            if ((void *)targetInst >= targetiab->storageStart &&
                    (void *)targetInst < targetiab->storageFinish &&
                    targetInst->alloc > 0) {
                targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
            }
        }
    }
}
```

❶ Guard against uninitialized link pointers as well as detect the end of the linked list.

❷ The link pointers that form the linked list of target instances are located `linkOffset` from the beginning of the target instance. So we need to do some pointer arithmetic to get the pointer to the beginning of the instance. We have to do this computation because it may be the case that the target is on several linked lists depending upon the relationships in which it participates.

Counting class based associations uses the basic counting primatives that we have already seen.

```
<<mrt forward references>>=
static void
mrtCountClassAssocRefs(
    MRT_AssociationRole const *participant,
    MRT_Class const *const assocClass) ;
```

To count class based associations, we have to count the references from each participant to the associator class. References from the participant to the associator class can be of any type. References from the associator class back to the participant are always singular.

```
<<mrt static functions>>=
static void
mrtCountClassAssocRefs(
    MRT_AssociationRole const *participant,
    MRT_Class const *const assocClass)
{
    switch (participant->storageType) {
    case mrtSingular:
        mrtCountSingularRefs(participant->classDesc, participant->storageOffset,
            assocClass) ;
        break ;
```

```
    case mrtArray:
        mrtCountArrayRefs(participant->classDesc, participant->storageOffset,
            assocClass) ;
        break ;

    case mrtLinkedList:
        mrtCountLinkedListRefs(participant->classDesc,
                participant->storageOffset, assocClass,
                participant->linkOffset) ;
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

In a reference type generalization, all of the references are always singular. However, we must iterate across all the subclasses of the generalization in order to count the references that the subclasses make back to the superclass.

```
<<mrt forward references>>=
static void
mrtCountGenRefs(
    MRT_RefGeneralization const *gen) ;
```

```
<<mrt static functions>>=
static void
mrtCountGenRefs(
    MRT_RefGeneralization const *gen)
{
    MRT_Class const *const superClass = gen->superclass.classDesc ;
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, superClass) ;                 // ❶
            mrt_InstIteratorMore(&iter) ; mrt_InstIteratorNext(&iter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        MRT_Instance *subInst = *(MRT_Instance **)
                ((uintptr_t)instref + gen->superclass.storageOffset) ;
        if (subInst != NULL) {
            MRT_Class const *const subClass = subInst->classDesc ;

            MRT_iab *subiab = mrtGetStorageProperties(subClass, NULL) ;
            if ((void *)subInst >= subiab->storageStart &&
                    (void *)subInst < subiab->storageFinish &&
                    subInst->alloc > 0) {
                subInst->refCount = mrtIncrRefCount(subInst->refCount) ;
            }
        }
    }

    MRT_RefSubClassRole const *subclass = gen->subclasses ;
    for (unsigned subcount = gen->subclassCount ; subcount != 0 ;   // ❷
            subcount--, subclass++) {
        mrtCountSingularRefs(subclass->classDesc, subclass->storageOffset,
                gen->superclass.classDesc) ;
    }
}
```

❶ Start by iterating over the instances of the superclass.

❷ The references from the subclass to the superclass is just an ordinary singular pointer reference. We already know how to count them.

For a union based generalization, all the references are singular. However, since the storage for the subclass instance is part of the superclass instance storage, we only have to iterate across the superclass instances to have access to both reference counts.

```
<<mrt forward references>>=
static void
mrtCountUnionRefs(
    MRT_UnionGeneralization const *gen) ;
```

```
<<mrt static functions>>=
static void
mrtCountUnionRefs(
    MRT_UnionGeneralization const *gen)
{
    MRT_Class const *const superClass = gen->superclass.classDesc ;
    MRT_InstIterator superIter ;
    for (mrt_InstIteratorStart(&superIter, superClass) ;
            mrt_InstIteratorMore(&superIter) ;
            mrt_InstIteratorNext(&superIter)) {
        MRT_Instance *superInst = mrt_InstIteratorGet(&superIter) ;
        MRT_Instance *subInst = (MRT_Instance *)
                ((uintptr_t)superInst + gen->superclass.storageOffset) ;

        if (subInst->alloc > 0) {
            subInst->refCount = mrtIncrRefCount(subInst->refCount) ;
        }
    }

    MRT_Class const *const *subclass = gen->subclasses ;
    for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
            subcount--, subclass++) {
        MRT_InstIterator subIter ;
        for (mrt_InstIteratorStart(&subIter, *subclass) ;
                mrt_InstIteratorMore(&subIter) ;
                mrt_InstIteratorNext(&subIter)) {
            MRT_Instance *subInst = mrt_InstIteratorGet(&subIter) ;
            MRT_Instance *superInst = (MRT_Instance *)
                    ((uintptr_t)subInst - gen->superclass.storageOffset) ;

            if (superInst->alloc > 0) {
                superInst->refCount = mrtIncrRefCount(superInst->refCount) ;
            }
        }
    }
}
```

Operations on Relationship Instances

Since the relationship descriptions have sufficient information to manage referential integrity, they also can be used to do the pointer manipulations required when instances of the relationships are created or deleted. It is important for the run-time to provide functions to perform these operations since checking referential integrity depends upon correct reference pointer manipulation.

Because we are using pointer values to manage instance identity and references[3], we need to be clear what operations must be performed for classes that participate in relationships.

[3] as opposed to attribute values if we were doing this using relational concepts

For class based associations:

- There is a one-to-one correspondence between instances of a class based association and instances of the associator class itself.
- Because of this correspondence, when an instance of an associator is created, the creation operation must be supplied with instance references to the participating instances. The run-time will make up the pointer references appropriately. The participating instances must not already be part of the relationship. This implies that the participating instances are newly created or the run-time will unlink them from the association before creating the new instance of the association.
- Conversely, deleting an instance of an associator class is sufficient to delete an instance of the class based association itself.
- Deleting an instance of a class participating in a class based association **does not** cause the corresponding instance (or instances) of the associator to be deleted.
- An operation to swap a different instance of a participating class must be provided.

For simple associations:

- There is a one-to-one correspondence between instances of the simple association and instances of the class which has the *referring* role.
- Because of this correspondence, when an instance of the referring class is created, the creation operation must be supplied with an instance reference to an instance of the class which plays the *referenced* role in the association.
- Conversely, deleting an instance of a referring class, unlinks the references to the referenced class.
- Deleting an instance of a referenced class, *does not* modify the reference in the referring class.
- An operation to swap a different instance of a referenced class must be provided.

For referenced based generalizations:

- When creating an instance of a subclass, the creation operation must be supplied with an instance reference to the superclass. The implication is that object creation must be in superclass to subclass order.
- Deleting an instance of the subclass, unlinks the references between the subclass and superclass. Deleting an instance of the superclass *does not* modify the reference in the subclass and the corresponding subclass instance is not deleted.
- An operation to reclassify subclass is also provided. This is a short hand for a delete / create sequence of a related subclass instance.

For union based generalizations:

- Subclass instances may not be directly created. It is necessary to create the superclass and reclassify the subclass instance to the desired subclass type. This may need to be repeated down a hierarchy if multiple generalization are defined. Note this implies the creation order for union based generalization is also from superclass to subclass order.
- Superclass or subclass instances may be deleted. In this case, there are no pointer references to modify, but referential integrity will be evaluated at the end of the data transaction.
- An operation to reclassify subclass is also provided. This is a short hand for a delete / create sequence of a related subclass instance.

Creating Relationship Links

The first part of supplying the operations on relationships is to be able to create the linkage between instances that refer to each other. We split this into two different functions, one for simple linkage and the other for associative linkage. Simple linkage forms the pointer references between only two participants in a relationship. This occurs for:

a. simple associations that are only between two participants,

b. associative classes when linking to only one participant, *i.e.* during a swap operation where one participant instance is being exchanged for another and

c. reference based generalizations

Associative linkage must account for the role of an associator class when linking together both participants in a class based association. This occurs when an instance of an associator class is created.

```
<<mrt internal external interfaces>>=
extern void
mrt_CreateSimpleLinks(
    MRT_Relationship const *const rel,
    void *const source,
    void *const target,
    bool isForward) ;
```

rel
A pointer to a relationship description for the relationship across which the relate operation is to happen.

source
A pointer to a class instance that serves the source role in the relationship.

target
A pointer to a class instance that serves the target role in the relationship.

isForward
A boolean to disambiguate the reflexive case. If `true`, then the link from `source` to `target` is in the forward direction. Otherwise, the link is to be made from `target` to `source`. This argument is ignored for relationships that are not reflexive class based associations.

This function is used to establish pointer links when instances which have simple association are created as well as when simple or class based association have their related instances "swapped".

For simple associations, `source` is an instance of the referring class and `target` is an instance of the referenced class. The `isForward` argument is ignored. For class based associations, `source` is an instance of the associator class and `target` is an instance of one of the participants. If the asssociation is reflexive, then the `isForward` argument determines if `target` is a target participant (true) or as source participant (false) . For reference generalizations, `source` is an instance of a subclass and `target` is an instance of the superclass. Union based generalizations have no links.

Since we have four different types of relationships (in terms of their storage properties), we consider each type as a separate case.

```
<<mrt external functions>>=
void
mrt_CreateSimpleLinks(
    MRT_Relationship const *const rel,
    void *const source,
    void *const target,
    bool isForward)
{
    assert(rel != NULL) ;
```

```
    assert(source != NULL) ;
    assert(target != NULL) ;

    switch (rel->relType) {
    case mrtSimpleAssoc: {
        <<mrt_CreateSimpleLinks: link simple association>>
    }
        break ;

    case mrtClassAssoc: {
        <<mrt_CreateSimpleLinks: link class association>>
    }
        break ;

    case mrtRefGeneralization: {
        <<mrt_CreateSimpleLinks: link reference generalization>>
    }
        break ;

    case mrtUnionGeneralization:
        // There are no pointer linkages for a union generalization.
        // N.B. fall through

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }

    mrtMarkRelationship(&rel, 1) ;      // ❶
}
```

❶ Since we are create links, we must mark the relationship to be evaluated at the end of the data transaction.

For simple associations, we must establish two sets of pointer linkages. From source to target is the primary referring direction. This is always singular and unconditional and so represented by a single pointer value. From target to source is in the back link direction and may be either singular or a linked list.

```
<<mrt_CreateSimpleLinks: link simple association>>=
MRT_SimpleAssociation const *const assoc = &rel->relInfo.simpleAssociation ;

MRT_Instance *srcInst = source ;
MRT_Instance *targetInst = target ;
if (assoc->source.classDesc != srcInst->classDesc ||
        assoc->target.classDesc != targetInst->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;                          // ❶
}

void *currentTarget =
        *(void **)((uintptr_t)source + assoc->source.storageOffset) ;

if (currentTarget != target) {                                       // ❷
    if (currentTarget != NULL) {
        mrtUnlinkBackref(&assoc->target, source, currentTarget) ;    // ❸
    }
    mrtLink(&assoc->source, source, target) ;                        // ❹
    mrtLink(&assoc->target, target, source) ;
}
```

❶ Make sure that the relationship descriptive information matches the classes of the instances we were handed.

❷ Just guard against the case where we are linking together what is already linked. Call it a big no op.

❸ We unlink any back references by the current target instance. This in effect causes the the current target to be half unrelated.

❹ Create a link for each leg of the relationship — from source to target and then from target to source.

Since the core of what is going on here is linking the two instances, let's look at what `mrtLink` does. We will use this function several more times later on.

`MrtLink` creates one leg of a relationship link from one instance ("from") to another ("to"). Like most of the access to relationship pointers we have already seen, `mrtLink` does some unsafe pointer arithmetic to find the location in the instance where references are stored. Then, depending upon the type of reference storage, updates the instance pointers.

```
<<mrt static functions>>=
static void
mrtLink(
    MRT_AssociationRole const *const fromRole,
    void *const fromInst,
    void *const toInst)
{
    void *linkStorage = (void *)((uintptr_t)fromInst + fromRole->storageOffset) ;
    switch (fromRole->storageType) {
    case mrtSingular: {
        void **toLink = linkStorage ;
        *toLink = toInst ;                                          // ❶
    }
        break ;

    case mrtArray:
        // can't link array types
        mrtFatalError(mrtStaticRelationship) ;
        break ;

    case mrtLinkedList: {
        MRT_LinkRef *toLinks =
                (MRT_LinkRef *)((uintptr_t)toInst + fromRole->linkOffset) ;
        if (toLinks->next != NULL && toLinks->prev != NULL) {
            mrtLinkRefRemove(toLinks) ;                             // ❷
        }

        MRT_LinkRef *fromList = linkStorage ;                       // ❸
        mrtLinkRefInsert(toLinks, fromList) ;
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

❶ For a singular pointer, overwriting the pointer value creates the link.

❷ If the "to" instances is already on a linked list somewhere, then we unlink it before placing it on the new list. We can do this since we do not need the list head to unlink an item from a list.

❸ The terminus of the back link list is in the `fromInst` and is located at the `storageOffset` within the instance. The link pointers in the `toInst` are at the `linkOffset` within the instance. Note the offset into `toInst` is actually given in the association description of the "from" instance. This arrangement is convenient in other cases, despite the confusion of using a from description to access something in the "to" instance.

For class based associations, the `mrt_CreateSimpleLinks` function updates only one of the pointer references, from the associator class to either the source or target participant of the relationship. The code is more complex here because we need to deal with several circumstances:

- The `source` argument is always the associative class, but we must figure out whether the `target` is a source participant instance or a target participant instance.
- We have to handle the reflexive case where the two participant classes are the same and the `isForward` argument resolves the direction.
- We must make sure that we are not creating a duplicate instance of the association. Since instances of an associative relationship correspond one-to-one to instances of the associator class, we have to make sure that there are no other associator class instances which have the same source and target participant instances. This is logically the same as enforcing an identifying constraint on the associator class since the referential attributes of an associator class form an identifier.

```
<<mrt_CreateSimpleLinks: link class association>>=
MRT_ClassAssociation const *const cassoc = &rel->relInfo.classAssociation ;
MRT_AssociatorRole const *const arole = &cassoc->associator ;
MRT_AssociationRole const *const srole = &cassoc->source ;
MRT_AssociationRole const *const trole = &cassoc->target ;

void *const associator = source ; // Change the variable names to keep things clear
void *const dest = target ;
if (arole->classDesc != ((MRT_Instance *)associator)->classDesc) { // ❶
    mrtFatalError(mrtRelationshipLinkage) ;
}

MRT_AssociationRole const *destrole ;
MRT_AttrOffset assocOffset ;
void *srcInst ;
void *targetInst ;

if (srole->classDesc == trole->classDesc) {                          // ❷
    // reflexive case
    if (trole->classDesc != ((MRT_Instance *)dest)->classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    if (isForward) {                                                 // ❸
        destrole = trole ;
        assocOffset = arole->forwardOffset ;
        srcInst = *(void **)((uintptr_t)associator + arole->backwardOffset) ;
        targetInst = dest ;
    } else {
        destrole = srole ;
        assocOffset = arole->backwardOffset ;
        srcInst = dest ;
        targetInst = *(void **)((uintptr_t)associator + arole->forwardOffset) ;
    }
} else {                                                             // ❹
    // non-reflexive case
    if (srole->classDesc == ((MRT_Instance *)dest)->classDesc) {
        // backward
        destrole = srole ;
        assocOffset = arole->backwardOffset ;
        srcInst = dest ;
        targetInst = *(void **)((uintptr_t)associator + arole->forwardOffset) ;
    } else if (trole->classDesc == ((MRT_Instance *)dest)->classDesc) {
        // forward
        destrole = trole ;
        assocOffset = arole->forwardOffset ;
        srcInst = *(void **)((uintptr_t)associator + arole->backwardOffset) ;
```

```
        targetInst = dest ;
    } else {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
}

void **p_assocRef = (void **)((uintptr_t)associator + assocOffset) ;
if (*p_assocRef != NULL) {
    mrtUnlinkBackref(destrole, associator, *p_assocRef) ;            // ❺
    *p_assocRef = NULL ;
}

if (srcInst != NULL && targetInst != NULL && arole->multiple == false) {
    mrtCheckDupAssociator(rel, srcInst, targetInst) ;               // ❻
}

*p_assocRef = dest ;                                                 // ❼
mrtLink(destrole, dest, associator) ;
```

❶ When invoked on a class based association, we insist that the "source" instance reference argument be an instance reference to the associator class. We introduce a new variable to prevent confusion of the `source` argument with the fact that it is now assumed to be an instance of the associator class.

❷ Check for the reflexive case.

❸ For the reflexive case, the `isForward` argument determines the direction of the destination of the link.

❹ For the non-reflexive case we can determine the direction since the classes of the two ends are different.

❺ Unlink the references between the associator and the destination, if any. This is a necessary step before we check for a duplicated associator instance. If this is really just an update of an existing association, then be must delete that association before testing that creating a new one would be a duplicate. Otherwise, link values laying around in the association class data structure could be interpreted as being part of a duplicate instance.

❻ Check that we are not duplicating an instance of the association. It is a fatal error to attempt to do so. Note we check that the source and target actually have valid instance pointers. It is possible for the association class **not** to be linked to anything (e.g. when it is first created) and so the source and/or target instance pointers may be `NULL`.

❼ Link the leg from the associative class to one of the participant instances and then in the opposite direction — from the participant instance to the associative class instance.

When relating reference generalizations, the reference from the subclass to the superclass is singular and unconditional. The reference from the superclass to the subclass is also singular.

```
<<mrt_CreateSimpleLinks: link reference generalization>>=
MRT_RefGeneralization const *const gen = &rel->relInfo.refGeneralization ;

MRT_Class const *const subclassClass =
        ((MRT_Instance *)source)->classDesc ;                        // ❶
int subclassCode = mrtFindRefGenSubclassCode(subclassClass, gen->subclasses,
        gen->subclassCount) ;

MRT_Class const *const superclassClass =
        ((MRT_Instance *)target)->classDesc ;                        // ❷
if (gen->superclass.classDesc != superclassClass) {
    mrtFatalError(mrtRelationshipLinkage) ;
}

void **p_superRef = (void **)((uintptr_t)source +
        gen->subclasses[subclassCode].storageOffset) ;
*p_superRef = target ;
```

```
void **p_subRef = (void **)((uintptr_t)target + gen->superclass.storageOffset) ;
*p_subRef = source ;
```

1. For reference generalizations, the "source" instance reference must be one of the subclasses that participates in the relationship. The primary reference in a generalization is from subclass instance to superclass instance.
2. The "target" instance reference must then be to the superclass of the generalization.

Finding the participating subclass of the generalization is a sequential search of the subclass roles in the relationship description.

```
<<mrt static functions>>=
static int
mrtFindRefGenSubclassCode(
    MRT_Class const *const subclassClass,
    MRT_RefSubClassRole const *subclasses,
    unsigned count)
{
    int subcode ;

    for (subcode = 0 ; subcode < count ; subcode++, subclasses++) {
        if (subclassClass == subclasses->classDesc) {
            return subcode ;
        }
    }

    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Create Associator Links

Linking class based associations must account for the nature of the associator class. A class based association is treated as being decomposed into two associations, one each between the participants and the associator class. The associator class has singular, unconditional references to each participant. The participants have back references to the associator whose type depends upon multiplicity and the static nature of the association.

```
<<mrt internal external interfaces>>=
extern void
mrt_CreateAssociatorLinks(
    MRT_Relationship const *rel,
    void *assoc,
    void *source,
    void *target) ;
```

rel
A pointer to a relationship description for the relationship across which the relate operation is to happen.

assoc
A pointer to a class instance that serves the associator role in the relationship.

source
A pointer to a class instance that serves the source role in the relationship.

target
A pointer to a class instance that serves the target role in the relationship.

```
<<mrt external functions>>=
void
mrt_CreateAssociatorLinks(
    MRT_Relationship const *rel,
    void *assoc,
    void *source,
    void *target)
{
    assert(rel != NULL) ;
    assert(assoc != NULL) ;
    assert(source != NULL) ;
    assert(target != NULL) ;

    assert(rel->relType == mrtClassAssoc) ;
    if (rel->relType != mrtClassAssoc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    MRT_ClassAssociation const *cassoc = &rel->relInfo.classAssociation ;
    MRT_AssociatorRole const *arole = &cassoc->associator ;

    assert(arole->classDesc == ((MRT_Instance *)assoc)->classDesc) ;
    if (arole->classDesc != ((MRT_Instance *)assoc)->classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    if (!arole->multiple) {
        mrtCheckDupAssociator(rel, source, target) ;    // ❶
    }

    mrt_CreateSimpleLinks(rel, assoc, source, false) ;  // ❷
    mrt_CreateSimpleLinks(rel, assoc, target, true) ;
}
```

❶ Problem arises when source and target are already related to each other through some other associator class instance. If that is the case, then we violate the identity contraint on the associator class and we would not have proper sets. So we have to navigate from source to see if we can find target.

❷ In keeping with the concept that a class based assocation is decomposed into two association legs, we need only create the links for both sides.

To make sure that there are no duplicated association instances, we must check that for any given pair of instances across a class based association, that we cannot traverse the association and find a match. The strategy used by `mrtCheckDupAssociator` is to start at the source instance and traverse the relationship in search of a matching target instance. This technique saves looking at all the associator class instances and searching for both the source and target pointer values. That code is easier to write, but will usually examine every instance of the associator class (since finding a duplicate is a rare happening). Each target instance linked to the source instance through the associator class must be compared to the target instance which is about to be set. If a match is found, then this is an attempt to create a duplicate instance of the association and a fatal error is declared. The complication arises in that we have to traverse the association using data found in the relationship description, *i.e.* using descriptive meta data. This means we have to perform all the pointer arithmetic ourselves.

The implementation considers the three cases that arise as a result of the three different ways that pointer linkage is stored. Of course, array storage used for static associations is not modified and so results in a fatal error.

```
<<mrt static functions>>=
static void
mrtCheckDupAssociator(
    MRT_Relationship const *rel,
    void *source,
```

```
    void *target)
{
    assert(rel != NULL) ;
    assert(rel->relType == mrtClassAssoc) ;
    assert(source != NULL) ;
    assert(target != NULL) ;

    MRT_ClassAssociation const *cassoc = &rel->relInfo.classAssociation ;
    MRT_AssociatorRole const *arole = &cassoc->associator ;
    MRT_AssociationRole const *srole = &cassoc->source ;

    switch (srole->storageType) {
    case mrtSingular: {
        <<mrtCheckDupAssociator: singular reference>>
    }
        break ;
    case mrtArray:
        mrtFatalError(mrtStaticRelationship) ;
        break ;

    case mrtLinkedList: {
        <<mrtCheckDupAssociator: linked list reference>>
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

If the source instance has a singular back link to the associator class instance there is only one possible target to which it is related.

```
<<mrtCheckDupAssociator: singular reference>>=
void *assocInst = *(void **)((uintptr_t)source + srole->storageOffset) ;
if (assocInst != NULL) {
    void *currentTarget =
            *(void **)((uintptr_t)assocInst + arole->forwardOffset) ;
    if (currentTarget == target) {
        mrtDupAssociatorError(rel) ;
    }
}
```

If the source instance has a linked list of back references to the associator class instance, then we have to set up an interation over the list to track down the related targets.

```
<<mrtCheckDupAssociator: linked list reference>>=
MRT_LinkRef *linksList =
        (MRT_LinkRef *)((uintptr_t)source + srole->storageOffset) ;     // ❶
for (MRT_LinkRef *assocLink = mrtLinkRefBegin(linksList) ;
        !(assocLink == NULL || assocLink == mrtLinkRefEnd(linksList)) ;
        assocLink = assocLink->next) {
    assert(srole->linkOffset != 0) ;
    void *assocInst = (void *)((uintptr_t)assocLink - srole->linkOffset) ;// ❷
    void *currentTarget = *(void **)((uintptr_t)assocInst + arole->forwardOffset) ;
    if (currentTarget == target) {
        mrtDupAssociatorError(rel) ;
        break ;
    }
}
```

❶ The linked list terminus for the back links is in the source instance.

❷ Recover the pointer to the beginning of the associative instance by subtracting off the offset to where the linked list pointers are stored.

Deleting Relationship Linkage

We must also provide a function to delete relationship linkage pointers.

```
<<mrt forward references>>=
static void mrtDeleteLinks(MRT_Relationship const * const *classRels,
        unsigned relCount, void *inst) ;
```

classRels
A pointer to an array of relationships description pointers that describe the relationships in which `inst` is a participant.

relCount
The number of elements in the `classRels` array.

inst
A pointer to the class instance that is to be unlinked from its relationship.

The implementation for deleting relationship links follows a similar pattern as for creating them. There are four distinct types of relationships and so we must consider each one.

```
<<mrt static functions>>=
static void
mrtDeleteLinks(
    MRT_Relationship const * const *classRels,
    unsigned relCount,
    void *inst)
{
    assert(inst != NULL) ;
    /*
     * Mark the transaction since we are updating the reference pointers.
     */
    mrtMarkRelationship(classRels, relCount) ;

    MRT_Instance *instref = inst ;
    MRT_Class const *const instclass = instref->classDesc ;

    for ( ; relCount != 0 ; relCount--, classRels++) {
        struct mrtrelationship const * const rel = *classRels ;
        switch (rel->relType) {
        case mrtSimpleAssoc: {
            <<mrtDeleteLinks: unlink simple association>>
        }
            break ;

        case mrtClassAssoc: {
            <<mrtDeleteLinks: unlink class based association>>
        }
            break ;

        case mrtRefGeneralization: {
            <<mrtDeleteLinks: unlink reference generalization>>
        }
```

```
            break ;

        case mrtUnionGeneralization:
            // For a union generalization, there are no pointer links.
            break ;

        default:
            mrtFatalError(mrtRelationshipLinkage) ;
            break ;
        }
    }
}
```

To unrelate simple associations, we must verify that the classes of the instance actually participate in the association and the perform the pointer operations to unlink the instances.

```
<<mrtDeleteLinks: unlink simple association>>=
MRT_SimpleAssociation const *assoc = &rel->relInfo.simpleAssociation ;

if (instclass == assoc->source.classDesc) {
    <<mrtDeleteLinks: unlink simple forward>>
} else if (instclass == assoc->target.classDesc) {
    <<mrtDeleteLinks: unlink simple backward>>
} else {
    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Unlinking a simple association in the forward direction implies that we are unlinking the primary reference. This primary reference is always singular and always non-NULL. When we are unlinking the primary reference we will follow the reference and try to unlink any back references.

```
<<mrtDeleteLinks: unlink simple forward>>=
MRT_AssociationRole const *sourceRole = &assoc->source ;

if (sourceRole->storageType == mrtSingular) {
    void **p_targetInst = (void **)
            ((uintptr_t)inst + sourceRole->storageOffset) ;
    MRT_Instance *targetInst = *p_targetInst ;
    *p_targetInst = NULL ; // ❶

    MRT_AssociationRole const *targetRole = &assoc->target ;
    if (targetInst != NULL && targetInst->alloc > 0 &&
            targetInst->classDesc == targetRole->classDesc) { // ❷
        mrtUnlinkBackref(targetRole, inst, targetInst) ;
    }
} else {
    // Simple forward association links are always singular.
    mrtFatalError(mrtRelationshipLinkage) ;
}
```

❶ Assigning `NULL` to the primary reference effectively unlinks the target instance.

❷ We only want to attempt to unlink the back reference if the target instance is still allocated and of the same class. We accomodate the `NULL` case also, since this can arise if the construction of the instance errored out because of a static relationships. Normally, instance don't just change their class out from under themselves. However, union based subclass instances do. Migrating a union subclass instance results in any back reference pointers to the instance remaining valid, but the class of the instance now occupying the memory has changed. Union subclasses are somewhat painful.

The case of unlinking back references for a simple association, occurs when the target instance is being deleted. In this case, we only delete the back references made by the target. In particular we make no attempt to follow the source references and delete

the primary reference pointer. If the referring instance is not later deleted, then referential integrity checking will find a pointer to an unallocated instance. This will be caught in referential integrity checking. we must be prepared for the fact that the back references are already gone. This is because it is possible to delete the referenced instance of a simple association before deleting the referring instance.

```
<<mrtDeleteLinks: unlink simple backward>>=
MRT_AssociationRole const *targetRole = &assoc->target ;

void *linkStorage = (void *)((uintptr_t)inst + targetRole->storageOffset) ;
switch (targetRole->storageType) {
case mrtSingular: {
    void **p_sourceInst = linkStorage ;
    *p_sourceInst = NULL ;                                    // ❶
}
    break ;

case mrtLinkedList: {                                         // ❷
    MRT_LinkRef *sourceList = linkStorage ;                   // ❸
    assert(sourceList->next != NULL && sourceList->prev != NULL) ;
    for (MRT_LinkRef *iter = mrtLinkRefBegin(sourceList) ;
            iter != mrtLinkRefEnd(sourceList) ; ) {
        MRT_LinkRef *sourceInst = iter ;
        iter = iter->next ;                                   // ❹
        mrtLinkRefRemove(sourceInst) ;
    }
}
    break ;

case mrtArray: {
    MRT_ArrayRef * alinks = linkStorage ;                     // ❺
    if (alinks->links != NULL) {
        // Can't unlink array type linkages.
        mrtFatalError(mrtStaticRelationship) ;
    }
}
    break ;

default:
    mrtFatalError(mrtRelationshipLinkage) ;
    break ;
}
```

❶ For the singular pointer case we can just write `NULL` to the back reference. Even if it was already `NULL`, it won't matter. The link is severed.

❷ In this case, the instance is being deleted and all of its back links must be removed.

❸ This is the list of back links to source instances.

❹ Be careful to advance the iterator before deleting the item from the list.

❺ We allow attempting to unlink array backlinks as long as there aren't really any backlinks. This case can arise in creating instances that participate in static relationships and then trying to delete them after catching the fatal error.

In this function we are trying to delete a single back link from the "target" to the "source". This is the back link deletion associated with deleting the primary reference pointer and we have tracked to a target instance.

```
<<mrt forward references>>=
static void
mrtUnlinkBackref(
    MRT_AssociationRole const *const targetRole,
```

```
    void *source,
    void *target) ;
```

```
<<mrt static functions>>=
static void
mrtUnlinkBackref(
    MRT_AssociationRole const *const targetRole,
    void *const source,
    void *const target)
{
    assert(source != NULL) ;
    assert(target != NULL) ;
    assert(targetRole != NULL) ;

    void *linkStorage =
            (void *)((uintptr_t)target + targetRole->storageOffset) ;
    switch (targetRole->storageType) {
    case mrtSingular: {
        void **p_sourceInst = linkStorage ;                              // ❶
        if (*p_sourceInst == source) {                                   // ❷
            *p_sourceInst = NULL ;
        }
    }
        break ;

    case mrtLinkedList: {
        MRT_LinkRef *srcblinks = (MRT_LinkRef *)
                ((uintptr_t)source + targetRole->linkOffset) ;           // ❸
        if (srcblinks->next != NULL && srcblinks->prev != NULL) {        // ❹
            MRT_LinkRef *targetList = linkStorage ;
            for (MRT_LinkRef *targetlink = mrtLinkRefBegin(targetList) ;
                    targetlink != mrtLinkRefEnd(targetList) ;
                    targetlink = targetlink->next) {
                if (targetlink == srcblinks) {
                    mrtLinkRefRemove(srcblinks) ;
                    break ;                                              // ❺
                }
            }
        }
    }
        break ;

    case mrtArray: {
        MRT_ArrayRef *alinks = linkStorage ;
        if (alinks->links != NULL) {                                     // ❻
            // Can't unlink array type linkages.
            mrtFatalError(mrtStaticRelationship) ;
        }
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

❶ The entity at the storageOffset is a simple pointer back to the source instance. Point to where the source reference is located in the target instance.

❷ We only want to unlink the backref if it is actually pointing back to the source.

❸ The source instance is linked onto a list whose list terminus is contained in the target instance. We must make sure the source instance is actually linked on the list before we try to remove it from the list. This means running the list to find the match. We only need a pointer to the links member in the source instance to remove it from the list (it is doubly linked). The links are offset into the source instance by the `linkOffset` given in the "target" role relationship descriptor.

❹ Check that the instance is actually linked on the list. If the source has been deleted previously, then it will have been removed from the list already.

❺ Once found and removed, we no longer need to iterate on the rest of the list.

❻ It's alright to unlink a static relationships as long as there were no links to begin with.

For class based associations, we deal with the association in two steps, between each participant and the associator class.

```
<<mrtDeleteLinks: unlink class based association>>=
MRT_ClassAssociation const *assoc = &rel->relInfo.classAssociation ;

if (instclass == assoc->associator.classDesc) {
    MRT_AssociatorRole const *assocRole = &assoc->associator ;
    MRT_AssociationRole const *sourceRole = &assoc->source ;
    MRT_AssociationRole const *targetRole = &assoc->target ;

    void **p_targetInst = (void **)
            ((uintptr_t)inst + assocRole->forwardOffset) ; // ❶
    MRT_Instance *targetInst = *p_targetInst ;
    *p_targetInst = NULL ;
    if (targetInst != NULL && targetInst->alloc > 0 &&
            targetInst->classDesc == targetRole->classDesc) {
        mrtUnlinkBackref(targetRole, inst, targetInst) ;
    }

    void **p_sourceInst = (void **)
            ((uintptr_t)inst + assocRole->backwardOffset) ;
    MRT_Instance *sourceInst = *p_sourceInst ;
    *p_sourceInst = NULL ;
    if (sourceInst != NULL && sourceInst->alloc > 0) {
        mrtUnlinkBackref(sourceRole, inst, sourceInst) ;
    }
}
```

❶ Start with the forward direction to the target. Point to where the target reference is located in the associator instance.

For reference generalizations, the references are singular pointers between the superclass and subclass instances. We must make sure the classes given actually participate in the generalization.

```
<<mrtDeleteLinks: unlink reference generalization>>=
MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;

if (instclass != gen->superclass.classDesc) {
    // Instance is a subclass instance
    int subclassCode = mrtFindRefGenSubclassCode(instclass, gen->subclasses,
            gen->subclassCount) ;
    // Obtain the pointer to the superclass instance.
    void **p_superInst = (void **)
            ((uintptr_t)inst + gen->subclasses[subclassCode].storageOffset) ;
    MRT_Instance *superInst = *p_superInst ;
    *p_superInst = NULL ;
    // NULL out the pointer in the superclass instance pointing to the subclass
    // instance. Watch for a NULL reference to the superclass instances.
    // This can happen if the subclass was simply created on it's on.
    if (superInst != NULL && superInst->alloc > 0) {
```

```
        void **p_subInst = (void **)
                ((uintptr_t)superInst + gen->superclass.storageOffset) ;
        *p_subInst = NULL ;
    }
} else {
    // Instance is a superclass instance
    // NULL out the pointer to the subclass instance only,
    // i.e. only the back reference.
    void **p_subInst = (void **)((uintptr_t)inst + gen->superclass.storageOffset) ;
    *p_subInst = NULL ;
}
```

Reclassifying Subclasses

Generalization relationships represent a disjoint union of the subclasses. Because of this property, we know that there is an unconditional relationship between a subclass instance and a superclass instance and an unconditional relationship between a superclass instance and exactly one subclass instance from among all the subclasses of the generalization. This situation leads to the concept of a reclassify operation to allow a related subclass instance to migrate from one subclass to another. It is a powerful concept to model modal operations in a domain. The run-time provides a function to accomplish the details.

```
<<mrt internal external interfaces>>=
extern void *
mrt_Reclassify(
    MRT_Relationship const *rel,
    void *sub,
    MRT_Class const *const newSubclass) ;
```

rel
: A pointer to a relationship description for the relationship across which the reclassify operation is to happen.

sub
: A pointer to a class instance that is to be reclassified.

newSubclass
: A pointer to the class description to which the currently related subclass instance is to be reclassified.

The `mrt_Reclassify` function migrates the class of the `sub` instance along the `rel` generalization to be of the new class, `newSubclass`. Conceptually, reclassification implies that the `sub` instance is unrelated and deleted and a new instance of `newSubclass` is created and related to the superclass instance to which `sub` was previously related.

The return value of the function is an instance pointer to the newly created subclass instance.

Since we support two different ways to store subclass instances, the reclassification considers each type separately.

```
<<mrt external functions>>=
void *
mrt_Reclassify(
    MRT_Relationship const *rel,
    void *sub,
    MRT_Class const *const newSubclass)
{
    assert(rel != NULL) ;
    assert(newSubclass != NULL) ;

    MRT_Instance *currentSubInst = sub ;
    assert(currentSubInst != NULL) ;
```

```
    assert(currentSubInst->alloc > 0) ;

    void *newSubInst = NULL ;

    if (rel->relType == mrtRefGeneralization) {
        <<mrt_Reclassify: reclassify reference generalization>>
    } else if (rel->relType == mrtUnionGeneralization) {
        <<mrt_Reclassify: reclassify union generalization>>
    } else {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    return newSubInst ;
}
```

For reference type generalizations, we must perform the complete delete / create sequence.

```
<<mrt_Reclassify: reclassify reference generalization>>=
MRT_RefGeneralization const *const gen = &rel->relInfo.refGeneralization ;

/*
 * Verify the subclass instance is an instance of that class that
 * is part of the relationship.
 */
int subclassCode = mrtFindRefGenSubclassCode(currentSubInst->classDesc,
        gen->subclasses, gen->subclassCount) ;
/*
 * Fetch the superclass instance via the reference in the subclass instance.
 */
MRT_Instance *super = *(MRT_Instance **)((uintptr_t)sub +
        gen->subclasses[subclassCode].storageOffset) ;
/*
 * Check that the subclass instance is related to a superclass that is
 * the correct one for the relationship.
 */
if (gen->superclass.classDesc != super->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;
}
/*
 * Verify that the new subclass is indeed a subclass of the relationship.
 * We don't care about the subtype code and are only using
 * mrtFindRefGenSubclassCode() to validate the new requested subclass.
 */
mrtFindRefGenSubclassCode(newSubclass, gen->subclasses, gen->subclassCount) ;
/*
 * Delete the old subclass instance.  Deleting will cause the subclass
 * instance to be unlinked from the generalization.
 */
mrt_DeleteInstance(sub) ;
/*
 * Create a new instance of the new subclass.
 */
newSubInst = mrt_CreateInstance(newSubclass, MRT_StateCode_IG) ;
/*
 * Create the links to the super class instance.
 */
mrt_CreateSimpleLinks(rel, newSubInst, super, true) ;
```

For generalizations implemented as unions, the reclassification sequence is much simpler. There is no need to delete and create a new instance of the new subclass. We need only transform the currently related instance into the new subclass.

```
<<mrt_Reclassify: reclassify union generalization>>=
```

```
MRT_UnionGeneralization const *const gen = &rel->relInfo.unionGeneralization ;
/*
 * Verify the subclass instance is an instance of that class that
 * is part of the relationship. We don't need the subclass code.
 */
MRT_Class const *const subClass = currentSubInst->classDesc ;
mrtFindUnionGenSubclassCode(subClass, gen->subclasses, gen->subclassCount) ;
/*
 * Compute the pointer to the superclass instance.
 */
MRT_Instance *super = (MRT_Instance *)((uintptr_t)currentSubInst -
        gen->superclass.storageOffset) ;
/*
 * Check that the subclass instance is related to a superclass that is
 * the correct one for the relationship.
 */
if (gen->superclass.classDesc != super->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;
}
/*
 * Check that the new subclass is one that is part of this generalization.
 * We don't actually need the subclass code itself.
 */
mrtFindUnionGenSubclassCode(newSubclass, gen->subclasses, gen->subclassCount) ;
/*
 * Clean up any relationship pointers in the currently related instance.
 */
mrtDeleteLinks(subClass->classRels, subClass->relCount, currentSubInst) ;
/*
 * The new instance occupies the same memory as the old one.
 */
newSubInst = currentSubInst ;
/*
 * Set up the memory for the subclass instance according to the new subclass.
 */
mrtInitializeInstance(newSubInst, newSubclass, MRT_StateCode_IG) ;
```

This function is the counterpart for finding a subclass among the subclasses for a union generalization.

```
<<mrt static functions>>=
static int
mrtFindUnionGenSubclassCode(
    MRT_Class const *const subclassClass,
    MRT_Class const *const *subclasses,
    unsigned count)
{
    int subcode ;

    for (subcode = 0 ; subcode < count ; subcode++, subclasses++) {
        if (subclassClass == *subclasses) {
            return subcode ;
        }
    }

    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Chapter 26

Managing Execution

In this section, we discuss the rules and policies for managing execution sequencing. There are two means available to domain activities to control the sequencing of execution.

- Invoke an ordinary function.
- Generate an event to the instance of a class.

Not much needs to be said about invoking functions. Control is transferred to the entry point and runs until the function is complete, transferring control back to next statement of the caller. Typically, such functions are organized into those that are associated with the domain as whole, a particular class or the instances of a class. Such organization may be helpful to the programmer, but since they are directly supported by the implementation language, the `micca` run-time does not get involved in mediating them.

The run-time does get involved with those computations that must leave off at some point, waiting for some other action in the system or the external environment, and then resume execution maintaining the past history. This type of execution is implemented as a state machine.

State Machine Rules

Each class that has lifecycle behavior may have a state model associated with it and each instance of that class will have a state variable that allows it to execute as an state machine independent of the other instances of the class. The run-time supports a Moore type state model.

In the Moore formulation of state models, activity code is associated with states and is executed upon entry into a state. This is distinguished from the Mealy formulation where actions are associated with the transitions and are executed upon exiting a state. Much writing and discussion has been wasted attempting to justify one type of state model over another. What we know is they are computationally equivalent, *i.e.* we can prove that there is no problem that you can solve with a Moore machine that cannot also be solved with a Mealy machine and *vice versa*. Whether your application is easier to describe with one type rather than the other is something that you alone may decide. Moore machines are the traditional formulation for Executable UML and they have the simplest implementation structures. What we specifically reject here is any use of hierarchical state models. They are unnecessary and add complication that is not welcome. The power of computation in Executable UML is derived from the interaction of simple state machines each of which is tied to the lifecycle of a particular class. If you have some state model that is large and complicated where you think some other kind of higher order structure is needed, the usual reason is that you have multiple classes masquerading as one and further refinement of your analysis is necessary. That is not usually a welcome answer to the situation, because if the analyst had been able to conceive of a better solution, he/she would probably have done so already. When state models are used to describe the lifecycle behavior of a well defined class (and not sets of classes or the domain as whole) there is no need for more complicated state execution schemes such as hierarchies.

Generally, state activities affect other computations in the domain by updating instance attribute values or by generating events to other instances. The important distinction here is that the application code of the state activities does not deal with actually dispatching the events nor does it control which event is dispatched next.

Event Types

There are three types of events:

1. Transition events that cause transitions in state machines.
2. Polymorphic events that are mapped at runtime across a generalization hierarchy to transition events.
3. Creations events that support asynchronous instance creation.

Creation events are transition events that are also associated with an instance creation. Our strategy for creation events is to create the instance in an inactive state before queuing the event that will activate the instance and cause a transition. We will have need to distinguish between the various event types and use an enumeration to accomplish that.

```
<<mrt internal simple types>>=
typedef enum {
    mrtTransitionEvent,
    mrtPolymorphicEvent,
    mrtCreationEvent
} MRT_EventType ;
```

The process of signaling an event involves the following steps:

1. Obtain an **Event Control Block** (ECB) from the free pool of ECB's.
2. Set the values of the fields in the ECB.
3. Queue the ECB for later dispatch.

Event Control Block

The Event Control Block (ECB) is the primary data structure for signaling and dispatching events.

```
<<mrt internal aggregate types>>=
typedef struct mrtecb {
    struct mrtecb *next ;
    struct mrtecb *prev ;
    MRT_EventCode eventNumber ;
    MRT_AllocStatus alloc ;
    MRT_Instance *targetInst ;
    MRT_Instance *sourceInst ;
    MRT_DelayTime delay ;
    MRT_DelayTime reload ;
    alignas(max_align_t) MRT_EventParams eventParameters ;
} MRT_ecb ;
```

`next, prev`
: Event queuing is done by doubly linked lists and the links are allocated as part of the ECB as the `next` and `prev` members.

`eventNumber`
: The event number. Events are also encoded as small zero based sequential integers that are unique only within the class to which they are associated. Event numbers are ultimately used as an array index.

`alloc`
: The `alloc` member is yet another part of *event-in-flight* detection. We will discuss that more below when we discuss event dispatch. For now, this member of the ECB holds the value of the allocation counter for the instance that is the target of the event. So when an event is generated, a copy of the current value of the `alloc` member of the instance is stored in the ECB.

targetInst
A pointer to the instance that is to receive the event.

sourceInst
A pointer to the instance that is the signaler of the event. If the event is generated outside the context of an instance (*e.g.* in an domain operation), then this member value is set to `NULL`. The `sourceInst` member also serves an important role in enforcing the rules for delayed events. More on that later.

delay
The number of milliseconds to delay before the event is posted.

reload
The number of milliseconds to delay for the second and subsequent times the event is signaled. For one-shot delayed events, `reload` is set to zero. For periodic delayed events, `reload` contains periodic delay interval.

eventParameters
Storage for any parameter values passed along with the event. We discuss event parameter data storage below.

Event codes are captured by small integer values.

```
<<mrt interface simple types>>=
typedef uint8_t MRT_EventCode ;
```

Delay times are in milliseconds and we wish to have a wide dynamic range of times.

```
<<mrt interface simple types>>=
typedef uint32_t MRT_DelayTime ;
```

Note that there is no notion of priority contained in the ECB. Some software architectures queue events in a priority order. That is not supported here. Frankly, if you need event priorities to make your system work, then you need to revisit your design or look for a software architecture that supports multiple threads of execution.

Like all the other data structures, there is a storage pool for ECB's and we define a size for it here that can be overridden on the compiler command line. Sizing the pool for ECB's can be difficult. It must be worst case allocation as running out of ECB's is a fatal system error. The pool must be sized to account for the maximum number of events that can be in flight at the same time. This includes delayed events, since they can be considered to be slow flying events.

```
<<mrt interface constants>>=
#ifndef MRT_EVENT_POOL_SIZE
#   define MRT_EVENT_POOL_SIZE 32
#endif /* MRT_EVENT_POOL_SIZE */
```

We must allocate the memory for the ECB storage. As usual, storage is just an array of structures.

```
<<mrt static data>>=
static MRT_ecb mrtECBPool[MRT_EVENT_POOL_SIZE] ;
```

Event Parameter Storage

We need to design how events that carry parameteric data will operate. In this formulation of state machines, events may carry additional parameters. Space has to be allocated for that data. The difficulty is that the parameter data must be given a type. There are a couple of solutions, neither of which is very satisfying. We could collect all the parameters from all the state machines in the system and create a giant union. This would properly allocate the amount of parameter storage required and provide a type safe manner to deal with that data. Unfortunately, the parameters to states are scattered in very many places in a system and gathering them together is a difficult undertaking.

Here we take the approach of providing a fixed amount of memory and letting state activities cast that memory into the appropriate type. Needless to say, this can also be a source of errors, but is much easier to manage. This choice makes sense for many systems. The number of states that use parametric data is usually small and using a fixed size works better than might be expected upon

first consideration. The important point here is that events can carry data with them. Many state machine formulations don't support this and it is very difficult to correctly manage memory lifetime without it. It is one of those things that you might not use very often but it is difficult to do without when you need it.

We fix the amount of memory used for event parameter storage, allowing it to be overridden by the defining the appropriate macro.

```
<<mrt interface constants>>=
#ifndef MRT_ECB_PARAM_SIZE
#   define MRT_ECB_PARAM_SIZE  32
#endif /* MRT_ECB_PARAM_SIZE */
```

```
<<mrt interface simple types>>=
typedef char MRT_EventParams[MRT_ECB_PARAM_SIZE] ;
```

The platform model has been very careful to insure the type signatures of events and state activities. When signaling an event, the code generator will generate code to insert event parameters into the data area using the signature associated with the event. In the state activity, the parameters will be moved into local variables according to the signature of the state activity.

Note that the parameters are passed by value but care must be taken when passing pointer references to data (*e.g.* `NUL` terminated strings passed as pointers). The run-time does nothing to manage the life time of the storage when values are passed by reference.

Event Queues

In this run-time architecture, we do asynchronous event dispatch from an event queue. This is one of the simplest ways to insure that we meet the requirement for state activities to run to completion. Since a queue is used, as a state activity executes and potentially signals other events, we know those events will not be dispatched until after the state activity completes. Therefore, there is no danger of a long complicated chain of event dispatching cycling back around to alter the state of the instance or potentially modify some data value that the state activity accesses after generating the event. The guarantee of run-to-completion for state activity execution is very important. We now examine the code that performs the queueing.

To serve as the head of the linked lists, we define an event queue structure that just contains the `next` and `prev` pointers.

```
<<mrt implementation aggregate types>>=
typedef struct mrteventqueue {
    MRT_ecb *next ;
    MRT_ecb *prev ;
} MRT_EventQueue ;
```

There are four queues that are used to manage events.

```
<<mrt static data>>=
static MRT_EventQueue eventQueue ;
static MRT_EventQueue tocEventQueue ;
static MRT_EventQueue delayedEventQueue ;
static MRT_EventQueue freeEventQueue ;
```

The `eventQueue` queue holds events waiting for immediate dispatch. The events in this queue are those signaled as part of an ongoing thread of control. The `tocEventQueue` queue holds events that start a new thread of control. Typically, those are events generated outside of a domain or delayed events finally being delivered. The `delayedEventQueue` queue holds events that are to be delivered by the run-time at some future time. Finally, the `freeEventQueue` queue holds those ECB's that are not currently begin used. From the data structures and the semantics of the queuing, a given ECB can be on at most one of the queues at any time. Most of the time each ECB is on exactly one of the queues, but there are short times when an ECB is not in any queue and a pointer to the ECB is held in a local variable.

The operations that are performed on event queues are associated with adding and removing elements. This code is very conventional and I'm sure you seen it or something very much like it many times before.

```
<<mrt implementation static inlines>>=
static inline MRT_ecb *
mrtEventQueueBegin(
    MRT_EventQueue *queue)
{
    return queue->next ;
}
```

```
<<mrt implementation static inlines>>=
static inline MRT_ecb *
mrtEventQueueEnd(
    MRT_EventQueue *queue)
{
    return (MRT_ecb *)queue ;
}
```

```
<<mrt implementation static inlines>>=
static inline bool
mrtEventQueueEmpty(
    MRT_EventQueue *queue)
{
    return mrtEventQueueBegin(queue) == mrtEventQueueEnd(queue) ;
}
```

```
<<mrt implementation static inlines>>=
static inline void
mrtEventQueueInsert(
    MRT_ecb *item,
    MRT_ecb *at)
{
    item->prev = at->prev ;
    item->next = at ;
    at->prev->next = item ;
    at->prev = item ;
}
```

```
<<mrt implementation static inlines>>=
static inline void
mrtEventQueueRemove(
    MRT_ecb *item)
{
    item->prev->next = item->next ;
    item->next->prev = item->prev ;
    item->prev = item->next = NULL ; // ❶
}
```

❶ Although it is not strictly necessary to NULL out the pointers when an item is removed from the queue, we do depend upon this to know that an item has been removed from a list and can be placed into a different list.

Since we have a pool of ECB's, we need some operations to manage the pool. We start with initialization. This places all the ECB's in the pool onto the free event queue.

```
<<mrt static functions>>=
static void
mrtECBPoolInit(void)
{
    assert(MRT_EVENT_POOL_SIZE >= 1) ;
    /*
```

```
     * Initialize the queue terminus structures.
     */
    eventQueue.next = eventQueue.prev = (MRT_ecb *)&eventQueue ;
    tocEventQueue.next = tocEventQueue.prev = (MRT_ecb *)&tocEventQueue ;
    delayedEventQueue.next = delayedEventQueue.prev = (MRT_ecb *)&delayedEventQueue ;
    freeEventQueue.next = freeEventQueue.prev = (MRT_ecb *)&freeEventQueue ;
    /*
     * Place all the event control blocks on the free event
     * queue.  Allocation occurs from there.
     */
    for (MRT_ecb *ecb = mrtECBPool ;
            ecb < mrtECBPool + MRT_EVENT_POOL_SIZE ; ecb++) {
        mrtEventQueueInsert(ecb, mrtEventQueueEnd(&freeEventQueue)) ;
    }
}
```

Event allocation is just removing an ECB from the free list. *N.B.* that running out of Event Control Blocks is fatal.

```
<<mrt static functions>>=
static inline MRT_ecb *
mrtECBalloc(void)
{
    if (mrtEventQueueEmpty(&freeEventQueue)) {
        mrtFatalError(mrtNoECB) ;
    }

    MRT_ecb *ecb = freeEventQueue.next ;
    mrtEventQueueRemove(ecb) ;
    memset(ecb, 0, sizeof(*ecb)) ; // ❶
    return ecb ;
}
```

❶ We zero out the ECB. Although not strictly necessary, it is convenient for debugging, especially when there are event parameters involved.

```
<<mrt static functions>>=
static inline void
mrtECBfree(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;

    mrtEventQueueInsert(ecb, mrtEventQueueEnd(&freeEventQueue)) ;
}
```

We will have need to find particular events in a queue. Events are identified by the source of the event, the target of the event and the number of the event.

```
<<mrt static functions>>=
static MRT_ecb *
mrtFindEvent(
    MRT_EventQueue *queue,
    MRT_Instance *sourceInst,
    MRT_Instance *targetInst,
    MRT_EventCode event)
{
    /*
     * Simple iteration through the list of events in the queue.
     */
    for (MRT_ecb *iter = mrtEventQueueBegin(queue) ;
```

```
            iter != mrtEventQueueEnd(queue) ;
            iter = iter->next) {
        if (iter->sourceInst == sourceInst && iter->targetInst == targetInst &&
                iter->eventNumber == event) {
            return iter ;
        }
    }
    return NULL ;
}
```

Event Signaling

Signaling an event is a very common operation for a state activity. In this section, we describe the code to accomplish event signaling.

Obtaining An ECB

As we mentioned previously, before an event can be signaled, we must obtain and fill in an ECB. We start our description with a function that does just that.

```
<<mrt internal external interfaces>>=
extern MRT_ecb *
mrt_NewEvent(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The numerical code for the event.

target
A pointer to the instance that is to receive the event.

source
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

The `mrt_NewEvent` function allocates an ECB structure, fills in the elements and returns the newly minted ECB.

```
<<mrt external functions>>=
MRT_ecb *
mrt_NewEvent(
    MRT_EventCode event,
    void *target,
    void *source)
{
    MRT_Instance *targetInst = target ;
    MRT_Instance *sourceInst = source ;

    assert(targetInst != NULL) ;
    assert(targetInst->alloc != 0) ;
    assert(event < targetInst->classDesc->eventCount) ;

    MRT_ecb *ecb = mrtECBalloc() ;
```

```
    ecb->eventNumber = event ;
    ecb->alloc = targetInst->alloc ;     // ❶
    ecb->targetInst = targetInst ;
    ecb->sourceInst = sourceInst ;
    ecb->delay = 0 ;                     // ❷
    ecb->reload = 0 ;

    return ecb ;
}
```

❶ Note that we initialize the ECB `alloc` member from the target instance. This is an essential part of detecting *event in flight* errors.

❷ Delayed or periodic event signaling will overwrite the `delay` and `reload` members as necessary. Most frequently, we are performing immediate signaling.

Posting an Event

Once you have obtained an ECB initialized for an event, then you need to fill in any event parameter data. Frequently, there are no parameters for an event. Then the ECB is ready to be placed on a queue. There is a distinction between events an instance sends to itself and those that an instance sends to a different instance. Self directed events are placed on the front of the event queue so that they are dispatched in preference to the non-self directed events. This is one of the fundamental state machine dispatch rules. Posting an event involves determining the correct place in the queue for the ECB. The decision is made based on the ECB values.

```
<<mrt internal external interfaces>>=
extern void
mrt_PostEvent(
    MRT_ecb *ecb) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

The `mrt_PostEvent` function queues the ECB pointed to by `ecb` to an appropriate event queue.

```
<<mrt external functions>>=
void
mrt_PostEvent(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;
    assert(ecb->targetInst != NULL) ;

        /*
         * The location in some event queue where the ECB will be inserted.
         */
    MRT_ecb *qloc ;

    if (ecb->sourceInst == NULL || mrtTransLevel > 0) {          // ❶
        qloc = mrtEventQueueEnd(&tocEventQueue) ;
    } else if (ecb->sourceInst != ecb->targetInst) {             // ❷
        qloc = mrtEventQueueEnd(&eventQueue) ;
    } else {                                                     // ❸
        for (qloc = mrtEventQueueBegin(&eventQueue) ;
                qloc != mrtEventQueueEnd(&eventQueue) &&
                    qloc->sourceInst == qloc->targetInst ;
```

```
                qloc = qloc->next) {
            // N.B. -- empty loop
        }
    }

    mrtEventQueueInsert(ecb, qloc) ;
}
```

1. All events that are signaled outside of a state activity start a new thread of control and the event is queued to the end of the thread of control event queue. If the `sourceInst` is NULL, then the event was signaled directly from a domain operation as this is what the `micca` code generator arranges. If the transaction level is non-zero, then we are in the middle of a synchronous service. This case can arise when a domain operation invokes an instance operation. The source of the event, in that case, is the instance, but the context is still outside of a state activity.
2. Ordinary transitioning events directed between distinct instances are queued to the end of the event queue.
3. Self directed events are queued to the front of the event queue, but we have to find the appropriate location. That location is the first event in the queue that is not self directed. The loop preserves the order of event dispatch in the highly unlikely (but not technically illegal) case that a state activity signals multiple self directed events.

One Step Event Signaling

For the case where the event has no parameters or the parameters have already been marshalled into a parameter block, a single function can create the event and post it.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalEvent(
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize) ;
```

`event`
The numerical code for the event.

`targetInst`
A pointer to the instance that is to receive the event. If `targetInst` is `NULL`, then no event is signaled.

`sourceInst`
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

`eventparams`
A pointer to the event parameters for the event. If the event requires no additional parameters, then this argument is passed as `NULL`.

`paramsize`
The number of bytes of event parameter data pointed to by `eventparams`. If the event requires no additional parameters, then this argument is passed as `0`.

```
<<mrt external functions>>=
void
mrt_SignalEvent(
    MRT_EventCode event,
    void *targetInst,
```

```
    void *sourceInst,
    void const *eventparams,
    size_t paramsize)
{
    if (targetInst == NULL) {
        return ;                                                    // ❶
    }

    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    if (eventparams != NULL) {
        assert(paramsize <= sizeof(ecb->eventParameters)) ;
        size_t toCopy = paramsize <= sizeof(ecb->eventParameters) ?
                paramsize : sizeof(ecb->eventParameters) ;
        memcpy(ecb->eventParameters, eventparams, toCopy) ;
    }

    mrt_PostEvent(ecb) ;
}
```

❶ We interpret signaling the `NULL` instance as a request to do nothing. This can be used to avoid tests against `NULL` when the result of a relationship navigation needs to be the target of the event.

Asynchronous Instance Creation

As we mentioned earlier, asynchronous instance creation is accomplished by a combination of creating an instance and then sending it an event. The transition caused by the event then causes a state activity to be executed. The creation is asynchronous in the sense that after an activity requests an asynchronous creation, the return from the request is immediate, but the instance does not come into existance until its creation event is dispatched.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst) ;
```

targetClass
A pointer to the class description for the instance that is to be created.

event
The number of the event to signal to the newly created instance.

eventparams
A pointer to the event parameters for the creation event.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`.

sourceInst
A pointer to the class instance that is the source of the creation event. If the creation event is signaled outside of a state activity, then this value should be `NULL`.

The `mrt_CreateInstanceAsync` function asynchronously creates an instance of the class described by `targetClass` and arranges for `event` to be signaled to the new instance. Note the event parameters must be provided. After the function returns, the instance has been created in an inactive state and the event has been posted. The return value is the pointer to the newly created instance and can be used to initialize the values of attribute of the instance.

```
<<mrt external functions>>=
void *
mrt_CreateInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst)
{
    assert(targetClass != NULL) ;
    assert(targetClass->edb != NULL) ;
    assert(targetClass->edb->creationState >= 0) ;
    assert(event < targetClass->edb->eventCount) ;

    MRT_Instance *targetInst = mrt_CreateInstance(targetClass,
            targetClass->edb->creationState) ;                    // ❶
    targetInst->alloc = -targetInst->alloc ;                      // ❷

    mrt_SignalEvent(event, targetInst, sourceInst, eventparams, paramsize) ;

    return targetInst ;
}
```

❶ We want the instance created in the pseudo-initial state.

❷ The `alloc` member is made negative for instances that are awaiting the dispatch of a creation event. Effectively the negative `alloc` value signals that the memory is reserved but the instance is not yet active. During event dispatch, this value will be turned back into a positive one. Note that the effect here is to use one of the instance slots as a temporary

buffer to hold the attribute values for the instance until the creation event is dispatched. This temporary usage must be accounted for when defining the number of instance memory slots for a class.

As we have already seen, subclass instances stored as a union require separate treatment. The following function is the asynchronous counterpart to `mrt_CreateUnionInstance`.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateUnionInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst,
    MRT_Relationship const *const genRel,
    void *super) ;
```

targetClass
A pointer to the class description for the instance that is to be created. The class described by `classDesc` must be a subclass of the relationship described by `genRel`.

event
The number of the event to signal to the newly created instance.

eventparams
A pointer to the event parameters for the creation event.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`.

sourceInst
A pointer to the class instance that is the source of the creation event. If the creation event is signaled outside of a state activity, then this value should be `NULL`.

genRel
A pointer to a relationship description for the generalization in which the instance is a union-based subclass. The `relType` field of `genRel` must be `mrtUnionGeneralization`.

super
A pointer to the superclass instance where the subclass instance is to be created. `super` must be an instance of the superclass described by `genRel`.

The `mrt_CreateUnionInstanceAsync` function asynchronously creates an instance of the union-based subclass described by `targetClass` and arranges for `event` to be signaled to the new instance. Note the event parameters must be provided. After the function returns, the instance has been created in an inactive state and the event has been posted. The return value is the pointer to the newly created instance and can be used to initialize the values of attribute of the instance.

Like the `mrt_CreateUnionInstance` function, the main change here is that the memory for the instance is located in the space allocated for its related superclass instance.

```
<<mrt external functions>>=
void *
mrt_CreateUnionInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst,
```

```
    MRT_Relationship const *const genRel,
    void *super)
{
    assert(targetClass != NULL) ;
    assert(targetClass->edb != NULL) ;
    assert(targetClass->edb->creationState >= 0) ;
    assert(event < targetClass->edb->eventCount) ;

    MRT_Instance *targetInst = mrt_CreateUnionInstance(targetClass,
            targetClass->edb->creationState, genRel, super) ;
    targetInst->alloc = -targetInst->alloc ;                          // ❶

    mrt_SignalEvent(event, targetInst, sourceInst, eventparams, paramsize) ;

    return targetInst ;
}
```

❶ As with non-union classes, we mark the instance as allocated but not active. When the creation event is dispatched, the allocation status is changed to indicate the instance is active.

Delayed Events

The concept of a delayed event is to request the run-time to post an event at some time in the future. This implies that the run-time has access to some type of timing facility by which it can know that a given amount of time has elapsed and this implies that the run-time will hold on to the ECB until that future time has arrived. We also interpret delayed events as being signaled from outside of an instance context. So, all delayed events start a new thread of control when they are dispatched.

There is one significant XUML rule associated with delayed events. There can be only one outstanding delayed event of a given event type between any sending / receiving pair of instances (which may be the same instance). This is another way of stating that delayed events are identified by their event name (or numerical encoding), the target instance and the source instance. There are a number of ways to interpret an attempt to generate what amounts to a duplicate delayed event. It could be considered an error, but that is inconvenient and goes against the grain of our attempts to minimize run-time errors. So the run-time regards an attempt to generate a delayed event of the same name between the same sending and receiving pair as a request to cancel the original event and create the new one at its newly given time. This turns out to be very convenient in practice, eliminating the need to perform checks. Cancelling and reinstating a new event turns out to be what is desired in most circumstances.

To understand the implementation of delayed events, it is necessary to understand the way the delayed event queue is maintained. The run-time has a delayed event queue where ECB's are placed awaiting to be posted. In servicing the delayed events, we are particularly trying to avoid doing any periodic computation. For example, we could treat the delayed event queue as a simple list and wake up periodically and run down the list decrementing time values and checking if any events have expired. Such a scheme is easy to implement, but in highly embedded and power sensitive application, periodic activity of this type is wasteful and deemed inappropriate.

In this implementation, we keep the delayed event queue in time relative order. This design meets two important criteria; only a single source of timing is used and there is no periodic execution activity. The cost of meeting these criteria is the price paid to find the appropriate place in the delayed event queue when a delayed event is requested.

There is sometimes a temptation for analysts to try to use delayed events for precise timing control over external interactions. For example, you could conceive of using delayed events as a means of generating a pulse width modulated (PWM) square wave to control a motor. Delayed events are generally **not** suitable for high speed, high precision timing such as that required by motor control. Since we are using only a single timing resource that is shared in the system, we are not able to achieve the precise jitter free timing that would be needed for a PWM. Most microcontrollers have many timing resources available in hardware and they should be used for functions that require high precision. What delayed events are most useful for is time outs for interactions in the millisecond to minutes range and for those functions where some jitter in the timing is acceptable (*e.g.* blinking an LED). The delayed event time is only guaranteed to be the minimum amount of time that is to elapse before the event is delivered.

There are three functions supplied for dealing with delayed events:

1. Post a delayed event.

2. Cancel a delayed event.
3. Query the remaining time for a delayed event.

The unit of time for delayed events is milliseconds.

Posting A Delayed Event

```
<<mrt internal external interfaces>>=
extern void
mrt_PostDelayedEvent(
    MRT_ecb *ecb,
    MRT_DelayTime time) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

time
The minimum number of milliseconds of time that must elapse before the event given by `ecb` is posted for dispatch.

The `mrt_PostDelayedEvent` function requests that the event given by `ecb` be dispatched no sooner than `time` milliseconds from now. The value of the `time` argument may be 0, in which case the event is posted for dispatch immediately. All delayed events start a new thread of control. Any request to post a delayed event of the same event number that already exists for some sending / receiving pair of instances results in the first event being canceled and a the event being posted at the given delay time.

```
<<mrt external functions>>=
void
mrt_PostDelayedEvent(
    MRT_ecb *ecb,
    MRT_DelayTime time)
{
    assert(ecb != NULL) ;

    mrtStopDelayedQueueTiming() ;                                    // ❶
    mrtRemoveDelayedEvent(ecb->sourceInst, ecb->targetInst,
            ecb->eventNumber) ;
    ecb->delay = mrtMsecToTicks(time) ;
    mrtInsertDelayedEvent(ecb) ;                                     // ❷

    mrtStartDelayedQueueTiming() ;                                   // ❸
}
```

❶ The timing queue must be stopped so we may examine it safely.

❷ Note that even if the requested time is zero, we insert the event into the delayed event queue anyway. This insures that it is ordered properly relative to any events that just happened to have expired at the time the queue was stopped.

❸ Starting the timing of the queue implies that any expired events are placed in the thread of control queue before the delayed queue timing is restarted.

We see here the distinction between posting a delayed event and the ordinary posting of an event. For delayed events, the event will end up on the delayed event queue or the thread of control queue. The determining factor is the delay time. If the delay is zero, then the event goes to the thread of control queue directly. Otherwise, it is placed on the thread of control queue after the delay time. Delayed events never go to the ordinary event queue.

There are five main actions for inserting a delayed event.

1. Convert the delay value from millisecond units to units of `ticks`. A tick is platform specific. Computers sometimes don't typically keep time in conventional human units, particularly small embedded systems as we are dealing with. However, we would like to run the delayed event queue in system specific units to avoid as much unnecessary conversion as we can. This conversion will be described below for each supported platform.
2. Stop the timing of the delayed event queue. More on this later but the goal of stopping the delayed event queue timing is to freeze the state of the queue so that we may operate on it.
3. Determine if there is already an event matching the one begin posted. This enforces the rule about not having two delayed events of the same type between the same sending / receiving pair. If one is found then it is removed.
4. Assured of no duplicates, the new event can be inserted into the timing queue.
5. Finally, the timing of the delayed queue is started.

Posting A Periodic Event

```
<<mrt internal external interfaces>>=
extern void
mrt_PostPeriodicEvent(
    MRT_ecb *ecb,
    MRT_DelayTime initial,
    MRT_DelayTime reload) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

initial
The minimum number of milliseconds of time that must elapse before the event given by `ecb` is first posted for dispatch.

reload
The minimum number of milliseconds of time that must elapse after the first posting of the `ecb` before it is reposted periodically after each `reload` interval.

The `mrt_PostPeriodicEvent` function requests that the event given by `ecb` be dispatched no sooner than `initial` milliseconds from now and that the `ecb` be reposted `reload` milliseconds after that. Reposting in `reload` milliseconds continues until the event is cancelled. The value of the `initial` argument may be 0, in which case the event is posted for dispatch immediately. The value of the `repost` argument must be non-zero if the event is to be treated as a periodic event. Invoking `mrt_PostPeriodicEvent` with a `repost` value of zero results in an delayed event. Like delayed events, all periodic events start a new thread of control. Any request to post a periodic event of the same event number that already exists for some sending / receiving pair of instances results in the first event being canceled and the event being posted at the given delay time.

```
<<mrt external functions>>=
void
mrt_PostPeriodicEvent(
    MRT_ecb *ecb,
    MRT_DelayTime initial,
    MRT_DelayTime reload)
{
    assert(ecb != NULL) ;
    ecb->reload = mrtMsecToTicks(reload) ;
    mrt_PostDelayedEvent(ecb, initial) ;
}
```

One Step Delayed Event Signaling

Just as for immediate events, delayed events without additional parameters or for which the parameters have been marshalled into a parameter block, a single function can both fill in an ECB and post it to the delayed event queue.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalDelayedEvent(
    MRT_DelayTime time,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize) ;
```

time
The minimum number of milliseconds of time that must elapse before the event is to be dispatched.

event
The numerical code for the event.

targetInst
A pointer to the instance that is to receive the event.

sourceInst
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

eventparams
A pointer to the event parameters for the event. If the event requires no additional parameters, then this argument is passed as `NULL`.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`. If the event requires no additional parameters, then this argument is passed as `0`.

```
<<mrt external functions>>=
void
mrt_SignalDelayedEvent(
    MRT_DelayTime time,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize)
{
    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    if (eventparams != NULL) {
        assert(paramsize <= sizeof(ecb->eventParameters)) ;
        size_t toCopy = paramsize <= sizeof(ecb->eventParameters) ?
                paramsize : sizeof(ecb->eventParameters) ;
        memcpy(ecb->eventParameters, eventparams, toCopy) ;
    }

    mrt_PostDelayedEvent(ecb, time) ;
}
```

One Step Periodic Event Signaling

Periodic events should not carry any additional parameters. It is not possible to change the parameter values between event dispatches and any information used for the parameter values can be available in the model when the event is received.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalPeriodicEvent(
    MRT_DelayTime initial,
    MRT_DelayTime reload,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst) ;
```

initial
The minimum number of milliseconds of time that must elapse before the event given by `ecb` is first posted for dispatch.

reload
The minimum number of milliseconds of time that must elapse after the first posting of the `ecb` before it is reposted periodically after each `reload` interval.

event
The numerical code for the event.

targetInst
A pointer to the instance that is to receive the event.

sourceInst
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

```
<<mrt external functions>>=
void
mrt_SignalPeriodicEvent(
    MRT_DelayTime initial,
    MRT_DelayTime reload,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst)
{
    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    mrt_PostPeriodicEvent(ecb, initial, reload) ;
}
```

Inserting Delayed Events

```
<<mrt static functions>>=
static void
mrtInsertDelayedEvent(
    MRT_ecb *ecb)
{
    MRT_ecb *iter = mrtEventQueueBegin(&delayedEventQueue) ;
    MRT_ecb *const end = mrtEventQueueEnd(&delayedEventQueue) ;
    for ( ; iter != end ; iter = iter->next) {
        if (ecb->delay < iter->delay) {                          // ❶
```

```
            iter->delay -= ecb->delay ;                          // ❷
            break ;
        } else {
            ecb->delay -= iter->delay ;
        }
    }
    mrtEventQueueInsert(ecb, iter) ;
}
```

❶ By keeping this comparison to be strictly less than, we preserve the order of event dispatch to match that of event generation.

❷ We are going to insert before the entry pointed to by `iter`. Therefore, we need to decrease its delay value by the amount of time that will have elapsed after the entry we are about to insert expires.

Removing Delayed Events

Since removing delayed events is something we do when canceling a delayed event, we factor out the code into a function.

```
<<mrt static functions>>=
static void
mrtRemoveDelayedEvent(
    MRT_Instance *sourceInst,
    MRT_Instance *targetInst,
    MRT_EventCode eventNumber)
{
    MRT_ecb *found = mrtFindEvent(&delayedEventQueue, sourceInst,
            targetInst, eventNumber) ;                           // ❶
    if (found != NULL) {
        if (found->next != mrtEventQueueEnd(&delayedEventQueue)) {   // ❷
            found->next->delay += found->delay ;
        }
    } else {
        found = mrtFindEvent(&tocEventQueue, sourceInst, targetInst,
                eventNumber) ;
    }
    if (found != NULL) {                                         // ❸
        mrtEventQueueRemove(found) ;
        mrtECBfree(found) ;
    }
}
```

❶ If the event already exists, remove it. First we search the delayed event queue. If we don't find the event there, it could have already expired and it would then be on the Thread of Control event queue. If we find it in either place, it is a duplicate event and it is removed. This is in keeping with our interpretation that a duplicate delayed event is a request to cancel the first and create one with a new delay time.

❷ If the event is not at the end of the queue, all the delay from the removed entry is added onto the next entry in the queue. This insures that the time associated with the deleted event is accounted for when subsequent events expire.

❸ If we get here and `found` is still `NULL`, then there was no delayed event matching the source / target / event and we just do nothing.

Canceling Delayed Events

Cancelling a delayed event is one of the more complicated delayed event operations. We must account for the various places where a delayed event may be queued.

A delayed event may be in one of two places:

1. In the delayed event queue awaiting either waiting for its time to expire or having already been marked as expired.
2. In the thread of control event queue awaiting dispatch.

We will have more to say about event dispatch below, but it is possible to try to cancel an event after its time has expired but before it has been delivered to the target instance. The mechanisms make the guarantee that after invoking `mrt_EventDelayCancel` the application can be assured that the event will **not** be delivered until such time when it is posted again. Note that it is possible to attempt to cancel a delayed event after it has already been delivered. This is not an error. Unfortunately, the run-time code cannot turn time backwards.

```
<<mrt internal external interfaces>>=
extern void
mrt_CancelDelayedEvent(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The number of the event.

targetInst
A pointer to the instance structure that is to receive the event.

sourceInst
A pointer to the instance structure that is sending the event. Events generated outside of a class instance set this argument to `NULL`.

The `mrt_CancelDelayedEvent` function cancels the delayed event given by `event` and signaled from `source` to `target`. After the return from this function the event is guaranteed not to be delivered.

```
<<mrt external functions>>=
void
mrt_CancelDelayedEvent(
    MRT_EventCode event,
    void *target,
    void *source)
{
    assert(target != NULL) ;

    mrtStopDelayedQueueTiming() ;                          // ❶
    mrtRemoveDelayedEvent(source, target, event) ;
    mrtStartDelayedQueueTiming() ;
}
```

❶ As before, we must stop the delayed queue so we can examine and manipulate it. We can't have delayed queue operations go on while we are walking and changing the queue contents.

Like all operations dealing with the delayed event queue, we must first put the queue in a state that we can examine it without asynchronous timing services modifying its state. Then we search the delayed event queue for the event to cancel. The only complication in the implementation is the need to search the thread of control event queue should the ECB have already been expired off of the delayed queue.

Time Remaining for a Delayed Event

The last provided operation on delayed events is to query the amount of time remaining for a particular delayed event. Since we hold the delayed events in sorted order of time differences, the task of determining the amount of remaining time involves

traversing the queue and summing the time increments of all the events in front of the event of interest. The only special case here is what to do if we don't find the delayed event at all. In that case, zero is returned.

```
<<mrt internal external interfaces>>=
extern MRT_DelayTime
mrt_RemainingDelayTime(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The number of the event.

targetInst
A pointer to the instance structure that is to receive the event.

sourceInst
A pointer to the instance structure that is sending the event. Events generated outside of a class instance set this argument to `NULL`.

The `mrt_RemainingDelayTime` function returns the amount of time remaining before the `event` is posted from `source` to `target`. The remaining time will be 0 if the event has already been posted or is such a delayed event does not exist. A return value of 0 does *not* imply that the event has already been dispatched.

The algorithm for computing the remaining time is to simply walk the delayed event queue from the beginning, summing up the set of time delays until we reach the ECB of delayed event.

```
<<mrt external functions>>=
MRT_DelayTime
mrt_RemainingDelayTime(
    MRT_EventCode event,
    void *target,
    void *source)
{
    MRT_Instance *targetInst = target ;
    MRT_Instance *sourceInst = source ;
    assert(targetInst != NULL) ;

    mrtStopDelayedQueueTiming() ;
    /*
     * Iterate through the delayed event time and sum all the delay times to
     * give the total amount of time remaining for the found event.
     */
    MRT_DelayTime remain = 0 ;
    MRT_ecb *iter ;
    MRT_ecb *endOfQueue = mrtEventQueueEnd(&delayedEventQueue) ;
    for (iter = mrtEventQueueBegin(&delayedEventQueue) ; iter != endOfQueue ;
            iter = iter->next) {
        remain += iter->delay ;
        if (iter->sourceInst == sourceInst && iter->targetInst == targetInst &&
                iter->eventNumber == event) {
            break ;
        }
    }
    mrtStartDelayedQueueTiming() ;
    /*
     * Return the amount of time remaining for the event.  If we didn't find
     * the event, the just return 0.
     */
```

```
    return iter == endOfQueue ? 0 : mrtTicksToMsec(remain) ;
}
```

Note that zero is returned if we did not find the event on the delayed queue. Returning zero does *not* tell us if the event has already been dispatched or might be still in flight on the event queue.

Expired Events in the Delayed Event Queue

When the delay time expires there are several ways to deal with the events that need to be dispatched. One approach would be simply to sync to the background and let code run there to remove expired events from the delayed event queue and restart the timer. Such a design is subject to problematic, unpredictable timing skew. After the timer source has expired and posted the synchronization request, the background may not execute immediately. Either a currently executing state activities or a previously queued sync function would delay the transfer of expired events from the delayed queue to the event queue. If there if another delayed event to expire, we would like to start the expiration timing without inserting some unpredictable time in between. Essentially, we would like to overlap the timing of the next delayed event with the time it takes to complete the current activities and execute the timer sync request.

This design chooses to remove each expired delayed event and pass the ECB as a parameter to a sync function which, when it runs, queues the ECB to the TOC event queue. Periodic events are handled similarly. The ECB is removed from the delayed event queue and a sync function is invoked. However, the parameters to the sync function are retrieved from the ECB rather than passing the ECB pointer itself. The ECB delay time is then reset to the reload time and then reinserted into the delayed event queue. When the sync function for periodic events runs, it must allocate a new ECB and fill some of the fields from the parameters passed to the sync function.

The concept of starting the delayed event queue timing is associated with starting the timing resource with the delay time of the event on the front of the delayed queue. We move the delay value from the head of the delayed event queue into the timer and zero the delay member.

```
<<mrt forward references>>=
static void mrtStartDelayedQueueTiming(void) ;
```

```
<<mrt static functions>>=
static void
mrtStartDelayedQueueTiming(void)
{
    mrtQueueExpiredDelayedEvents() ;                          // ❶

    if (!mrtEventQueueEmpty(&delayedEventQueue)) {
        MRT_ecb *ecb = mrtEventQueueBegin(&delayedEventQueue) ;
        assert(ecb->delay != 0) ;
        mrtSysTimerStart(ecb->delay) ;
        ecb->delay = 0 ;
    }
}
```

❶ In those cases where we “lost” the race to stop the timing queue, we expire any events on the queue that have a delay time of zero. This also handles the case of a delayed event request with a zero delay value.

The following function moves any events that have expired onto the TOC queue. These are events that were on the delayed event queue because we lost the race against the timing mechanism or there was a request for a zero delay.

```
<<mrt forward references>>=
static void mrtQueueExpiredDelayedEvents(void) ;
```

```
<<mrt static functions>>=
static void
mrtQueueExpiredDelayedEvents(void)
{
```

```
    MRT_ecb *iter = mrtEventQueueBegin(&delayedEventQueue) ;
    MRT_ecb *const end = mrtEventQueueEnd(&delayedEventQueue) ;
    while (iter != end) {
        if (iter->delay == 0) {
            MRT_ecb *expired = iter ;
            iter = iter->next ;
            mrtEventQueueRemove(expired) ;
            if (expired->reload != 0) {                          // ❶
                MRT_ecb *reloaded = mrtECBalloc() ;
                memcpy(reloaded, expired, sizeof(*reloaded)) ;
                mrtInsertDelayedEvent(reloaded) ;
            }
            mrtEventQueueInsert(expired, mrtEventQueueEnd(&tocEventQueue)) ;
        } else {
            break ;
        }
    }
}
```

❶ Additional work for a periodic event. We have to clone the event placing one copy in the TOC queue and putting the other back into the delayed event queue.

An analogous operation is needed to stop the queue timing. Any remaining time is set back into the first entry on the queue. This puts the queue into a state where ECB's can be inserted, deleted or summed to find the time remaining for an event.

```
<<mrt forward references>>=
static void mrtStopDelayedQueueTiming(void) ;
```

```
<<mrt static functions>>=
static void
mrtStopDelayedQueueTiming(void)
{
    /*
     * Avoid operating on an empty queue.
     */
    if (!mrtEventQueueEmpty(&delayedEventQueue)) {
        /*
         * Stop the timer, obtaining the residual time.
         */
        MRT_DelayTime remain = mrtSysTimerStop() ;
        MRT_ecb *ecb = mrtEventQueueBegin(&delayedEventQueue) ;
        assert(ecb->delay == 0) ;
        ecb->delay = remain ;
    }
}
```

Conceptually, starting and stopping the queue timing moves the time value of the first ECB on the delay queue into and out of the real timing resource (whatever that may be). So when the the delayed event queue timing is running, at least the first ECB on the queue will have a zero delay time. When it is stopped, we must insure that the first queued ECB has a non-zero delay time.

The `mrt_TimerExpireService` function is provided for the asynchronous timer execution to call that performs task of expiring events and finding the next time that needs to be placed in the timer.

```
<<mrt external interfaces>>=
/*
 * Must be invoked from interrupt service level only!
 */
extern MRT_DelayTime
mrt_TimerExpireService(void) ;
```

```
<<mrt external functions>>=
MRT_DelayTime
mrt_TimerExpireService(void)
{
    MRT_DelayTime next_delay = 0 ;
    /*
     * Iterate along the delayed event queue.
     */
    MRT_ecb *iter = mrtEventQueueBegin(&delayedEventQueue) ;
    MRT_ecb *const end = mrtEventQueueEnd(&delayedEventQueue) ;
    while (iter != end) {
        if (iter->delay == 0) {
            MRT_ecb *expired = iter ;
            iter = iter->next ;                                          // ❶
            mrtEventQueueRemove(expired) ;

            if (expired->reload == 0) {
                MRT_DelayedEventParams *params =
                    (MRT_DelayedEventParams *)mrt_SyncRequest(mrtExpireDelayedEvent) ;
                params->ecb = expired ;
            } else {
                MRT_PeriodicEventParams *params =
                    (MRT_PeriodicEventParams *)mrt_SyncRequest(mrtExpirePeriodicEvent) ;
                params->eventNumber = expired->eventNumber ;
                params->alloc = expired->alloc ;
                params->targetInst = expired->targetInst ;
                params->sourceInst = expired->sourceInst ;

                expired->delay = expired->reload ;
                mrtInsertDelayedEvent(expired) ;
                iter = mrtEventQueueBegin(&delayedEventQueue) ;          // ❷
            }
        } else {
            /*
             * Stop at the first non-zero delay time.  This marks the boundary
             * of events that need additional delay time.  The first such event
             * is the next amount of time to delay.
             */
            next_delay = iter->delay ;
            iter->delay = 0 ;
            break ;
        }
    }

    return next_delay ;
}
```

❶ We must advance the iterator before removing the event since the removal will invalidate the iterator.

❷ Once we have reinserted an ECB into the delayed event queue, our iterator is invalid. So, we start at the beginning of the queue looking for additional ECB's which have expired.

1. The function assumes it is called from interrupt service level, *i.e.* it assumes that it may touch the delayed event queue with impunity since it cannot be interrupted by other code that might manipulate the state of those queues.
2. The function returns the next time to expire. The caller is responsible for placing that time into the timer facility if it is non-zero. If the returned time is zero, then the timer facility is not currently needed and should remain stopped.

Notice that the function `mrtExpireDelayedEvent` or `mrtExpirePeriodicEvent` is queued as as sync function to perform the last step of queuing the ECB for dispatch. The run-time uses its own internal facilities to post this function as a sync

function. This insures that delayed event posting is no different than any other interrupt synchronization and preserves the notion that the event queues are not accessed by interrupt service code and therefore do not need to be guarded by a critical section.

```
<<mrt implementation aggregate types>>=
typedef struct mrtdelayedeventparams {
    MRT_ecb *ecb ;
} MRT_DelayedEventParams ;
```

```
<<mrt static functions>>=
static void
mrtExpireDelayedEvent(
    MRT_SyncParams const *params)
{
    MRT_DelayedEventParams *dep = (MRT_DelayedEventParams *)params ;
    mrtEventQueueInsert(dep->ecb, mrtEventQueueEnd(&tocEventQueue)) ;
}
```

```
<<mrt implementation aggregate types>>=
typedef struct mrtperiodiceventparams {
    MRT_AllocStatus alloc ;
    MRT_EventCode eventNumber ;
    MRT_Instance *targetInst ;
    MRT_Instance *sourceInst ;
} MRT_PeriodicEventParams ;
```

```
<<mrt static functions>>=
static void
mrtExpirePeriodicEvent(
    MRT_SyncParams const *params)
{
    MRT_PeriodicEventParams *pep = (MRT_PeriodicEventParams *)params ;
    MRT_ecb *ecb = mrtECBalloc() ;
    ecb->eventNumber = pep->eventNumber ;
    ecb->alloc = pep->alloc ;
    ecb->targetInst = pep->targetInst ;
    ecb->sourceInst = pep->sourceInst ;
    mrtEventQueueInsert(ecb, mrtEventQueueEnd(&tocEventQueue)) ;
}
```

Timing Considerations

With timing being such a common activity in programming, there are very many system specific situations that arise in obtaining timing services on any particular platform. When running on top of an operating system, it will provide the necessary timing services. Unfortunately, the interface to those services varies from OS to OS. On bare metal platforms, generally you will have to get timer peripherals and interrupts involved. We will do what we can here to factor away the essential logic from the platform specific, but note that getting delayed event services running on any particular platform will require some additional work.

We will try to make as few assumptions about the available timing services of the platform as we can. Here are the constraints on the timing services:

- The is only a single source of timing. That timing source allows us to specify some time value to it and it will respond with some notification (*e.g.* an interrupt) when the given time has elapsed.
- It is possible to stop the timing and determine how much time remains before it expires.
- It is not acceptable to execute code periodically that does nothing. This is to account for battery powered devices that cannot afford to wake up and check a timing queue only to find out that there is nothing to do. Activity must be strictly event driven and that implies that we can get positive notification when a time period has elapsed.

Event Dispatch

Finally, we arrive at the point where we can discuss event dispatching. Up until this time, we have been concerned with signaling events, *i.e.* queuing events to be delivered. Now we examine the means by which events are delivered to target instances.

First, we must clarify that despite the fact we have been discussing the *event queue* as if it were a single entity, there are in fact two queues used for dispatching events. The reason for two queues is that we need a way to determine the boundaries of a *thread of control*.

All events that originate outside of a state activity or are the result of a delayed event start a thread of control. The thread of control ends when all the subsequent events that originate from the event starting the thread of control have been dispatched and any activities resulting from the event dispatch has been run. A thread of control is an important concept because its boundaries determine when the data model must be consistent and when we can check the referential integrity.

So our strategy is to use one queue to hold events that start a thread of control and another queue to hold the events that are awaiting dispatch for the ongoing thread of control. When the queue for the ongoing thread of control is empty, then we can start a new thread of control by using events queued to the thread of control queue. When both queues are empty, there is no work to be done and we must wait for subsequent interactions of the system with the outside world.

Dispatching An Event From a Queue

When the run-time needs to dispatch an event it invokes the `mrtDispatchEventFromQueue` function. This function takes as a parameter the queue from which the event is to be dispatched.

```
<<mrt forward references>>=
static bool mrtDispatchEventFromQueue(MRT_EventQueue *queue) ;

<<mrt static functions>>=
static bool
mrtDispatchEventFromQueue(
    MRT_EventQueue *queue)
{
    static MRT_ecb *ecb = NULL ;            // ❶

    if (ecb != NULL) {                      // ❷
        mrtECBfree(ecb) ;
        ecb = NULL ;
    }

    if (!mrtEventQueueEmpty(queue)) {
        ecb = queue->next ;
        mrtEventQueueRemove(ecb) ;
        mrtDispatchEvent(ecb) ;             // ❸
        mrtECBfree(ecb) ;
        ecb = NULL ;

        return true ;
    }

    return false ;
}
```

❶ There is a tricky complication in dispatching an event. It is possible for the event dispatch to cause a fatal error. It is also possible to override the fatal error handler to deal with system specific issues. The overridden fatal error handler can `longjmp` out of the run-time code. Indeed this is the preferred way for testing code. However, if this happens, we could loose the ECB that caused the fatal error. So, while we are dispatching the event, the ECB is stored in a static variable and that static variable is used to indicate the state of completion of the event dispatch.

❷ If the ECB is still around, we presume that we did *not* complete the function after the invocation of `mrtDispatchEvent`. In that case we free our dangling ECB pointer. All event dispatch comes through this function, so a single pointer suffices to save the state of the last dispatch.

❸ Invoking `mrtDispatchEvent` causes transitions in the state maching and can potentially generate a fatal system error.

There are two distinct types of events: transitioning events and polymorphic events. The event types are distinct in that the numerical encoding of the events is used in different ways. In the platform model, all the events for a given state model must have distinct names. The distinction in the platform model between Transitioning Event and Deferred Event is the difference we now see between an ordinary transition event and a polymorphic event. The code generator will assign a unique integer to each event. The set of transition events will be encoded sequentially starting at 0 up to E - 1, where E is the number of transitioning events in the state model. Polymorphic events for the class are then encoded starting at E up to E + P - 1, where P is the number of polymorphic events. Note that only superclasses that have both polymorphic events (either defined or inherited) and a state model of its own will have both transition events and polymorphic events defined for it. However, all events have a unique name in the platform model and a unique numerical encoding in the run-time.

Encoding the event number in this fashion makes it convenient to identify an event for a class when, for example, it is in a delayed event queue. As we will see below, at dispatch time, any polymorphic event numbers will have the number of transition events for the class (E in the discussion above) subtracted from their encoded number so that they can be used as an array index into the data structures used for polymorphic event mapping.

There is a secondary usage of transition events for asynchronous instance creation. A transition event that is dispatched when the current state of the instance is the pseudo-initial creation state is deemed a creation event. We wish to treat creation events separately because of the design strategy we have used to implement asynchronous instance creation. The strategy creates the instance in the creation state, marks its memory slot as allocated but inactive, and queues a transition event. When the event is dispatched, the instance memory slot must be marked active and a transition event dispatch happens. By marking the instance as inactive, but allocated, we can make sure the instance data is not accessed during the time between when it was created and when the creation event is actually dispatched. But, because there is additional processing to be done before a creation event is dispatched as a transition event, we have to be able to identify a transition event as being using in an asynchronous instance creation context.

We provide separate functions to dispatch each event type.

```
<<mrt forward references>>=
static void mrtDispatchTransitionEvent(MRT_ecb *ecb) ;
static void mrtDispatchPolymorphicEvent(MRT_ecb *ecb) ;
static void mrtDispatchCreationEvent(MRT_ecb *ecb) ;
```

Once an event has been chosen for dispatch, the logic to determine the type of the event and, therefore, which dispatch function to invoke is shown below.

```
<<mrt forward references>>=
static void mrtDispatchEvent(MRT_ecb *ecb) ;
```

```
<<mrt static functions>>=
static void
mrtDispatchEvent(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;

    MRT_Instance *targetInst = ecb->targetInst ;
    assert(targetInst != NULL) ;

    MRT_Class const *const classDesc = targetInst->classDesc ;
    assert(classDesc != NULL) ;

    MRT_edb const *const edb = classDesc->edb ;
    if (edb == NULL) {
        mrtDispatchPolymorphicEvent(ecb) ;                    // ❶
    } else {
```

```
        if (ecb->eventNumber < edb->eventCount) {                    // ❷
            if (targetInst->currentState == edb->creationState) {    // ❸
                mrtDispatchCreationEvent(ecb) ;
            } else {
                mrtDispatchTransitionEvent(ecb) ;
            }
        } else {
            assert(classDesc->pdb != NULL) ;
            assert(ecb->eventNumber - edb->eventCount <
                    classDesc->pdb->eventCount) ;
            mrtDispatchPolymorphicEvent(ecb) ;                       // ❹
        }
    }
}
```

❶ If there is no event dispatch block for the class, then all the events of the class must be polymorphic.

❷ If a class has both transitioning events and polymorphic events, the code generator insures that the transitioning events are always numbered starting from 0.

❸ A creation events is just a transitioning event dispatched when we are in the creation state.

❹ If a class has both transitioning events and polymorphic events, the polymorphic events are numbered starting at the end of the sequence for transitioning events.

Event typing is discovered by first determining if there are any transition events. If not, then we must have a polymorphic event (and consequently, its numbering will have started at 0). Otherwise, if the event number falls in the range of transition events, then it must be either an ordinary transition event or a transition event used as a creation event. Finally, for the case where there are both transition and polymorphic events defined for the class, events outside of the transition event range must be dispatched as polymorphic events.

In the sections below, we consider the dispatch of each particular event type.

Transition Event Dispatch

The most frequent event type to dispatch is that of a transitioning event to a state machine. We refer to these event types as, *transition*, only to distinguish them from the more complicated and less frequent used polymorphic and creation event types.

Dispatching an event in its simplest terms involves using the current state of the instance and the event number contained in an ECB as indices into the transition matrix. The transition matrix is the same for all instances of a class. The entry in the transition matrix is the new state to which a transition is to be made. There are a few additional rules needed to account for *ignored* and *can't happen* events.

Ignored events (IG) cause no transition. Events that are ignored can be thought of as an optimization on the state transition graph. Ignored events could be handled by adding a new state to which the ignored event makes a transition and that new state has all the other outbound transitions that the original state had. Clearly, having the concept of ignored events saves much clutter in the state transition graph.

When the analyst considers a transition to be a logical impossibility, then it is declared as a can't happen (CH) event. In the micca run-time, a can't happen transition is treated as a fatal system error. This is a policy decision of the architecture, so **do not** assume that *can't happen* means *shouldn't happen* or *has a low probability of happening*. In this architecture, can't happen means absolutely impossible to happen and if it does happen then there has been a tear in the logic / space / time continuum and the only course available is to give up and declare a fatal error.

The data structure used for transitioning event dispatch is called an **Event Dispatch Block** (EDB). The code generator supplies an EDB for each class (and assigner) that has a state model. Below we will see how all this ties together. For now, we discuss the data structure and how it is used.

```
<<mrt internal aggregate types>>=
typedef struct mrteventdispatchblock {
    MRT_DispatchCount stateCount ;
    MRT_DispatchCount eventCount ;
    MRT_StateCode initialState ;
    MRT_StateCode creationState ;
    MRT_StateCode const *transitionTable ;
    MRT_PtrActivityFunction const *activityTable ;
    bool const *finalStates ;

#       ifndef MRT_NO_NAMES
    char const *const *stateNames ;
#       endif /* MRT_NO_NAMES */
} MRT_edb ;
```

stateCount
The number of states in the transition matrix.

eventCount
The number of events in the transition matrix.

initialState
The number of the state that is the default state when an instance is created synchronously.

creationState
The number of the state that is the default state when an instance is created asynchronously.

transitionTable
A pointer to the transition matrix.

activityTable
A pointer to the state activities.

finalStates
A pointer to a boolean array that determines if a state is a final state.

stateNames
A pointer to an array of character pointers to the names of the class states. This information is used in tracing event dispatch.

The dimensions of the state transition matrix are `stateCount` rows by `eventCount` columns. The counts are held as small integers.

```
<<mrt internal simple types>>=
typedef uint8_t MRT_DispatchCount ;
```

The transition table is in state major order, *i.e.* the current state is used to index conceptual rows and the event number is used to index conceptual columns. The dimensions of the transition table are captured in the EDB to allow run time bounds checking during event dispatch.

The basic transition algorithm is to use the current state of an instance and the event number of an event as the indices into the transition matrix. The entry in the transition matrix is the new state. Notice the very simple data structures required for Moore state machines.

The new state is used as an index into the `activityTable`. The activity table is an array of function pointers to the activity associated with each state.

```
<<mrt internal simple types>>=
typedef void MRT_ActivityFunction(void *const, void const *const) ;
typedef MRT_ActivityFunction *MRT_PtrActivityFunction ;
```

Since Moore state machines associate the activity with the state, that code segment is supplied as a function matching the prototype above. The first argument is a pointer to the instance receiving the event. It is `void` typed and state activities are expected to recover the correct type by casting the pointer to the be of the proper class data structure. The second argument is a pointer to the event parameters. Again, the correct type is recovered in the state activity to match the parameter signature of the activity. Notice that assigning back into event parameters does not make any sense as the parameter values are discarded after the state activity completes.

One other feature of the state machine dispatch rules regards final states. A state may be marked as final and if so, then the run-time will destroy the instance when the state activity is completed. The `finalStates` member points to an array, indexed by state number, that specifies if a particular state is indeed a final state. As is frequently the case, the class may have no final states. In this case, `finalState` member may be set to `NULL` to indicate this fact and save the storage of the final state booleans (*i.e.* there is no need to have an array of `false` values).

```
<<mrt static functions>>=
static void
mrtDispatchTransitionEvent(
    MRT_ecb *ecb)
{
    MRT_Instance *const targetInst = ecb->targetInst ;
    MRT_edb const *const edb = targetInst->classDesc->edb ;
    assert(edb != NULL) ;
    assert(edb->stateCount > targetInst->currentState) ;
    assert(edb->eventCount > ecb->eventNumber) ;
    /*
     * Check for the "event-in-flight" error. This occurs when an instance is
     * deleted while there is an event for that instance in the event queue.
     * For this architecture, such occurrences are considered as run-time
     * detected analysis errors.
     */
    if (targetInst->alloc != ecb->alloc) {
        mrtEventInFlightError(ecb->sourceInst, ecb->eventNumber, targetInst) ;
    }
    /*
     * Fetch the new state from the transition table.
     */
    MRT_StateCode newState = *(edb->transitionTable +
            targetInst->currentState * edb->eventCount + ecb->eventNumber) ;

#       ifndef MRT_NO_TRACE
    /*
     * Trace the transition.
     */
    mrtTraceTransitionEvent(ecb->eventNumber, ecb->sourceInst,
            ecb->targetInst, targetInst->currentState, newState) ;
#       endif /* MRT_NO_TRACE */

    /*
     * Check for a can't happen transition.
     */
    if (newState == MRT_StateCode_CH) {
        mrtCantHappenError(targetInst, targetInst->currentState,
                ecb->eventNumber) ;
    } else if (newState != MRT_StateCode_IG) {
        assert(newState < edb->stateCount) ;
        /*
         * We update the current state to reflect the transition before
         * executing the activity for the state.
         */
        targetInst->currentState = newState ;
        /*
         * Invoke the state activity if there is one.
         */
```

```
        MRT_PtrActivityFunction activity = edb->activityTable[newState] ;
        if (activity) {
            activity(targetInst, &ecb->eventParameters) ;
        }
        /*
         * Check if we have entered a final state. If so, the instance is
         * deleted.
         */
        if (edb->finalStates && edb->finalStates[newState]) {
            mrt_DeleteInstance(targetInst) ; // ❶
        }
    }
}
```

❶ This is where asynchronous instance deletion occurs. If the state is designated as final, then the run-time deletes the instance after its state activity completes.

The processing for dispatching a transition event follows directly from the definitions. After the check for an event-in-flight error, we perform the indexing into the transition matrix. The indexing expression results from the need to treat a linear set of bytes as a two dimensional matrix. We can't type it any differently since we have different sized transition matrices for each different state model. After obtaining the new state, we must determine if we are actually going to make a transition or if the event is to be ignored or considered a fatal error. Assuming that we are transitioning, then the associated state activity is found and executed. Note that empty state activity may be dispensed with and a `NULL` inserted into the action table. After the action, we check if the instance entered a final state.

We are finally in a position to explain the event-in-flight error in detail. Only one analysis error is detected at run-time, the delivery of an event to an instance that has been deleted. Because events are queued, it is possible for an event to be generated for an instance and then while the event is on the queue awaiting to be delivered, the target instance is deleted by some other code executing. For a single threaded architecture, this is considered an analysis error. Deliverying events to deleted instances should never happen! The analytical model is responsible for insuring that instance deletion is accomplished only after there are no events awaiting to be delivered. However, it can happen and the run-time detects and catches this.

A significant difficulty arises in systems that use distinct memory pools for the instances of each class. If an instance is destroyed and another one created, they may very well end up in exactly the same array slot and therefore have exactly the same instance pointer value. So, a pathological case where an event is generated for an instance, the instance is deleted and then re-created while the event is queued could end up delivering the event to the newly recreated instance. Quite the wrong thing to do.

The strategy used here is to vary the number in the `alloc` field of the instance each time it is allocated. Then a copy of the `alloc` field is placed in the ECB when the event is queued. In effect, the real identifier of an instance, for event dispatch purposes, is its pointer plus the value of the `alloc` field. When dispatched, the values of the `alloc` fields in the two structures must match or else the target instance has been destroyed and re-created in the same memory slot. Of course, the observant reader will have seen that in the case where the target instance is destroyed and recreated 16,000 times while the event is queued will result in the event being dispatched to the wrong instance. This is considered such a remote possibility as to be of no practical concern.

Polymorphic Event Dispatch

Polymorphic events in their full generality can be complex, but they are based on a simple idea. In fact, there is nothing going on in the dispatch of polymorphic events that could not otherwise be handled in the state activity. Strictly speaking, polymorphic events must be considered an optimization, but a very convenient and significant one. Previously, we have described the rules concerning polymorphic events.

When a polymorphic event is dispatched, we must traverse the generalization from the superclass to the subclass to determine the type of the subclass. Conceptually, determining which subclass is related to a particular instance of a superclass is not difficult. There are two fundamental steps in dispatching a polymorphic event:

1. Determining which subtype instance is currently related to a supertype instance.

2. Mapping the polymorphic event encoding in the supertype to an event encoding in the subtype.

In order to accomplish the first step, the run-time has know how the generalization relationship is stored in the instances. The run-time supports two different schemes of storing the generalization relationship, either as a pointer reference or as a union member of the superclass structure.

Both relationship storage techniques have their uses and we will not discuss the pros and cons of one choice over another here. If the generalization relationship is stored as a reference, then the superclass instances will contain an instance pointer to a subtype. If the generalization relationship is stored as a union, then the superclass structure will a member that is a union of the data types of all the subtypes of the generalization hierarchy.

As we will see, if we can locate where in the superclass instance structure the subclass encoding and the subclass reference or union are located, then we can determine the type of the subclass instance to which a particular superclass instance is related. To do that we assume that the there is a data type that can hold the byte offset from the beginning of the superclass instance structure to the required information. As you can probably imagine, this will be a tricky piece of code since it must pick out information from an arbitrary data structure in a generic fashion.

```
<<mrt internal simple types>>=
typedef size_t MRT_AttrOffset ;
```

Polymorphic Event Mapping

We now turn our attention to the actual mapping of polymorphic events. The mapping is analogous to the mapping of current state and event to a new state for normal event dispatch. For polymorphic events, the mapping is from subclass code of the currently related subclass and polymorphic event number to a new event. The data structure required for this is given by:

```
<<mrt internal aggregate types>>=
typedef struct mrtgendispatchblock {
    struct mrtrelationship const *relship ;
    MRT_EventCode const *eventMap ;
} MRT_gdb ;
```

relship
A pointer to the relationship description for the relationship across which the polymorphic event will be dispatched.

eventMap
The `eventMap` member is a pointer to the mapping of polymorphic events for the generalization. This mapping is indexed in major order by subclass code and in minor order by polymorphic event number.

A key realization here is that as we are mapping along a generalization relationship, a given polymorphic event may be mapped into a normal event where it will be consumed by the state machine of the class or it may be delegated further down the hierarchy. To be delegated further implies that a polymorphic event will be mapped into yet another polymorphic event to be further mapped in a subsequent dispatch.

Now we can tie it all together. For a superclass class that has associated polymorphic events the code generator supplies a Polymorphic Dispatch Block (PDB) to direct the run-time as to how to perform the mapping of polymorphic events to transitioning events.

```
<<mrt internal aggregate types>>=
typedef struct mrtpolydispatchblock {
    MRT_DispatchCount eventCount ;
    MRT_DispatchCount genCount ;
    struct mrtgendispatchblock const *genDispatch ;

#       ifndef MRT_NO_NAMES
    char const *const *genNames ;
#       endif /* MRT_NO_NAMES */
} MRT_pdb ;
```

eventCount
The `eventCount` member holds the number of polymorphic events associated with the superclass. Like transition events, polymorphic events are encoded as zero based sequential integers so they may be used as array indices in the mapping process.

genCount
The `genCount` member holds the number of generalization that originate at the superclass class.

genDispatch
The `genDispatch` member holds a pointer to an array of Generalization Dispatch Blocks. The array contains `genCount` elements.

genNames
A pointer to an array of generalization relationship names. The array is of length, `genCount`.

Now we can give the code for polymorphic event dispatch.

```
<<mrt static functions>>=
static void
mrtDispatchPolymorphicEvent(
    MRT_ecb *ecb)
{
    MRT_Instance *superInst = ecb->targetInst ;
    MRT_Class const *const superClassDesc = superInst->classDesc ;
    MRT_pdb const *const pdb = superClassDesc->pdb ;
    assert(pdb != NULL) ;
    assert(pdb->genCount > 0) ;
    /*
     * Check for the "event-in-flight" error. We must make sure the
     * superclass instance still exists.
     */
    if (superInst->alloc != ecb->alloc) {
        mrtEventInFlightError(ecb->sourceInst, ecb->eventNumber, superInst) ;
    }
    /*
     * Compute the base offset for polymorphic event numbering.  This base
     * offset will be used to turn the polymorphic event number into an array
     * index.
     */
    MRT_edb const *const edb = superClassDesc->edb ;
    MRT_EventCode eventOffset = edb == NULL ? 0 : edb->eventCount ;
    assert(ecb->eventNumber >= eventOffset) ;
    assert(ecb->eventNumber - eventOffset < pdb->eventCount) ;
    /*
     * Save the original event number.  We intend to reuse the same ECB for each
     * event we dispatch and will need this and the super class instance pointer
     * values should there be more than one generalization associated with this
     * superclass.
     */
    MRT_EventCode origEvent = ecb->eventNumber ;
    /*
     * For each generalization that originates at the superclass an event is
     * generated down that generalization to one of the subclasss.
     */
    MRT_gdb const *gdb = pdb->genDispatch ;
    for (unsigned gnum = 0 ; gnum < pdb->genCount ; gdb++, gnum++) {
        MRT_Relationship const *const rel = gdb->relship ;
        MRT_Instance *subInst ;
        int subclassCode ;

        /*
         * Find the target instance reference and the class of the target
```

```
         * instance. How we do this depends upon how the generalization is
         * stored in the superclass instance.
         */
        if (rel->relType == mrtRefGeneralization) {
            /*
             * When the generalization is implemented via a pointer, we need an
             * extra level of indirection to fetch the address of the subclass.
             */
            MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;
            subInst = *(MRT_Instance **)
                    ((uintptr_t)superInst + gen->superclass.storageOffset) ;
            assert(subInst != NULL) ;
            /*
             * We must also guard against the possibility that the subclass was
             * unrelated from the superclass before the polymorphic event was
             * dispatched.
             */
            if (subInst == NULL) {
                mrtFatalError(mrtRelationshipLinkage) ;
            }
            assert(subInst->classDesc != NULL) ;
            subclassCode = mrtFindRefGenSubclassCode(subInst->classDesc,
                    gen->subclasses, gen->subclassCount) ;
        } else if (rel->relType == mrtUnionGeneralization) {
            /*
             * When the generalization is implemented by a union, we need only
             * point to the address of the subclass since it is contained
             * within the superclass.
             */
            MRT_UnionGeneralization const *gen =
                    &rel->relInfo.unionGeneralization ;
            subInst = (MRT_Instance *)
                    ((uintptr_t)superInst + gen->superclass.storageOffset) ;
            assert(subInst->classDesc != NULL) ;
            subclassCode = mrtFindUnionGenSubclassCode(subInst->classDesc,
                    gen->subclasses, gen->subclassCount) ;
        } else {
            mrtFatalError(mrtRelationshipLinkage) ;
        }
        /*
         * Check that our subclass instance is indeed allocated and usable.  We
         * are trying to guard against the possiblity that the subclass
         * instance was deleted before the polymorphic event was delivered.
         */
        assert(subInst->alloc > 0) ;
        if (subInst->alloc <= 0) {
            mrtEventInFlightError(ecb->sourceInst, origEvent, subInst) ;
        }
        /*
         * Update the target and allocation status in the ECB to match
         * that of the subclass instance, which is where the event
         * is now directed.
         */
        ecb->targetInst = subInst ;
        ecb->alloc = subInst->alloc ;
        /*
         * Fetch the event number for the subclass from the polymorphic
         * mapping.  The class of the subclass related to the superclass
         * determines the mapped value for the event.  Note we must subtract
         * off any offset in the event encoding that was consumed by the
         * transition events.
         */
```

```
        ecb->eventNumber = *(gdb->eventMap + subclassCode * pdb->eventCount +
                origEvent - eventOffset) ;

#           ifndef MRT_NO_TRACE
        /*
         * Trace the transition.
         */
        mrtTracePolymorphicEvent(origEvent, ecb->sourceInst, superInst,
                subclassCode, gnum, ecb->eventNumber) ;
#           endif /* MRT_NO_TRACE */

        mrtDispatchEvent(ecb) ; // ❶
    }
}
```

❶ Since a polymorphic event may map to either another polymorphic event or to a transition event, we will use the general dispatch function to recursively dispatch the mapped event.

The code loops through all of the generalizations for which the `superInst` is a superclass. The vast majority of the time there is only one generalization. The strategy used here is to reuse the ECB that was carrying the polymorphic event as the ECB for the remapped events. This saves allocating a new ECB and avoids any problem that there may not be an ECB available at that time. The mapping of a polymorphic event is a function of the superclass target and the polymorphic event number. So we hold them in local variables as we overwrite the necessary fields in the ECB to hold the mapped event information.

The core of the algorithm is to determine the subclass code of the related subclass instance and use that as the row index into the polymorphic event map for the generalization. The event number (appropriately offset by the number of transition events) is then used as the column index to find the mapping entry. That mapping entry contains a new event number. The new target of the event is the currently related subclass instance. As we have discussed, this may be a stored as a pointer or may be a union member of the superclass instance structure. For the pointer case, we fetch the pointer from its location in the superclass structure. For the union case, the location in the superclass structure is the beginning of the union, *i.e.* we down cast to the subclass member. We fill in the `alloc` field to enable the event in flight detection just in case the mapped event causes a transition. Finally the newly minted ECB is recursively dispatched and the next generalization is considered. Recursively dispatching the event preserves the order of delivery of the events. Note that when there are multiple generalization hierarchies for the event, the dispatch order is determined by the order the code generator decides for generalization dispatch blocks.

Creation Event Dispatch

Fortunately, creation events are much simpler than polymorphic events. Creation event dispatch fixes up the `alloc` field of the target and the ECB before normal event dispatch. No additional data structures are required.

```
<<mrt static functions>>=
static void
mrtDispatchCreationEvent(
    MRT_ecb *ecb)
{
#       ifndef MRT_NO_TRACE
    /*
     * Trace the transition.
     */
    mrtTraceCreationEvent(ecb->eventNumber, ecb->sourceInst, ecb->targetInst,
            ecb->targetInst->classDesc) ;
#       endif /* MRT_NO_TRACE */

    assert(ecb->alloc == ecb->targetInst->alloc) ;
    assert(ecb->alloc < 0) ;
    assert(ecb->targetInst->alloc < 0) ;
    assert(ecb->targetInst->currentState ==
        ecb->targetInst->classDesc->edb->creationState) ;
```

```
    ecb->alloc = ecb->targetInst->alloc = -ecb->targetInst->alloc ; // ❶
    mrtDispatchTransitionEvent(ecb) ;
}
```

❶ By negating the `alloc` value, we show that the instance is active.

Chapter 27

Bridging Domains

All but the simplest of systems will contain more that one domain. A domain represents a coherent subject matter with its own set of rules and policies. Domains are also the unit of encapsulation and reuse.

Consider a simple example of a domain that controls a chemical reaction vessel. One aspect of synthesizing something in a reaction vessel is controlling the temperature of the vessel. Such systems typically delegate control to another domain so that a Reaction Management domain would delegate setting and maintaining the reaction vessel temperature to a Signal I/O domain.

From the point of view of Reaction Management, the reaction vessel has heaters, coolers, pumps, valves and such things and it is its responsibility to sequence the vessel operations to accomplish the synthesis. Reaction Management does not know the details of what it is controlling. It wants to set the vessel temperature to 57C and maintain it there for some time period. How that happens in terms of heaters, thermal load and other such physical considerations is delegated.

From the point of view of Signal I/O, it knows that it is controlling output actuators with input sensors as feed back. That output point number 57 corresponds to a reactor vessel is not in its scope of concern.

The concept of a domain is very much related to the concept of *separation of concerns*. But that which is separated must be combined if we are to obtain a useful system. The difference in the semantic view points of domains must be bridged by translating the assumptions and dependencies of one domain into the services provided by other domains.

There are several approaches to bridging. The most elegant approach is *implicit bridgeing* which is related to *aspect oriented programming* concepts. In this style of bridging, we would use a separate means to describe how operations in one domain would be mapped to side effects in another domain. For example, we would want to be able to state that when the state activity in the Reaction Management domain updates the reactor vessel set point attribute we would want the temperature to be transferred to output point number 57 of the Signal I/O domain. The advantage of this type of bridging is that the individual domains are not modified and the domain semantics are not disturbed. The bridge interactions are defined outside of the domain itself. It is then a code generator's task to generate the domain code so that the bridge mapping is implemented. The disadvantage of this type of bridging is that it is very difficult to implement both the means to specify the domain interactions and the code generator required to generate the bridging code.

Another approach to bridging is termed, *explicit bridging*. In this form, a domain makes explicit invocations to an *external entity*. The invocations demonstrate the explicit requirements that the domain delegates and the service it requires. Bridges then map the external entity invocations to interactions with other domains that provide the required service. Explicit bridging is a workable technique but can lead to a large number of domain operations being needed to service a client domain's dependencies. Often, a client domain and service domain will have *counterpart classes*. These are classes that represent different aspects of the same entity but do so in the semantics of the individual domains. In such cases, model level operations in the client domain, *e.g.* updating an attribute value, are bridged to a similar model operations in the service domain, *e.g* updating the counterpart class attribute with a scaled value. If it is necessary to code a domain operation for each such model level operation, the service domain will be unnecessarily complicated and its reuse will be limited since the details of how model level actions need to take place will very much depend upon the context in which the service domain is used.

Making available the ability to do some model level operations such as updating attributes or signaling events from outside of the domain can make explicit bridging much easier and prevent cluttering the external interfaces of service domains with domain operations specifically built to support the use of the service domain in one particular system. Doing so breaks the encapsulation of the domain, albeit in a very controlled manner.

`Micca` takes the following view of bridging:

- Domain designs should provide domain operations for those computations that cannot be accomplished by a single simple model level operation. Operations such as navigating a relationship or other more complicated activities of instance creation and relationship linkage require processing code to accomplish and that code is best gathered in a domain operation.
- The `micca` code generator provides the means to break encapsulation of a domain for simple model level actions such as signaling an event or reading a class instance attribute.
- A domain must be populated before it can be bridged. If there is an initial instance population, bridges usually must account for that. If the population of instances for a class is static, this will usually greatly simplify the bridge.
- Bridge operation code is manually coded to map the semantics of one domain onto another and, in general, is specific to the use of the bridged domains in a particular system and, therfore, *not* reusable.
- Dynamic activity in one domain may have to be reflected in the bridge. For example, instance creation in one domain may have to result in creating an instance of a counterpart class in a service domain. In such cases, the bridge itself may have to track the dynamics of the two domains.

These consideration make domain bridging abstract and complicated. Sadly, the methodological fundamentals of bridging seem to be lacking and much of what is done to build systems from bridged domains smacks of being *ad hoc*. There is much more that could be said about bridging that space does not allow for here. Projects are cautioned that bridging domains can be a significant activity for which plans need to be made.

In this section, we discuss the facilities provided by `micca` to perform operations on a domain from outside the domain itself. These facilities are intended to be used by bridge operation code. Conceptually, `micca` builds a *portal* into the domain and supplies a set of functions that will reach through the portal and perform operations in the domain. The set of operations that can be performed is very limited but is a useful set for implementing bridge operations.

Portal Data Structures

The only externally scoped identifiers in the generated “C” code file are those of the domain operations. All other generated code contains identifiers that are file static in scope. This is done to prevent collisions and contention of names at link time. As a result, other translation units have no access to the internals of a domain as was the design goal.

Since we intend to provide some operations on the internals of a domain we need a way to identify the entities upon which we are operating. Internally, a `micca` generated domain uses pointers to refer to instances, class descriptors and other essential data. We do *not* want to expose addresses of the internals of a domain to the outside. Since all class instances are really elements of the class storage array, the index of an instance in the storage array serves as a convenient external identifier. The `micca` code generator will place a set of “C” preprocessor definitions in the generated header file to numerically encode the classes, instances and attributes. It is these small integer numbers that will serve as identifiers outside of the domain and which we will use as array indices (with appropriate bounds checking) inside the domain.

```
<<mrt interface simple types>>=
typedef unsigned short MRT_ClassId ;
typedef unsigned short MRT_InstId ;
typedef unsigned short MRT_AttrId ;
typedef size_t MRT_AttrSize ;
typedef unsigned short MRT_AssignerId ;
```

To operate on the internals of a domain requires a data structure that maps the numerically encoded identifiers to the internals of domain.

```
<<mrt interface aggregate types>>=
struct mrtdomainportal ;
typedef struct mrtdomainportal MRT_DomainPortal ;

<<mrt internal aggregate types>>=
struct mrtdomainportal {
```

```
    unsigned classCount ;
    MRT_Class const *classes ;
    unsigned assignerCount ;
    MRT_Class const *assigners ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} ;
```

classCount
The number of classes in the domain.

classes
A pointer to an array of class descriptions for the domain. The array contains `classCount` elements.

assignerCount
The number of assigners in the domain.

assigners
A pointer to an array of class description for the assigners. The array contains `assignerCount` elements.

name
A pointer to a NUL terminated character array giving the name of the domain.

The domain portal is a collection of class descriptions for the classes and assigners in the domain. Note that relationships are not accessible via the portal.

For each domain, the code generator will create an externally scoped variable of the above type. The name of the variable follows the form `<domain name>__PORTAL`, *i.e.* the suffix `__PORTAL` is appended to the domain name. This variable forms the *portal* into the domain and below we describe the operations on the domain that are available via the portal.

Portal Access Functions

In this section we describe the set of operations that are available through the portal.

Portal Errors

All the portal access functions return an integer value. Non-negative return values indicate success and return the requested information. Negative return values indicate errors as described below.

```
<<mrt interface constants>>=
    // No such class.
#define MICCA_PORTAL_NO_CLASS       (-1)
    // No such instance.
#define MICCA_PORTAL_NO_INST        (-2)
    // No such attribute.
#define MICCA_PORTAL_NO_ATTR        (-3)
    // Instance slot is not in use.
#define MICCA_PORTAL_UNALLOC        (-4)
    // Class does not have a state model.
#define MICCA_PORTAL_NO_STATE_MODEL (-5)
    // No such event for the class.
#define MICCA_PORTAL_NO_EVENT       (-6)
    // No such state for the class.
#define MICCA_PORTAL_NO_STATE       (-7)
```

```
    // Class does not support dynamic instances.
#define MICCA_PORTAL_NO_DYNAMIC     (-8)
    // Operation not allowed on a dependent attribute.
#define MICCA_PORTAL_DEPENDENT_ATTR (-9)
    // Operation failed from insufficient space to transfer value.
#define MICCA_PORTAL_SIZE_ERROR     (-10)
```

It is useful to have a function to translate the error code number into a canonical string for the error.

```
<<mrt external interfaces>>=
extern char const *
mrt_PortalErrorString(
    int portalErrorCode) ;
```

portalErrorCode
A negative or zero value as returned from a portal function.

The `mrt_PortalErrorString` function translates a portal error code to a human readable string. By special dispensation, a value of zero for `portalErrorCode` is accepted and a string reference is returned for it. The value of `NULL` is returned for unknown error code values or if the run time was compiled with the pre-processor macro, `MRT_NO_NAMES`, defined.

```
<<mrt external functions>>=
char const *
mrt_PortalErrorString(
    int portalErrorCode)
{
#       ifndef MRT_NO_NAMES

    static char const * const portalErrStrings[] = {
        "No error",                                     // ❶
        "No such class",
        "No such instance",
        "No such attribute",
        "Instance slot is not in use",
        "Class does not have a state model",
        "No such event for the class",
        "No such state for the class",
        "Class does not support dynamic instances",
        "Operation not allowed on a dependent attribute",
        "Operation failed from insufficient space to transfer value",
    } ;

    assert(portalErrorCode >= MICCA_PORTAL_SIZE_ERROR && portalErrorCode <= 0) ;

    if (portalErrorCode < MICCA_PORTAL_SIZE_ERROR || portalErrorCode > 0) {
        return NULL ;
    }

    return portalErrStrings[-portalErrorCode] ;         // ❷

#       else

    return NULL ;

#       endif /* MRT_NO_NAMES */
}
```

❶ We'll allow zero as an error code since it is used to indicate success by several of the portal functions. So we need a place for the success message in the string array.

❷ Negating the error code turns it into an index. We have already validated the argument value so this is safe to do.

References to Attributes

Internally, the portal access functions often need to find a class instance. We have factored that into a static function.

```
<<mrt static functions>>=
static int
mrtPortalGetInstRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_Instance **ref)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const pclass = portal->classes + classId ;
    if (instId >= pclass->instCount) {
        return MICCA_PORTAL_NO_INST ;
    }

    MRT_Instance *instance = mrtIndexToInstance(pclass, instId) ;
    if (instance == NULL) {
        return MICCA_PORTAL_UNALLOC ;
    }

    if (instance->classDesc != pclass) {
        return MICCA_PORTAL_NO_INST ;                          // ❶
    }

    if (ref) {
        *ref = instance ;
    }
    return 0 ;
}
```

❶ Normally, the class descriptor pointer in the instance matches that obtained from the portal classes array because we used the class descriptor to convert from an index to an instance reference. However, for union subclasses, it is possible to give an instance id that maps to a storage slot in the superclass storage which currently contains an instance of a different subtype class. This test checks for that circumstance.

There is one case where we expose some internal pointer information and this is for attributes. Attributes implemented as arrays don't pass well by value in "C". Obtaining a reference to the attribute is often the best way to deal with passing its value around.

```
<<mrt external interfaces>>=
extern int
mrt_PortalGetAttrRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void **pref,
    MRT_AttrSize *size) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
The number of the attribute to be read. This number is generated by micca and placed in the domain header file.

pref
A pointer to a memory pointer where the attribute reference is placed. If this argument is `NULL` then no reference is returned.

size
A pointer to where the size of the attribute is placed. If this argument is `NULL` then no size information is returned.

`mrt_GetAttrRef` obtains a pointer to the storage location for an attribute and the size of that storage location. The return value is 0, if the attribute storage could be found. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalGetAttrRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void **pref,
    MRT_AttrSize *size)
{
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    MRT_Attribute const *attr = class->classAttrs + attrId ;
    if (attr->type != mrtIndependentAttr) {
        return MICCA_PORTAL_DEPENDENT_ATTR ;
    }
```

```
    if (pref) {
        *pref = (void *)((uintptr_t)instref + attr->access.offset) ;
    }
    if (size) {
        *size = attr->size ;
    }
    return 0 ;
}
```

Reading Attributes

Instance attribute values may be read through the portal.

```
<<mrt external interfaces>>=
extern int
mrt_PortalReadAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void *dst,
    MRT_AttrSize dstSize) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
The number of the attribute to be read. This number is generated by micca and placed in the domain header file.

dst
A pointer to memory where the attribute value is placed.

dstSize
The number of bytes pointed to by `dst`.

`mrt_PortalReadAttr` reads an attribute value from a domain. No more than `dstSize` bytes will be placed in the memory pointed to by `dst`. If the return value is non-negative, then it represents the actual number of bytes read. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalReadAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void *dst,
    MRT_AttrSize dstSize)
{
    assert(dst != NULL) ;
```

```
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    MRT_Attribute const *attr = class->classAttrs + attrId ;
    MRT_AttrSize srcSize = attr->size ;
    if (srcSize > dstSize) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    if (attr->type == mrtIndependentAttr) {
        void *src = (void *)((uintptr_t)instref + attr->access.offset) ;
        memcpy(dst, src, srcSize) ;
    } else {
        attr->access.formula(instref, dst, srcSize) ;
    }
    return srcSize ;
}
```

Updating Attributes

Instance attribute values may be updated through the portal.

```
<<mrt external interfaces>>=
extern int
mrt_PortalUpdateAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void const *src,
    MRT_AttrSize srcSize) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
The number of the attribute to be updated. This number is generated by micca and placed in the domain header file.

src
A pointer to memory from where the attribute value is taken.

srcSize
The number of bytes pointed to by `src`.

`mrt_PortalUpdateAttr` updates an attribute value in a domain. If the return value is non-negative, then it represents the actual number of bytes copied into the attribute storage location. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalUpdateAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void const *src,
    MRT_AttrSize srcSize)
{
    assert(src != NULL) ;

    void *dst ;
    MRT_AttrSize dstSize ;

    int result = mrt_PortalGetAttrRef(portal, classId, instId, attrId, &dst,
            &dstSize) ;
    if (result != 0) {
        return result ;
    }

    if (srcSize > dstSize) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    assert(dst != NULL) ;
    memcpy(dst, src, srcSize) ;
```

```
    return srcSize ;
}
```

Signaling Events

The portal allows signalling an event to a class instance.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to signal.

eventParameters
A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

paramSize
The number of bytes pointed to by `eventParameters`.

`mrt_PortalSignalEvent` signals an ordinary or polymorphic event to the given instance. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    int result ;
    MRT_ecb *ecb ;

    result = mrtPortalNewECB(portal, classId, instId, eventNumber,
            eventParameters, paramSize, &ecb) ;
    if (result == 0) {
        mrt_PostEvent(ecb) ;
    }
```

```
    return result ;
}
```

Internally, we factor obtaining an ECB into its own function.

```
<<mrt static functions>>=
static int
mrtPortalNewECB(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_ecb **ecbRef)
{
    assert(ecbRef != NULL) ;

    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const classDesc = instref->classDesc ;
    assert(classDesc != NULL) ;
    if (classDesc->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= classDesc->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    if (paramSize > sizeof(MRT_EventParams)) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    MRT_ecb *ecb = mrt_NewEvent(eventNumber, instref, NULL) ;
    if (eventParameters != NULL) {
        memcpy(ecb->eventParameters, eventParameters, paramSize) ;
    }
    *ecbRef = ecb ;
    return 0 ;
}
```

Signaling Delayed Events

Signaling delayed events is similar to immediate events, but adds a delay time argument.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_DelayTime delay) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to signal.

eventParameters
A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

paramSize
The number of bytes pointed to by `eventParameters`.

delay
The minimum number of milliseconds to delay before the event is dispatched.

`mrt_PortalSignalDelayedEvent` signals an ordinary or polymorphic event to the deliverd to an instance after a delay time. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_DelayTime delay)
{
    MRT_ecb *ecb ;
    int result = mrtPortalNewECB(portal, classId, instId, eventNumber,
            eventParameters, paramSize, &ecb) ;
    if (result == 0) {
        mrt_PostDelayedEvent(ecb, delay) ;
    }

    return result ;
}
```

Canceling Delayed Events

A delayed event may be cancelled.

```
<<mrt external interfaces>>=
extern int
mrt_PortalCancelDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to signal.

`mrt_PortalCancelDelayedEvent` cancels a delayed event. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalCancelDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber)
{
    MRT_Instance *instref ;

    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    mrt_CancelDelayedEvent(eventNumber, instref, NULL) ;
    return 0 ;
}
```

Remaining Delay Time

You may also request the time remaining for a delayed event.

```
<<mrt external interfaces>>=
extern int
mrt_PortalRemainingDelayTime(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    MRT_DelayTime *delayRef) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to query.

delayRef
A pointer to a location where the remaining delay time is placed.

The `mrt_PortalCancelDelayedEvent` function queries the amount of time remaining before the event with `eventNumber` is dispatched to the `instId` instance of class, `classId`. The value is returned indirectly via the `delayRef` pointer. The returned delay value is zero if the delay time has already elapsed or if no matching delayed event could be found. The return value of the function is zero if successful and the value pointed to by `delayRef` is valid.

```
<<mrt external functions>>=
int
mrt_PortalRemainingDelayTime(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    MRT_DelayTime *delayRef)
{
    assert(delayRef != NULL) ;

    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }
```

```
    MRT_DelayTime delay = mrt_RemainingDelayTime(eventNumber, instref, NULL) ;
    if (delayRef) {
        *delayRef = delay ;
    }
    return 0 ;

}
```

Synchronous Instance Creation

```
<<mrt external interfaces>>=
extern int
mrt_PortalCreateInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode initialState) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

initialState
The number of the state into which the new created instance is placed. If the class given by `classId` has no state model, then `initialState` must be given as `MRT_StateCode_IG`. If the class given by `classId` has as state model, then if `initialState` is given as `MRT_StateCode_IG`, the instance will be placed in the default initial state as it was defined when the state model was configured. Otherwise, `initialState` must be a valid state number for the class and the new instance is created in that state. N.B. no state activity is executed in any case.

```
<<mrt external functions>>=
int
mrt_PortalCreateInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode initialState)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->containment != NULL) {
        return MICCA_PORTAL_NO_DYNAMIC ;
    }

    if (initialState != MRT_StateCode_IG) {
        if (class->edb == NULL) {
            return MICCA_PORTAL_NO_STATE_MODEL ;
        }
        if (initialState <= MRT_StateCode_CH ||
                initialState >= class->edb->stateCount ||
                initialState == class->edb->creationState) {
            return MICCA_PORTAL_NO_STATE ;
```

```
        }
    }
    void *inst = mrt_CreateInstance(class, initialState) ;
    return mrt_InstanceIndex(inst) ;
}
```

Asynchronous Instance Creation

```
<<mrt external interfaces>>=
extern int
mrt_PortalCreateInstanceAsync(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

eventNumber
The number of the creation event.

eventParameters
A pointer to the parameters for the event. If the event has no supplemental parameters, this argument should be NULL.

paramSize
The number of bytes pointed to by `eventParameters`.

The `mrt_PortalCreateInstanceAsync` function sends `eventNumber` as a creation event to the class given by `classId`. Any parameters of the event are pointed to by `eventParameters` and are `paramSize` bytes in length. The newly created instance is placed in its pseudo-initial state and the `eventNumber` event is signaled to it. N.B. a transition occurs when the event is delivered resulting in the execution of a state activity.

```
<<mrt external functions>>=
int
mrt_PortalCreateInstanceAsync(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->containment != NULL) {                    // ❶
        return MICCA_PORTAL_NO_DYNAMIC ;
    }
```

```
    if (class->edb == NULL) {                          // ❷
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->edb->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    MRT_Instance *inst = mrt_CreateInstanceAsync(class,
            eventNumber, eventParameters, paramSize, NULL) ;
    return mrt_InstanceIndex(inst) ;
}
```

❶ Union subclass instances cannot be created asynchronously.

❷ Asynchronous creation requires an event dispatch block.

Deleting Instances

```
<<mrt external interfaces>>=
extern int
mrt_PortalDeleteInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

The `mrt_PortalDeleteInstance` deletes the `instId` instance of the `classId` class.

```
<<mrt external functions>>=
int
mrt_PortalDeleteInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId)
{
    MRT_Instance *inst = NULL ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &inst) ;
    if (result == 0) {
        assert(inst != NULL) ;
        mrt_DeleteInstance(inst) ;
    }

    return result ;
}
```

Signaling Events To Assigners

The portal allows signalling an event to an assigner instance.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalEventToAssigner(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

portal
A pointer to the portal structure for the domain.

assignerId
The number of the assigner. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class. Single assigners have only one instance and so `instId` must be 0 for them. Multiple assigners may have an `instId` value greater than 0.

eventNumber
The event number of the event to signal.

eventParameters
A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

paramSize
The number of bytes pointed to by `eventParameters`.

`mrt_PortalSignalEventAssigner` signals a transitioning event to the given assigner instance. Polymorphic and creation events cannot be signaled to an assigner. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalEventToAssigner(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    MRT_Instance *instref = NULL ;
    int result = mrtPortalGetAssignerRef(portal, assignerId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    MRT_Class const *const asnClass = instref->classDesc ;
    MRT_edb const *edb = asnClass->edb ;
    assert(edb != NULL) ;
    if (eventNumber >= edb->eventCount) {
```

```
        return MICCA_PORTAL_NO_EVENT ;
    }

    if (paramSize > sizeof(MRT_EventParams)) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    MRT_ecb *ecb = mrt_NewEvent(eventNumber, instref, NULL) ;
    if (eventParameters != NULL) {
        memcpy(ecb->eventParameters, eventParameters, paramSize) ;
    }
    mrt_PostEvent(ecb) ;
    return 0 ;
}
```

Similar to obtaining an instance reference, we factor out code to obtain a reference to an assigner.

```
<<mrt static functions>>=
static int
mrtPortalGetAssignerRef(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_Instance **ref)
{
    assert(portal != NULL) ;

    if (assignerId >= portal->assignerCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const asnClass = portal->assigners + assignerId ;
    if (instId >= asnClass->instCount) {
        return MICCA_PORTAL_NO_INST ;
    }

    MRT_iab *iab = asnClass->iab ;
    assert(iab != NULL) ;
    MRT_Instance *instref = (MRT_Instance *)
            ((uintptr_t)iab->storageStart + iab->instanceSize * instId) ;
    if (instref->alloc <= 0) {
        return MICCA_PORTAL_UNALLOC ;
    }

    if (ref) {
        *ref = instref ;
    }
    return 0 ;
}
```

Obtaining Class Current State

The portal supports reading the current state of an instance of a class having a state model.

```
<<mrt external interfaces>>=
extern int
mrt_PortalInstanceCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

The `mrt_PortalInstanceCurrentState` function returns the number of the current state of the `instId` instance of `classId` class.

```
<<mrt external functions>>=
int
mrt_PortalInstanceCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId)
{
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (class->edb == NULL) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    return instref->currentState ;
}
```

Obtaining Assigner Current State

The portal supports reading the current state of an assigner of a class having a state model.

```
<<mrt external interfaces>>=
extern int
mrt_PortalAssignerCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

assignerId
The number of the assigner. This number is generated by micca and placed in the domain header file.

instId
The number of the assigner instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the assigner.

The `mrt_PortalAssignerCurrentState` function returns the number of the current state of the `instId` instance of `assignerId` assigner.

```
<<mrt external functions>>=
int
mrt_PortalAssignerCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId)
{
    MRT_Instance *instref = NULL ;
    int result = mrtPortalGetAssignerRef(portal, assignerId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    return instref->currentState ;
}
```

Domain Introspection

For testing and other purposes, it is useful to be able to inquire about the characteristics of a domain at run-time. The functions in this section expose other domain portal meta-information.

```
<<mrt external interfaces>>=
extern char const *
mrt_PortalDomainName(
    MRT_DomainPortal const *const portal) ;
```

portal
A pointer to a domain portal data structure.

The `mrt_PortalDomainName` function returns a pointer to a `NUL` terminated string giving the name of the domain described by the portal data pointed to by `portal`. If the domain was compiled with the preprocessor symbol `MRT_NO_NAMES` defined, then the return value is `NULL`.

```
<<mrt external functions>>=
char const *
mrt_PortalDomainName(
    MRT_DomainPortal const *const portal)
{
    assert(portal != NULL) ;

#       ifndef MRT_NO_NAMES
    return portal->name ;
#       else
    return NULL ;
#       endif /* MRT_NO_NAMES */
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalDomainClassCount(
    MRT_DomainPortal const *const portal) ;
```

portal
A pointer to a domain portal data structure.

The function `mrt_PortalDomainClassCount` returns the number of classes in the domain. Classes may be identified to the other portal functions using integer numbers ranging from 0 to the return value of `mrt_PortalDomainClassCount` minus 1.

```
<<mrt external functions>>=
int
mrt_PortalDomainClassCount(
    MRT_DomainPortal const *const portal)
{
    assert(portal != NULL) ;

    return portal != NULL ? portal->classCount : 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the class name is placed.

The `mrt_PortalClassName` function obtains the name of the *classId* class in the domain described by *portal*. The class name is returned via `nameRef` as a pointer to a `NUL` terminated string. If the domain was compiled with the preprocessor symbol `MRT_NO_NAMES` defined, then the value placed in the location pointed to by *nameRef* is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

#       ifndef MRT_NO_NAMES
    MRT_Class const *const class = portal->classes + classId ;
    if (nameRef) {
        *nameRef = class->name ;
    }
#       else
    if (nameRef) {
        *nameRef = NULL ;
    }
#       endif /* MRT_NO_NAMES */

    return 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassAttributeCount` function returns the number of attributes in the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is non-negative and represents the number of attributes of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    return class->attrCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassInstanceCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassInstanceCount` function returns the maximum number of instances that may be created for the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of instance storage slots for the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassInstanceCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;
```

```
    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    return class->instCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassEventCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassEventCount` function returns the number of events in the state model of the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of events defined for the state model of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassEventCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    return class->eventCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassStateCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassStateCount` function returns the number of states in the state model of the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of states defined for the state model of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassStateCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    MRT_edb const *edb = class->edb ;
    assert(edb != NULL) ;

    return edb != NULL ? edb->stateCount : 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

attrId
The number of the attribute. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the attribute name is placed.

The `mrt_PortalClassAttributeName` function obtains the name of the *attrId* attribute in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    if (nameRef) {
#           ifndef MRT_NO_NAMES
        *nameRef = class->classAttrs[attrId].name ;
#           else
        *nameRef = NULL ;
#           endif /* MRT_NO_NAMES */
    }

    return 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeSize(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

attrId
The number of the attribute. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassAttributeSize` function returns the number of bytes occupied by the *attrId* attribute in the *classId* of the domain described by *portal*. If successful, the return value of the function is positive and represents the number of bytes of storage allocated to the attribute. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeSize(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    return class->classAttrs[attrId].size ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassEventName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

eventNumber
The number of the event. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the event name is placed.

The `mrt_PortalClassEventName` function obtains the name of the *eventNumber* event of the state model in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassEventName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

#       ifndef MRT_NO_NAMES
    char const *const *eventNames = class->eventNames ;
    if (eventNames != NULL && nameRef != NULL) {
        *nameRef = eventNames[eventNumber] ;
    }
#       else
    if (nameRef != NULL) {
        *nameRef = NULL ;
    }
#       endif /* MRT_NO_NAMES */

    return 0 ;
```

```
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassStateName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode stateCode,
    char const **nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

stateCode
The number of the state. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the state name is placed.

The `mrt_PortalClassStateName` function obtains the name of the *stateCode* state of the state model in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassStateName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode stateCode,
    char const **nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    MRT_edb const *edb = class->edb ;
    if (edb == NULL) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (stateCode < 0 || stateCode >= edb->stateCount) {
        return MICCA_PORTAL_NO_STATE ;
    }

    if (nameRef) {
#           ifndef MRT_NO_NAMES
        *nameRef = edb->stateNames[stateCode] ;
#           else
        *nameRef = NULL ;
#           endif /* MRT_NO_NAMES */
    }
```

```
    return 0 ;
}
```

Chapter 28

Asynchronous Execution

When the main loop was discussed, we made a brief mention of asynchronous execution in the context of executing synchronization functions. It is now time to discuss how the run-time deals with asynchronous execution. First we discuss some background and then present the means used by the run-time to synchronize the two execution contexts.

Until now, the discussion of data management and execution sequencing have assumed that we have a single execution context. However, all modern computer architectures support the concept of an interrupt. An interrupt signaled by some external hardware causes the processor to stop executing and transfer control to a different section of code so that immediate action can be taken on the cause of the interrupt. After the external condition is handled, execution usually resumes where it was interrupted. The need for asynchronous execution was recognized early in computer architecture design as interaction with the external environment is much of what makes a computer a useful machine.

The exact details of how this happens is different for every computer architecture. Some computers offer very sophisticated schemes that include arbitrating the priority of multiple competing interrupt sources. Most offer only a single priority of interrupt processing or place the burden of prioritization on the programmer or external hardware. The `micca` run-time is intended for highly embedded systems and, in such systems, achieving the low execution overhead is of great value. So the model of asynchronous execution used by the run-time mirrors that which is provided directly by the hardware architecture. Techniques for having multiple, scheduled execution contexts are very well known. None of them is used here. Here we are only interested in a very simple model of asynchronous execution that closely mirrors what is provided by the computing hardware. Amazingly, this is much less restrictive than might be first imagined.

Nesting interrupts is not forbidden, but is definitely not encouraged. No restriction is put on what may happen at interrupt service. This is usually a potential source of error in many systems. Because the execution is asynchronous to domain model execution, any access to the domain data structures from interrupt service **must not** be attempted. It is ultimately unsafe and results in horribly difficult to detect timing windows being generated wherein things will mysteriously and irreproducibly fail.

However, it is usually necessary for an interrupt to communicate back to the domain models. Usually, the interrupt signals some change of condition in the environment that has been detected by the hardware. The interrupt may not be able to do everything required to handle the situation and frequently additional computation is required to resolve what has happened. The run-time provides a way for an interrupt to request synchronization with the running background. This is accomplished by having the interrupt service code perform a synchronization request.

```
<<mrt external interfaces>>=
extern MRT_SyncParams *
mrt_SyncRequest(
    MRT_SyncFunc syncfunc) ;
```

syncfunc
A pointer to a synchronization function.

The `mrt_SyncRequest` function is called from interrupt service routines to request that the synchronization function, `syncfunc`, be executed in the background at the first safe opportunity. The return value of the function is a pointer to a parameter area where arguments to `syncfunc` may be placed which are then made available when the function is invoked.

The request is in the form of a function, with optional parameters, that is to be executed at the first **safe** opportunity. As we saw before, the synchronization functions are executed in between state activities to insure that the domain data are in a coherent state.

To implement deferring the sync function, the run-time provides a sync queue where requests for background synchronization execution are placed. The queue is implemented in an array. Like all resources in the run-time, the storage required for the queue is fixed at compile time.

```
<<mrt interface constants>>=
#ifndef MRT_SYNC_QUEUE_SIZE
#   define MRT_SYNC_QUEUE_SIZE 10
#endif /* MRT_SYNC_QUEUE_SIZE */
```

The number of queue entries can be sized appropriately for the system. Generally, the number of queue slots must be sized to handle any cluster of interrupts that go off at nearly the same time.

It is necessary to provide interrupts the ability to pass parameters to the background sync functions. This is often data that must be sampled coincident with the interrupt in order to capture the correct external state. To avoid problems with variable life times, data is passed by value by placing it in a parameter area.

This is not a good strategy for passing a large amount of data from the interrupt context to the background. In those cases, it is necessary to manage memory between the background and the interrupts. No facilities are provided by the run-time to do this as it usually must to be constructed *ad hoc* to suit the particular needs of the data transfer. It is often sufficient to manage a memory pool in the background and use the synchronization to the background to allow interrupt service to return memory to the pool when it is no longer needed. The archetypal example is communications buffers. Background code can manage a buffer pool. When outgoing messages are transmitted, synchronization to the background can be used to return the buffer to the pool. Incoming messages can be placed in pre-queued buffers and when the message is complete, synchronizing to the background passes buffer pointers to be processed and returned to the pool.

Sync Function and Parameters

For our purposes here, it is only necessary to define some data structure that can be used by the interrupt service code to place data that will be delivered to the background function. The same considerations that were discussed for event parameters apply for sync function parameters. So we use the same strategy and simply adopt the same data structure for sync function parameters as for event parameters.

```
<<mrt interface aggregate types>>=
typedef MRT_EventParams MRT_SyncParams ;
```

There is no meaningful return value of a sync function and they are simply passed a pointer to their parameter data area. The prototype for a sync function is as follows.

```
<<mrt interface aggregate types>>=
typedef void (*MRT_SyncFunc)(MRT_SyncParams const *) ;
```

Sync Queue

Each element of the sync queue is a pointer to the sync function and a place where the interrupt service code may place the values of the parameters.

```
<<mrt implementation aggregate types>>=
typedef struct mrtsyncblock {
    MRT_SyncFunc function ;
    alignas(max_align_t) MRT_SyncParams params ;     // ❶
} MRT_SyncBlock ;
```

❶ *N.B.* the use of a `C11` alignment feature.

The sync queue is stored in an array and we use a couple of pointers to keep track of the head and tail.

```
<<mrt implementation aggregate types>>=
typedef struct mrtsyncqueue {
    MRT_SyncBlock *head ;
    MRT_SyncBlock *tail ;
} MRT_SyncQueue ;
```

Following our usual pattern, we allocate storage for the sync queue entries as an array and for the control structure that tracks the queue boundaries within the sync queue entry array.

```
<<mrt static data>>=
static MRT_SyncBlock mrtSyncQueueStorage[MRT_SYNC_QUEUE_SIZE] ;
static MRT_SyncQueue mrtSyncQueue = {
    .head = mrtSyncQueueStorage,
    .tail = mrtSyncQueueStorage,
} ;
```

There are only three operations on the sync queue. First we must be able to determine if the queue is empty. Emptiness is determined when the queue head is equal to the queue tail. For this type of queue operation, the head is where the next entry is removed and the tail is where the next entry is inserted.

```
<<mrt forward references>>=
static inline bool mrtSyncQueueEmpty(void) ;
```

```
<<mrt static functions>>=
static inline bool
mrtSyncQueueEmpty(void)
{
    return mrtSyncQueue.head == mrtSyncQueue.tail ;
}
```

We need to be able to add a sync function to the queue. Since the sync queue is shared between the background and foreground, the operations on the queue must be protected in a critical section.

```
<<mrt forward references>>=
static MRT_SyncParams *mrtSyncQueuePut(MRT_SyncFunc f) ;
```

```
<<mrt static functions>>=
static MRT_SyncParams *
mrtSyncQueuePut(
    MRT_SyncFunc f)
{
    MRT_SyncParams *params = NULL ;

    mrtBeginCriticalSection() ;
```

```
    MRT_SyncBlock *tail = mrtSyncQueue.tail ;                          // ❶
    MRT_SyncBlock *newTail = tail + 1 ;
    if (newTail >= mrtSyncQueueStorage + COUNTOF(mrtSyncQueueStorage)) {
        newTail = mrtSyncQueueStorage ;
    }

    if (newTail != mrtSyncQueue.head) {                                // ❷
        mrtSyncQueue.tail = newTail ;
        tail->function = f ;
        params = &tail->params ;
    }

    mrtEndCriticalSection() ;

    return params ;
}
```

❶ The `tail` pointer points to the location where the next sync queue function was placed. Incrementing the tail pointer must account for wrapping around the array boundary.

❷ Overflow is detected as the queue being in the state where the next insertion location equals the head of the queue (*i.e.* the next insertion would overwrite the sync block on the head of the queue). Note the sync queue tail pointer is not modified until after we are assured there is no overflow. This insures that should an overflow error happen, the queue is left in a consistent state.

The consumer of the sync queue entries is the main loop.

```
<<mrt forward references>>=
static MRT_SyncBlock *mrtSyncQueueGet(void) ;
```

```
<<mrt static functions>>=
static MRT_SyncBlock *
mrtSyncQueueGet(void)
{
    MRT_SyncBlock *block = NULL ;

    mrtBeginCriticalSection() ;

    if (!mrtSyncQueueEmpty()) {
        block = mrtSyncQueue.head ;                                    // ❶
        if (++mrtSyncQueue.head >=
                mrtSyncQueueStorage + COUNTOF(mrtSyncQueueStorage)) {  // ❷
            mrtSyncQueue.head = mrtSyncQueueStorage ;
        }
    }

    mrtEndCriticalSection() ;

    return block ;
}
```

❶ The `head` pointer points to the next sync queue block to consider. Fetch the current sync queue block and point to the next one.

❷ Wrap around the end of the array if we have advanced the `head` pointer past the end of the storage array.

Since the main loop runs with interrupts enabled, obtaining a sync queue entry must be done in a critical section. We remove entries from the head of the queue. Otherwise, the only complexity in the code is to account for wrap around in the the array storage.

Dispatching Synchronization Functions

Above we saw that when an event is dispatched, the `mrtDispatchTOCEvent` function is called. This function performs the background synchronization before dispatching a single event.

```
<<mrt forward references>>=
static inline bool mrtInvokeOneSyncFunction(void) ;
```

Invoking a sync function is a simple matter of pulling it off the sync queue and passing a pointer to the parameters to it. We return a value indicating if a function was called.

```
<<mrt static functions>>=
static inline bool
mrtInvokeOneSyncFunction(void)
{
    bool didInvoke = false ;
    do {
        MRT_SyncBlock const *block = mrtSyncQueueGet() ;

        if (block == NULL) {
            break ;                                 // ❶
        }
        assert(block->function != NULL) ;           // ❷
        if (block->function != NULL) {
            block->function(&block->params) ;
            didInvoke = true ;
        }
    } while (!didInvoke) ;

    return didInvoke ;
}
```

❶ The value of `didInvoke` should always be false at this statement.

❷ There should be no `NULL` function pointers placed in the sync queue, but should it happen, we just skip that entry and go on to consider the next one.

Chapter 29

Event Dispatch Tracing

Debugging event driven, callback, state machine based applications can be rather more complicated than conventional, linear flow code. When class instances generate events to other instances and it can be hard to determine the exact sequence of execution by simply examining the source code. Indeed, part of the intent here is to factor away from the application the details of sequencing execution. Setting a breakpoint in the action of a state is easy enough, the difficulties arise when trying to determine where to set a breakpoint to catch the results of the next event dispatch. Given that many events will be flying around a given program, it is very useful to be able to collect the set of event dispatches in chronological order.

To help in debugging, the run-time can be conditionally compiled to support tracing the event dispatch. After the code is properly compiled, a pointer to a trace callback function may be registered and then each event dispatched will result in the function being called with the information about the dispatch.

It should be noted that dealing with trace information can be very difficult. For small embedded systems, there may not be sufficient space to store the strings that give names to states and events (the `MRT_NO_NAMES` macro can be defined to remove the string information). This means that only numbers are available to the tracing code and there is substantial effort required to back translate the numbers into strings that are meaningful to a human.

Because tracing also affects the performance of the run-time code, it may also be excluded by defining the `MRT_NO_TRACE` macro. Most projects will want to define both `MRT_NO_NAMES` and `MRT_NO_TRACE` in code compiled for a release. This, along with defining `NDEBUG`, will result in significantly less initialized data space usage in the executables.

Trace Information

Since there are three types of events, there are three distinct sets of information generated when an event is dispatched. There is information common to all events and information specific to each event type.

```
<<mrt interface trace aggregate types>>=
struct mrttraceinfo ;
typedef struct mrttraceinfo MRT_TraceInfo ;

<<mrt internal trace aggregate types>>=
struct mrttraceinfo {
    <<common trace fields>>
    union {
        <<transition trace fields>>
        <<polymorphic trace fields>>
        <<creation trace fields>>
    } info ;
} ;
```

Common Trace Data

```
<<common trace fields>>=
MRT_EventType eventType ;
MRT_EventCode eventNumber ;
MRT_Instance *sourceInst ;
MRT_Instance *targetInst ;
```

eventType
The type of the event that was dispatched.

eventNumber
The number of the event that was dispatched.

sourceInst
A pointer to the instance that was the source of the dispatched event.

targetInst
A pointer to the instance that was the target of the dispatched event.

Transition Event Trace Data

transitionTrace

```
<<transition trace fields>>=
struct transitiontrace {
    MRT_StateCode currentState ;
    MRT_StateCode newState ;
} transitionTrace ;
```

currentState
The current state of the instance before the event dispatch.

newState
The new state entered as a result of the transition.

Polymorphic Event Trace Data

polyTrace

```
<<polymorphic trace fields>>=
struct polytrace {
    MRT_SubclassCode subcode ;
    MRT_DispatchCount genNumber ;
    MRT_EventCode mappedEvent ;
} polyTrace ;
```

subcode
The subclass code of the currently related instance.

genNumber
The number of the generalization down which the event was dispatched.

mappedNumber
The new event number to which the polymorphic event mapped.

The subclass type is encoded as a small integer value.

```
<<mrt internal trace simple types>>=
typedef uint8_t MRT_SubclassCode ;
```

Creation Event Trace Data

creationTrace

```
<<creation trace fields>>=
struct creationtrace {
    MRT_Class const *targetClass ;
} creationTrace ;
```

targetClass
A pointer to the class structure for the target of the creation event.

Access to Trace Information

Event tracing information is passed out of the run-time by having the application register a callback function. That function takes a pointer to the trace information as its argument.

```
<<mrt interface trace aggregate types>>=
typedef void (*MRT_TraceHandler)(MRT_TraceInfo const *) ;
```

The function is registered with the run-time by invoking:

```
<<mrt trace external interfaces>>=
extern MRT_TraceHandler
mrt_RegisterTraceHandler(
    MRT_TraceHandler handler) ;
```

handler
A pointer to a function that handles trace information.

The `mrt_RegisterTraceHandler` function is used to supply a trace callback function. Supplying a non-NULL value for *handler* also enables tracing. The return value of the the function is the previous value of the callback. Tracing can be turned off by invoking `mrt_RegisterTraceHandler` with `NULL`.

The trace handling function is invoked with a pointer to the trace information for the event being dispatched. The data pointed to by the argument is not valid after the return from the trace handler function. So handler functions must copy the trace data if they wish it to be available after their return.

The current value of the tracing function is stored in a static variable.

```
<<mrt trace static data>>=
static MRT_TraceHandler mrtTraceHandler ;
```

```
<<mrt trace external functions>>=
MRT_TraceHandler
mrt_RegisterTraceHandler(
    MRT_TraceHandler handler)
{
    MRT_TraceHandler oldhandler = mrtTraceHandler ;
    mrtTraceHandler = handler ;
    return oldhandler ;
}
```

The implementation of registering a handler is simply to record the function pointer in a variable.

For each type of event dispatch, the run-time calls a specific function to determine if tracing is enabled and to marshal the trace information into the proper data structure.

```
<<mrt trace static functions>>=
static inline void
mrtTraceTransitionEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_StateCode currentState,
    MRT_StateCode newState)
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtTransitionEvent,
            .eventNumber = event,
            .sourceInst = source,
            .targetInst = target,
            .info.transitionTrace = {
                .currentState = currentState,
                .newState = newState
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

```
<<mrt trace static functions>>=
static inline void
mrtTracePolymorphicEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_SubclassCode subclass,
    MRT_DispatchCount genNumber,
    MRT_EventCode newEvent)
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtPolymorphicEvent,
            .eventNumber = event,
            .sourceInst = source,
            .targetInst = target,
            .info.polyTrace = {
                .subcode = subclass,
                .genNumber = genNumber,
                .mappedEvent = newEvent
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

```
<<mrt trace static functions>>=
static inline void
mrtTraceCreationEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_Class const *class)
```

```
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtCreationEvent,
            .eventNumber = event,
            .sourceInst = source,
            .targetInst = target,
            .info.creationTrace = {
                .targetClass = class
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

Obtaining Tracing Output

A default trace handler is supplied that simply prints the trace information to the standard output using `printf`. Two version are supplied. One for when strings are present and the other for when they are not and only numbers can be printed.

```
<<mrt trace static functions>>=
#ifndef MRT_NO_STDIO
#ifndef MRT_NO_NAMES
static void
mrtPrintTraceInfo(
    MRT_TraceInfo const *traceInfo)
{
    char const *sourceName ;
    char const *sourceClassName ;
    char sourceIdNum[32] ;

    if (traceInfo->sourceInst == NULL) {
        sourceName = "?" ;
        sourceClassName = "?" ;
    } else {
        sourceClassName = traceInfo->sourceInst->classDesc->name ;
        sourceName = traceInfo->sourceInst->name ;
        if (sourceName == NULL) {
            unsigned instid = mrt_InstanceIndex(traceInfo->sourceInst) ;
            snprintf(sourceIdNum, sizeof(sourceIdNum), "%u", instid) ;
            sourceName = sourceIdNum ;
        }
    }

    char const *targetName = traceInfo->targetInst->name ;
    char targetIdNum[32] ;
    if (targetName == NULL) {
        unsigned instid = mrt_InstanceIndex(traceInfo->targetInst) ;
        snprintf(targetIdNum, sizeof(targetIdNum), "%u", instid) ;
        targetName = targetIdNum ;
    }

    switch (traceInfo->eventType) {
    case mrtTransitionEvent: {
        MRT_StateCode newState = traceInfo->info.transitionTrace.newState ;
        char const *newStateName ;
        if (newState == MRT_StateCode_IG) {
            newStateName = "IG" ;
        } else if (newState == MRT_StateCode_CH) {
```

```
            newStateName = "CH" ;
        } else {
            newStateName = traceInfo->targetInst->classDesc->edb->stateNames[
                traceInfo->info.transitionTrace.newState] ;
        }

        printf("%s: Transition: %s.%s - %s -> %s.%s: %s ==> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->targetInst->classDesc->name, targetName,
            traceInfo->targetInst->classDesc->edb->stateNames[
                traceInfo->info.transitionTrace.currentState],
            newStateName) ;
    }
        break ;

    case mrtPolymorphicEvent: {
        MRT_Relationship const *rel = traceInfo->targetInst->classDesc->pdb->
                genDispatch[traceInfo->info.polyTrace.genNumber].relship ;
        MRT_Class const *subclass ;
        char const *subname = NULL ;
        if (rel->relType == mrtRefGeneralization) {
            subclass = rel->relInfo.refGeneralization.
                subclasses[traceInfo->info.polyTrace.subcode].classDesc ;
            subname = subclass->name ;
        } else if (rel->relType == mrtUnionGeneralization) {
            subclass = rel->relInfo.unionGeneralization.
                subclasses[traceInfo->info.polyTrace.subcode] ;
            subname = subclass->name ;
        } else {
            printf("%s: bad relationship type in polymorphic event, %d\n",
                mrtTimestamp(), rel->relType) ;
            break ;
        }
        printf("%s: Polymorphic: %s.%s - %s -> %s.%s: %s - %s -> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->targetInst->classDesc->name, targetName,
            traceInfo->targetInst->classDesc->pdb->genNames[
                traceInfo->info.polyTrace.genNumber],
            subclass->eventNames[traceInfo->info.polyTrace.mappedEvent],
            subname) ;
    }
        break ;

    case mrtCreationEvent:
        printf("%s: Creation: %s.%s - %s -> %s ==> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->info.creationTrace.targetClass->name,
            targetName) ;
        break ;

    default:
        printf("%s: Unknown trace event type, \"%u\"",
                mrtTimestamp(), traceInfo->eventType) ;
        break ;
    }
}
#   else  /* MRT_NO_NAMES is defined */
static void
mrtPrintTraceInfo(
```

```
    MRT_TraceInfo const *traceInfo)
{
    switch (traceInfo->eventType) {
    case mrtTransitionEvent:
        printf("%s: Transition: %p - %u -> %p: %u ==> %u\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->targetInst,
                traceInfo->info.transitionTrace.currentState,
                traceInfo->info.transitionTrace.newState) ;
        break ;

    case mrtPolymorphicEvent:
        printf("%s: Polymorphic: %p - %u -> %p: %u - %u -> %d\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->targetInst,traceInfo->info.polyTrace.genNumber,
                traceInfo->info.polyTrace.mappedEvent,
                traceInfo->info.polyTrace.subcode) ;
        break ;

    case mrtCreationEvent:
        printf("%s: Creation: %p - %u -> %p ==> %p\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->info.creationTrace.targetClass,
                traceInfo->targetInst) ;
        break ;

    default:
        printf("%s: Unknown trace event type, \"%u\"",
                mrtTimestamp(), traceInfo->eventType) ;
        break ;
    }
}
#endif /* MRT_NO_NAMES */
#endif /* MRT_NO_STDIO */
```

```
<<mrt forward references>>=
#ifndef MRT_NO_TRACE
static char const *mrtTimestamp(void) ;
#endif /* MRT_NO_TRACE */
```

Tracing Strategies

Clearly, tracing can generate data at a rather high rate and can be rather intrusive upon the execution of the system. Several strategies may be used to deal with the trace data. If possible, all the trace data can be dumped in a raw form out a communications interface and let some other program decode and display it. That may still be too intrusive and sometimes it is best to filter the trace data based on the target instance pointer value. In this way you may trace the event dispatches on only a subset of instances. Several different filtering schemes, such as source instance or classes, can be envisioned.

Another possibility is to store trace information in a memory area in some sort of circular queue arrangement. Then it is possible for the application to start and stop such tracing and achieve “logic analyzer” type triggering functionality. The trace information can then be extracted from memory and analyzed.

When running in a POSIX environment, one can assume reasonable I/O facilities. The POSIX version of the run-time includes default trace handling to timestamp and print the trace data in human readable form.

You will also note that the trace information has no timing data associated with it. This type of data is so system specific that it is left to the tracing callback to supply. If you have a free running cycle counter in your system, this can be a good indicator of relative time and the trace callback function can add this to the data set supplied by the run-time. Your system may also have

source of clocked timing data that can also be used as a timing reference. In either case, augmenting the trace data with some sort of relative time information is very valuable.

Tracing can also be used as a framework for testing. If a domain is built to run in a testing framework where tracing is enabled, then recording all the trace information allows one to determine the amount of *transition* coverage a test set causes. The goal is to develop test sets that drive the domain with appropriate data so that all state transitions are taken. Tracing allows the recording of what transitions a given thread of control causes. Since in most well designed state machines, state activity code is small and does not contain complicated or intricate internal program flow, causing all state activities to be executed is often close to complete statement coverage. As an added benefit, state machines can be considered as directed graphs. A depth first traversal of a directed graph can be be used to determine a spanning tree for the graph. Traversing a spanning tree for a graph insures that all nodes in the graph are visited and the event sequence given by the spanning tree can guide the generation of test set data and can help to minimize the number of test cases required to ensure adequate coverage.

Chapter 30

Error Handling

Until now we have glossed over the subject of how to handle errors in the `micca` run-time. In XUML, the domains assume a perfect architecture in the sense that no formal mechanism is provided to signal architectural errors back to the application domains. This makes sense because the application models are meant to be implementation independent and able to be run on a variety of underlying platforms. However, an error policy, in much the same terms as data and execution policies, must be put into place. The details of the error handling policy will vary between software architectures, so it is important to state them clearly. For the `micca` run-time, the following principles guide error handling.

- To that extent possible, the run-time operations should not report errors back to the application. For implementation languages that do not support exception handling, the usual technique of returning error codes is not very effective. Either by accident or sloth, many error codes are not checked. Even when the error code is checked, there is little recovery recourse for the application. For example, it does little good to know that we are unable to generate an event because we do not have sufficient ECB resources when there is nothing a state activity can do to free up the required resources.
- Errors that result from exhausted resources or analysis errors detected at run-time are **fatal**. Exactly how fatal errors are acted upon is platform dependent and may result in terminating a program or completely resetting the system. Regardless of the consequence of a fatal error, the assumption is that the program can no longer continue to run.

With these principles in mind, we define a set of error conditions that are detected by the run-time. All these conditions are fatal and are handled by invoking a fatal error handler.

```
<<mrt interface simple types>>=
typedef enum {
    mrtCantHappen = 1,
    mrtEventInFlight,
    mrtNoECB,
    mrtNoInstSlot,
    mrtUnallocSlot,
    mrtSyncOverflow,
    mrtRefIntegrity,
    mrtTransOverflow,
    mrtInstSetOverflow,
    mrtStaticRelationship,
    mrtRelationshipLinkage,
    mrtDupAssociator,
    mrtPanic,

#       ifdef _POSIX_C_SOURCE
    mrtTimerOpFailed,
    mrtSignalOpFailed,
    mrtSelectWaitFailed,
#       endif /* _POSIX_C_SOURCE */
} MRT_ErrorCode ;
```

We will need a string representation of the error codes to make human readable messages.

```
<<mrt static data>>=
static char const * const mrtErrorMsgs[] = {
    [0] = "no error",     /* place holder */
    [mrtNoECB] = "no available Event Control Blocks\n",
    [mrtSyncOverflow] = "synchronization queue overflow\n",
    [mrtTransOverflow] = "transaction markings overflow\n",
    [mrtInstSetOverflow] = "instance set overflow: instance number %u\n",
    [mrtStaticRelationship] = "attempt to modify static relationship\n",
    [mrtRelationshipLinkage] = "invalid instance linkage operation or value\n",
    [mrtPanic] = "panic: %s\n",

#       ifndef MRT_NO_NAMES
    [mrtCantHappen] = "can't happen transition: %s.%s: %s - %s -> CH\n",
    [mrtEventInFlight] = "event-in-flight error: %s.%s - %s -> %s.%s\n",
    [mrtNoInstSlot] = "no available instance slots: %s\n",
    [mrtUnallocSlot] = "unallocated instance slot: %u in class %s\n",
    [mrtRefIntegrity] = "referential integrity check failed: %s\n",
    [mrtDupAssociator] = "duplicate associator instance: %s\n",
#       else
    [mrtCantHappen] = "can't happen transition: %p: %u - %u -> CH\n",
    [mrtEventInFlight] = "event-in-flight error: %p - %u -> %p\n",
    [mrtNoInstSlot] = "no available instance slots: %p\n",
    [mrtUnallocSlot] = "unallocated instance slot: %u in class %p\n",
    [mrtRefIntegrity] = "referential integrity check failed: %p\n",
    [mrtDupAssociator] = "duplicate associator instance: %p\n",
#       endif /* MRT_NO_NAMES */

#       ifdef _POSIX_C_SOURCE
    [mrtTimerOpFailed] = "interval timer operation failed: %s\n",
    [mrtSignalOpFailed] = "signal operation failed: %s\n",
    [mrtSelectWaitFailed] = "blocking on pselect() failed: %s\n",
#       endif /* _POSIX_C_SOURCE */
} ;
```

For some of the error messages, there are parameters which containing naming information. Above we defined two versions of the format string for the error message. When names are not present, the best we can usually do is to print pointer addresses. Below we define inline functions to handle the difference when naming information is included and when it is not. This allows us to have a single interface and avoid sprinkling conditional compilation directives through the code.

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtCantHappenError(
    MRT_Instance *const targetInst,
    MRT_StateCode currentState,
    MRT_EventCode eventNumber)
{
    mrtFatalError(mrtCantHappen,
        targetInst->classDesc->name,
        targetInst->name ? targetInst->name : "?",
        targetInst->classDesc->edb->stateNames[currentState],
        targetInst->classDesc->eventNames[eventNumber]) ;
}

#else

noreturn static void inline
mrtCantHappenError(
    MRT_Instance *const targetInst,
```

```
    MRT_StateCode currentState,
    MRT_EventCode eventNumber)
{
    mrtFatalError(mrtCantHappen, targetInst, currentState, eventNumber) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES
noreturn static void inline
mrtEventInFlightError(
    MRT_Instance *const sourceInst,
    MRT_EventCode eventNumber,
    MRT_Instance *const targetInst)
{
    mrtFatalError(mrtEventInFlight,
            sourceInst ? sourceInst->classDesc->name : "?",
            sourceInst ? sourceInst->name : "?",
            targetInst->classDesc->eventNames[eventNumber],
            targetInst->classDesc->name,
            targetInst->name ? targetInst->name : "?") ;
}

#else

noreturn static void inline
mrtEventInFlightError(
    MRT_Instance *const sourceInst,
    MRT_EventCode eventNumber,
    MRT_Instance *const targetInst)
{
    mrtFatalError(mrtEventInFlight, sourceInst, eventNumber, targetInst) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtNoInstSlotError(
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtNoInstSlot, classDesc->name) ;
}

#else

noreturn static void inline
mrtNoInstSlotError(
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtNoInstSlot, classDesc) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES
```

```
noreturn static void inline
mrtUnallocSlotError(
    MRT_InstId slot,
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtUnallocSlot, slot, classDesc->name) ;
}

#else

noreturn static void inline
mrtUnallocSlotError(
    MRT_InstId slot,
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtUnallocSlot, slot, classDesc) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtRefIntegrityError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtRefIntegrity, rel->name) ;
}

#else

noreturn static void inline
mrtRefIntegrityError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtRefIntegrity, rel) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtDupAssociatorError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtDupAssociator, rel->name) ;
}

#else

noreturn static void inline
mrtDupAssociatorError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtDupAssociator, rel) ;
}

#endif /* MRT_NO_NAMES */
```

Everywhere else the run-time operations have been crafted to avoid error possibilities. For example, as discussed in delayed event generation, we interpret the attempt to generate a duplicate delayed event as wishing to cancel the existing one and instantiate a new one. This semantic interpretation avoids generating an error and avoids all the additional code require in state activities that generate delayed events.

Exactly how fatal errors are handled will depend upon the specifics of how the platform handles errors. We define an interface for a fatal error handler.

```
<<mrt interface aggregate types>>=
typedef void (*MRT_FatalErrorHandler)(MRT_ErrorCode, char const *, va_list) ;
```

The interface is patterned after `vprintf`, giving a format string and a pointer to a variable length argument list.

```
<<mrt forward references>>=
static void
mrtDefaultFatalErrorHandler(
    MRT_ErrorCode errNum,
    char const *fmt,
    va_list ap) ;
```

The system provides a default fatal error handler, a message is printed to the standard error stream.

```
<<mrt static functions>>=
static void
mrtDefaultFatalErrorHandler(
    MRT_ErrorCode errNum,
    char const *fmt,
    va_list ap)
{
#       ifndef MRT_NO_STDIO
    vfprintf(stderr, fmt, ap) ;
#       endif /* MRT_NO_STDIO */
}
```

A pointer to the fatal error handler is initialized with the default one.

```
<<mrt static data>>=
static MRT_FatalErrorHandler mrtErrHandler = mrtDefaultFatalErrorHandler ;
```

Because fatal error handling is usually so platform specific and because of the need to test fatal error paths, we provide the ability to delegate the consequence of the fatal error.

```
<<mrt external interfaces>>=
extern MRT_FatalErrorHandler
mrt_SetFatalErrorHandler(
    MRT_FatalErrorHandler newHandler) ;
```

`newHandler`
A pointer to a fatal error handler function.

The `MRT_FatalErrorHandler` function install `newHandler` as the fatal error handler and return the previous error handler function pointer.

```
<<mrt external functions>>=
MRT_FatalErrorHandler
mrt_SetFatalErrorHandler(
    MRT_FatalErrorHandler newHandler)
{
    MRT_FatalErrorHandler prevHandler = mrtErrHandler ;
```

```
    if (newHandler) {
        mrtErrHandler = newHandler ;
    }
    return prevHandler ;
}
```

The run-time internally calls `mrtFatalError`.

```
<<mrt forward references>>=
noreturn static void
mrtFatalError(MRT_ErrorCode errNum, ...) ;
```

```
<<mrt static functions>>=
noreturn static void
mrtFatalError(
    MRT_ErrorCode errNum,
    ...)
{
    va_list ap ;

    assert(mrtErrHandler != NULL) ;
    assert(errNum < COUNTOF(mrtErrorMsgs)) ;

    va_start(ap, errNum) ;
    mrtErrHandler(errNum, mrtErrorMsgs[errNum], ap) ;
    va_end(ap) ;
    /*
     *  If the handler does return, we insist that all errors
     *  are fatal. So we abort().
     */
    abort() ;
}
```

As we see in the code, we insist that there is an error handler. There is always the default one and a different handler can be specified if necessary. Finally, `abort()` is called should the error handler return.

Avoiding Fatalities

In this error handling strategy, every run-time detected error is fatal. Although the details of the processing for fatal errors can be delegated, in most systems of the class we consider here, fatal errors usually result in a system reset. Under the vast majority of circumstances, that is the desired behavior. However, there are some particular circumstances where causing a fatal error is not the desired behavior.

Consider the case where some external stimulus results in an event being generated. If the stimulus occurs more frequently than events can be processed, then the run-time will run out of event control blocks causing a fatal error. As an example, consider the arrival of a communications packet. If somewhere during the processing of the packet an event is generated, then if packets arrive too fast a fatal error can be generated. In effect it would provide a means for an external stimulus to cause the system to crash. Certainly for the case of a communications packet, the preferred behavior would be to drop the packet and let higher level protocol deal with the necessary retries, etc. It is then necessary to be able to determine if generating an event would be successful.

In this section we describe functions that can be used to avoid run-time requests that would exhaust an underlying resource and therefore cause a fatal system error. It should be emphasized that these functions are **not** intended for ordinary or casual use. Under the vast majority of circumstances, such as when one state machine generates an event to another state machine, event generation and other such activities should continue to assume that there are no resources that can be consumed. System analysis and testing should then determine the appropriate sizing for the various resource pools. The capability described in this section is to handle unusual and extraordinary circumstances where hardware failure or failure to abide by communications protocols could force the system into a fatal error situation.

Note also that the alternative provided here causes the external stimulus that would cause the fatal error condition effectively to be ignored. For some system requirements that is not an acceptable solution. For example, consider a digital input line that is used to generate an interrupt and that interrupt signals an external condition monitored by hardware, say the maximum extent of motion of physical robot arm. If this interrupt arrives at a very fast rate, one might conclude the hardware has failed. Ignoring the interrupt might do little other than mask a problem that should cause a fatal error condition and potentially reset the system. The conclusion is that providing a means of avoiding fatal error conditions is not intented to serve as an overall error handling policy. Careful analysis and consideration is still required. If an interrupt arrives faster than expected and, because of what the interrupt represents, it cannot be ignored, that fact and the response to it must be deduced by the interrupt service code (*e.g.* by determining the interrupt frequency) and actions appropriate to the system must be taken. It can be a difficult problem to solve and the functions provided here are too generic to be used indescrimately.

There are three internal run-time resources that can be exhausted.

- Class instances can be dynamically created and each class has its own instance pool.
- Event control blocks are used for generating and dispatching state machine events.
- The foreground / background synchronization queue has a fixed number of slots and excessive synchronization requests from interrupt service routines can fill the queue.

Checking for Available Instance Space

The `mrt_CanCreateInstance()` function provides a means determine if there is space available to create a new instance of a class.

```
<<mrt internal external interfaces>>=
extern bool
mrt_CanCreateInstance(
    MRT_Class const *const classDesc) ;
```

classDesc
A pointer to the class data for which the space check is made.

`mrt_CanCreateInstance` returns `true` if there at least one instance of `classDesc` may be created without exhausting the memory pool of instances for the class.

```
<<mrt external functions>>=
bool
mrt_CanCreateInstance(
    MRT_Class const *const classDesc)
{
    assert(classDesc != NULL) ;

    /*
     * Search for an empty slot in the pool.
     */
    return mrtFindInstSlot(classDesc->iab) != NULL ;
}
```

Checking for Event Blocks

```
<<mrt external interfaces>>=
extern bool
mrt_CanSignalEvent(void) ;
```

The `mrt_CanSignalEvent` function returns true if it is possible to signal an event and *not* cause a system error.

```
<<mrt external functions>>=
bool
mrt_CanSignalEvent(void)
{
    return !mrtEventQueueEmpty(&freeEventQueue) ;
}
```

Non-Fatal Background Synchronization

```
<<mrt external interfaces>>=
MRT_SyncParams *
mrt_TrySyncRequest(
    MRT_SyncFunc syncfunc) ;
```

`syncfunc`
A pointer to a synchronization function.

The `mrt_TrySyncRequest` function attempts to queue `syncfunc` as a synchronization function. The return value of the function is `NULL` if the attempt failed because of lack of space in the sync queue. Otherwise, the return value is non-NULL and points to space where the parameters of `syncfunc` may be placed.

Causing Fatalities

There are rare cases when executing outside of the state machine dispatch mechanism where situations are detected where the execution state cannot be resolved and it is necessary to assert a *panic* situation. For this situation, the run time provides the `mrt_Panic` function.

```
<<mrt external interfaces>>=
extern noreturn void
mrt_Panic(
    char const *format,
    ...) ;
```

`format`
A `printf` style format string.

`...`
A variable length list of arguments matching the `format` string conversion specifiers.

The `mrt_Panic` formats an output string according to the `format` argument and the other invocation parameters and forces an error situation.

```
<<mrt external functions>>=
noreturn void
mrt_Panic(
```

```
    char const *format,
    ...)
{
    char outbuf[128] ;
    va_list ap ;

    va_start(ap, format) ;
    vsnprintf(outbuf, sizeof(outbuf), format, ap) ;
    va_end(ap) ;

    mrtFatalError(mrtPanic, outbuf) ;
}
```

Chapter 31

Linked List Operations

The run-time supplies a set of linked list operations for use in manipulating the lists of instance pointers used to realize and navigate associations. They are the usual circular, doubly linked list manipulations and are presented here without little further comment.

```
<<mrt internal static inlines>>=
static inline MRT_LinkRef *
mrtLinkRefBegin(
    MRT_LinkRef const *list)
{
    return list->next ;
}
```

```
<<mrt internal static inlines>>=
static inline MRT_LinkRef *
mrtLinkRefEnd(
    MRT_LinkRef const *list)
{
    return (MRT_LinkRef *)list ;
}
```

```
<<mrt internal static inlines>>=
static inline bool
mrtLinkRefEmpty(
    MRT_LinkRef const *list)
{
    return list->next == list ;
}
```

```
<<mrt internal static inlines>>=
static inline bool
mrtLinkRefNotEmpty(
    MRT_LinkRef const *list)
{
    return list->next != list ;
}
```

Note that an empty list is one where the list terminus points back to itself.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefInit(
    MRT_LinkRef *ref)
{
```

```
    ref->next = ref->prev = ref ;
}
```

Because the list is doubly linked, we only have to have a pointer to the item to remove it.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefRemove(
    MRT_LinkRef *item)
{
    item->prev->next = item->next ;
    item->next->prev = item->prev ;
    item->next = item->prev = NULL ; // ❶
}
```

❶ We set the values of the `next` and `prev` members to `NULL` when the item has been removed from a list. This gives a handy test to determine if the item is in any list. This is useful in some of the manipulations of linked lists of references in order to maintain such lists as a set.

The insert function inserts an item before a given point in the list. Since it is usually called with a list head as the insertion place, the net effect is to place the item last in the list.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefInsert(
    MRT_LinkRef *item,
    MRT_LinkRef *at)
{
    if (item->next != NULL || item->prev != NULL) { // ❶
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    item->prev = at->prev ;
    item->next = at ;
    at->prev->next = item ;
    at->prev = item ;
}
```

❶ We also test that they are `NULL` before inserting an item into a list to insure that we are not inserting a duplicate. Inserting an item that is already in another list destroys the pointer structure of the list.

Chapter 32

POSIX Specific Code

At this point we have reached the end of the generic code and must now begin to account for the platform differences in the way timing and asynchronous execution is handled. This software architecture can be run on a conventional POSIX platform. This includes Linux, Mac OS X and even Cygwin. The primary purpose of making the run-time function in a POSIX environment is simulation. Often, a domain can be executed for testing and simulation purposes on a conventional computer more easily than in the target environment. With I/O and disk storage, testing and tracing logic is often much easier. It is also perfectly reasonable to implement an application using `micca` and targeting a POSIX platform.

POSIX Critical Sections

We start with a discussion of how to implement a critical section in POSIX. In POSIX, a *signal* is the means provided for asynchronous execution. It is unfortunate that the same word, *signal*, is used in POSIX parlance as we have used as the mechanism for causing state machine transitions. Signals function in much the same way as interrupts do on a micro-controller. Signals can interrupt the running process and then return to it after they complete. There are times when we must insure that execution is *not* interrupted by asynchronous signal execution. The functions in this section accomplish that.

The technique used is to maintain a signal mask of all the signals that are under control of the run-time.

```
<<posix static data>>=
static sigset_t mrtSigMask ;
```

This signal mask can then be used to control signal execution as necessary. Note that an application can call low level signal handling primitives and manage subsets of signals outside of the run-time. This is definitely discouraged as it is likely to introduce race conditions in the run-time. The proper behavior of the run-time depends upon it having knowledge of the signals that are used by the application. Many programs just use the default behavior for a signal, *e.g.* exiting the program on `SIGHUP`. The run-time need not know about such default behavior, especially if the behavior terminates the process.

To keep the run-time informed as to the signals an application uses, they can register a signal function that will be called when the signal is received.

```
<<posix external interfaces>>=
typedef void (*MRT_SignalFunc)(int) ;

extern void
mrt_RegisterSignal(
    int sigNum,
    MRT_SignalFunc func) ;
```

sigNum
The number of the signal being registered.

func
A pointer to the function that is to be executed when the signal happens.

The function `mrt_RegisterSignal` registers the `func` function to be called when `sigNum` is signaled to the process. If `func` is `NULL`, then the signal's behavior is reset to its default behavior.

The implementation of `mrt_RegisterSignal` accounts for both registering the signal action with the operating system and maintaining our signal mask that is used in foreground / background synchronization.

```
<<posix external functions>>=
void
mrt_RegisterSignal(
    int sigNum,
    MRT_SignalFunc func)
{
    assert(sigNum > 0) ;

    struct sigaction action ;
    if (func) {
        action.sa_handler = func ;
        sigaddset(&mrtSigMask, sigNum) ;
    } else {
        action.sa_handler = SIG_DFL ;
        sigdelset(&mrtSigMask, sigNum) ;
    }
    sigfillset(&action.sa_mask) ;
    action.sa_flags = 0 ;

    int sigresult = sigaction(sigNum, &action, NULL) ;
    if (sigresult != 0) {
        mrtFatalError(mrtSignalOpFailed, strerror(errno)) ;
    }
}
```

We set up signal handlers to run uninterrupted by other signals. This is accomplished by filling the `sa_mask` member of the `sigaction` structure. This is simplifies keeping track of what is going on.

Starting a critical section just means that we must block all the managed signals.

```
<<posix static functions>>=
static inline void
mrtBeginCriticalSection(void)
{
    if (sigprocmask(SIG_BLOCK, &mrtSigMask, NULL) != 0) {
        mrtFatalError(mrtSignalOpFailed, strerror(errno)) ;
    }
}
```

The end of a critical section is equally easily accomplished by unblocking the managed signals.

```
<<posix static functions>>=
static inline void
mrtEndCriticalSection(void)
{
    if (sigprocmask(SIG_UNBLOCK, &mrtSigMask, NULL) != 0) {
        mrtFatalError(mrtSignalOpFailed, strerror(errno)) ;
    }
}
```

POSIX Timing Interfaces

In this section we present the timing interface for POSIX systems. In this timing scheme, the interval timer that measures real time is used and notifications of elapsed time arrive via SIGALRM.

In an embedded system, time is usually measured in units of clock ticks, where the amount of real time represented by a clock tick will vary from system to system. It is useful then to run the delayed event queue in hardware device units rather than a conventional time measure so that we may avoid the conversion computation each time the delayed event queue timing is started or stopped.

So, we introduce a couple of functions to convert between clock ticks and milliseconds and *vice versa*. Since the POSIX interface operates at a higher level, in the POSIX case there is no transformation. Note that there is a bit of data type slight of hand going on here. We use the `MRT_DelayTime` type to hold values of milliseconds and clock ticks. In practice this is not a problem, but the type will have to be chosen to account for the largest values held either of the uses of the data type.

```
<<posix static functions>>=
static inline MRT_DelayTime
mrtMsecToTicks(
    MRT_DelayTime msec)
{
    return msec ;
}
```

```
<<posix static functions>>=
static inline MRT_DelayTime
mrtTicksToMsec(
    MRT_DelayTime ticks)
{
    return ticks ;
}
```

For the POSIX run-time, masking the timer used for delayed events means that we must block `SIGALRM`.

```
<<posix static functions>>=
static void
mrtSysTimerMask(void)
{
    /*
     * Make sure SIGALRM does not go off.
     */
    sigset_t mask ;
    sigemptyset(&mask) ;
    sigaddset(&mask, SIGALRM) ;
    if (sigprocmask(SIG_BLOCK, &mask, NULL) != 0) {
        mrtFatalError(mrtSignalOpFailed, strerror(errno)) ;
    }
}
```

The corresponding unmasking operation follows the same pattern.

```
<<posix static functions>>=
static void
mrtSysTimerUnmask(void)
{
    /*
     * Allow SIGALRM to notify us.
     */
    sigset_t mask ;
    sigemptyset(&mask) ;
    sigaddset(&mask, SIGALRM) ;
    if (sigprocmask(SIG_UNBLOCK, &mask, NULL) != 0) {
        mrtFatalError(mrtSignalOpFailed, strerror(errno)) ;
    }
}
```

To start a timer we supply the number of ticks we want to expire before we are notified.

```
<<posix static functions>>=
static void
mrtSysTimerStart(
    MRT_DelayTime time)
{
    struct itimerval delayedEventTimer ;

    delayedEventTimer.it_interval.tv_sec = 0 ;
    delayedEventTimer.it_interval.tv_usec = 0 ;
    delayedEventTimer.it_value.tv_sec = time / 1000 ;
    delayedEventTimer.it_value.tv_usec = (time % 1000) * 1000 ;

    if (setitimer(ITIMER_REAL, &delayedEventTimer, NULL) != 0) {
        mrtFatalError(mrtTimerOpFailed, strerror(errno)) ;
    }
    mrtSysTimerUnmask() ;
}
```

The code initializes the **real** interval timer to use in servicing the delayed event queue. We set the `it_interval` member, which represents the next value to be loaded into the timer, to 0. Then when the time given by the `it_value` member expires the timer is stopped. The system timer is specified in microseconds and our time value is in milliseconds, so some conversion must be performed. Upon expiration, SIGALRM is generated. Notice that we exit the function with SIGALRM unblocked.

Stopping the timer returns the amount of time that had not elapsed.

```
<<posix static functions>>=
static MRT_DelayTime
mrtSysTimerStop(void)
{
    mrtSysTimerMask() ;
    /*
     * Fetch the remaining time.
     */
    struct itimerval delayedEventTimer ;
    if (getitimer(ITIMER_REAL, &delayedEventTimer) != 0) {
        mrtFatalError(mrtTimerOpFailed, strerror(errno)) ;
    }
    /*
     * Convert the returned time into milliseconds.
     */
    MRT_DelayTime remain =
            delayedEventTimer.it_value.tv_sec * 1000 +
            delayedEventTimer.it_value.tv_usec / 1000 ;
    /*
```

```
     * Set the current timer value to zero to turn it off.
     */
    memset(&delayedEventTimer, 0, sizeof(delayedEventTimer)) ;
    if (setitimer(ITIMER_REAL, &delayedEventTimer, NULL) != 0) {
        mrtFatalError(mrtTimerOpFailed, strerror(errno)) ;
    }

    return remain ;
}
```

Stopping the timer must make sure that SIGALRM does not expire during the process of getting things stopped.

We need a function to execute when `SIGALRM` goes off. It must service the delayed event queue and then restart the timer, if necessary.

```
<<posix static functions>>=
static void
mrtSysTimerExpire(
    int signum)
{
    MRT_DelayTime nextTime = mrt_TimerExpireService() ;
    if (nextTime != 0) {
        mrtSysTimerStart(nextTime) ;
    }
}
```

Since the timing services use `SIGALRM`, the signal registration function is used to insure that `SIGALRM` is serviced.

```
<<posix static functions>>=
static void
mrtInitSysTimer(void)
{
    mrt_RegisterSignal(SIGALRM, mrtSysTimerExpire) ;
}
```

POSIX Async Execution Interface

The POSIX view of a process includes the notion of *signals*. Signals are a form of asynchronous execution, and reasonably correspond to the interrupts of a bare metal system. As we have seen in the timer services above, we can use signals to access a variety of services on a POSIX system.

In this section we fill out the asynchronous execution interfaces using signals. As we will see, POSIX systems also require that you deal with their I/O interface in order to properly handle execution sequencing. For now, we present an interface that allows an application to deal with the asynchronous aspects of signals. As expected, the interface allows an application to register a sync function that will be executed at the first safe opportunity after the signal has expired.

```
<<posix external functions>>=
MRT_SyncParams *
mrt_SyncRequest(
    MRT_SyncFunc f)
{
    MRT_SyncParams *params = mrtSyncQueuePut(f) ;
    if (params == NULL) {
        mrtFatalError(mrtSyncOverflow) ;
    }
    return params ;
}
```

The alternate interface is also easily implemented.

```
<<posix external functions>>=
MRT_SyncParams *
mrt_TrySyncRequest(
    MRT_SyncFunc f)
{
    return mrtSyncQueuePut(f) ;
}
```

POSIX I/O Interface

On POSIX platforms, the run-time must also supply services to handle I/O. The reason for this is that there are two means of awakening a sleeping process, receiving a signal and servicing a I/O file descriptor.[1] On bare metal systems, I/O is frequently accomplished on an *ad hoc* basis and the single mechanism of the sync queue is usually sufficient. However, POSIX makes a distinction between signals and I/O operations on file descriptors and an interface needs to be provided to deal with servicing file descriptors that require attention.

We model this interface after the one for signals. The idea is that a set of callback functions can be registered on a file descriptor for reading, writing or an exception. When the condition is satisfied the callback is invoked. So we must define an I/O callback function as:

```
<<posix external interfaces>>=
typedef void (*MRT_FDServiceFunc)(int) ;
```

When the callback is invoked, it is passed the value of the file descriptor that requires service.

The run-time provide two functions for I/O. One registers the callbacks for a file descriptor and the other removes a file descriptor from consideration.

```
<<posix external interfaces>>=
extern void
mrt_RegisterFDService(
    int fd,
    MRT_FDServiceFunc readService,
    MRT_FDServiceFunc writeService,
    MRT_FDServiceFunc exceptService) ;
```

fd
: A file descriptor as returned from `open`, `socket` or any other system calls that create file descriptors.

readService
: A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is readable or `NULL` if no callback is registered.

writeService
: A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is writable or `NULL` if no callback is registered.

exceptService
: A pointer to a callback function that will be registered for the file descriptor and invoked when the file descriptor is in an exception condition or `NULL` if no callback is registered. In practice, exception conditions are used only for reading OOB (out of bands) data on a TCP socket.

The corresponding remove function has the following interface.

[1] There are other ways to integrate signals and I/O in POSIX systems. The use of a `pselect` based approach is a design decision.

```
<<posix external interfaces>>=
extern void
mrt_UnregisterFDService(
    int fd,
    bool rmRead,
    bool rmWrite,
    bool rmExcept) ;
```

fd
A file descriptor as returned from `open`, `socket` or any other system calls that create file descriptors.

rmRead
A boolean indicated whether or not the file descriptor should have its read callback unregistered.

rmWrite
A boolean indicated whether or not the file descriptor should have its write callback unregistered.

rmExcept
A boolean indicated whether or not the file descriptor should have its exception callback unregistered.

The implementation of these two functions requires some internal data structures to track the file descriptor sets. File descriptor sets are handed to `pselect` to indicate how a process is to be awakened for I/O servicing.

To track the callback functions we need a data structure.

```
<<posix implementation aggregate types>>=
typedef struct mrtfdservicemap {
    bool set ;
    MRT_FDServiceFunc read ;
    MRT_FDServiceFunc write ;
    MRT_FDServiceFunc except ;
} MRT_FDServiceMap ;
```

set
A boolean indicating whether or not the entry is in use.

read
A pointer to the read callback registered for the file descriptor or `NULL` if no callback is registered.

write
A pointer to the write callback registered for the file descriptor or `NULL` if no callback is registered.

except
A pointer to the exception callback registered for the file descriptor or `NULL` if no callback is registered.

Following our familiar pattern, we define an array of mapping entries that defines a pool for storing the entries that map file descriptor state to callbacks. This array is indexed by file descriptor value.

```
<<posix static data>>=
static struct mrtfdservicemap mrtFDServicePool[FD_SETSIZE] ;
```

The value of `FD_SETSIZE` is determined by the system and is the maximum number of file descriptors that can be in a `fd_set` given to `select`.

One complication of using `pselect` as a means of registering intent on multiple file descriptors is that you must keep track of the largest value of a file descriptor in the set handed to `pselect`. This is an argument to `pselect` (and `select`). Fortunately, UNIX file descriptors operate in a rather predictable manner. Each process has file descriptors 0, 1, and 2 open when the process is started. Creating a new file descriptor (*e.g.* by opening a file) will allocate the next largest unused file descriptor number.

This rule applies to the three file descriptors opened by default for a process. So, for example, closing file descriptor 2 and then opening a new file will result in file descriptor 2 being reused. All this makes tracking the maximum file descriptor number relatively easy. We only need a single integer variable.

```
<<posix static data>>=
static int mrtMaxFD = -1 ;
```

Since `mrtMaxFD` holds the maximum value of the file descriptors that have been registered, the value -1 indicates that there are no registered file descriptors.

We need variables to hold the three sets of file descriptors needed by `pselect`.

```
<<posix static data>>=
static fd_set mrtReadFDS ;
static fd_set mrtWriteFDS ;
static fd_set mrtExceptFDS ;
```

Finally, we can talk about the implementation of the I/O registration functions.

```
<<posix external functions>>=
void
mrt_RegisterFDService(
    int fd,
    MRT_FDServiceFunc readService,
    MRT_FDServiceFunc writeService,
    MRT_FDServiceFunc exceptService)
{
    assert(fd >= 0 && fd < FD_SETSIZE) ;
    MRT_FDServiceMap *fds = mrtFDServicePool + fd ;

    fds->read = readService ;
    if (readService) {
        FD_SET(fd, &mrtReadFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mrtReadFDS) ;
    }

    fds->write = writeService ;
    if (writeService) {
        FD_SET(fd, &mrtWriteFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mrtWriteFDS) ;
    }

    fds->except = exceptService ;
    if (exceptService) {
        FD_SET(fd, &mrtExceptFDS) ;
        fds->set = true ;
    } else {
        FD_CLR(fd, &mrtExceptFDS) ;
    }

    if (fds->read == NULL && fds->write == NULL && fds-> except == NULL) {
        if (fds->set && fd >= mrtMaxFD) {
            --mrtMaxFD ;
        }
        fds->set = false ;
    } else if (fds->set && fd > mrtMaxFD) {
        mrtMaxFD = fd ;
    }
}
```

The function simply tests the various callback functions and if they are not `NULL`, then the file descriptor is added to the appropriate set. As written, `mrt_RegisterFDService` may be used to modify file descriptors already registered. The last bit of logic is there in case `mrt_RegisterFDService` is used to effectively remove the file descriptor by supplying three `NULL` callback function pointers. We must also account for the maximum file descriptor value that has been registered.

Unregistering a file descriptor from consideration is also straight forward.

```
<<posix external functions>>=
void
mrt_UnregisterFDService(
    int fd,
    bool rmRead,
    bool rmWrite,
    bool rmExcept)
{
    assert(fd >= 0 && fd < FD_SETSIZE) ;
    MRT_FDServiceMap *fds = mrtFDServicePool + fd ;

    if (rmRead) {
        fds->read = NULL ;
        FD_CLR(fd, &mrtReadFDS) ;
    }

    if (rmWrite) {
        fds->write = NULL ;
        FD_CLR(fd, &mrtWriteFDS) ;
    }

    if (rmExcept) {
        fds->except = NULL ;
        FD_CLR(fd, &mrtExceptFDS) ;
    }

    if (fds->read == NULL && fds->write == NULL && fds-> except == NULL &&
            fd >= mrtMaxFD) {
        mrtMaxFD = fd - 1 ;
    }
}
```

We also need something to initialize the file descriptor sets that we are maintaining.

```
<<posix static functions>>=
static void
mrtInitFDService(void)
{
    FD_ZERO(&mrtReadFDS) ;
    FD_ZERO(&mrtWriteFDS) ;
    FD_ZERO(&mrtExceptFDS) ;
}
```

POSIX Suspending Execution

The main loop detects when there is nothing left to do and suspends execution. Here we present how that suspension happens for the POSIX version of the run-time.

This design is based on using `pselect` to suspend a process until either a signal occurs or a file descriptor requires service. The `mrtWait` function is called by the main loop when there is no work currently to be done. It is invoked inside of a critical section. This is an important entry condition for `mrtWait`. In the POSIX case, this means that `mrtWait` must be invoked with all the registered signals blocked. We then use `pselect` to atomically enable all signals and block the process.

```
<<posix static functions>>=
static void
mrtWait(void)
{
    mrtBeginCriticalSection() ;
    if (mrtSyncQueueEmpty()) {
        /*
         * Copy the file descriptor sets since "pselect" modifies them in place
         * upon return.
         */
        fd_set readfds ;
        memcpy(&readfds, &mrtReadFDS, sizeof(readfds)) ;
        fd_set writefds ;
        memcpy(&writefds, &mrtWriteFDS, sizeof(writefds)) ;
        fd_set exceptfds ;
        memcpy(&exceptfds, &mrtExceptFDS, sizeof(exceptfds)) ;
        /*
         * Allow all the signals during the select.
         */
        sigset_t mask ;
        sigemptyset(&mask) ;
        /*
         * "mrtMaxFD" holds the maximum value of any registered file
         * descriptor. We must add one to get the number of file descriptors
         * "pselect" is to consider.
         */
        int r = pselect(mrtMaxFD + 1, &readfds, &writefds,
                &exceptfds, NULL, &mask) ;
        if (r == -1) {
            if (errno != EINTR) {
                mrtFatalError(mrtSelectWaitFailed, strerror(errno)) ;
            }
            /*
             * Got a signal while waiting. We go back to the main loop on the
             * assumption that something has been placed in the sync queue.
             */
        } else {
            /*
             * Dispatch the service functions for the file descriptors.
             */
            MRT_FDServiceMap *s = mrtFDServicePool ;
            for (int fd = 0 ; r > 0 && fd <= mrtMaxFD ; fd++, s++) {
                /*
                 * Do exceptions first. This is only important for sockets, but
                 * without going first the OOB data processing won't work.
                 */
                if (FD_ISSET(fd, &exceptfds)) {
                    assert(s->except != NULL) ;
                    s->except(fd) ;
                    --r ;
                }
                if (FD_ISSET(fd, &readfds)) {
                    assert(s->read != NULL) ;
                    s->read(fd) ;
                    --r ;
                }
                if (FD_ISSET(fd, &writefds)) {
                    assert(s->write != NULL) ;
                    s->write(fd) ;
                    --r ;
                }
```

```
            }
        }
    }
    mrtEndCriticalSection() ;
}
```

By far, most of the work in `mrtWait` is to deal with the file descriptor status changes. The file descriptor sets must be copied before being handed to `pselect` since it modifies them in place. After we determine that it was a file descriptor status change that caused us to wake up, we must go through and find all file descriptors that had a status change and invoke the callback function.

The way that we are using `pselect` in this circumstance may seem a bit backwards. Upon entry to `mrtWait`, we start a critical section where the registered signals are blocked. The signal mask given to `pselect` is empty, meaning that `pselect` will allow all signals while the process sleeps. Upon the return from `pselect` we will be back to the state where the registered signals are blocked. Thus we avoid the race condition where we have determined that the sync queue is empty, but asynchronous execution that might affect the sync queue arrives before we can put the process to sleep. This is exactly the type of race condition `pselect` is used to prevent.

It is also worth noting we do *not* use any time out in the `pselect` invocation. All timing is done via delayed events, and they are signalled via `SIGALRM` and managed on the delayed event queue as discussed before.

POSIX Tracing

Since we have ready access to time information on POSIX, we can put together a convenience function to format the current time of day into a timestamp. The format of the timestamp makes it easy to sort based on the string representation.

```
<<posix trace static functions>>=
static char const *
mrtTimestamp(void)
{
    static char timestamp[128] ;

    struct timeval now ;
    if (gettimeofday(&now, NULL) != 0) {
        return "unknown" ;
    }

    struct tm *ltime ;
    ltime = localtime(&now.tv_sec) ;
    if (ltime == NULL) {
        return strerror(errno) ;
    }

    int tlen = strftime(timestamp, sizeof(timestamp), "%FT%T", ltime) ;
    if (tlen == 0) {
        return strerror(errno) ;
    }

    int flen = snprintf(timestamp + tlen, sizeof(timestamp) - tlen,
            ".%03u.%03u", (unsigned)(now.tv_usec / 1000),
            (unsigned)(now.tv_usec % 1000)) ;
    if (flen > (sizeof(timestamp) - tlen)) {
        return "too big" ;
    }

    return timestamp ;
}
```

POSIX Initialization

Here we present the POSIX version of the required internal initialization.

```
<<posix static functions>>=
static inline void
mrtPlatformInit(void)
{
    sigemptyset(&mrtSigMask) ;
}
```

```
<<posix external functions>>=
void
mrt_Initialize(void)
{
    mrtPlatformInit() ;
    mrtECBPoolInit() ;
    mrtTransactionsInit() ;
    mrtInitSysTimer() ;
    mrtInitFDService() ;
    setvbuf(stdout, NULL, _IOLBF, 0) ; // ❶
#       if !(defined(MRT_NO_TRACE) || defined(MRT_NO_STDIO))
    mrt_RegisterTraceHandler(mrtPrintTraceInfo) ; // ❷
#       endif /* !defined(MRT_NO_TRACE) && !defined(MRT_NO_STDIO) */
}
```

❶ Set up line buffering on stdout. This make sure we see output even if we are piping the output to some other command.

❷ We always register a tracing function unless it is specifically compiled out.

POSIX Include Files

We will need a number of POSIX specific include files.

```
<<posix includes>>=
#include <signal.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>
```

POSIX Files

Here we present the two literate program roots for the run-time. We adopt a naming convention for the chunk names with the understanding that they will be renamed to “micca_rt.h” and “micca_rt.c” when the files are created by tangling.

Run Time Header Header for POSIX

```
<<micca_rt_posix.h>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
```

```
<<version info>>
*/

#ifndef MICCA_RT_H_
#define MICCA_RT_H_

<<common header file definitions>>

<<posix external interfaces>>

#endif /* MICCA_RT_H_ */
```

Run Time Code File for POSIX

```
<<micca_rt_posix.c>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

#define _POSIX_C_SOURCE 200809L

#include "micca_rt.h"
#include "micca_rt_internal.h"
<<posix includes>>

/*
 * Constants
 */
<<mrt implementation constants>>

/*
 * Data Types
 */
<<mrt implementation simple types>>
<<mrt implementation aggregate types>>
<<posix implementation aggregate types>>

/*
 * Forward References
 */
<<mrt forward references>>
<<posix forward references>>

/*
 * Static Data
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static data>>
#   endif /* MRT_NO_TRACE */

<<mrt static data>>
<<posix static data>>

/*
 * Static Inline Functions
 */
```

```
<<mrt implementation static inlines>>

/*
 * Static Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static functions>>
<<posix trace static functions>>
#   endif /* MRT_NO_TRACE */

<<posix static functions>>
<<mrt static functions>>

/*
 * External Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace external functions>>
#   endif /* MRT_NO_TRACE */

<<posix external functions>>
<<mrt external functions>>
```

Chapter 33

ARM v7-M Specific Code

A version of the run-time code is also given for the ARMv7-M architecture. This is the processor architecture for the ARM® Cortex-M series of processors.

ARM processors are always a bit different from those provided by a single manufacturer. ARM only specifies the processor core and instruction set. The chip delivered by a particular company usually consist of a a system on a chip (SOC) that includes both the ARM core and a set of SOC specific peripherals, such as timers and ADC's. This means that some of the support for the ARMv7-M architecture is generic and some is SOC specific.

We show run-time adaptations for two different ARM SOC's:

- The EFM32 Giant Gecko is available from Silicon Labs and features an ARM Cortex-M3 core.
- The MSP432 is available from Texas Instruments and features an ARM Cortex-M4 core.

Controlling basic functions in the processor is generic and we present generic code that is common to both chips.

The code to control vendor specific peripherals might be clumsy were it not for the CMSIS initiative by ARM[1]. Vendors supply "C" header files with the preferred naming conventions for use on their chips. This greatly simplifies being able to provide common core related code and vendor specific code.

ARM v7-M Constants

```
<<arm7m constants>>=
#define MRT_MAX_CLOCK_TICKS     (UINT32_MAX - 1)
```

ARM v7-M Critical Sections

```
<<arm7m static functions>>=
static inline void
mrtBeginCriticalSection(void)
{
    __disable_irq() ;
}
```

[1] CMSIS involves many different parts. Here we deal only with the core aspects of the CMSIS standard.

```
<<arm7m static functions>>=
static inline void
mrtEndCriticalSection(void)
{
    __enable_irq() ;
}
```

ARM v7-M Async Execution Interface

When posting a sync function, we must make sure no interrupt service is executed and so put a critical section around the call to `mrtSyncQueuePut`

```
<<arm7m external functions>>=
MRT_SyncParams *
mrt_SyncRequest(
    MRT_SyncFunc f)
{
    MRT_SyncParams *params = mrtSyncQueuePut(f) ;
    if (params == NULL) {
        mrtFatalError(mrtSyncOverflow) ;
    }

    return params ;
}
```

The alternate, non-fatal interface is also easily implemented.

```
<<arm7m external functions>>=
MRT_SyncParams *
mrt_TrySyncRequest(
    MRT_SyncFunc f)
{
    MRT_SyncParams *params = mrtSyncQueuePut(f) ;

    return params ;
}
```

ARM v7-M Initialization

```
<<arm7m external functions>>=
void
mrt_Initialize(void)
{
    mrtPlatformInit() ;
    mrtECBPoolInit() ;
    mrtTransactionsInit() ;
    mrtInitSysTimer() ;
#       if !(defined(MRT_NO_TRACE) || defined(MRT_NO_STDIO))
    mrt_RegisterTraceHandler(mrtPrintTraceInfo) ;
#       endif /* !defined(MRT_NO_TRACE) && !defined(MRT_NO_STDIO) */
}
```

EFM32GG Includes

```
<<efm32gg includes>>=

#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_emu.h"
#include "em_rtc.h"
#include "em_rmu.h"
#include "em_burtc.h"

#ifdef MRT_SWO_OUTPUT
#   include "bsp_trace.h"
#endif /* MRT_SWO_OUTPUT */
```

EFM32GG Timing Interfaces

```
<<efm32gg constants>>=
#define MRT_TIMER_FREQUENCY     1024
/*
 * for a timer frequency of 1024 ticks / sec,
 * max delay is 4,194,303 s == 69,905 min == 1,165 hr == 48.5 days
 */
#define MRT_MAX_MSEC_DELAY\
    ((MRT_MAX_CLOCK_TICKS / (uint32_t)(MRT_TIMER_FREQUENCY)) * (uint32_t)(1000))
```

```
<<efm32gg static functions>>=
static inline MRT_DelayTime
mrtMsecToTicks(
    MRT_DelayTime msec)
{
    if (msec > MRT_MAX_MSEC_DELAY) {
        msec = MRT_MAX_MSEC_DELAY ;
    }
    /*
     * Use 64 bit arithmetic to avoid overflow.
     */
    return ((uint64_t)msec * (uint64_t)(MRT_TIMER_FREQUENCY) +
            (uint64_t)(1000 / 2)) / (uint64_t)(1000) ;
}
```

```
<<efm32gg static functions>>=
static inline MRT_DelayTime
mrtTicksToMsec(
    MRT_DelayTime ticks)
{
    return ((uint64_t)ticks * UINT64_C(1000) +
            UINT64_C(MRT_TIMER_FREQUENCY / 2)) /
            UINT64_C(MRT_TIMER_FREQUENCY) ;
}
```

```
<<efm32gg static functions>>=
static void
mrtInitSysTimer(void)
{
    RTC_Reset() ;
```

```
    /*
     * The Low Frequency Crystal is driving the RTC.
     * It's frequency is 32,768 Hz.
     * Set RTC prescaler to divide by 32 to get a 1024 Hz timer frequency.
     */
    CMU_ClockDivSet(cmuClock_RTC, cmuClkDiv_32) ;
    /*
     * Enable Low Freq A Clock to run the RTC.
     */
    CMU_ClockEnable(cmuClock_RTC, true) ;
    RTC_Init_TypeDef init = RTC_INIT_DEFAULT ;
    init.enable = false ;
    RTC_Init(&init) ;
    /*
     * Enable interrupt at the NVIC.
     */
    NVIC_ClearPendingIRQ(RTC_IRQn) ;
    NVIC_EnableIRQ(RTC_IRQn) ;
}
```

```
<<efm32gg static functions>>=
static void
mrtSysTimerMask(void)
{
    RTC_IntDisable(RTC_IFC_COMP0) ;
}
```

```
<<efm32gg static functions>>=
static void
mrtSysTimerUnmask(void)
{
    RTC_IntEnable(RTC_IFC_COMP0) ;
}
```

```
<<efm32gg static functions>>=
static void
mrtSysTimerStart(
    MRT_DelayTime time)
{
    RTC_CompareSet(0, time) ;
    RTC_IntClear(RTC_IFC_COMP0) ;
    mrtSysTimerUnmask() ;
    RTC_Enable(true) ;
}
```

```
<<efm32gg static functions>>=
static MRT_DelayTime
mrtSysTimerStop(void)
{
    /*
     * There is a race between when we read the CNT register and when we
     * actually get the timer stopped where it may expire. If it expires, the
     * interrupt flag will be set and we can use that as an indication that the
     * CNT register is not valid (i.e. it was cleared to zero when the timer
     * expired).  This is different than if we had just been enabled and
     * requested to stop before one tick went by (in our case that's ~ 244
     * microsecond) and count would have been zero. So this is a bit tricky in
     * here.
     */
    MRT_DelayTime remain ;
```

```
    mrtSysTimerMask() ;
    if (RTC->CTRL & RTC_CTRL_EN) {
        remain = RTC->COMP0 ;
        remain -= RTC->CNT ;
        /*
         * Check that the timer didn't expire, as is indicated by the interrupt
         * flag, while we were trying to read the registers.
         */
        if (RTC->IF & RTC_IF_COMP0) {
            remain = 0 ;
        }
    } else {
        remain = 0 ;
    }
    RTC_Enable(false) ; // timer disabled

    return remain ;
}
```

```
<<efm32gg external functions>>=
void
RTC_IRQHandler(void)
{
    RTC_Enable(false) ; // timer disabled
    RTC_IntClear(RTC_IFC_COMP0) ;
    MRT_DelayTime nextTime = mrt_TimerExpireService() ;
    if (nextTime != 0) {
        mrtSysTimerStart(nextTime) ;
    }
}
```

EFM32GG Initialization

```
<<efm32gg static functions>>=
static void
mrtPlatformInit(void)
{
    /*
     * Workaround for chip errata.
     */
    CHIP_Init() ;
    /*
     * Set stack align, so ISR's are truly ordinary "C" functions.
     */
    SCB->CCR |= SCB_CCR_STKALIGN_Msk ;
    /*
     * Update the global notion of the system clock frequency.  The device
     * drivers need this to calculate various dividers properly.
     */
    SystemCoreClockUpdate() ;
    /*
     * High frequency peripheral clock.
     */
    CMU_ClockDivSet(cmuClock_HFPER, cmuClkDiv_1) ;
    CMU_ClockEnable(cmuClock_HFPER, true) ;
    /*
     * Enable the LF Crystal Oscillator.
     */
```

```
    CMU_OscillatorEnable(cmuOsc_LFXO, true, true) ;
    /*
     * Enable the Ultra-Low Frequency Oscillator for use by the Backup
     * Real Time clock.
     */
    CMU_OscillatorEnable(cmuOsc_ULFRCO, true, true) ;
    /*
     * Enable the low energy clock bus.
     */
    CMU_ClockEnable(cmuClock_CORELE, true) ;
    /*
     * Select the external crystal oscillator for the low frequency clocks.
     */
    CMU_ClockSelectSet(cmuClock_LFA, cmuSelect_LFXO) ;
    CMU_ClockSelectSet(cmuClock_LFB, cmuSelect_LFXO) ;
    /*
     * Enable the Backup Realtime Clock as a timestamp.
     * N.B. we must enable the Backup Unit itself before the clock
     * is programmable.
     */
     RMU_ResetControl(rmuResetBU, rmuResetModeClear) ;
     BURTC_Init_TypeDef buinit = BURTC_INIT_DEFAULT ;
     buinit.clkDiv = burtcClkDiv_2 ;
     /*
      * set to 1 kHz frequency, default is 2.
      * This will give us a 1 msec free running clock.
      */
     BURTC_Init(&buinit) ;

#       ifdef MRT_SWO_OUTPUT
    /*
     * Retarget instrumentation I/O to the SWO output so it will
     * show up in the debugger output console.
     */
    BSP_TraceSwoSetup() ;
#       endif /* MRT_SWO_OUTPUT */
}
```

EFM32GG Suspending Execution

```
<<efm32gg static functions>>=
static void
mrtWait(void)
{
    mrtBeginCriticalSection() ;

    if (mrtEventQueueEmpty(&tocEventQueue)) {       // ❶
        EMU_EnterEM2(true) ;
    }

    mrtEndCriticalSection() ;
}
```

❶ By the time `mrtWait` is invoked, we know that the both event queues had gone empty. However, between the time the test for empty event queues is made and we reach here, an interrupt could have gone off. So to make sure there is no pending work to be done, *i.e.* to make sure we didn't loose the race with the interrupt, we must test if there are any queued TOC events.

EFM32GG Tracing

```
<<efm32gg trace static functions>>=
static char const *
mrtTimestamp(void)
{
    static char tsbuf[16] ;

    uint32_t ts = BURTC_CounterGet() ;

    snprintf(tsbuf, sizeof(tsbuf), "%03lu.%03lu", ts / 1000, ts % 1000) ; // ❶
    return tsbuf ;
}
```

❶ Recall that the BURTC was set up to be a free running 1 msec counter. The timestamp is split into a seconds portion and a milliseconds portion.

EFM32GG Files

```
<<micca_rt_efm32gg.h>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

#ifndef MICCA_RT_H_
#define MICCA_RT_H_

<<common header file definitions>>

#endif /* MICCA_RT_H_ */
```

```
<<micca_rt_efm32gg.c>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

/*
 * Includes
 */
#include "micca_rt.h"
#include "micca_rt_internal.h"
<<efm32gg includes>>

/*
 * Constants
 */
<<mrt implementation constants>>
<<arm7m constants>>
<<efm32gg constants>>
```

```
/*
 * Data Types
 */
<<mrt implementation simple types>>
<<mrt implementation aggregate types>>

/*
 * Forward References
 */
<<mrt forward references>>

/*
 * Static Data
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static data>>
#   endif /* MRT_NO_TRACE */

<<mrt static data>>
<<efm32gg static data>>

/*
 * Static Inline Functions
 */
<<mrt implementation static inlines>>

/*
 * Static Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static functions>>
<<efm32gg trace static functions>>
#   endif /* MRT_NO_TRACE */

<<arm7m static functions>>
<<efm32gg static functions>>
<<mrt static functions>>

/*
 * External Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace external functions>>
<<efm32gg trace external functions>>
#   endif /* MRT_NO_TRACE */

<<arm7m external functions>>
<<efm32gg external functions>>
<<mrt external functions>>
```

MSP432 Includes

```
<<msp432 includes>>=
#define NO_MSP_CLASSIC_DEFINES

#include "msp.h"

#define TIMER_DIVIDER   256
```

```
extern uint32_t SystemCoreClock ;
```

MSP432 Timing Interfaces

```
<<msp432 includes>>=
#define MRT_MAX_MSEC_DELAY ((MRT_MAX_CLOCK_TICKS / mrtTimerFrequency) * 1000UL)

<<msp432 static data>>=
static uint32_t mrtTimerFrequency ;
```

```
<<msp432 static functions>>=
static inline MRT_DelayTime
mrtMsecToTicks(
    MRT_DelayTime msec)
{
    /*
     * for 3 MHz divided down by 256
     * max delay is 366,503,875 ms
     * == 3,665,038 s == 61083 min == 1,018 hr == 42 days
     */
    if (msec > MRT_MAX_MSEC_DELAY) {
        msec = MRT_MAX_MSEC_DELAY ;
    }
    /*
     * We must avoid overflow if the requested number of
     * msecs is large.
     */
    return msec < MRT_MAX_CLOCK_TICKS / mrtTimerFrequency ?
            ((msec * mrtTimerFrequency) + 1000UL / 2UL) / 1000UL :
            ((msec + 1000UL / 2UL) / 1000UL) * mrtTimerFrequency ;
}
```

```
<<msp432 static functions>>=
static inline MRT_DelayTime
mrtTicksToMsec(
    MRT_DelayTime ticks)
{
    return ticks < MRT_MAX_CLOCK_TICKS / 1000UL ?
        ((ticks * 1000UL) + mrtTimerFrequency / 2UL) / mrtTimerFrequency :
        ((ticks + mrtTimerFrequency / 2UL) / mrtTimerFrequency) * 1000UL ;
}
```

```
<<msp432 static functions>>=
static void
mrtInitSysTimer(void)
{
    /*
     * for 3 MHz, dividing down by 256 ==> 11718
     */
    mrtTimerFrequency = SystemCoreClock / (uint32_t)256 ;

    TIMER32_1->CONTROL =
            TIMER32_CONTROL_PRESCALE_2 |
            TIMER32_CONTROL_SIZE |
            TIMER32_CONTROL_ONESHOT ;
    TIMER32_1->INTCLR = 0 ;
    NVIC_ClearPendingIRQ(T32_INT1_IRQn) ;
```

```
    NVIC_EnableIRQ(T32_INT1_IRQn) ;
}
```

```
<<msp432 static functions>>=
static void
mrtSysTimerMask(void)
{
    BITBAND_PERI(TIMER32_1->CONTROL, TIMER32_CONTROL_IE_OFS) = 0 ;
}
```

```
<<msp432 static functions>>=
static void
mrtSysTimerUnmask(void)
{
    TIMER32_1->INTCLR = 0 ;
    BITBAND_PERI(TIMER32_1->CONTROL, TIMER32_CONTROL_IE_OFS) = 1 ;
}
```

```
<<msp432 static functions>>=
static void
mrtSysTimerStart(
    MRT_DelayTime time)
{
    TIMER32_1->LOAD = time ;
    mrtSysTimerUnmask() ;
    BITBAND_PERI(TIMER32_1->CONTROL, TIMER32_CONTROL_ENABLE_OFS) = 1 ;
}
```

```
<<msp432 static functions>>=
static MRT_DelayTime
mrtSysTimerStop(void)
{
    mrtSysTimerMask() ;
    BITBAND_PERI(TIMER32_1->CONTROL, TIMER32_CONTROL_ENABLE_OFS) = 0 ;
    return TIMER32_1->VALUE ;
}
```

```
<<msp432 external functions>>=
void
T32_INT1_IRQHandler(void)
{
    TIMER32_1->INTCLR = 0 ;
    MRT_DelayTime nextTime = mrt_TimerExpireService() ;
    if (nextTime != 0) {
        mrtSysTimerStart(nextTime) ;
    } else {
        mrtSysTimerStop() ;
    }
}
```

MSP432 Initialization

```
<<msp432 static functions>>=
static void
mrtPlatformInit(void)
{
}
```

MSP432 Suspending Execution

```
<<msp432 static functions>>=
static void
mrtWait(void)
{
    mrtBeginCriticalSection() ;

    if (mrtEventQueueEmpty(&tocEventQueue)) {
        __DSB() ; // ❶
        __WFI() ;
    }

    mrtEndCriticalSection() ;
}
```

❶ ARM recommends a data synchronization barrier instruction before going to sleep to insure any writes have been fully synced to memory before going to sleep and having clocks and other things turned off. We don't need it in this pariticular case since no write operation has been performed. However, we will follow the recommendation as it is only a single instruction and we don't want some change in the future to create a mysterious problem.

MSP432 Tracing

```
<<msp432 trace static functions>>=
static char const *
mrtTimestamp(void)
{
    return "0" ;
}
```

MSP432 Files

```
<<micca_rt_msp432.h>>=
/*
<<copyright info>>

Micca version:
<<version info>>
*/

#ifndef MICCA_RT_H_
#define MICCA_RT_H_

<<common header file definitions>>

#endif /* MICCA_RT_H_ */
```

```
<<micca_rt_msp432.c>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
```

```
*/

/*
 * Includes
 */
#include "micca_rt.h"
#include "micca_rt_internal.h"
<<msp432 includes>>

/*
 * Constants
 */
<<mrt implementation constants>>
<<arm7m constants>>

/*
 * Data Types
 */
<<mrt implementation simple types>>
<<mrt implementation aggregate types>>

/*
 * Forward References
 */
<<mrt forward references>>

/*
 * Static Data
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static data>>
#   endif /* MRT_NO_TRACE */

<<mrt static data>>
<<msp432 static data>>

/*
 * Static Inline Functions
 */
<<mrt implementation static inlines>>

/*
 * Static Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static functions>>
<<msp432 trace static functions>>
#   endif /* MRT_NO_TRACE */

<<arm7m static functions>>
<<msp432 static functions>>
<<mrt static functions>>

/*
 * External Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace external functions>>
#   endif /* MRT_NO_TRACE */

<<arm7m external functions>>
<<msp432 external functions>>
```

```
<<mrt external functions>>
```

Chapter 34

MSP430 Specific Code

```
<<msp430 includes>>=
#include "msp430.h"

#ifndef MRT_TIMER_FREQUENCY
#   define MRT_TIMER_FREQUENCY     4096UL
#endif /* MRT_TIMER_FREQUENCY */
#define MRT_MAX_CLOCK_TICKS     (UINT32_MAX - 1)
#define MRT_MAX_MSEC_DELAY\
        ((MRT_MAX_CLOCK_TICKS / MRT_TIMER_FREQUENCY) * 1000UL)
```

MSP430 Critical Sections

```
<<msp430 static functions>>=
static inline void
mrtBeginCriticalSection(void)
{
    __disable_interrupt() ;
}
static inline void
mrtEndCriticalSection(void)
{
    __enable_interrupt() ;
}
```

MSP430 Timing Interfaces

```
<<msp430 static functions>>=
static inline MRT_DelayTime
mrtMsecToTicks(
    MRT_DelayTime msec)
{
    if (msec > MRT_MAX_MSEC_DELAY) {
        msec = MRT_MAX_MSEC_DELAY ;
    }
    /*
     * We must avoid overflow if the requested number of
     * msecs is large.
     */
```

```
    return msec < MRT_MAX_CLOCK_TICKS / MRT_TIMER_FREQUENCY ?
            ((msec * MRT_TIMER_FREQUENCY) + 1000UL / 2UL) / 1000UL :
            ((msec + 1000UL / 2UL) / 1000UL) * MRT_TIMER_FREQUENCY ;
}
```

```
<<msp430 static functions>>=
static inline MRT_DelayTime
mrtTicksToMsec(
    MRT_DelayTime ticks)
{
    return ticks < MRT_MAX_CLOCK_TICKS / 1000UL ?
        ((ticks * 1000UL) + MRT_TIMER_FREQUENCY / 2UL) / MRT_TIMER_FREQUENCY :
        ((ticks + MRT_TIMER_FREQUENCY / 2UL) / MRT_TIMER_FREQUENCY) * 1000UL ;
}
```

```
<<msp430 static functions>>=
static void
mrtSysTimerMask(void)
{
    TA0CCTL0 &= ~CCIE ;
}
```

```
<<msp430 static functions>>=
static void
mrtSysTimerUnmask(void)
{
    TA0CCTL0 &= ~CCIFG ;
    TA0CCTL0 |= CCIE ;
}
```

```
<<msp430 static functions>>=
static inline uint16_t
readTimer(void)
{
#   define  SUCCESSIVE_MATCHES  3

    uint16_t cmp1Reg ;
    uint16_t cmp2Reg ;
    uint8_t cnt ;

    cnt = 1 ;
    cmp1Reg = TA0R ;
    do {
        cmp2Reg = TA0R ;
        cnt = cmp1Reg == cmp2Reg ? cnt + 1 : 1 ;
        cmp1Reg = cmp2Reg ;
    } while (cnt < SUCCESSIVE_MATCHES) ;

    return cmp2Reg ;

#   undef SUCCESSIVE_MATCHES
}
```

```
<<msp430 static functions>>=
static void
setNextTime(
    uint16_t clockTicks)
{
    _disable_interrupts() ;
```

```
    if (clockTicks == 1) {
        clockTicks++ ;
    }
    TA0CCR0 = readTimer() + clockTicks ;
    mrtSysTimerUnmask() ;

    _enable_interrupts() ;
}
```

```
<<msp430 static functions>>=
static void
mrtSysTimerStart(
    MRT_DelayTime time)
{
    clockCnts = time >> 16 ;
    setNextTime(time) ;
}
```

```
<<msp430 static data>>=
/*
 * We keep a higher precision time indication by counting Timer A overflows.
 * This variable holds the number of Timer A overflow interrupts that have
 * occurred. When concatenated with the value of the TA0R register from
 * Timer A and treated as a 32 bit quantity, the value represented is the
 * time since the system started as a U(20,12) fixed radix point number
 * in units of seconds (this is because we are running timer A off of the
 * 32 KHz clock divided down by 8 ==> 4096 ticks / sec). For 20 integer bits of
 * seconds, the clock time value rolls over after ~1,000,000 seconds or
 * about 12.1 days.
 *
 * Use the function, "sysTimeTicks()" to obtain the time value.
 */
static uint16_t timerTicks ;
static uint16_t clockCnts ;
```

```
<<msp430 static functions>>=
/*
 * To get a high resolution time value we combine the in memory counter that
 * counts timer overflows with the current value of the timer register (TA0R).
 * However we must be careful here because the counter is shared with an
 * interrupt service routine. So we make this function a "monitor" function and
 * we must check that the interrupt flag is not set right after we read the
 * timer value. If it is, then the memory counter will be off by one since
 * there is a pending interrupt that has not been serviced.
 */
uint32_t
sysTimeTicks(void)
{
    _disable_interrupts() ;

    uint16_t tvalue = readTimer() ;
    /*
     * If the timer rolls over between the time we enter this function and the
     * time we get here, then the interrupt flag will be set. In that case, the
     * memory counter will be off by one (since the interrupt service routine
     * will NOT have been run as we are in a "monitor" function).  Also we can
     * assume the timer value is simply zero, reflecting the timer value when
     * the roll over occurs (as it will not take a tick's worth of time to
     * execute to here.
     *
     * If there is no roll over of the timer, then the time is simply the
```

```
     * concatenation of the memory counter value and the timer value.
     */
    bool rollover = (TA0CTL & TAIFG) != 0 ;
    _enable_interrupts() ;

    return rollover ?
        (((uint32_t)timerTicks + 1) << 16) :
        (((uint32_t)timerTicks << 16) | tvalue) ;
}
```

```
<<msp430 static functions>>=
static MRT_DelayTime
mrtSysTimerStop(void)
{
    MRT_DelayTime remain ;

    mrtSysTimerMask() ;
    uint16_t timer = readTimer() ;
    /*
     * Account that the timer might go off between when we
     * stop it and when we read it.
     */
    if (TA0CCTL0 & CCIFG) {
        remain = clockCnts ?
                (MRT_DelayTime)(clockCnts - 1) << 16 : 0 ;
    } else {
        remain = TA0CCR0 - timer ;
        remain += (MRT_DelayTime)clockCnts << 16 ;
    }
    return remain ;
}
```

```
<<msp430 static functions>>=
void
__attribute__((interrupt(TIMER0_A0_VECTOR)))
timerA0ISR(void)
{
    if (clockCnts == 0) {
        MRT_DelayTime next = mrt_TimerExpireService() ;
        if (next) {
            mrtSysTimerStart(next) ;
        }
        __low_power_mode_off_on_exit() ;
    } else {
        /*
         * Loading the compare register with the timer
         * value causes us to count an entire 16 bits worth
         * of ticks.
         */
        TA0CCR0 = TA0R ;
        --clockCnts ;
    }
}
```

```
<<msp430 static functions>>=
static void
mrtInitSysTimer(void)
{
    /*
     * Stop and Clear the timer.
     */
```

```
    TA0CTL = TACLR ;
    /*
     * Set up clock source to ACLK and Divide down by 8.
     */
    TA0CTL |= TASSEL_1 | ID_3 ;
    TA0CCTL0 = 0 ;
    TA0CCTL1 = 0 ;
    TA0CCTL2 = 0 ;
    /*
     * Start the timer in continuous mode.
     * Enable the timer overflow interrupt.
     */
    TA0CTL |= MC_2 | TAIE ;
    /*
     * Zero out the upper level counter that keeps track of ticks.
     */
    timerTicks = 0 ;
}
```

MSP430 Async Execution Interface

```
<<msp430 external functions>>=
MRT_SyncParams *
mrt_SyncRequest(
    MRT_SyncFunc f)
{
    MRT_SyncParams *params = mrtSyncQueuePut(f) ;
    if (params == NULL) {
        mrtFatalError(mrtSyncOverflow) ;
    }
}
```

```
<<msp430 external functions>>=
MRT_SyncParams *
mrt_TrySyncRequest(
    MRT_SyncFunc f)
{
    return mrtSyncQueuePut(f) ;
}
```

MSP430 Suspending Execution

```
<<msp430 static functions>>=
static void
mrtWait(void)
{
    mrtBeginCriticalSection() ;
    if (mrtSyncQueueEmpty()) {
        __bis_SR_register(LPM0_bits | GIE) ;
    } else {
        mrtEndCriticalSection() ;
    }
}
```

MSP430 Initialization

```
<<msp430 external functions>>=
void
mrt_Initialize(void)
{
    mrtECBPoolInit() ;
    mrtTransactionsInit() ;
    mrtInitSysTimer() ;
}
```

MSP430 Tracing

```
<<msp430 trace static functions>>=
static char const *
mrtTimestamp(void)
{
    return "0" ;
}
```

MSP430 Files

```
<<micca_rt_msp430.h>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

#ifndef MICCA_RT_H_
#define MICCA_RT_H_

<<common header file definitions>>

#endif /* MICCA_RT_H_ */
```

```
<<micca_rt_msp430.c>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

/*
 * Includes
 */
#include "micca_rt.h"
#include "micca_rt_internal.h"
<<msp430 includes>>

/*
```

```
 * Constants
 */
<<mrt implementation constants>>

/*
 * Data Types
 */
<<mrt implementation simple types>>
<<mrt implementation aggregate types>>

/*
 * Forward References
 */
<<mrt forward references>>
<<msp430 forward references>>

/*
 * Static Data
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static data>>
#   endif /* MRT_NO_TRACE */

<<mrt static data>>
<<msp430 static data>>

/*
 * Static Inline Functions
 */
<<mrt implementation static inlines>>

/*
 * Static Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static functions>>
<<msp430 trace static functions>>
#   endif /* MRT_NO_TRACE */

<<msp430 static functions>>
<<mrt static functions>>

/*
 * External Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace external functions>>
#   endif /* MRT_NO_TRACE */

<<msp430 external functions>>
<<mrt external functions>>
```

Chapter 35

Common Header File Definitions

In this section, the arrangement of literal program chucks that are common to all header files for the run-time is given.

```
<<common header file definitions>>=
/*
 * Includes
 */
<<mrt interface includes>>

/*
 * Constants
 */
<<mrt interface constants>>

/*
 * Preprocessor Defines
 */
#ifndef COUNTOF
#   define COUNTOF(a)   (sizeof(a) / sizeof(a[0]))
#endif /* COUNTOF */

/*
 * Data Types
 */
<<mrt interface simple types>>
<<mrt interface aggregate types>>

#   ifndef MRT_NO_TRACE
<<mrt interface trace aggregate types>>
#   endif /* MRT_NO_TRACE */

/*
 * External Functions
 */
<<mrt external interfaces>>
#   ifndef MRT_NO_TRACE
<<mrt trace external interfaces>>
#   endif /* MRT_NO_TRACE */
```

Chapter 36

Internal Header File

The run-time code uses an internal header file to separate declarations that should not be generally exposed. This header file, called `micca_rt_internal.h`, must be included in the generated domain code, but contains definitions that are not needed by bridge code.

```
<<mrt interface includes>>=
#include <stddef.h>
#include <stdbool.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdnoreturn.h>
#include <assert.h>
```

```
<<micca_rt_internal.h>>=
/*
<<edit warning>>
<<copyright info>>

Micca version:
<<version info>>
*/

#ifndef MICCA_RT_INTERNAL_H_
#define MICCA_RT_INTERNAL_H_

/*
 * Standard Includes
 */
#ifndef MRT_NO_STDIO
#   include <stdio.h>
#endif /* MRT_NO_STDIO */

#include <stdlib.h>
#include <string.h>

#if __STDC_VERSION__ >= 201112L         /* ❶ */
#   include <stdalign.h>
#   include <stdnoreturn.h>
#else
#   ifndef noreturn
#       ifdef __CC_ARM
#           define noreturn __declspec(noreturn)
#       else /* __CC_ARM */
#           define noreturn
#       endif /* __CC_ARM */
```

```
#   endif /* noreturn */
#   ifndef alignas
#       define alignas(x)
#   endif /* alignas */
#endif /* __STDC_VERSION__ >= 201112L */

/*
 * Constants
 */
<<mrt internal constants>>

/*
 * Data Types
 */
<<mrt internal simple types>>
<<mrt internal aggregate types>>
#   ifndef MRT_NO_TRACE
<<mrt internal trace simple types>>
<<mrt internal trace aggregate types>>
#   endif /* MRT_NO_TRACE */

/*
 * External Functions
 */
<<mrt internal external interfaces>>

/*
 * Static Inline Functions
 */
<<mrt internal static inlines>>

#endif /* MICCA_RT_INTERNAL_H_ */
```

❶ Although we target the C11 standard syntax, there are really only a few C11 features that we use. So, for those stuck with a C99 compiler, we can use the preprocessor the remove the C11 dependencies. The most serious implication is over the `alignas` macro. We seek to have the most liberal alignment we can obtain, and if that is not the default by a C99 compiler, then the declaration of event parameters and sync function parameters could cause problems.

Part VI

Generating “C” Code

In the previous part of the book, we described the "C" code that forms the run time support for `micca` translated domains. In this part, we show how "C" code is generated to implement the domain.

Chapter 37

Introduction

This is the fourth and final major section of the micca translation scheme. We have already seen the platform specific model and how it captures the properties of a domain implementation. The micca domain specific language is used to populate the platform model. The micca run time provides functions to map model level actions onto the target platform.

The remaining component is to generate "C" code for a domain. Code generation consist of two major pieces:

- Generating initialized "C" variables that supply the data values required by the run time.
- Generating the "C" code for model level operations in state activities and domain operations. This facilitates interfacing to the run time code.

Micca does not compile an action language into the target "C" for state activities. Rather, it passes along the "C" code provided for the activities wrapped up as functions. However, to insulate the user supplied "C" code from the details of the run time function interfaces and the data structures used, micca performs an additional level of macro expansion on the state activities and operations code. Thus the supplied "C" code for activities contains embedded macros and these are expanded to support interfacing to the run time and performing other model level activities.

In the next section we discuss generating the data required by the run time. Afterward, we describe generating "C" code to help interface user supplied state activity code to the run time.

Generating a Domain

One of the domain operations provide by micca is a generate command. The usual workflow is to configure a domain, populate the domain and then generate the code files.

```
<<micca configuration>>=
operation generate {args} {
    return [@Gen@::miccaGenerate $args]
}
```

Since micca is a rosea based application, we will perform the code generation in a child namespace to prevent any possible name collisions. This follows the same pattern we used for the configuration command.

```
<<generation commands namespace>>=
namespace eval @Gen@ {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>

    <<generation helpers namespace>>
    <<generation support namespace>>
```

```
    <<generation header namespace>>
    <<generation code namespace>>

    namespace path {
        ::micca
        ::micca::@Config@::Helpers
        ::micca::@Gen@::Helpers
        ::rosea::InstCmds
    }

    <<generation data>>
    <<generation commands>>
}
```

Generating Run Time Data

`Micca` creates two "C" files for each domain. The first is a header file that contains interfacing information for the domain. The second is a code file that contains all the run time data and "C" code for the domain.

A "C" compiler requires a large amount of type information in a specific order. Generally, one is required to declare symbols before they are defined. The ordering has to be quite precise. One way to accomplish this is to use a template. The template is organized in the order required by the compiler and contains embedded commands. When expanded, the commands embedded in the template query the platform model and produce "C" code output.

Most languages have template expansion libraries and Tcl is no different. We will use the `textutil::expander` package from `tcllib`. We will also have use for creating nicely adjusted text blocks, typically as comments, and will be using `textutil::adjust` for that purpose.

```
<<required packages>>=
package require textutil::expander
package require textutil::adjust
```

Because we are using a template expansion technique to generate the code files, the implementation of code generation has the flavor of a report generator. Commands imbedded in the template make queries on the populated platform model and emit a *report* of the query, which in this case is a set of "C" language statements. Although the reports generated are intended to be consumed by a compiler rather than a human, the design of the generator is very similar to generating reports from a database. In this case the database is the platform model as populated by a configuration script and the reports are "C" code placed in files and intended to be processed later by a compiler.

The code generation is accomplished in four steps.

1. Parse command line arguments.
2. Verify domains are populated.
3. Set up template expansion.
4. Generate files.

```
<<generation commands>>=
proc miccaGenerate {arglist} {
    variable errcount 0

    <<miccaGenerate: parse command line arguments>>
    <<miccaGenerate: verify domains are populated>>
    <<miccaGenerate: set up template expansion>>
    <<miccaGenerate: generate files>>

    if {$errcount > 0} {
```

```
        tailcall DeclError GENERATE_ERRORS $errcount
    }
    return $genfiles
}
```

```
<<error code formats>>=
GENERATE_ERRORS     {%d code generation error(s)}
```

We set up options for the generation in a dictionary. The default values are given and can be overridden by those passed in.

```
<<miccaGenerate: parse command line arguments>>=
variable options
set options [dict create\
    expanderror     fail\
    stubexternalops false\
    lines           false\
]
set options [dict merge $options $arglist]
```

We will insist that each domain have exactly one population for it. This is accomplished by finding all the populations associated with the domains and looking for those where there is not exactly one population given for the domain.

```
<<miccaGenerate: verify domains are populated>>=
set badpops [pipe {
    Population findAll |
    findUnrelated ~ R100
}]

forAllRefs badpop $badpops {
    set domainName [readAttribute $badpop Name]
    log::error "for domain, \"$domainName\", no population is given"
    incr errcount
}

if {$errcount > 0} {
    tailcall DeclError GENERATE_ERRORS $errcount
}
```

The `expander` package creates a command that, when invoked with a template, performs the expansion. Here we set up that command and specify some options to it.

```
<<miccaGenerate: set up template expansion>>=
textutil::expander expand
expand setbrackets <% %>
expand errmode [dict get $options expanderror]
```

Generating the files happens by iterating over all the domains and generating a header file and a code file for each one.

```
<<miccaGenerate: generate files>>=
set genfiles [list]
try {
    forAllRefs domainRef [Domain findAll] {
        assignAttribute $domainRef {Name domain} {Interface interface}\
            {Prologue prologue} {Epilogue epilogue}

        GatherClassProperties $domain

        lappend genfiles $domain.h
        <<miccaGenerate: generate header file>>
        lappend genfiles $domain.c
        <<miccaGenerate: generate code file>>
```

```
    }
} finally {
    rename expand {} ; # ❶
}
```

❶ The template expander creates a new command and we need to delete it so that there won't be any conflict the next time `micccaGenerate` is invoked.

To generate the header file, we set up some variable in the namespace where the template expansion occurs. These variables define the domain context for the header generation. We perform the expansion in the `GenHeader` namespace so that the command embedded in the template resolve without qualification and we don't have any name conflicts between those commands and other code generation commands.

```
<<miccaGenerate: generate header file>>=
set GenHeader::domain $domain
set GenHeader::interface $interface
set GenSupport::domain $domain

expand evalcmd "namespace eval [namespace current]::GenHeader"
set hchan [::open $domain.h w]
try {
    variable headerTemplate
    puts $hchan [expand expand $headerTemplate]
} on error {result opts} {
    # puts $::errorInfo
    return -options $opts $result
} finally {
    chan close $hchan
}
```

Generating the code files is done in a similar manner. In this case, we perform the generation in the `GenCode` namespace.

```
<<miccaGenerate: generate code file>>=
set GenCode::domain $domain
set GenCode::prologue $prologue
set GenCode::epilogue $epilogue

CreateActivityCommands $domain

expand evalcmd "namespace eval [namespace current]::GenCode"
set cchan [::open $domain.c w]
try {
    variable codeTemplate
    puts $cchan [expand expand $codeTemplate]
} on error {result opts} {
    #puts $::errorInfo
    return -options $opts $result
} finally {
    chan close $cchan
    namespace delete GenActivity
}
```

Common Class Queries

There are a number of queries about the properties of the classes in a domain that are common and frequent enough to warrant factoring the queries into a more convenient form. We store these class properties in a relation value that is held in an ordinary namespace variable. The required query is quite complex. We intend to hold everything in a single relation value and this will cover both assigners and classes, the two being very similar in this context.

```
<<generation helper commands>>=
proc GatherClassProperties {domain} {
    variable ClassProperties

    <<GatherClassProperties: gather class properties>>
    <<GatherClassProperties: gather assigner properties>>

    return
}
```

N.B. that we calculate the properties of ordinary classes first. This is important because our calculations for multiple assigners depends upon knowledge that we gather about ordinary classes.

For classes, we, as usual, must distinguish between union-based subclass and all other classes. Because union subclasses are stored in the same memory allocated to the superclass, we need to set the allocation for them differently than for non-union based classes. For non-union classes, the population specifies exactly the required storage. For union subclasses, we must use the population characteristics of the superclass.

```
<<GatherClassProperties: gather class properties>>=
<<GatherClassProperties: non-union subclass properties>>
set ClassProperties $classes

<<GatherClassProperties: union subclass properties>>
set ClassProperties [relation union $ClassProperties $subclasses]
```

Also note here, that we are storing the calculated properties into a variable as they are determined. Again this lets us use past results for the next phases.

Like other queries of this type, the strategy is to the `rvajoin` to deal with information that might be missing. In this case, we need to deal with the face that there may not be any initial instances defined for a class. Since one property we are interested in is the total number of slots to allocate in the class storage pool, we want to be able to treat an empty instance population as requiring zero additional storage slots. The `rvajoin` allows us to do that, where a normal join would delete the class from the result.

```
<<GatherClassProperties: non-union subclass properties>>=
set classRefs [FindNonUnionSubclasses $domain] ;                    # ❶
set cpopRefs [findRelated $classRefs R104 {~R101 ElementPopulation}\
        {~R105 ClassPopulation}]

set allocs [pipe {
    deRef $cpopRefs |
    relation project % Class Allocation |
    relation rename % Class Name
} {} |%]

set insts [pipe {
    findRelated $cpopRefs ~R102 |
    deRef % |
    relation project % Class Number |
    relation rename % Class Name
} {} |%]

set classes [pipe {
    deRef $classRefs |
    relation project ~ Name Number |
    relation rename ~ Number ClassNumber |
    relation extend ~ stup\
        Declaration string {"struct [tuple extract $stup Name]"}\
        Reference string {"struct [tuple extract $stup Name] *"}\
        ConstReference string {"struct [tuple extract $stup Name] const *"}\
        StorageVariable string {"[tuple extract $stup Name]__POOL"}\
```

```
        ClassVariable string {"${domain}__CLASSES"} |
    relation join ~ $allocs |
    ralutil::rvajoin ~ $insts Instances |
    relation extend ~ itup\
        InitialInstance int {
            [relation cardinality [tuple extract $itup Instances]]} \
        TotalInstance int {
            [relation cardinality [tuple extract $itup Instances]] +
            [tuple extract $itup Allocation]} |
    relation eliminate ~ Instances |
    relation rename ~ ClassNumber Number
}]
# puts [relformat $classes classes]
```

❶ This finds all classes which are **not** a subclass of a union-based generalization.

For classes which **are** a subclass of a union-based generalization, we want the total allocation for the subclass to be the same as the ultimate superclass of the generalization. We say, ultimate superclass, since generalizations may extend multiple levels and it is the subclass at the "top" which determine the storage requirements.

We start the query by finding the set of ultimate superclasses for the union generalizations. A union superclass is an ultimate superclass if it does **not** also serve a role as a union subclass. Then we can compute the transitive closure of the union subclasses over all union superclasses. The result of the transitive closure can then be joined against the ultimate superclasses to give us the name of the class that governs the total allocation for the union-based generalization.

```
<<GatherClassProperties: union subclass properties>>=
set usubRefs [UnionSubclass findWhere {$Domain eq $domain}]
set usubs [pipe {
    deRef $usubRefs |
    relation eliminate ~ Domain Role
}]
# puts [relformat $usubs "Union Subclasses"]

set usupers [pipe {
    UnionSuperclass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation eliminate ~ Domain Role |
    relation rename ~ Class Superclass
}]
# puts [relformat $usupers "Union Superclasses"]

# The use of semiminus here gives the set of union-bases superclasses
# which are *not* also used as subclasses.
set ultimate_supers [pipe {
    relation semiminus $usubs $usupers -using {Class Superclass} |
    relation join ~ $classes -using {Superclass Name} |
    relation project ~ Superclass TotalInstance
}]
# puts [relformat $ultimate_supers "Ultimate Union Superclasses"]

# We need this relation to recover the class number after the
# transitive closure. TCLRal insists (as a probable mis-feature)
# that a transitive close be performed on a binary relation.
set domclasses [pipe {
    Class findWhere {$Domain eq $domain} |
    deRef ~ |
    relation eliminate ~ Domain
}]

set subinsts [pipe {
    findRelated $usubRefs R47 R40 R41 R104 {~R101 ElementPopulation}\
```

```
        {~R105 ClassPopulation} ~R102 |
    deRef % |
    relation project % Class Number |
    relation rename % Class Name
} {} |%]
# puts [relformat $subinsts "Union Subclass instances"]

set subclasses [pipe {
    relation join $usubs $usupers\
            -using {Relationship Relationship} |
    relation project ~ Class Superclass |
    relation tclose ~ |
    relation join ~ $ultimate_supers |
    relation eliminate ~ Superclass |
    relation join $domclasses ~ -using {Name Class} |
    relation extend ~ stup\
        Declaration string {"struct [tuple extract $stup Name]"}\
        Reference string {"struct [tuple extract $stup Name] *"}\
        ConstReference string {"struct [tuple extract $stup Name] const *"}\
        StorageVariable string {""}\
        ClassVariable string {"${domain}__CLASSES"} |
    relation rename ~ Number ClassNumber |
    ralutil::rvajoin ~ $subinsts Instances |
    relation extend ~ itup\
        Allocation int {"0"}\
        InitialInstance int {
            [relation cardinality [tuple extract $itup Instances]]} |
    relation eliminate ~ Instances |
    relation rename ~ ClassNumber Number
}]
# puts [relformat $subclasses "Union Subclass"]
```

The query for the assigners can be grouped as common items that apply to both single and multi assigners and those properties specifice to each type.

```
<<GatherClassProperties: gather assigner properties>>=
<<GatherClassProperties: common assigner properties>>

<<GatherClassProperties: single assigner properties>>
set ClassProperties [relation union $ClassProperties $sassigners]

<<GatherClassProperties: multi assigner properties>>
set ClassProperties [relation union $ClassProperties $massigners]
```

The common properties query starts at the **AssignerStateModel** and uses `rvajoin` to obtain the information in **R53** as relation valued attributes. This is an easy technique for discriminating a generalization. Note the assigners are tagged by ascending value of the Association name. This number is used as part of assigning assigner ID's for external use.

```
<<GatherClassProperties: common assigner properties>>=
set assigners [pipe {
    AssignerStateModel findWhere {$Domain eq $domain} |
    deRef ~ |
    ralutil::rvajoin ~ $::micca::SingleAssigner Single |
    ralutil::rvajoin ~ $::micca::MultipleAssigner Multiple |
    relation tag ~ Number -ascending Association |
    relation extend ~ stup\
        Declaration string {"struct [tuple extract $stup Association]"}\
        Reference string {"struct [tuple extract $stup Association] *"}\
        ConstReference string\
                {"struct [tuple extract $stup Association] const *"}\
        StorageVariable string {"[tuple extract $stup Association]__POOL"}\
        ClassVariable string {"${domain}__ASSIGNERS"}
```

```
}]
# puts "assigners =\n$assigners"
```

Single assigners have a single instance defined automatically.

```
<<GatherClassProperties: single assigner properties>>=
set sassigners [pipe {
    relation restrictwith $assigners {[relation isnotempty $Single]} |
    relation eliminate ~ Domain Single Multiple |
    relation rename ~ Association Name |
    relation extend ~ stup\
        Allocation int {0}\
        InitialInstance int {1}\
        TotalInstance int {1}
}]
# puts [relformat $sassigners sassigners]
```

Multiple assigners can have population values specified during configuration creating instances which form the initial instance population for the assigner. The computation of the storage for multiple assigners is a bit different. The total number of multiple assigner instances is always the same number of instances as the class which partitions the multiple assigner. The `allocation` command is not supported for multiple assigners, so there is an implied value of the allocation which is the total number of instances minus those defined as part of the initial instance population. Here we compute all the fields just like a class or single assigner but with consideration of the rules for the differences in multiple assigners.

The computation of the total number of multiple assigner instances is complicated by a couple of considerations.

1. There may be no initial population of the multi assigner and we must deduce that the total number of instances is just that of the partitioning class.
2. If the multiple assigner is partitioned by a union subclass, then it is possible that the union subclass has no initial instances and has no allocation defined for it in the configuration. This is allowed for union subclasses since their storage is tied to the ultimate superclass of the generalization (indeed there is a warning issued of there is an `allocation` for a union subclass in the population).
3. Even if there is an initial instance population for the multiple assigner instances, computing the total instance allocation must account for the previous consideration.

The easiest implementation technique is to use the class property information we have already gathered. In particular, we have already calculated the total allocation information for ordinary classes, including union subclasses. So the strategy here is to compute the `TotalInstance` information knowing the partitioning class, find any initial multiple assigner instances and subtract that count from the total to compute what would have been the required allocation count.

```
<<GatherClassProperties: multi assigner properties>>=
set partClassPop [pipe {
    MultipleAssigner findWhere {$Domain eq $domain} |
    findRelated ~ R54 |
    deRef ~ |
    relation eliminate ~ Number |
    relation rename ~ Name Class |
    relation extend ~ itup TotalInstance int {
            [GetClassProperty [tuple extract $itup Class] TotalInstance]}
}]
# puts [relformat $partClassPop partClassPop]

# Then we can find the number of intial instances of the multiple assigner
# and deduce what the "implied" allocation for non-initial instances
# would have been.
set massigners [pipe {
    relation restrictwith $assigners {[relation isnotempty $Multiple]} |
    relation rename ~ Number AssignerNumber |
    ralutil::rvajoin ~ $::micca::MultipleAssignerInstance Instances |
```

```
    relation ungroup ~ Multiple |
    relation join ~ $partClassPop |
    relation extend ~ stup\
        InitialInstance int\
            {[relation cardinality [tuple extract $stup Instances]]} |
    relation extend ~ atup\
        Allocation int {[tuple extract $atup TotalInstance] -\
            [tuple extract $atup InitialInstance]} |
    relation eliminate ~ Domain Single Class Instances |
    relation rename ~ Association Name AssignerNumber Number
}]
# puts [relformat $massigners massigners]
```

Having computed all the properties, we need some procedures to access them.

```
<<generation helper commands>>=
proc GetClassProperty {class prop} {
    variable ClassProperties
    set cprop [relation restrictwith $ClassProperties {$Name eq $class}]
    if {[relation isnotempty $cprop]} {
        return [relation extract $cprop $prop]
    }

    error "unknown property, \"$prop\", for class, \"$class\""
}
```

The class descriptor array element for a given class is commonly needed.

```
<<generation helper commands>>=
proc GetClassDescriptor {domain className} {
    return [string cat\
        [GetClassProperty $className ClassVariable]\
        \[ [GetClassProperty $className Number] \]

    ]
}
```

Chapter 38

Header Files

The header file generated for a domain contains interfacing information. The header file is included in the generated "C" code file and, typically, is included by bridge code that is mapping the dependencies of one domain onto another.

Following our usual pattern, we will execute the header generation commands from the template in a child namespace to avoid any name conflicts.

The `GenHeader` namespace is a child of `::micca::@Gen@` and follows our usual pattern of imports, command path and contents.

```
<<generation header namespace>>=
namespace eval GenHeader {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>

    namespace path {
        ::micca
        ::micca::@Gen@::Helpers
        ::micca::@Config@::Helpers
        ::rosea::InstCmds
    }

    <<generation header data>>
    <<generation header commands>>
}
```

The template for the header file is shown below. Each embedded command is then shown in the following sections.

```
<<generation data>>=
set headerTemplate [textutil::adjust::undent {
    <%banner%>
    #ifndef <%headerFileGuard%>
    #define <%headerFileGuard%>
    <%interface%>
    #include "micca_rt.h"
    <%interfaceTypeAliases%>
    <%domainOpDeclarations%>
    <%externalOpDeclarations%>
    <%eventParamDeclarations%>
    <%portalIds%>
    <%portalDeclaration%>
    #endif /* <%headerFileGuard%> */
}]
```

The generation of the header files is accomplished by iterating over all the domains and placing the output of the template expansion into a file. We use some variables to provide context to the embedded expansion commands.

The commands embedded in the template all follow a similar pattern. The return values of the commands are placed in the output of the template expansion.

Header File Guard

The generated header file includes definitions of preprocessor symbols to prevent the header file from being included multiple times.

```
<<generation header commands>>=
proc headerFileGuard {} {
    variable domain
    return [string toupper [string trim $domain]]_H_
}
```

Domain Interface

Arbitrary code may be placed in the generated header file by invoking the `interface` command as part of the domain configuration. The text provided during configuration is simply passed into the generated header file.

```
<<generation header commands>>=
proc interface {} {
    variable interface

    return [string cat\
        [comment "Domain Interface Contents"]\
        [textutil::adjust::undent [string trim $interface]]\
    ]
}
```

Type Aliases

The domain configuration can contain type aliases. Normally, the type alias information is placed in the generated code file. However, if a type alias name is used in a context where it would be exposed outside of the domain, *e.g.* as a parameter to a domain operation, then the type alias definition is placed in the generated header file. There are three circumstances where a type is needed external to the domain:

1. As the return type or parameter type of a domain operation.
2. As the return type or parameter type of an external operation.
3. As an argument to an event (since events can be signaled by the portal operations).

We will have need to compute the type aliases needed by the interface more than once, so we factor the code into a procedure.

```
<<generation helper commands>>=
proc FindInterfaceTypeAliases {domain} {
    set domRef [Domain findWhere {$Name eq $domain}]

    <<FindInterfaceTypeAliases:: find type alias names>>
    <<FindInterfaceTypeAliases:: find aliases in domain operations>>
    <<FindInterfaceTypeAliases:: find aliases in external operations>>
    <<FindInterfaceTypeAliases:: find aliases in events>>
    <<FindInterfaceTypeAliases:: union the combination>>
}
```

We compute a list of all the type alias names. This is used below in the queries for the various types of parameters types to determine if they are indeed one of the type aliases.

```
<<FindInterfaceTypeAliases:: find type alias names>>=
set aliases [pipe {
    findRelated $domRef ~R7 |
    deRef %
} {} |%]
# puts [relformat $aliases aliases]
```

Query the return types and parameters of the domain operations.

```
<<FindInterfaceTypeAliases:: find aliases in domain operations>>=
set doRefs [pipe {
    findRelated $domRef ~R5 |
    deRef % |
    relation semijoin % $aliases\
            -using {Domain Domain ReturnDataType TypeName}
} {} |%]
# puts [relformat $doRefs doRefs]
set dopRefs [pipe {
    findRelated $domRef ~R5 ~R6 |
    deRef % |
    relation semijoin % $aliases\
            -using {Domain Domain DataType TypeName}
} {} |%]
# puts [relformat $dopRefs dopRefs]
```

The same type of query is needed for the external operations.

```
<<FindInterfaceTypeAliases:: find aliases in external operations>>=
set eops [findRelated $domRef ~R1 {~R2 ExternalEntity} ~R10]
set eoRefs [relation semijoin [deRef $eops] $aliases\
            -using {Domain Domain ReturnDataType TypeName}]
# puts [relformat $eoRefs eoRefs]
set eopRefs [pipe {
    findRelated $eops ~R11 |
    deRef % |
    relation semijoin % $aliases\
            -using {Domain Domain DataType TypeName}
} {} |%]
# puts [relformat $eopRefs eopRefs]
```

And finally, we examine the event parameters.

```
<<FindInterfaceTypeAliases:: find aliases in events>>=
set argRefs [pipe {
    Argument findWhere {$Domain eq $domain} |
    deRef ~ |
    relation semijoin ~ $aliases\
            -using {Domain Domain DataType TypeName}
}]
# puts [relformat $argRefs argRefs]
```

The union of these sets is then the set of type aliases that will need to be made available outside of the domain.

```
<<FindInterfaceTypeAliases:: union the combination>>=
return [relation union $doRefs $dopRefs $eoRefs $eopRefs $argRefs]
```

The implementation of the `interfaceTypeAliases` iterates over the type aliases that are used in the domain interfaces and emits the corresponding "C" `typedef` statement.

```
<<generation header commands>>=
proc interfaceTypeAliases {} {
    variable domain
    append result [comment "Type Aliases"]
    relation foreach taRef [FindInterfaceTypeAliases $domain] {
        relation assign $taRef TypeName TypeDefinition
        append result "typedef "\
            [typeCheck composeDeclaration $TypeDefinition $TypeName] " ;\n"
    }

    return $result
}
```

Domain Operation Declarations

"C" language programs are composed of *declarations* and *definitions*. A declaration makes symbol names and types known to the compiler. A definition associates specific data or code statements to a symbol. Header files contain a lot of declarations. Here we start with declarations for the domain operations. Domain operations are just ordinary "C" functions but are given external scope.

```
<<generation header commands>>=
proc domainOpDeclarations {} {
    variable domain
    set result [comment "Domain Operations External Declarations"]

    set opRefs [DomainOperation findWhere {$Domain eq $domain}]
    set params [deRef [findRelated $opRefs ~R6]]
    set ops [pipe {
        deRef $opRefs |
        relation project ~ Domain Name ReturnDataType Comment |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $params Parameters
    }]
    # puts [relformat $ops ops]

    relation foreach op $ops {
        relation assign $op
        if {$Comment ne {}} {
            append result [comment $Comment]
        }
        if {[relation isempty $Parameters]} {
            set pdecl void
        } else {
            set plist [list]
            relation foreach op_param $Parameters -ascending Number {
                relation assign $op_param Name DataType
                lappend plist [typeCheck composeDeclaration $DataType $Name]
            }
            set pdecl [join $plist {, }]
        }

        append result "extern $ReturnDataType\
                ${Domain}_${Operation}\($pdecl\) ;\n" ; # ❶
    }

    return $result
}
```

❶ Note that we prepend the domain name to the operation name in order to avoid naming conflicts in the global namespace. Naming conventions are necessary when dealing with a language like “C” which does not have support for namespaces.

External Operation Declarations

The declarations for external operations follows the same pattern as for domain operations.

```
<<generation header commands>>=
proc externalOpDeclarations {} {
    variable domain
    set result [comment "External Operations Declarations"]

    set opRefs [ExternalOperation findWhere {$Domain eq $domain}]
    set params [deRef [findRelated $opRefs ~R11]]
    set ops [pipe {
        deRef $opRefs |
        relation project ~ Domain Entity Name ReturnDataType Comment |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $params Parameters
    }]
    # puts [relformat $ops ops]

    relation foreach op $ops {
        relation assign $op
        if {$Comment ne {}} {
            append result [comment [string trim $Comment \n]]
        }
        if {[relation isempty $Parameters]} {
            set pdecl void
        } else {
            set plist [list]
            relation foreach op_param $Parameters -ascending Number {
                relation assign $op_param Name DataType
                lappend plist [typeCheck composeDeclaration $DataType $Name]
            }
            set pdecl [join $plist {, }]
        }

        append result "extern $ReturnDataType\
                ${Domain}_${Entity}_${Operation}__EOP\($pdecl\) ;\n" ; # ❶
    }

    return $result
}
```

❶ Note that we prepend the domain name and the external entity name to the operation name in order to avoid naming conflicts in the global namespace.

Event Parameter Declarations

Since events can be signaled via the portal functions, we need to make the parameter signature of an event available in the header file. Typically, bridge code will need to supply any parameter an event may need and will need a structure declaration to do that.

```
<<generation header commands>>=
proc eventParamDeclarations {} {
    variable domain
```

```
    set result [comment "Event Parameter Structure Declarations"]
    set evtparams [pipe {
        relation restrictwith $::micca::Event {$Domain eq $domain} |
        relation join ~ $::micca::EventSignature $::micca::ParameterSignature\
                $::micca::Parameter $::micca::Argument |
        relation project ~ Model Event Name Position DataType |
        relation group ~ Params Name Position DataType |
        relation group ~ Events Event Params
    }]
    #puts [relformat $evtparams evtparams]
    relation foreach evtparam $evtparams -ascending Model {
        relation assign $evtparam
        relation foreach event $Events {
            relation assign $event
            set pstructname "struct ${domain}_${Model}_${Event}__EPARAMS"
            append result "$pstructname \{\n"
            relation foreach param $Params -ascending Position {
                relation assign $param DataType Name
                append result\
                    "    [typeCheck composeDeclaration $DataType $Name] ;\n"
            }
            append result "\} ;\n"
            append result\
                "#if __STDC_VERSION__ >= 201112L\n"\
                "static_assert(sizeof($pstructname) <= sizeof(MRT_EventParams), "\
                "\"Parameters for class or assigner, $Model, event,\
                    $Event, are too large\") ;\n"\
                "#endif /* __STDC_VERSION__ >= 201112L */\n" ; # ❶
        }
    }

    return $result
}
```

❶ The static assertion about the size of event parameters is an important protection to insure there are not memory overruns. However, we relent and ignore the problem when faced with a compiler that is less capable than C11.

Portal Function Constants

The run time code provides a set of portal functions that allow bridge code to tunnel simple model level operations, such a signaling an event, into a domain. The functions require integer constants that encode identifiers for the various entities that can be accessed via the portal. Those constants are placed in the generated header file.

We generate information on several aspects of the domain:

- Classes
- Instances
- Attributes
- Events
- States
- Assigners

```
<<generation header commands>>=
proc portalIds {} {
    variable domain

    <<portalIds:: generate class information>>
    <<portalIds:: generate assigner information>>
    <<portalIds:: generate event information>>
    <<portalIds:: generate state information>>

    return $result
}
```

For classes, we generate encodings for the class itself, its attributes and its initial instances.

```
<<portalIds:: generate class information>>=
append result [comment "Numeric encoding of classes, attributes and\
        instances used by the portal functions"]
set classRefs [Class findWhere {$Domain eq $domain}]
append result "#define [string toupper ${domain}_CLASSCOUNT]\
        [refMultiplicity $classRefs]\n"
forAllRefs classRef $classRefs {
    assignAttribute $classRef
    <<portalIds:: generate class encodings>>
    <<portalIds:: generate attribute encodings>>
    <<portalIds:: generate instance encodings>>
}
```

Each class was given a unique number within the domain when the platform model was populated. This is the basis for the class id encodings.

```
<<portalIds:: generate class encodings>>=
append result [comment "Class: $Name Attribute Encoding"]
set classid [string toupper ${Domain}_${Name}_CLASSID]
append result "#define $classid $Number\n"
```

Attributes are encoded in alphabetical order. Note that the portal does not allow access to relationships or the pointer values that are used to navigate a relationship.

```
<<portalIds:: generate attribute encodings>>=
set attrRefs [findRelated $classRef ~R20 {~R25 PopulatedComponent}\
        {~R21 Attribute}]
append result "#define [string toupper ${Domain}_${Name}_ATTRCOUNT]\
        [refMultiplicity $attrRefs]\n"
set attrNumber -1
relation foreach attr [deRef $attrRefs] -ascending Name {
    relation assign $attr {Name attrName}
    set attrid [string toupper ${Domain}_${Name}_${attrName}_ATTRID]
    append result "#define $attrid [incr attrNumber]\n"
}
```

We generate encodings for instances that are part of the initial instance population.

```
<<portalIds:: generate instance encodings>>=
set instRefs [findRelated $classRef ~R20 ~R103]
append result [comment "Class: $Name Instance Encoding"]
append result "#define [string toupper ${Domain}_${Name}_INSTCOUNT]\
        [GetClassProperty $Name TotalInstance]\n"
forAllRefs inst $instRefs {
    assignAttribute $inst Instance {Number InstNumber}
    set instid [string toupper ${Domain}_${Name}_${Instance}_INSTID]
    append result "#define $instid $InstNumber\n"
}
```

Assigners are treated much like classes.

```
<<portalIds:: generate assigner information>>=
set assigners [pipe {
    AssignerStateModel findWhere {$Domain eq $domain} |
    deRef ~ |
    relation tag ~ Number -ascending Association
}]
# puts [relformat ${::micca::@Gen@::Helpers::ClassProperties} ClassProperties]
relation foreach assigner $assigners {
    relation assign $assigner
    append result [comment "Assigner: $Association"]
    set prefix [string toupper ${Domain}_${Association}]
    append result\
        "#define ${prefix}_ASSIGNERID $Number\n"\
        "#define ${prefix}_INSTCOUNT\
            [GetClassProperty $Association TotalInstance]\n"
}
```

The encoding for event numbers is obtained straight from the platform model as they are numbered there. In this case we query the **Event** class.

```
<<portalIds:: generate event information>>=
set mevents [pipe {
    Event findWhere {$Domain eq $domain} |
    deRef ~ |
    relation eliminate ~ Domain |
    relation group ~ Events Event Number
}]
relation foreach mevent $mevents -ascending Model {
    relation assign $mevent
    append result [comment "Class: $Model Event Encoding"]
    append result "#define [string toupper ${domain}_${Model}_EVENTCOUNT]\
        [relation cardinality $Events]\n"
    relation foreach event $Events -ascending Number {
        relation assign $event
        append result "#define\
            [string toupper ${domain}_${Model}_${Event}_EVENT] $Number\n"
    }
}
```

The encoding for state numbers is also obtained straight from the platform model as they are numbered there. In this case we query the **State** class.

```
<<portalIds:: generate state information>>=
set mstates [pipe {
    StatePlace findWhere {$Domain eq $domain && $Name ne "@"} |
    deRef ~ |
    relation project ~ Model Name Number |
    relation group ~ States Name Number
}]
relation foreach mstate $mstates -ascending Model {
    relation assign $mstate
    append result [comment "Class: $Model State Encoding"]
    append result "#define [string toupper ${domain}_${Model}_STATECOUNT]\
        [relation cardinality $States]\n"
    relation foreach state $States -ascending Name {
        relation assign $state
        append result "#define\
            [string toupper ${domain}_${Model}_${Name}_STATE] $Number\n"
    }
}
```

Portal Data Structure Declaration

The portal functions require a data structure and we must emit the declaration of that data structure.

```
<<generation header commands>>=
proc portalDeclaration {} {
    variable domain
    append result\
        [comment "Domain Portal Declaration"]\
        "extern MRT_DomainPortal const ${domain}__PORTAL ;\n"
}
```

Chapter 39

Generating Code Files

In this section we describe the code used to generate the domain code file. We follow the pattern established for the header file. A namespace is used to prevent any naming conflicts for the template expansion commands.

```
<<generation code namespace>>=
namespace eval GenCode {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>

    namespace path {
        ::micca
        ::micca::@Gen@::Helpers
        ::micca::@Config@::Helpers
        ::rosea::InstCmds
    }

    namespace import [namespace parent]::expand
    namespace import [namespace parent]::GenSupport::ExpandActivity

    <<generation code data>>
    <<generation code commands>>
}
```

The template for code generation is much more complicated. In the code file we will find it necessary to emit many forward declaration in addition to the required definitions. For example, it is necessary to emit forward declarations for all instance operations since we do not know in which activity one of them might be invoked.

```
<<generation data>>=
set codeTemplate [textutil::adjust::undent {
    <%banner%>
    #include "micca_rt.h"
    #include "micca_rt_internal.h"
    <%domainInclude%>
    <%prologueDeclarations%>

    #ifndef MRT_NO_NAMES
        #define MRT_DOMAIN_NAME <%domainNameString%>
    #endif /* MRT_NO_NAMES */

    #if (defined(MRT_INSTRUMENT) && defined(MRT_NO_NAMES))
        #error "cannot define both MRT_INSTRUMENT and MRT_NO_NAMES"
    #endif /* (defined(MRT_INSTRUMENT) && defined(MRT_NO_NAMES)) */

    #ifdef MRT_INSTRUMENT
```

```
    static char const mrtDomainName[] = MRT_DOMAIN_NAME ;
        #ifdef BOSAL
            #include "bosal.h"
            #ifndef MRT_INSTRUMENT_ENTRY
                #define MRT_INSTRUMENT_ENTRY\
                    bsl_Printf("%s: %s: %s %d\n", mrtDomainName,\
                            __func__, __FILE__, __LINE__) ;
            #endif /* MRT_INSTRUMENT_ENTRY */
            #ifndef MRT_DEBUG
                #define MRT_DEBUG(...) bsl_Printf(__VA_ARGS__)
            #endif /* MRT_DEBUG */
        #endif /* BOSAL */
        #ifndef MRT_INSTRUMENT_ENTRY
            #define MRT_INSTRUMENT_ENTRY\
                printf("%s: %s: %s %d\n", mrtDomainName, __func__, __FILE__,\
                        __LINE__) ;
        #endif /* MRT_INSTRUMENT_ENTRY */
        #ifndef MRT_DEBUG
            #define MRT_DEBUG(...) printf(__VA_ARGS__)
        #endif /* MRT_DEBUG */
    #else
        #define MRT_INSTRUMENT_ENTRY
        #define MRT_DEBUG(...)
    #endif /* MRT_INSTRUMENT */
    <%implementationTypeAliases%>
    <%forwardClassDeclarations%>
    <%forwardRelationshipDeclaration%>
    <%classDeclarations%>
    <%assignerDeclarations%>
    <%stateParamDeclarations%>
    <%operationDeclarations%>
    <%ctorDeclarations%>
    <%dtorDeclarations%>
    <%formulaDeclarations%>
    <%activityDeclarations%>
    <%storageDeclarations%>
    <%nameDefinitions%>
    <%iabDefinitions%>
    <%edbDefinitions%>
    <%pdbDefinitions%>
    <%classDefinitions%>
    <%assignerDefinitions%>
    <%relationshipDefinitions%>
    <%classInstanceDefinitions%>
    <%assignerInstanceDefinitions%>
    <%operationDefinitions%>
    <%ctorDefinitions%>
    <%dtorDefinitions%>
    <%formulaDefinitions%>
    <%activityDefinitions%>
    <%domainCtorDefinition%>
    <%domainOpDefinitions%>
    <%externalOpDefinitions%>
    <%portalDefinition%>
    <%epilogueDeclarations%>
}]
```

In the following sections, we show the procedures that are invoked as part of the code template expansion.

Including the Domain Header File

The generated code file includes the generated header file automatically.

```
<<generation code commands>>=
proc domainInclude {} {
    variable domain

    return "#include \"$domain.h\"\n"
}
```

Domain Name as a String

The generated code file defines a preprocessor macro with the domain name, making it available for debugging and instrumentation printout.

```
<<generation code commands>>=
proc domainNameString {} {
    variable domain

    return "\"$domain\""
}
```

Domain Prologue

Any prologue text defined during domain configuration is placed early in the generated code file.

```
<<generation code commands>>=
proc prologueDeclarations {} {
    variable prologue

    return [string cat\
        [comment "Domain Prologue"]\
        [textutil::adjust::undent [string trim $prologue \n]]\
    ]
}
```

Implementation Type Aliases

The type aliases that are not placed in the header file are placed in the code file.

```
<<generation code commands>>=
proc implementationTypeAliases {} {
    variable domain
    set aliases [pipe {
        Domain findWhere {$Name eq $domain} |
        findRelated % ~R7 |
        deRef %
    } {} |%]

    set impaliases [relation minus $aliases [FindInterfaceTypeAliases $domain]] ; # ❶
    append result [comment "Type Aliases"]
    relation foreach taRef $impaliases {
        relation assign $taRef TypeName TypeDefinition
```

```
        append result\
            "typedef "\
            [typeCheck composeDeclaration $TypeDefinition $TypeName]\
            " ;\n"
    }

    return $result
}
```

❶ The set difference between all the type aliases and those used in the domain interface is the set that we are interested in here.

Forward Class Declarations

We need to introduce the structure names for the classes early in the generated code file. No definition of the structure is given here. That will come later, but since class structures have pointers to other class structures we need to tell the compiler about the names. Sometimes the pointer references between classes are so intertwined that we have no choice but to forward declare the class. Rather than attempting to figure out some order that does not reference a class structure name before it is defined, we just emit forward declarations for all the classes. There is no reason to do the compiler's job here.

```
<<generation code commands>>=
proc forwardClassDeclarations {} {
    variable domain

    set classNames [pipe {
        Class findWhere {$Domain eq $domain} |
        deRef ~ |
        relation list ~ Name -ascending Number
    }]
    set result [comment "Class structure forward declarations"]
    foreach className $classNames {
        append result "struct $className ;\n"
    }

    return $result
}
```

Forward Relationship Declaration

The same reasoning applies to the array that holds the relationship description information. Class definitions will make reference to relationship descriptions and we need to have its name known to the compiler.

```
<<generation code commands>>=
proc forwardRelationshipDeclaration {} {
    variable domain
    if {[isNotEmptyRef [Relationship findAll]]} {
        set fwrd "static MRT_Relationship const ${domain}__RSHIPS\[\] ;\n"
    } else {
        set fwrd {}
    }
    return [string cat\
        [comment "Relationship descriptors forward declaration"]\
        $fwrd\
    ]
}
```

Class Declarations

To declare the structures of the classes has one complication. As usual, union subclasses have to be treated separately. Because the class structure of union based generalization nests the subclasses as part of the superclass structure, we must emit the union subclass declarations in the correct order for the compiler. Otherwise, the compiler will complain as it must know the size of every member of a structure as the structure is being declared.

The strategy here is start with the ultimate superclasses and find those ultimate superclasses that are part of a union based generalization. An ultimate superclass is one at the top of a generalization hierarchy, *i.e.* a superclass that is not also a subclass of some other generalization relationship. For those ultimate superclasses that are part of a union based generalization we can emit the class declarations in the tree in reverse order. After dealing with the union superclasses then the non-union ones can be declared in an arbitrary order.

```
<<generation code commands>>=
proc classDeclarations {} {
    variable domain

    set result [comment "Class structure declarations"]

    set ultimates [pipe {
        FindUltimateSuperclasses $domain |
        findRelated % {~R48 UnionSuperclass}
    } {} |%]
    append result [DeclareUnionSubclassStructures $ultimates]

    set remaining [FindNonUnionSubclasses $domain]
    forAllRefs classRef $remaining {
        append result [DeclareClassStructure $classRef]
    }

    return $result
}
```

Since the union subclasses form a tree, we will use a recursive procedure to walk the implied tree. The order of traversal will be depth first which insures that union subclasses are defined before they must be used in the definition of the including union superclass.

```
<<generation code commands>>=
proc DeclareUnionSubclassStructures {superRefs} {
    if {[isEmptyRef $superRefs]} {
        return
    }

    set subRefs [findRelated $superRefs R44 ~R45]
    forAllRefs subRef $subRefs {
        set newsupers [pipe {
            deRef $subRef |
            relation semijoin ~ $::micca::UnionSuperclass\
                -using {Domain Domain Class Class} |
            ::rosea::Helpers::ToRef ::micca::UnionSuperclass ~
        }] ; # ❶

        append result [DeclareUnionSubclassStructures $newsupers] ; # ❷

        append result [DeclareClassStructure [findRelated $subRef R47 R40 R41]] ; # ❸
    }

    return $result
}
```

❶ By performing the `semijoin` only across the Domain and Class attributes, we in effect are asking if the union subclass is also a union superclass regardless of the relationship.

❷ Recursively define the subclass structures first so we get a depth first walk.

❸ Emit the declaration for the current subclass node.

We can find the classes not part of a union generalization by the difference of all the classes and those that are union subclasses.

```
<<generation helper commands>>=
proc FindNonUnionSubclasses {domain} {
    set usubs [UnionSubclass findWhere {$Domain eq $domain}]
    return [pipe {
        Class findWhere {$Domain eq $domain} |
        deRef ~ |
        relation semiminus [deRef $usubs] ~ -using {Domain Domain Class Name} |
        ::rosea::Helpers::ToRef ::micca::Class ~
    }]
}
```

Class structure definitions consist of three components:

1. The base instance data structure as required and defined by the run time code.
2. Attributes defined by the class.
3. References used to implement relationship navigation.

Of the three, references are the most complicated in that they come in a larger number of variations.

```
<<generation code commands>>=
proc DeclareClassStructure {classRef} {
    assignAttribute $classRef {Name className}

    append result\
        "struct $className \{\n"\
        "    MRT_Instance base__INST ;\n"

    set compRefs [findRelated $classRef ~R20] ; # ❶

    <<DeclareClassStructure: attribute declarations>>
    <<DeclareClassStructure: reference declarations>>

    append result "\} ;\n"
}
```

❶ Find all the class components for the given class.

Declaring the structure members for attributes is a matter of finding the attributes and emiting “C” declarations, knowing the data type and attribute name. The only minor complication is for attributes that are specified to be arrays.

```
<<DeclareClassStructure: attribute declarations>>=
set attrRefs [findRelated $compRefs {~R25 PopulatedComponent}\
    {~R21 Attribute} {~R29 IndependentAttribute} R29] ; # ❶
forAllRefs attrRef $attrRefs {
    assignAttribute $attrRef {Name attrName} DataType
    append result "    [typeCheck composeDeclaration $DataType $attrName] ;\n"
}
```

❶ This query finds all the attributes of the class that are independent attributes. Those are the ones that will have a structure member.

As can be seen from the portion of the platform model that deals with classes there are a lot of different types of references. In the code below we assiduously consider each different type.

```
<<DeclareClassStructure: reference declarations>>=
set refRefs [findRelated $compRefs {~R25 PopulatedComponent} {~R21 Reference}] ; # ❶
<<DeclareClassStructure: superclass references>>
<<DeclareClassStructure: associator references>>
<<DeclareClassStructure: association references>>

set genRefs [findRelated $compRefs {~R25 GeneratedComponent}] ; # ❷
<<DeclareClassStructure: subclass references>>
<<DeclareClassStructure: subclass containers>>
<<DeclareClassStructure: link containers>>
<<DeclareClassStructure: complementary references>>
```

❶ First we deal with those references that can be populated. Referring to the platform model will make this clearer.

❷ The other references are generated to support navigating relationships in both directions.

A superclass reference declaration takes the form of a simple pointer to the superclass structure.

```
<<DeclareClassStructure: superclass references>>=
set superRefs [findRelated $refRefs {~R23 SuperclassReference} ~R91\
        {~R47 ReferringSubclass} R37 ~R36]
forAllRefs superRef $superRefs {
    assignAttribute $superRef {Relationship attrName} {Class className}
    append result "    [GetClassProperty $className Reference]$attrName ;\n"
}
```

Associative classes make two references, one to each class participating in the association, which are grouped together in an unnamed structure.

```
<<DeclareClassStructure: associator references>>=
set atorRefs [findRelated $refRefs {~R23 AssociatorReference} ~R93 R42]
forAllRefs atorRef $atorRefs {
    assignAttribute $atorRef {Name attrName}

    set sourceDecl [pipe {
        findRelated $atorRef ~R34 |
        readAttribute % Class |
        GetClassProperty % Reference
    } {} |%]
    set targetDecl [pipe {
        findRelated $atorRef ~R35 |
        readAttribute % Class |
        GetClassProperty % Reference
    } {} |%]

    append result\
        "    struct \{\n"\
        "        ${targetDecl}forward ;\n"\
        "        ${sourceDecl}backward ;\n"\
        "    \} $attrName ;\n"
}
```

Simple association references also resolve to a single pointer declaration to the referenced class.

```
<<DeclareClassStructure: association references>>=
set assocRefs [findRelated $refRefs {~R23 AssociationReference} ~R90 R32 ~R33]
forAllRefs assocRef $assocRefs {
    assignAttribute $assocRef {Relationship attrName} {Class className}
    append result "    [GetClassProperty $className Reference]$attrName ;\n"
}
```

Generated references deal with the "other" side of a relationship. When a class serves as a superclass, it has a pointer to its related subclass instance. For the superclass, we use a `void` pointer since the type of the subclass instance varies.

```
<<DeclareClassStructure: subclass references>>=
set subRefs [findRelated $genRefs {~R24 SubclassReference}]
forAllRefs subRef $subRefs {
    assignAttribute $subRef {Name attrName}
    append result "    void *$attrName ;\n"
}
```

Union superclasses use a union to serve as the container for subclass instances.

```
<<DeclareClassStructure: subclass containers>>=
set contRefs [findRelated $genRefs {~R24 SubclassContainer}]
forAllRefs contRef $contRefs {
    assignAttribute $contRef {Name attrName}

    set subRefs [findRelated $contRef ~R96 R44 ~R45]
    append result "    union \{\n"
    foreach subName [relation list [deRef $subRefs] Class] {
        append result "        "\
            "[GetClassProperty $subName Declaration] $subName ;\n"
    }
    append result "    \} $attrName ;\n"
}
```

For class instances that are part of a linked list, we have to provide storage in the class structure for the list links.

```
<<DeclareClassStructure: link containers>>=
set linkRefs [findRelated $genRefs {~R24 LinkContainer}]
forAllRefs linkRef $linkRefs {
    append result "    MRT_LinkRef [readAttribute $linkRef Name] ;\n"
}
```

Complementary references are used by associations to implement the back references. There are several types depending upon the multiplicity and whether the association is static.

```
<<DeclareClassStructure: complementary references>>=
set complRefs [findRelated $genRefs {~R24 ComplementaryReference}]
forAllRefs complRef $complRefs {
    set singRef [findRelated $complRef {~R26 SingularReference} R26]
    if {[isNotEmptyRef $singRef]} {
        set attrName [readAttribute $singRef Name]
        set refedClass [FindReferencedClass $singRef]
        append result\
            "    [GetClassProperty $refedClass Reference]$attrName ;\n"
        continue
    }

    set arrayRef [findRelated $complRef {~R26 ArrayReference} R26]
    if {[isNotEmptyRef $arrayRef]} {
        set attrName [readAttribute $arrayRef Name]
        set classref [GetClassProperty [FindReferencedClass $arrayRef] Reference]
        append result\
```

```
            "    struct \{\n"\
            "        ${classref}const *links ;\n"\
            "        unsigned count ;\n"\
            "    \} $attrName ;\n"
        continue
    }

    set linkRef [findRelated $complRef {~R26 LinkReference} R26]
    if {[isNotEmptyRef $linkRef]} {
        set attrName [readAttribute $linkRef Name]
        append result "    MRT_LinkRef $attrName ;\n"
        continue
    }
}
```

For complementary references, we need to be able to find the class to which they refer. We factor out the mucking around the platform model that is necessary to determine the class referenced by a complementary reference.

```
<<generation helper commands>>=
proc FindReferencedClass {compRef} {
    set destRef [findRelated $compRef {~R28 BackwardReference} ~R94]

    # backward, simple
    set refing [findRelated $destRef {~R38 SimpleReferencedClass} R33 ~R32]
    if {[isNotEmptyRef $refing]} {
        set refedClass [readAttribute $refing Class]
    } else {
        # backward, target
        set src [findRelated $destRef {~R38 TargetClass} R35 ~R42]
        if {[isNotEmptyRef $src]} {
            set refedClass [readAttribute $src Class]
        } else {
            # forward, source
            set trg [findRelated $compRef {~R28 ForwardReference} ~R95 R34 ~R42]
            if {[isNotEmptyRef $trg]} {
                set refedClass [readAttribute $trg Class]
            }
        }
    }
    return $refedClass
}
```

Assigner Declarations

An assigner is declared in much the same manner as a class. This gives us the ability to deal with the state model of an assigner using the same code as for classes. There are a few different rules for assigners:

- Single assigners have no attributes and are never created dynamically. Their single instance is created as part of the initial instance population.
- Multiple assigners have a single attribute and may be created dynamically.

```
<<generation code commands>>=
proc assignerDeclarations {} {
    variable domain

    append result [comment "Single Assigner Structure Declarations"]
```

```
    set singles [SingleAssigner findWhere {$Domain eq $domain}]
    forAllRefs assigner $singles {
        assignAttribute $assigner
        append result\
            "struct $Association \{\n"\
            "    MRT_Instance base__INST ;\n"\
            "\} ;\n"
    }

    append result [comment "Multiple Assigner Structure Declarations"]

    set multis [MultipleAssigner findWhere {$Domain eq $domain}]
    forAllRefs assigner $multis {
        assignAttribute $assigner
        append result\
            "struct $Association \{\n"\
            "    MRT_Instance base__INST ;\n"\
            "    [GetClassProperty $Class Reference] idinstance ;\n"\
            "\} ;\n"
    }

    return $result
}
```

State Parameter Declarations

The parameter signatures for state activities are strictly private to the domain and so their structure declarations are placed in the generated code file.

```
<<generation code commands>>=
proc stateParamDeclarations {} {
    variable domain
    set result [comment "State Parameter Structure Declarations"]

    set stateargs [pipe {
        State findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Name State |
        relation join ~ $::micca::StateSignature $::micca::ParameterSignature\
                $::micca::Parameter $::micca::Argument |
        relation project ~ Model State Name Position DataType |
        relation group ~ Params Name Position DataType
    }]
    #puts [relformat $stateargs stateargs]

    relation foreach statearg $stateargs -ascending {Model State} {
        relation assign $statearg
        set pstructname "struct ${Model}_${State}__SPARAMS"
        append result "$pstructname \{\n"
        relation foreach param $Params -ascending Position {
            relation assign $param DataType Name
            append result\
                "    [typeCheck composeDeclaration $DataType $Name] ;\n"
        }
        append result "\} ;\n"
        append result\
            "#if __STDC_VERSION__ >= 201112L\n"\
            "static_assert(sizeof($pstructname) <= sizeof(MRT_EventParams), "\
            "\"Parameters for class or assigner, $Model, state,\
```

```
                $State, are too large\") ;\n"\
            "#endif /* __STDC_VERSION__ >= 201112L */\n"
    }

    return $result
}
```

Operation Declarations

Both class based and instance based operations must have names that are unique within the class, so we can emit the forward declarations for both types of operations in a single query.

```
<<generation code commands>>=
proc operationDeclarations {} {
    variable domain
    set result [comment "Operation Forward Declarations"]

    set ops [pipe {
        Operation findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $::micca::OperationParameter Parameters
    }]

    relation foreach op $ops {
        relation assign $op
        if {[relation isempty $Parameters]} {
            set pdecl void
        } else {
            set plist [list]
            relation foreach op_param $Parameters -ascending Number {
                relation assign $op_param Name DataType
                lappend plist [typeCheck composeDeclaration $DataType $Name]
            }
            set pdecl [join $plist {, }]
        }

        append result\
            "static $ReturnDataType ${Class}_$Operation\($pdecl\) ;\n"
    }

    return $result
}
```

Constructor Declarations

Pointers to constructors are part of the IAB for a class and so we need to have forward declarations for them.

```
<<generation code commands>>=
proc ctorDeclarations {} {
    variable domain
    set result [comment "Class Constructor Forward Declarations"]

    set ctors [Constructor findWhere {$Domain eq $domain}]
    forAllRefs ctor $ctors {
        assignAttribute $ctor {Class className}
        append result\
```

```
            "static void ${className}__CTOR\(void *const) ;\n"
    }

    return $result
}
```

Destructor Declarations

Destructor declarations follow the same pattern as constructors.

```
<<generation code commands>>=
proc dtorDeclarations {} {
    variable domain
    set result [comment "Class Destructor Forward Declarations"]

    set dtors [Destructor findWhere {$Domain eq $domain}]
    forAllRefs dtor $dtors {
        assignAttribute $dtor {Class className}
        append result\
            "static void ${className}__DTOR\(void *const) ;\n"
    }

    return $result
}
```

Formula Declarations

Formula declarations arise for dependent attributes.

```
<<generation code commands>>=
proc formulaDeclarations {} {
    variable domain
    set result [comment "Dependent Attribute Formula Forward Declarations"]

    set deps [DependentAttribute findWhere {$Domain eq $domain}]
    forAllRefs dep $deps {
        assignAttribute $dep {Class className} {Name attrName}
        append result\
            "static void ${className}_${attrName}__FORMULA\(void const *const,\
                void *const, MRT_AttrSize) ;\n"
    }

    return $result
}
```

Activity Declarations

State activities must also be declared so that pointers to them can be used as activity table initializer values.

```
<<generation code commands>>=
proc activityDeclarations {} {
    variable domain
    set result [comment "State Activities Forward Declarations"]
```

```
    set classes [pipe {
        State findWhere {$Domain eq $domain && [string trim $Activity] ne {}} |
        deRef ~ |
        relation eliminate ~ Domain Activity File Line IsFinal |
        relation group ~ States Name
    }]

    relation foreach class $classes -ascending Model {
        relation assign $class
        relation foreach state $States {
            relation assign $state
            append result "static void ${Model}_${Name}__ACTIVITY\("\
                "void *const s__SELF, "\
                "void const *const p__PARAMS) ;\n"
        }
    }

    return $result
}
```

Storage Declarations

Because the relationship pointers in the initial instance population initializers will involve address expressions into the storage pool of the instances of a class, we need to make the storage pool names known to the compiler. Later, we will put down the initializers, if any. If there is not an initial instance population for a class, then this declaration is sufficient to allocate the memory for the instance storage.

```
<<generation code commands>>=
proc storageDeclarations {} {
    variable domain
    set result [comment "Class Instance Storage Forward Declarations"]

    set cpops [pipe {
        FindNonUnionSubclasses $domain |
        findRelated % R104 {~R101 ElementPopulation} {~R105 ClassPopulation}
    } {} |%]

    forAllRefs cpop $cpops {
        assignAttribute $cpop {Class className}
        set var [GetClassProperty $className StorageVariable]
        set totalinsts [GetClassProperty $className TotalInstance]
        append result "static struct $className $var \[$totalinsts\] ;\n"
    }

    set singles [SingleAssigner findWhere {$Domain eq $domain}]
    forAllRefs single $singles {
        assignAttribute $single Association
        set var [GetClassProperty $Association StorageVariable]
        set totalinsts [GetClassProperty $Association TotalInstance]
        append result "static struct ${Association} $var\[$totalinsts\] ;\n"
    }

    set multis [MultipleAssigner findWhere {$Domain eq $domain}]
    forAllRefs multi $multis {
        assignAttribute $multi Association
        set var [GetClassProperty $Association StorageVariable]
        set totalinsts [GetClassProperty $Association TotalInstance]
        append result "static struct ${Association} $var\[$totalinsts\] ;\n"
    }
```

```
    return $result
}
```

Name Definitions

The naming information for classes, relationships, states, events, etc. is placed in the generated code file. Note that the names can be compiled out by defining the `MRT_NO_NAMES` preprocessor symbol. Naming consumes a lot of memory and is most useful only during debugging.

```
<<generation code commands>>=
proc nameDefinitions {} {
    variable domain
    append result\
        [comment "Domain Naming Definitions"]\
        "#ifndef MRT_NO_NAMES\n"

    set storageType "static char const"

    set classRefs [Class findWhere {$Domain eq $domain}]
    forAllRefs classRef $classRefs {
        set className [readAttribute $classRef Name]
        append result "$storageType ${className}__NAME\[\] = \"$className\" ;\n"
    }

    set relRefs [Relationship findWhere {$Domain eq $domain}]
    forAllRefs relRef $relRefs {
        set relName [readAttribute $relRef Name]
        append result "$storageType ${relName}__NAME\[\] = \"$relName\" ;\n"
    }

    set stateRefs [StatePlace findWhere {$Domain eq $domain}]
    forAllRefs stateRef $stateRefs {
        assignAttribute $stateRef {Model modelName} {Name stateName}
        set namevar [expr {$stateName eq "@" ? "AT" : $stateName}]
        append result "$storageType ${modelName}_${namevar}__SNAME\[\] =\
                \"$stateName\" ;\n"
    }

    set eventRefs [Event findWhere {$Domain eq $domain}]
    forAllRefs eventRef $eventRefs {
        assignAttribute $eventRef {Model modelName} {Event eventName}
        append result "$storageType ${modelName}_${eventName}__ENAME\[\] =\
                \"$eventName\" ;\n"
    }

    set saRefs [SingleAssigner findWhere {$Domain eq $domain}]
    if {[isNotEmptyRef $saRefs]} {
        append result "$storageType single_assigner_instance__NAME\[\] =\
                \"assigner\" ;\n"
    }

    set maRefs [MultipleAssigner findWhere {$Domain eq $domain}]
    if {[isNotEmptyRef $maRefs]} {
        append result "$storageType multi_assigner_attribute__NAME\[\] =\
                \"idinstance\" ;\n"
    }

    append result "#endif /* MRT_NO_NAMES */\n"
```

```
    return $result
}
```

Instance Allocation Block Definitions

An Instance Allocation Block defines the characteristics of the instance storage pool to the run-time code.

```
<<generation code commands>>=
proc iabDefinitions {} {
    variable domain
    set result [comment "Instance Allocation Block Definitions"]

    forAllRefs classRef [FindNonUnionSubclasses $domain] {
        append result [DefineIABMembers $classRef false]
    }

    set unionRefs [pipe {
        UnionSubclass findWhere {$Domain eq $domain} |
        findRelated ~ R47 R40 R41
    }]
    forAllRefs classRef $unionRefs {
        append result [DefineIABMembers $classRef true]
    }

    set singles [SingleAssigner findWhere {$Domain eq $domain}]
    forAllRefs single $singles {
        assignAttribute $single Association
        append result\
            "static MRT_iab ${Association}__IAB = \{\n"\
            "    .storageStart = &${Association}__POOL\[0\],\n"\
            "    .storageFinish = &${Association}__POOL\[1\],\n"\
            "    .storageLast = &${Association}__POOL\[0\],\n"\
            "    .alloc = 1,\n"\
            "    .instanceSize = sizeof\(struct ${Association}\),\n"\
            "    .construct = NULL,\n"\
            "    .destruct = NULL,\n"\
            "    .linkCount = 0,\n"\
            "    .linkOffsets = NULL\n"\
            "\} ;\n"
    }

    set multis [MultipleAssigner findWhere {$Domain eq $domain}]
    forAllRefs multi $multis {
        set initial [refMultiplicity [findRelated $multi ~R106]]
        assignAttribute $multi Association {Class idclass}
        set total [GetClassProperty $idclass TotalInstance]

        append result\
            "static MRT_iab ${Association}__IAB = \{\n"\
            "    .storageStart = &${Association}__POOL\[0\],\n"\
            "    .storageFinish = &${Association}__POOL\[$total\],\n"\
            "    .storageLast = &${Association}__POOL\[[expr {$total - 1}]\],\n"\
            "    .alloc = $initial,\n"\
            "    .instanceSize = sizeof\(struct ${Association}\),\n"\
            "    .construct = NULL,\n"\
            "    .destruct = NULL,\n"\
            "    .linkCount = 0,\n"\
            "    .linkOffsets = NULL\n"\
```

```
            "\} ;\n"
    }

    return $result
}
```

```
<<generation helper commands>>=
proc DefineIABMembers {classRef isUnionSubclass} {
    assignAttribute $classRef {Name className}

    set linkRefs [findRelated $classRef ~R20 {~R25 GeneratedComponent}\
        {~R24 ComplementaryReference} {~R26 LinkReference}]
    if {[isEmptyRef $linkRefs]} {
        set linkOffsets NULL
    } else {
        set linkOffsets ${className}__LINKS
        set linkCount [refMultiplicity $linkRefs]
        append result\
            "static MRT_AttrOffset const $linkOffsets\[$linkCount\] = \{\n"
        set linkMembers [relation list [deRef $linkRefs] Name]
        foreach linkMember $linkMembers {
            append result\
                "    "\
                "offsetof\([GetClassProperty $className Declaration],\
                $linkMember\),\n"
        }
        append result "\} ;\n"
    }

    append result\
        "static MRT_iab ${className}__IAB = \{\n"

    if {$isUnionSubclass} {
        append result\
            "    .storageStart = NULL,\n"\
            "    .storageFinish = NULL,\n"\
            "    .storageLast = NULL,\n"
    } else {
        set nelements [GetClassProperty $className TotalInstance]
        append result\
            "    .storageStart = &${className}__POOL\[0\],\n"\
            "    .storageFinish = &${className}__POOL\[$nelements\],\n"\
            "    .storageLast = &${className}__POOL\[[expr {$nelements - 1}]\],\n"
    }
    set ctor [pipe {
        findRelated $classRef ~R8 |
        expr {[isEmptyRef %] ? "NULL" : "[readAttribute % Class]__CTOR"}
    } {} |%]
    set dtor [pipe {
        findRelated $classRef ~R9 |
        expr {[isEmptyRef %] ? "NULL" : "[readAttribute % Class]__DTOR"}
    } {} |%]
    append result\
        "    .alloc = [GetClassProperty $className InitialInstance],\n"\
        "    .instanceSize = sizeof\([GetClassProperty $className Declaration]\),\n"\
        "    .construct = $ctor,\n"\
        "    .destruct = $dtor,\n"\
        "    .linkCount = [refMultiplicity $linkRefs],\n"\
        "    .linkOffsets = $linkOffsets\n"\
        "\} ;\n"

    return $result
```

```
}
```

Event Dispatch Block Definitions

The Event Dispatch Block is the critical data structure used to dispatch events to the state models.

```
<<generation code commands>>=
proc edbDefinitions {} {
    variable domain
    set result [comment "Event Dispatch Block Definitions"]

    upvar #0 ::micca::@Gen@::options options

    forAllRefs smodel [StateModel findWhere {$Domain eq $domain}] {
        assignAttribute $smodel {Model className}

        append result [GenerateTransitionTable $smodel]
        append result [GenerateActivityTable $smodel]
        set finalstates [GenerateFinalStates $smodel result]

        append result [GenerateStateNames $smodel]

        append result "static MRT_edb const ${className}__EDB = \{\n"

        set stateRefs [findRelated $smodel ~R55]
        set cstateRef [findRelated $smodel ~R56]
        append result\
            "    .stateCount = [expr {[refMultiplicity $stateRefs] +\
                [refMultiplicity $cstateRef]}],\n"

        set eventRefs [findRelated $smodel ~R87]
        append result\
            "    .eventCount = [refMultiplicity $eventRefs],\n"\
            "    .initialState = [readAttribute [findRelated $smodel R58 R57]\
                    Number],\n"
        set crstate [expr {[isNotEmptyRef $cstateRef] ?\
            [readAttribute [findRelated $cstateRef R57] Number] :\
            "MRT_StateCode_IG"}]
        append result\
            "    .creationState = $crstate,\n"\
            "    .transitionTable = ${className}__TTAB,\n"\
            "    .activityTable = ${className}__ATAB,\n"\
            "    .finalStates = $finalstates,\n"

        append result\
            "        #ifndef MRT_NO_NAMES\n"\
            "    .stateNames = ${className}__SNAMES,\n"\
            "        #endif /* MRT_NO_NAMES */\n"

        append result\
            "\} ;\n"
    }

    return $result
}
```

The transition table is key to event dispatch. The data structure is a matrix of *state* rows and *events* columns. However, the matrix is held as a one-dimensional array since that is a consistent single data type that can be declared for the EDB. The dispatch code

deals with finding the correct state row and event column by performing the required scaling of row access by the number of event columns.

In the platform model, the **Transition Place** only holds those transitions that are explicitly mentioned in `transition` statements. To compute the whole transition matrix, we have to combine the explicit transition with the default ones.

We compute the transition matrix in three steps:

1. Compute the entries for the explicitly mentioned transitions.
2. Compute the default transitions not explicitly given.
3. Combine the two groups into the transition matrix.

```
<<generation code commands>>=
proc GenerateTransitionTable {smodelref} {
    variable domain
    assignAttribute $smodelref {Model className}

    <<GenerateTransitionTable: compute specified transitions>>
    <<GenerateTransitionTable: compute default transitions>>
    <<GenerateTransitionTable: compute transition matrix>>

    set result "static MRT_StateCode const ${className}__TTAB\[\] = \{\n"
    relation foreach transition $transmatrix -ascending {StateNumber EventNumber} {
        relation assign $transition
        append result "    $NewStateNumber, // $State - $Event -> $NewState\n"
    } ; # ❶
    append result "\} ;\n"

    return $result
}
```

❶ Order here is first by state number and then by event number. This is the order that the run-time code requires.

Transitions specified in a `transition` statement are one of two types. Either a transition to a state or one of the non-transitioning destinations of “IG” or “CH”.

```
<<GenerateTransitionTable: compute specified transitions>>=
set tplaces [pipe {
    TransitionPlace findWhere {$Domain eq $domain && $Model eq $className} |
    deRef ~
}]
# puts [relformat $tplaces tplaces]

set statetrans [pipe {
    relation join $tplaces $::micca::StateTransition |
    relation join ~ $::micca::StatePlace\
        -using {Domain Domain Model Model NewState Name} |
    relation extend ~ stup\
        NewStateNumber string {[tuple extract $stup Number]} |
    relation eliminate ~ Number
}] ; # ❶
# puts [relformat $statetrans statetrans]

set nontrans [pipe {
    relation join $tplaces $::micca::NonStateTransition |
    relation extend ~ ntup\
        NewState string {[tuple extract $ntup TransRule]}\
        NewStateNumber string {"MRT_StateCode_[tuple extract $ntup TransRule]"} |
    relation eliminate ~ TransRule
}] ; # ❷
# puts [relformat $nontrans nontrans]
```

❶ The transitions to an actual state need to pick up the number for the new state. The numerical encoding is what the run-time code actually uses.

❷ For non-transitioning destinations, the new state number is actually a symbolic constant. This is the reason that the **NewStateNumber** attribute is a string.

The default transitions are computed by difference. First we compute all possible transitions and then subtract off the ones that were specified as a **TransitionPlace** instance. All possible transitions is just the Cartesian product of the states and events.

```
<<GenerateTransitionTable: compute default transitions>>=
set smodel [deRef $smodelref]
set srcstates [pipe {
    relation semijoin $smodel $::micca::StatePlace |
    relation eliminate ~ Number |
    relation rename ~ Name State
}] ; # ❶
# puts [relformat $srcstates srcstates]

set events [pipe {
    findRelated $smodelref ~R87 |
    deRef %
} {} |%]
# puts [relformat $events events]

set defrule [readAttribute $smodelref DefaultTrans]
# puts "defrule = \"$defrule\""

set deftrans [pipe {
    relation join $srcstates $events |
    relation minus ~ $tplaces |
    relation extend ~ dtup NewState string {$defrule} |
    relation update ~ ftup {[tuple extract $ftup State] eq "@"}\
        {tuple update $ftup NewState CH} |
    relation extend ~ ttup\
        NewStateNumber string {"MRT_StateCode_[tuple extract $ttup NewState]"}
}] ; # ❷
# puts [relformat $deftrans deftrans]
```

❶ We need to clean up some attributes here.

❷ The `join` here effectively computes the Cartesian product since the only attributes in common are `Domain` and `Model`.

The final transition matrix is just a union of the three types of transitions, namely, explicitly specified transitions, explicitly specified non-state transitions and default transitions. We also need to join on the state and event numbers since they are used to order the transition matrix during output.

```
<<GenerateTransitionTable: compute transition matrix>>=
set transmatrix [pipe {
    relation union $statetrans $nontrans $deftrans |
    relation join ~ $::micca::StatePlace\
        -using {Domain Domain Model Model State Name} |
    relation rename ~ Number StateNumber |
    relation join ~ $::micca::Event |
    relation rename ~ Number EventNumber |
    relation eliminate ~ Domain Model
}]
# puts [relformat $transmatrix transmatrix]
```

```
<<generation code commands>>=
proc GenerateActivityTable {smodel} {
```

```
    variable domain
    assignAttribute $smodel {Model className}

    set result "static MRT_PtrActivityFunction const ${className}__ATAB\[\] = \{\n"

    set places [pipe {
        StatePlace findWhere {$Domain eq $domain && $Model eq $className} |
        deRef ~
    }]
    relation foreach place $places -ascending Number {
        set state [relation semijoin $place $::micca::State]
        if {[relation isnotempty $state]} {
            relation assign $state Name Activity
            if {[string trim $Activity] eq {}} {
                append result "    NULL, // $Name\n"
            } else {
                append result "    ${className}_${Name}__ACTIVITY, // $Name\n"
            }
        } else {
            set cstate [relation semijoin $place $::micca::CreationState]
            if {[relation isnotempty $cstate]} {
                append result "    NULL, // [relation extract $cstate Name]\n"
            }
        }
    }

    append result "\} ;\n"
}
```

```
<<generation code commands>>=
proc GenerateFinalStates {smodel resultVar} {
    set tstates [findRelatedWhere $smodel ~R55 {$IsFinal}]
    if {[isEmptyRef $tstates]} {
        return NULL
    }
    upvar 1 $resultVar result
    variable domain
    assignAttribute $smodel {Model className}
    set places [pipe {
        StatePlace findWhere {$Domain eq $domain && $Model eq $className} |
        deRef ~
    }]

    append result "static bool const ${className}__TSTATES\[\] = \{\n"
    relation foreach place $places -ascending Number {
        set state [relation semijoin $place $::micca::State]
        if {[relation isnotempty $state]} {
            relation assign $state Name IsFinal
            append result "    "\
                [expr {$IsFinal ? "true" : "false"}]\
                ", // $Name\n"
        } else {
            set cstate [relation semijoin $place $::micca::CreationState]
            if {[relation isnotempty $cstate]} {
                append result "    false, // [relation extract $cstate Name]\n"
            }
        }
    }

    append result "\} ;\n"
    return ${className}__TSTATES
```

```
}
```

```
<<generation code commands>>=
proc GenerateStateNames {smodel} {
    variable domain
    assignAttribute $smodel {Model className}

    set states [pipe {
        StatePlace findWhere {$Domain eq $domain && $Model eq $className} |
        deRef ~
    }]

    append result\
        "#ifndef MRT_NO_NAMES\n"\
        "static char const * const ${className}__SNAMES\[\] = \{\n"

    relation foreach state $states -ascending Number {
        relation assign $state Name
        set Name [string map {@ AT} $Name]
        append result "    ${className}_${Name}__SNAME,\n"
    }

    append result\
        "\} ;\n"\
        "#endif /* MRT_NO_NAMES */\n"

    return $result
}
```

Polymorphic Dispatch Block Definitions

Polymorphic event dispatch involves mapping events at run time across a generalization. Here the data structures required by the run-time are more complicated.

```
<<generation code commands>>=
proc pdbDefinitions {} {
    variable domain

    set rnames {}
    set pmaps {}
    set gdbs {}
    set pdbs {}

    set deRefs [DeferredEvent findWhere {$Domain eq $domain}]
    set superRefs [findRelated $deRefs ~R86]
    if {[isEmptyRef $superRefs]} {
        return
    }

    set deferred [pipe {
        findRelated $superRefs {R86 DeferralPath} |
        deRef ~ |
        relation join ~ $::micca::DeferredEvent $::micca::Event |
        relation eliminate ~ Domain Role |
        relation rename ~ Model Superclass Number SuperNumber
    }]
    # puts [relformat $deferred deferred]

    set nonlocals [pipe {
```

```
        NonLocalEvent findWhere {$Domain eq $domain} |
        deRef % |
        relation join % $::micca::Relationship\
            -using {Domain Domain Relationship Name} |
        relation rename % Number RelNumber
    } {} |%]
    # puts [relformat $nonlocals nonlocals]

    set polys [relation join $nonlocals\
            $::micca::DeferredEvent $::micca::Event]
    # puts [relformat $polys polys]

    set trans [relation join $nonlocals\
            $::micca::TransitioningEvent $::micca::Event]
    # puts [relformat $trans trans]

    set supers [pipe {
        relation union $polys $trans |
        relation eliminate ~ Domain Role |
        relation join ~ $deferred |
        relation group ~ EventMap Event SuperNumber Number |
        relation group ~ SubMap Model EventMap |
        relation group ~ Generalizations Relationship RelNumber SubMap
    }]
    # puts [relformat $supers supers]

    relation foreach super $supers {
        relation assign $super

        set gdbvar ${Superclass}__GDBS
        append gdbs "static MRT_gdb $gdbvar\[\] = \{\n"

        set rnamesvar ${Superclass}__RNAMES
        append rnames "static char const *const $rnamesvar\[\] = \{\n"

        relation foreach gen $Generalizations -ascending Relationship {
            relation assign $gen

            append rnames "    ${Relationship}__NAME,\n"

            set pmapvar ${Superclass}_${Relationship}__PMAP
            append pmaps "static MRT_EventCode const $pmapvar\[\] = \{\n"

            relation foreach submap $SubMap -ascending Model {
                relation assign $submap
                relation foreach eventmap $EventMap -ascending Number {
                    relation assign $eventmap
                    append pmaps "    $Number, // $Event for $Model\n"
                }
            }
            append pmaps "\} ;\n"

            append gdbs\
                "    \{\n"\
                "        .relship = &${domain}__RSHIPS\[$RelNumber\],\n"\
                "        .eventMap = $pmapvar,\n"\
                "    \},\n"
        }
        append rnames "\} ;\n"
        append gdbs "\} ;\n"

        append pdbs\
```

```
            "static MRT_pdb const ${Superclass}__PDB = \{\n"\
            "    .eventCount = [relation cardinality $EventMap],\n"\
            "    .genCount = [relation cardinality $Generalizations],\n"\
            "    .genDispatch = $gdbvar,\n"\
            "        #ifndef MRT_NO_NAMES\n"\
            "    .genNames = $rnamesvar,\n"\
            "        #endif /* MRT_NO_NAMES */\n"\
            "\} ;\n"
    }


    append result\
        [comment "Polymorphic Event Dispatch Block Definitions"]\
        "#ifndef MRT_NO_NAMES\n"\
        $rnames\
        "#endif /* MRT_NO_NAMES */\n"\
        $pmaps $gdbs $pdbs

    return $result
}
```

Class Description Definitions

The class descriptions are another run-time data structure that is used extensively. It is the key focus of how a particular instance is to be treated since all instances of a class exhibit the same behavior.

The strategy here is to build a single, rather large relation value that contains attributes for the values of the `MRT_Class` structure elements and then iterate over the relation value to output the "C" structure member initializer values.

```
<<generation code commands>>=
proc classDefinitions {} {
    variable domain
    set front [comment "Class Description Definitions"]

    set classRefs [Class findWhere {$Domain eq $domain}]

    set classinrel [pipe {
        findRelated $classRefs {~R41 ClassRole} |
        deRef % |
        relation eliminate % Role
    } {} |%]
    set assocrels [pipe {
        relation join $classinrel $::micca::Association -using\
            {Domain Domain Relationship Name} |
        relation eliminate ~ IsStatic
    }]
    set genrels [relation join $classinrel $::micca::Generalization\
            -using {Domain Domain Relationship Name}]

    set rels [pipe {
        relation union $assocrels $genrels |
        relation join ~ $::micca::Relationship\
            -using {Domain Domain Relationship Name} |
        relation rename ~ Class Name Number RelNumber
    }]
    # puts [relformat $rels rels]

    set attrs [pipe {
        findRelated $classRefs ~R20 {~R25 PopulatedComponent} {~R21 Attribute} |
        deRef % |
```

```
        ralutil::rvajoin % $::micca::IndependentAttribute Indep |
        ralutil::rvajoin % $::micca::DependentAttribute Dep |
        relation rename % Name Attribute Class Name
    } {} |%]
    # puts [relformat $attrs attrs]

    set usubs [pipe {
        findRelated $classRefs {~R41 ClassRole} {~R40 Subclass}\
            {~R47 UnionSubclass} |
        deRef % |
        relation join % $::micca::Relationship\
            -using {Domain Domain Relationship Name} |
        relation rename % Class Name Number SuperNumber |
        relation eliminate % Role Relationship
    } {} |%]
    # puts [relformat $usubs usubs]

    set allevents [pipe {
        Event findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Model Name Number EvtNumber
    }]
    # puts [relformat $allevents allevents]

    set classes [pipe {
        deRef $classRefs |
        relation extend ~ ctup IAB string {"&[tuple extract $ctup Name]__IAB"} |
        ralutil::rvajoin ~ $allevents AEvents |
        ralutil::rvajoin ~\
            [relation rename $::micca::TransitioningEvent Model Name] TEvents |
        ralutil::rvajoin ~\
            [relation rename $::micca::DeferredEvent Model Name] PEvents |
        relation extend ~ etup\
            EDB string {
                [relation isempty [tuple extract $etup TEvents]] ?\
                "NULL" : "&[tuple extract $etup Name]__EDB"}\
            PDB string {
                [relation isempty [tuple extract $etup PEvents]] ?\
                "NULL" : "&[tuple extract $etup Name]__PDB"}\
            eventNames string {
                [relation isempty [tuple extract $etup AEvents]] ? "NULL" :\
                "[tuple extract $etup Name]__ENAMES"} |
        relation eliminate ~ TEvents PEvents |
        ralutil::rvajoin ~ $rels Relationships |
        relation extend ~ rtup\
            relCount int {
                [relation cardinality [tuple extract $rtup Relationships]]}\
            classRels string {
                [relation cardinality [tuple extract $rtup Relationships]] == 0 ?\
                "NULL" : "[tuple extract $rtup Name]__CRELS"} |
        ralutil::rvajoin ~ $attrs Attributes |
        relation extend ~ atup\
            attrCount int {
                [relation cardinality [tuple extract $atup Attributes]]}\
            classAttrs string {
                [relation isempty [tuple extract $atup Attributes]] ?\
                "NULL" : "[tuple extract $atup Name]__CATTRS"} |
        ralutil::rvajoin ~ $usubs UnionSubs |
        relation extend ~ utup\
            containment string {
                [relation isempty [tuple extract $utup UnionSubs]] ?\
                "NULL" :\
```

```
            "&${domain}__RSHIPS\[[relation extract\
                [tuple extract $utup UnionSubs]\
                SuperNumber]\].relInfo.unionGeneralization.superclass"} |
        relation eliminate ~ Domain

    }]
    # puts [relformat $classes classes]

    append result "static MRT_Class const\
            ${domain}__CLASSES\[[relation cardinality $classes]\] = \{\n"

    set relrefs {}
    set attrrefs {}
    set namerefs "#ifndef MRT_NO_NAMES\n"
    relation foreach class $classes -ascending Number {
        relation assign $class

        if {[relation isnotempty $Relationships]} {
            append relrefs\
                "static MRT_Relationship const *const\
                    ${Name}__CRELS\[\] = \{\n"

            relation foreach rel $Relationships {
                relation assign $rel
                append relrefs\
                    "    &${domain}__RSHIPS\[$RelNumber\], // $Relationship\n"
            }

            append relrefs "\} ;\n"
        }

        if {[relation isnotempty $Attributes]} {
            append attrrefs\
                "static MRT_Attribute const ${Name}__CATTRS\["\
                [relation cardinality $Attributes]\
                "\] = \{\n"

            relation foreach attr $Attributes -ascending Attribute {
                relation assign $attr
                append attrrefs\
                    "    \{\n"\
                    "        .size = sizeof($DataType),\n"
                if {[relation isnotempty $Indep]} {
                    set offset "offsetof([GetClassProperty $Name Declaration],\
                            $Attribute)"
                    append attrrefs\
                        "        .type = mrtIndependentAttr,\n"\
                        "        .access.offset = $offset,\n"
                } else {
                    set formfunc "${Name}_${Attribute}__FORMULA"
                    append attrrefs\
                        "        .type = mrtDependentAttr,\n"\
                        "        .access.formula = $formfunc,\n"
                }
                append attrrefs\
                    "            #ifndef MRT_NO_NAMES\n"\
                    "        .name = \"$Attribute\"\n"\
                    "            #endif /* MRT_NO_NAMES */\n"\
                    "    \},\n"
            }

            append attrrefs "\} ;\n"
```

```
        }

        if {[relation isnotempty $AEvents]} {
            append namerefs\
                "static char const *const ${Name}__ENAMES\["\
                [relation cardinality $AEvents]\
                "\] = \{\n"

            relation foreach event $AEvents -ascending EvtNumber {
                relation assign $event {Event eventName}
                append namerefs "    ${Name}_${eventName}__ENAME,\n"
            }

            append namerefs "\} ;\n"
        }

        append result\
            "    \{ // $Name\n"\
            "        .iab = $IAB,\n"\
            "        .eventCount = [relation cardinality $AEvents],\n"\
            "        .edb = $EDB,\n"\
            "        .pdb = $PDB,\n"\
            "        .relCount = $relCount,\n"\
            "        .classRels = $classRels,\n"\
            "        .attrCount = $attrCount,\n"\
            "        .classAttrs = $classAttrs,\n"\
            "        .instCount = [GetClassProperty $Name TotalInstance],\n"\
            "        .containment = $containment,\n"\
            "            #ifndef MRT_NO_NAMES\n"\
            "        .name = ${Name}__NAME,\n"\
            "        .eventNames = $eventNames,\n"\
            "            #endif /* MRT_NO_NAMES */\n"\
            "    \},\n"
    }
    append namerefs "#endif /* MRT_NO_NAMES */\n"

    append result "\} ;\n"

    return [string cat $front $namerefs $relrefs $attrrefs $result]
}
```

Assigner Definitions

```
<<generation code commands>>=
proc assignerDefinitions {} {
    variable domain

    append result [comment "Assigner Class Description Definitions"]

    set events [pipe {
        TransitioningEvent findWhere {$Domain eq $domain} |
        deRef ~ |
        relation join ~ $::micca::Event |
        relation rename ~ Model Association
    }]
    set assigners [pipe {
        AssignerStateModel findWhere {$Domain eq $domain} |
        deRef ~ |
        ralutil::rvajoin ~ $::micca::SingleAssigner Single |
```

```
        ralutil::rvajoin ~ $::micca::MultipleAssigner Multiple |
        relation join ~ $events |
        relation group ~ Events Event Number
    }]

    set result {}
    set attrResult {}
    set enameResult {}
    if {[relation isnotempty $assigners]} {
        append enameResult "#ifndef MRT_NO_NAMES\n"

        set nassign [relation cardinality $assigners]
        append result\
            "static MRT_Class const ${domain}__ASSIGNERS\[$nassign\]= \{\n"\

        relation foreach assigner $assigners -ascending Association {
            relation assign $assigner

            append enameResult\
                "static char const *const ${Association}__ENAMES\["\
                [relation cardinality $Events]\
                "\] = \{\n"

            relation foreach event $Events -ascending Number {
                relation assign $event {Event eventName}
                append enameResult "    ${Association}_${eventName}__ENAME,\n"
            }

            append enameResult "\} ;\n"

            append result\
                "    \{ // $Association\n"\
                "        .iab = &${Association}__IAB,\n"\
                "        .eventCount = [relation cardinality $Events],\n"\
                "        .edb = &${Association}__EDB,\n"\
                "        .pdb = NULL,\n"
            if {[relation isnotempty $Single]} {
                append result\
                    "        .attrCount = 0,\n"\
                    "        .classAttrs = NULL,\n"\
            } else {
                relation assign $Multiple
                append attrResult\
                    "static MRT_Attribute const ${Association}__CATTRS\[1\] = \{\n"\
                    "    \{\n"\
                    "        .size = sizeof([GetClassProperty $Class Reference]),\n"\
                    "        .type = mrtIndependentAttr,\n"\
                    "        .access.offset = offsetof(struct $Association,\
                               idinstance),\n"\
                    "            #ifndef MRT_NO_NAMES\n"\
                    "        .name = multi_assigner_attribute__NAME\n"\
                    "            #endif /* MRT_NO_NAMES */\n"\
                    "    \}\n"\
                    "\} ;\n"
                append result\
                    "        .attrCount = 1,\n"\
                    "        .classAttrs = ${Association}__CATTRS,\n"\
            }
            set ninst [GetClassProperty $Association TotalInstance]
            append result\
                "        .instCount = $ninst,\n"\
                "        .relCount = 0,\n"\
```

```
                "        .classRels = NULL,\n"\
                "        .containment = NULL,\n"\
                "            #ifndef MRT_NO_NAMES\n"\
                "        .name = ${Association}__NAME,\n"\
                "        .eventNames = ${Association}__ENAMES,\n"\
                "            #endif /* MRT_NO_NAMES */\n"\
                "    \},\n"
        }
        append enameResult "#endif /* MRT_NO_NAMES */\n"

        append result "\} ;\n"
    }

    return [string cat $enameResult $attrResult $result]
}
```

Relationship Description Definitions

The calculation of relationship descriptions breaks down along the lines of the four different types of relationships found in the platform model.

```
<<generation code commands>>=
proc relationshipDefinitions {} {
    variable domain
    set subroles {}

    set relRefs [Relationship findWhere {$Domain eq $domain}]
    if {[isEmptyRef $relRefs]} {
        return
    }
    append result "static MRT_Relationship const\
        ${domain}__RSHIPS\[[refMultiplicity $relRefs]\] = \{\n"

    set indent [string repeat { } 4]
    set indent2 [string repeat $indent 2]
    set indent3 [string repeat $indent 3]
    set indent4 [string repeat $indent 4]

    forAllRefs relRef $relRefs {
        assignAttribute $relRef {Name relName} Number
        append result "$indent\[$Number\] = \{ // $relName\n"

        <<relationshipDefinitions: simple associations>>
        <<relationshipDefinitions: class based associations>>
        <<relationshipDefinitions: reference generalizations>>
        <<relationshipDefinitions: union generalizations>>
    }

    append result "\} ;\n"

    return [string cat\
        [comment "Relationship Description Definitions"]\
        $subroles\
        $result\
    ]
}
```

```
<<relationshipDefinitions: simple associations>>=
set typeRef [findRelated $relRef {~R30 Association} {~R31 SimpleAssociation}]
```

```
if {[isNotEmptyRef $typeRef]} {
    set sourceRef [findRelated $typeRef ~R32]
    assignAttribute $sourceRef {Class sClass} {Conditionality sCond}\
        {Multiplicity sMult}
    set sNum [readAttribute [findRelated $sourceRef R40 R41] Number]

    set targetRef [findRelated $typeRef ~R33]
    assignAttribute $targetRef {Class tClass}
    set tNum [readAttribute [findRelated $targetRef R38 R40 R41] Number]

    set tCond false
    set tMult false
    set stype mrtSingular
    set slink 0

    set tcomp [findRelated $targetRef R38 R94 R28]
    lassign [FindRelOffsets $tcomp] ttype tlink

    set sstore "offsetof([GetClassProperty $sClass Declaration],\
            $relName)"
    set tstore "offsetof([GetClassProperty $tClass Declaration],\
            [readAttribute $tcomp Name])"

    append result\
        "$indent2.relType = mrtSimpleAssoc,\n"\
        "$indent2.relInfo.simpleAssociation = \{\n"

    append result\
        "$indent3.source = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$sNum\], // $sClass\n"\
        "$indent4.cardinality = [MapToCardinality $tCond $tMult],\n"\
        "$indent4.storageType = $stype,\n"\
        "$indent4.storageOffset = $sstore,\n"\
        "$indent4.linkOffset = $slink\n"\
        "$indent3\},\n"

    append result\
        "$indent3.target = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$tNum\], // $tClass\n"\
        "$indent4.cardinality = [MapToCardinality $sCond $sMult],\n"\
        "$indent4.storageType = $ttype,\n"\
        "$indent4.storageOffset = $tstore,\n"\
        "$indent4.linkOffset = $tlink\n"\
        "$indent3\}\n"

    append result\
        "$indent2\},\n"\
        "${indent3}#ifndef MRT_NO_NAMES\n"\
        "$indent2.name = ${relName}__NAME\n"\
        "${indent3}#endif /* MRT_NO_NAMES */\n"\
        "$indent\},\n"
    continue
}
```

```
<<relationshipDefinitions: class based associations>>=
set typeRef [findRelated $relRef\
        {~R30 Association} {~R31 ClassBasedAssociation}]
if {[isNotEmptyRef $typeRef]} {
    set sourceRef [findRelated $typeRef ~R34]
    assignAttribute $sourceRef {Class sClass} {Conditionality sCond}\
        {Multiplicity sMult}
    set sNum [readAttribute [findRelated $sourceRef R40 R41] Number]
```

```
    set targetRef [findRelated $typeRef ~R35]
    assignAttribute $targetRef {Class tClass} {Conditionality tCond}\
        {Multiplicity tMult}
    set tNum [readAttribute [findRelated $targetRef R38 R40 R41] Number]

    set asstorRef [findRelated $typeRef ~R42]
    assignAttribute $asstorRef {Class aClass} {Multiplicity aMult}
    set aNum [readAttribute [findRelated $asstorRef R40 R41] Number]

    set scomp [findRelated $sourceRef R95 R28]
    lassign [FindRelOffsets $scomp] stype slink

    set tcomp [findRelated $targetRef R38 R94 R28]
    lassign [FindRelOffsets $tcomp] ttype tlink

    set sstore "offsetof([GetClassProperty $sClass Declaration],\
            [readAttribute $scomp Name])"
    set tstore "offsetof([GetClassProperty $tClass Declaration],\
            [readAttribute $tcomp Name])"

    append result\
        "$indent2.relType = mrtClassAssoc,\n"\
        "$indent2.relInfo.classAssociation = \{\n"

    append result\
        "$indent3.source = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$sNum\], // $sClass\n"\
        "$indent4.cardinality = [MapToCardinality $tCond $tMult],\n"\
        "$indent4.storageType = $stype,\n"\
        "$indent4.storageOffset = $sstore,\n"\
        "$indent4.linkOffset = $slink,\n"\
        "$indent3\},\n"

    append result\
        "$indent3.target = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$tNum\], // $tClass\n"\
        "$indent4.cardinality = [MapToCardinality $sCond $sMult],\n"\
        "$indent4.storageType = $ttype,\n"\
        "$indent4.storageOffset = $tstore,\n"\
        "$indent4.linkOffset = $tlink\n"\
        "$indent3\},\n"

    set aClassDecl [GetClassProperty $aClass Declaration]
    set aClassForw "offsetof($aClassDecl, ${relName}.forward)"
    set aClassBack "offsetof($aClassDecl, ${relName}.backward)"
    append result\
        "$indent3.associator = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$aNum\], // $aClass\n"\
        "$indent4.forwardOffset = $aClassForw,\n"\
        "$indent4.backwardOffset = $aClassBack,\n"\
        "$indent4.multiple = $aMult,\n"\
        "$indent3\}\n"

    append result\
        "$indent2\},\n"\
        "${indent3}#ifndef MRT_NO_NAMES\n"\
        "$indent2.name = ${relName}__NAME\n"\
        "${indent3}#endif /* MRT_NO_NAMES */\n"\
        "$indent\},\n"
    continue
```

```
}
```

```
<<relationshipDefinitions: reference generalizations>>=
set typeRef [findRelated $relRef\
        {~R30 Generalization} {~R43 ReferenceGeneralization}]
if {[isNotEmptyRef $typeRef]} {
    set superRef [findRelated $typeRef ~R36]
    assignAttribute $superRef {Class superClass}
    set supNum [readAttribute [findRelated $superRef R48 R40 R41] Number]

    set subRefs [findRelated $typeRef ~R37]
    set subClasses [findRelated $subRefs R47 R40 R41]

    set subs [pipe {
        deRef $subRefs |
        relation join ~ [deRef $subClasses]\
            -using {Domain Domain Class Name} |
        relation extend ~ stup\
            classDesc string {
                "&${domain}__CLASSES\[[tuple extract $stup Number]\],\
                // [tuple extract $stup Class]"}\
            storageOffset string {
                "offsetof([GetClassProperty\
                    [tuple extract $stup Class] Declaration],\
                    $relName)"} |
        relation project ~ Class classDesc storageOffset
    }]

    append subroles "static struct mrtrefsubclassrole const\
            ${relName}__SROLES\[[refMultiplicity $subRefs]\] = \{\n"
    relation foreach sub $subs -ascending Class {
        relation assign $sub
        append subroles\
            "    \{\n"\
            "        .classDesc = $classDesc,\n"\
            "        .storageOffset = $storageOffset\n"\
            "    \},\n"
    }
    append subroles "\} ;\n"

    append result\
        "$indent2.relType = mrtRefGeneralization,\n"\
        "$indent2.relInfo.refGeneralization = \{\n"

    append result\
        "$indent3.superclass = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$supNum\], // $superClass\n"\
        "$indent4.storageOffset =\
            offsetof([GetClassProperty $superClass Declaration],\
            $relName)\n"\
        "$indent3\},\n"

    append result\
        "$indent3.subclassCount = [refMultiplicity $subRefs],\n"\
        "$indent3.subclasses = ${relName}__SROLES\n"

    append result\
        "$indent2\},\n"\
        "${indent3}#ifndef MRT_NO_NAMES\n"\
        "$indent2.name = ${relName}__NAME\n"\
        "${indent3}#endif /* MRT_NO_NAMES */\n"\
        "$indent\},\n"
```

```
    continue
}
```

```
<<relationshipDefinitions: union generalizations>>=
set typeRef [findRelated $relRef\
        {~R30 Generalization} {~R43 UnionGeneralization}]
if {[isNotEmptyRef $typeRef]} {
    set superRef [findRelated $typeRef ~R44]
    assignAttribute $superRef {Class superClass}
    set supNum [readAttribute [findRelated $superRef R48 R40 R41] Number]
    set subRefs [findRelated $typeRef ~R45]
    set subClasses [findRelated $subRefs R47 R40 R41]
    set subs [pipe {
        deRef $subRefs |
        relation join ~ [deRef $subClasses]\
            -using {Domain Domain Class Name} |
        relation extend ~ stup\
            classDesc string {
                "&${domain}__CLASSES\[[tuple extract $stup Number]\],\
                // [tuple extract $stup Class]"} |
        relation project ~ Class classDesc
    }]

    append subroles "static MRT_Class const * const\
            ${relName}__SROLES\[[refMultiplicity $subRefs]\] = \{\n"
    relation foreach sub $subs -ascending Class {
        relation assign $sub
        append subroles "    $classDesc\n"
    }
    append subroles "\} ;\n"

    append result\
        "$indent2.relType = mrtUnionGeneralization,\n"\
        "$indent2.relInfo.unionGeneralization = \{\n"

    append result\
        "$indent3.superclass = \{\n"\
        "$indent4.classDesc = &${domain}__CLASSES\[$supNum\], // $superClass\n"\
        "$indent4.storageOffset =\
            offsetof([GetClassProperty $superClass Declaration],\
            $relName)\n"\
        "$indent3\},\n"

    append result\
        "$indent3.subclassCount = [refMultiplicity $subRefs],\n"\
        "$indent3.subclasses = ${relName}__SROLES\n"

    append result\
        "$indent2\},\n"\
        "${indent3}#ifndef MRT_NO_NAMES\n"\
        "$indent2.name = ${relName}__NAME\n"\
        "${indent3}#endif /* MRT_NO_NAMES */\n"\
        "$indent\},\n"
    continue
}
```

We use a helper function to compute the byte offset to link list pointers used for dynamic multiple reference pointer storage. If it's not a linked list storage scheme, then the offset is always 0.

```
<<generation helper commands>>=
proc FindRelOffsets {complRef} {
    set typeRef [findRelated $complRef {~R26 SingularReference}]
```

```
    set offset 0
    if {[isNotEmptyRef $typeRef]} {
        set type mrtSingular
    } else {
        set typeRef [findRelated $complRef {~R26 ArrayReference}]
        if {[isNotEmptyRef $typeRef]} {
            set type mrtArray
        } else {
            set typeRef [findRelated $complRef {~R26 LinkReference}]
            set type mrtLinkedList
            assignAttribute [findRelated $typeRef ~R27]\
                {Class lClass} {Name lcomp}
            set offset "offsetof([GetClassProperty $lClass Declaration], $lcomp)"
        }
    }
    return [list $type $offset]
}
```

Map separate conditionality and multiplicity values to a single encoded enumerator used by the run-time.

```
<<generation code commands>>=
proc MapToCardinality {cond mult} {
    if {$cond && !$mult} {
        return mrtAtMostOne
    } elseif {!$cond && !$mult} {
        return mrtExactlyOne
    } elseif {$cond && $mult} {
        return mrtZeroOrMore
    } elseif {!$cond && $mult} {
        return mrtOneOrMore
    }
}
```

Class Instance Definitions

```
<<generation code commands>>=
proc classInstanceDefinitions {} {
    variable domain
    variable staticMultiRefs {}

    set classRefs [FindNonUnionSubclasses $domain]
    set classpops [FindClassPopulation $classRefs]

    set result [comment "Instance Pool Definitions"]
    relation foreach classpop $classpops -ascending ClassNumber {
        relation assign $classpop

        append result\
            "static struct $Class "\
            [GetClassProperty $Class StorageVariable]\
            "\[[GetClassProperty $Class TotalInstance]\] = \{\n"

        relation foreach inst $Instances -ascending InstNumber {
            append result [GenInstanceInitializers $Class $ClassNumber $inst]
        }

        append result "\} ;\n"
    }
```

```
    if {$staticMultiRefs ne {}} {
        set staticMultiRefs [string cat\
            [comment "Static Reference Definitions"] $staticMultiRefs]
    }
    return [string cat $staticMultiRefs $result]
}
```

```
<<generation code commands>>=
proc FindClassPopulation {classRefs} {
    set classes [pipe {
        deRef $classRefs |
        relation rename ~ Name Class Number ClassNumber
    }]

    set compRefs [findRelated $classRefs ~R20]

    # Populated Components
    set popCompRefs [findRelated $compRefs {~R25 PopulatedComponent}]

    # Attribute Components
    set attrs [pipe {
        findRelated $popCompRefs {~R21 Attribute} {~R29 IndependentAttribute}\
                {~R19 ValueInitializedAttribute} R19 R29 |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $attrs attrs]

    # Reference Components
    # For the populated reference components we will go ahead and
    # compute the class to which the reference is made.
    set refCompRefs [findRelated $popCompRefs {~R21 Reference}]

    # Superclass Reference Component
    set screfs [findRelated $refCompRefs {~R23 SuperclassReference}]
    set refedsuper [pipe {
        findRelated $screfs ~R91 {~R47 ReferringSubclass} R37 ~R36 |
        deRef % |
        relation eliminate % Role |
        relation rename % Class SuperClass Relationship Component
    } {} |%]
    set superrefs [pipe {
        deRef $screfs |
        relation rename ~ Name Component |
        relation join ~ $refedsuper
    }]
    # puts [relformat $superrefs superrefs]

    # Associator Reference Components
    set arrefs [findRelated $refCompRefs {~R23 AssociatorReference}]
    set cbainfo [pipe {
        findRelated $arrefs ~R93 |
        deRef % |
        relation project % Domain Class Relationship |
        relation rename % Class AssociatorClass |
        relation join % $::micca::SourceClass |
        relation eliminate % Role Conditionality Multiplicity |
        relation rename % Class SourceClass |
        relation join % $::micca::TargetClass |
        relation eliminate % Role Conditionality Multiplicity |
        relation rename % Class TargetClass AssociatorClass Class\
                Relationship Component
```

```
    } {} |%]
    set atorrefs [pipe {
        deRef $arrefs |
        relation rename ~ Name Component |
        relation join ~ $cbainfo
    }]
    # puts [relformat $atorrefs atorrefs]

    # Association Reference Components
    set asrrefs [findRelated $refCompRefs {~R23 AssociationReference}]
    set srcinfo [pipe {
        findRelated $asrrefs ~R90 R32 ~R33 |
        deRef % |
        relation eliminate % Role |
        relation rename % Class ReferencedClass Relationship Component
    } {} |%]
    set assocrefs [pipe {
        deRef $asrrefs |
        relation rename ~ Name Component |
        relation join ~ $srcinfo
    }]
    # puts [relformat $assocrefs assocrefs]

    # Generated components
    set genCompRefs [findRelated $compRefs {~R25 GeneratedComponent}]

    # Subclass Reference Components
    set subrefs [pipe {
        findRelated $genCompRefs {~R24 SubclassReference} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $subrefs subrefs]

    # Subclass Container
    set subconts [pipe {
        findRelated $genCompRefs {~R24 SubclassContainer} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $subconts subconts]

    # Link Container
    set linkconts [pipe {
        findRelated $genCompRefs {~R24 LinkContainer} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $linkconts linkconts]

    # Complementary Reference
    set complRefs [findRelated $genCompRefs {~R24 ComplementaryReference}]

    # Singular Reference
    set singRefs [pipe {
        findRelated $complRefs {~R26 SingularReference} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $singRefs singRefs]

    # Array Reference
```

```
    set arrayRefs [pipe {
        findRelated $complRefs {~R26 ArrayReference} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $arrayRefs arrayRefs]

    # Link Reference
    set linkRefs [pipe {
        findRelated $complRefs {~R26 LinkReference} |
        deRef % |
        relation rename % Name Component
    } {} |%]
    # puts [relformat $linkRefs linkRefs]

    set classpops [pipe {
        findRelated $classRefs R104 {~R101 ElementPopulation}\
            {~R105 ClassPopulation} ~R102 |
        deRef % |
        relation rename % Number InstNumber |
        relation join % $classes $::micca::SpecifiedComponentValue |
        ralutil::rvajoin % $attrs Attributes |
        ralutil::rvajoin % $superrefs SuperRefs |
        ralutil::rvajoin % $atorrefs AssociatorRefs |
        ralutil::rvajoin % $assocrefs AssociationRefs |
        ralutil::rvajoin % $subrefs SubRefs |
        ralutil::rvajoin % $subconts SubclassContainers |
        ralutil::rvajoin % $linkconts LinkContainers |
        ralutil::rvajoin % $singRefs SingularRefs |
        ralutil::rvajoin % $arrayRefs ArrayRefs |
        ralutil::rvajoin % $linkRefs LinkRefs |
        relation group % Components Component Value Attributes SuperRefs\
            AssociatorRefs AssociationRefs SubRefs SubclassContainers\
            LinkContainers SingularRefs ArrayRefs LinkRefs |
        relation group % Instances Instance InstNumber Components |
        relation eliminate % Domain
    } {} |%]
    # puts [relformat $classpops classpops]

    return $classpops
}
```

```
<<generation code commands>>=
proc GenInstanceInitializers {className classNumber inst} {
    variable domain
    set indent [string repeat { } 4]
    set indent2 [string repeat $indent 2]
    set indent3 [string repeat $indent 3]
    set indent4 [string repeat $indent 4]

    relation assign $inst

    set sp [pipe {
        StateModel findById Domain $domain Model $className |
        findRelated ~ R58 R57
    }]
    set initstate [expr {[isNotEmptyRef $sp] ?\
        [readAttribute $sp Number] : "MRT_StateCode_IG"}]

    append result\
        "$indent\{ // $Instance\n"\
        "$indent2.base__INST = \{\n"\
```

```
        "$indent3.classDesc = &${domain}__CLASSES\[$classNumber\],\n"\
        "$indent3.alloc = [expr {$InstNumber + 1}],\n"\
        "$indent3.currentState = $initstate,\n"\
        "$indent3.refCount = 0,\n"\
        "${indent4}#ifndef MRT_NO_NAMES\n"\
        "$indent3.name = \"$Instance\"\n"\
        "${indent4}#endif /* MRT_NO_NAMES */\n"\
        " $indent2\},\n"

    relation foreach comp $Components {
        relation assign $comp
        if {[relation isnotempty $Attributes]} {
            append result "$indent2.$Component = $Value,\n"
            continue
        }

        if {[relation isnotempty $SuperRefs]} {
            set superclass [relation extract $SuperRefs SuperClass]
            append result\
                "$indent2.$Component = "\
                [GenInstanceAddress $domain $superclass $Value],\n
            continue
        }

        if {[relation isnotempty $AssociatorRefs]} {
            relation assign $AssociatorRefs
            # the painful reflexive case again!
            if {$SourceClass eq $TargetClass} {
                # N.B. the inversion. the target instance is the
                # one referenced in the forward direction
                set sinstname [dict get $Value backward]
                set tinstname [dict get $Value forward]
            } else {
                set sinstname [dict get $Value $SourceClass]
                set tinstname [dict get $Value $TargetClass]
            }
            set sourceaddr [GenInstanceAddress $domain $SourceClass\
                $sinstname]
            set targetaddr [GenInstanceAddress $domain $TargetClass\
                $tinstname]

            append result\
                "$indent2.$Component = \{\n"\
                "$indent3.forward = $targetaddr,\n"\
                "$indent3.backward = $sourceaddr\n"\
                "$indent2\},\n"
            continue
        }

        if {[relation isnotempty $AssociationRefs]} {
            set refvalue [GenInstanceAddress $domain\
                    [relation extract $AssociationRefs ReferencedClass]\
                    $Value]
            append result "$indent2.$Component = $refvalue,\n"
            continue
        }

        if {[relation isnotempty $SubRefs]} {
            lassign $Value subclass subinstname
            set subinstaddr [GenInstanceAddress $domain $subclass $subinstname]
            append result "$indent2.$Component = $subinstaddr,\n"
            continue
```

```
        }

        if {[relation isnotempty $SubclassContainers]} {
            lassign $Value subclass subinst

            set subRef [Class findWhere {$Domain eq $domain &&\
                $Name eq $subclass}]
            set subpops [FindClassPopulation $subRef]
            # puts [relformat $subpops subpops]

            relation assign $subpops {Class subclassName}\
                {ClassNumber subclassNumber} {Instances subInstances}
            set instpop [relation restrictwith $subInstances\
                {$Instance eq $subinst}]

            append result\
                "$indent2.$Component.$subclass = "\
                [string trimleft [indentCode [GenInstanceInitializers\
                    $subclassName $subclassNumber $instpop] 8]]
            continue
        }

        if {[relation isnotempty $LinkContainers]} {
            lassign [dict get $Value next] nclass ninst ncomp
            set nextaddr [GenInstanceAddress $domain $nclass $ninst]
            lassign [dict get $Value prev] pclass pinst pcomp
            set prevaddr [GenInstanceAddress $domain $pclass $pinst]
            append result\
                "$indent2.$Component = \{\n"\
                "$indent3.next = $nextaddr.$ncomp,\n"\
                "$indent3.prev = $prevaddr.$pcomp\n"\
                "$indent2\},\n"
            continue
        }

        if {[relation isnotempty $SingularRefs]} {
            if {$Value eq "@nil@"} {
                set refaddr NULL
            } else {
                lassign $Value refclass refinst
                set refaddr [GenInstanceAddress $domain $refclass $refinst]
            }
            append result "$indent2.$Component = $refaddr,\n"
            continue
        }

        if {[relation isnotempty $ArrayRefs]} {
            if {$Value eq "@nil@"} {
                set refcount 0
                set refaddr NULL
            } else {
                lassign $Value refclass refinsts
                set refcount [llength $refinsts]
                set refaddr ${Component}_${className}_${InstNumber}

                # Build reference array
                variable staticMultiRefs
                append staticMultiRefs\
                    "static [GetClassProperty $refclass Reference]\
                        const $refaddr\[$refcount\] = \{\n"
                foreach inst $refinsts {
                    append staticMultiRefs\
```

```
                        "$indent[GenInstanceAddress $domain $refclass $inst],\n"
                }
                append staticMultiRefs "\} ;\n"
            }
            append result\
                "$indent2.$Component = \{\n"\
                "$indent3.links = $refaddr,\n"\
                "$indent3.count = $refcount\n"\
                "$indent2\},\n"
            continue
        }

        if {[relation isnotempty $LinkRefs]} {
            lassign [dict get $Value next] nclass ninst ncomp
            set nextaddr [GenInstanceAddress $domain $nclass $ninst]
            lassign [dict get $Value prev] pclass pinst pcomp
            set prevaddr [GenInstanceAddress $domain $pclass $pinst]
            append result\
                "$indent2.$Component = \{\n"\
                "$indent3.next = $nextaddr.$ncomp,\n"\
                "$indent3.prev = $prevaddr.$pcomp\n"\
                "$indent2\},\n"
            continue
        }
    }

    append result "$indent\},\n"

    return $result
}
```

Assigner Instance Definitions

```
<<generation code commands>>=
proc assignerInstanceDefinitions {} {
    variable domain
    set result {}

    append result [comment "Single Assigner Instance Definitions"]
    set singles [SingleAssigner findWhere {$Domain eq $domain}]
    forAllRefs assigner $singles {
        set initstate [pipe {
            findRelated $assigner R53 R50 R58 R57 |
            readAttribute ~ Number
        }]
        assignAttribute $assigner
        set instnum [GetClassProperty $Association Number]
        append result\
            "static struct $Association ${Association}__POOL\[1\] = \{\n"\
            "    \{\n"\
            "        .base__INST = \{\n"\
            "            .classDesc = &${domain}__ASSIGNERS\[$instnum\],\n"\
            "            .alloc = 1,\n"\
            "            .currentState = $initstate,\n"\
            "            .refCount = 0,\n"\
            "                #ifndef MRT_NO_NAMES\n"\
            "            .name = single_assigner_instance__NAME\n"\
            "                #endif /* MRT_NO_NAMES */\n"\
            "        \}\n"\
```

```
            "    \}\n"\
            "\} ;\n"
    }

    append result [comment "Multiple Assigner Instance Definitions"]
    set multis [MultipleAssigner findWhere {$Domain eq $domain}]
    forAllRefs multi $multis {
        assignAttribute $multi
        set initstate [pipe {
            findRelated $multi R53 R50 R58 R57 |
            readAttribute ~ Number
        }]

        set multiinsts [findRelated $multi ~R106]
        if {[isNotEmptyRef $multiinsts]} {
            set totalinsts [GetClassProperty $Class TotalInstance]
            set instnum [GetClassProperty $Association Number]
            append result "static struct $Association\
                    ${Association}__POOL\[$totalinsts\] = \{\n"
            forAllRefs assigner $multiinsts {
                assignAttribute $assigner
                set allocnum 0
                append result\
                    "    \[$Number\] = \{\n"\
                    "        .base__INST = \{\n"\
                    "            .classDesc = &${domain}__ASSIGNERS\[$instnum\],\n"\
                    "            .alloc = [incr allocnum],\n"\
                    "            .currentState = $initstate,\n"\
                    "            .refCount = 0,\n"\
                    "                #ifndef MRT_NO_NAMES\n"\
                    "            .name = \"$Instance\"\n"\
                    "                #endif /* MRT_NO_NAMES */\n"\
                    "        \},\n"\
                    "        .idinstance = [GenInstanceAddress $domain $IdClass\
                           $IdInstance]\n"\
                    "    \},\n"\
            }
            append result "\} ;\n"
        }
    }

    return $result
}
```

Operation Definitions

```
<<generation code commands>>=
proc operationDefinitions {} {
    variable domain

    set ops [pipe {
        Operation findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $::micca::OperationParameter Parameters
    }]

    set result [comment "Operation Definitions"]
    relation foreach op $ops {
```

```
        relation assign $op
        append result\
            "static $ReturnDataType\n${Class}_$Operation"

        if {[relation isempty $Parameters]} {
            append result "\(void\)\n"
            set syms [relation create {
                Name string Ctype string Type string Class string
            }]
        } else {
            set pdecls {}
            relation foreach param $Parameters -ascending Number {
                relation assign $param
                append pdecls\
                    [typeCheck composeDeclaration $DataType $Name]\
                    ,\n
            }
            set pdecls [string trimright $pdecls ",\n"]\)\n
            append result\
                "\(\n"\
                [indentCode $pdecls]

            set syms [pipe {
                relation project $Parameters Name DataType |
                relation rename ~ DataType Ctype |
                relation extend ~ stup\
                    Type string {{}}\
                    Class string {{}} |
                relation update ~ utup {
                    [tuple extract $utup Name] eq "self"} {
                    tuple update $utup Type Reference Class $Class
                }
            }]
        }

        append result\
            [blockcomment $Body]\
            "\{\n"\
            "    MRT_INSTRUMENT_ENTRY\n"\
            [linedirective $Line $File]\
            [ExpandActivity "$Operation operation" $Body $syms]\
            "\}\n"
    }

    return $result
}
```

Constructor Definitions

```
<<generation code commands>>=
proc ctorDefinitions {} {
    return [TorDefinitions Constructor]
}
```

```
<<generation code commands>>=
proc TorDefinitions {which} {
    variable domain
    set suffixmap [dict create\
        Constructor CTOR\
```

```
        Destructor DTOR\
    ]
    set result [comment "$which Definitions"]

    set tors [$which findWhere {$Domain eq $domain}]

    forAllRefs tor $tors {
        assignAttribute $tor

        set selfdecl [GetClassProperty $Class Reference]const
        set syms [relation create {
            Name string Ctype string Type string Class string
        } [list Name self Ctype $selfdecl Type Reference Class $Class]]
        append result\
            "static void\n${Class}__[dict get $suffixmap $which]\(\n"\
            "    void *const s__SELF\)\n"\
            [blockcomment $Body]\
            "\{\n"\
            "    MRT_INSTRUMENT_ENTRY\n"\
            "    $selfdecl self = s__SELF ;\n"\
            [linedirective $Line $File]\
            [ExpandActivity "$Class $which" $Body $syms]\
            "\}\n"
    }

    return $result
}
```

Destructor Definitions

```
<<generation code commands>>=
proc dtorDefinitions {} {
    return [TorDefinitions Destructor]
}
```

Formula Definitions

```
<<generation code commands>>=
proc formulaDefinitions {} {
    variable domain
    set result [comment "Dependent Attribute Formula Definitions"]

    set deps [DependentAttribute findWhere {$Domain eq $domain}]

    forAllRefs dep $deps {
        assignAttribute $dep
        set type [readAttribute [findRelated $dep R29] DataType]

        set selfdecl [GetClassProperty $Class ConstReference]const
        set syms [relation create {
            Name string Ctype string Type string Class string
        }\
            [list Name self Ctype $selfdecl Type Reference Class $Class]\
            [list Name $Name Ctype "void *const" Type {} Class {}]\
            [list Name size Ctype MRT_AttrSize Type {} Class {}]\
        ]
```

```
        set asgnType [typeCheck assignmentType [UnaliasType $domain $type]]
        switch -exact -- [dict get $asgnType type] {
            scalar {
                set attrDecl "$type *const $Name"
            }
            string -
            array {
                set attrDecl "[dict get $asgnType base] *const $Name"
            }
            default {
                error "unknown assignment type, \"[dict get $asgnType type]\""
            }
        }
        append result\
            "static void\n${Class}_${Name}__FORMULA\(\n"\
            "    void const *const s__,\n"\
            "    void *const v__,\n"\
            "    MRT_AttrSize size\)\n"\
            [blockcomment $Formula]\
            "\{\n"\
            "    MRT_INSTRUMENT_ENTRY\n"\
            "    $selfdecl self = s__ ;\n"\
            "    $attrDecl = v__ ;\n"\
            [linedirective $Line $File]\
            [ExpandActivity "$Class $Name Formula" $Formula $syms]\
            "\}\n"
    }

    return $result
}
```

Activity Definitions

```
<<generation code commands>>=
proc activityDefinitions {} {
    variable domain
    set result {}

    set action_states_ref [State findWhere\
            {$Domain eq $domain && [string trim $Activity] ne {}}]

    if {[isNotEmptyRef $action_states_ref]} {
        set action_classes [pipe {
            deRef $action_states_ref |
            relation eliminate ~ Domain IsFinal |
            relation group ~ States Name Activity File Line
        }]
        #puts [relformat $action_classes action_classes]

        append result [comment "State Activities Definitions"]

        relation foreach class $action_classes -ascending Model {
            relation assign $class
            append result "#define MRT_CLASS_NAME \"$Model\"\n"
            relation foreach state $States {
                relation assign $state
                set selfdecl [GetClassProperty $Model Reference]const
                append result\
                    "static void\n${Model}_${Name}__ACTIVITY\(\n"\
```

```
                    "    void *const s__SELF,\n"\
                    "    void const *const p__PARAMS)\n"\
                    [blockcomment $Activity]\
                    "\{\n"\
                    "        #define MRT_STATE_NAME \"$Name\"\n"\
                    "    MRT_INSTRUMENT_ENTRY\n"\
                    "    $selfdecl self = s__SELF ;\n"

                set statesig [pipe {
                    relation restrict $::micca::StateSignature ssig_tup {
                        [tuple extract $ssig_tup Domain] eq $domain &&
                        [tuple extract $ssig_tup Model] eq $Model &&
                        [tuple extract $ssig_tup State] eq $Name} |
                    relation semijoin ~ $::micca::ParameterSignature |
                    ::rosea::Helpers::ToRef ::micca::ParameterSignature ~
                }]

                if {[isNotEmptyRef $statesig]} {
                    set signame ${Model}_${Name}__SPARAMS
                    append result\
                        "    struct $signame const *const pp__PARAMS = p__PARAMS ;\n"
                    set params [FindParamsFromSig $statesig]
                    relation foreach param $params -ascending Position {
                        set asgnment [GenParamAssignment $domain $param]
                        append result [indentCode $asgnment]
                    }

                    set syms [pipe {
                        relation project $params Name Declaration |
                        relation rename ~ Declaration Ctype |
                        relation extend ~ stup\
                            Type string {{}}\
                            Class string {{}}
                    }]
                } else {
                    set syms [relation create {
                        Name string Ctype string Type string Class string
                    }]
                }
                set syms [relation insert $syms [list Name self Ctype $selfdecl\
                        Type Reference Class $Model]]

                append result\
                    [linedirective $Line $File]\
                    [ExpandActivity "$Model $Name activity" $Activity $syms]

                append result\
                    "        #undef MRT_STATE_NAME\n"\
                    "\}\n"
            }
            append result "#undef MRT_CLASS_NAME\n"
        }
    }

    return $result
}
```

```
<<generation helper commands>>=
proc FindParamsFromSig {psigRef} {
    return [pipe {
        findRelated $psigRef ~R79 |
        deRef % |
```

```
        relation join % $::micca::Argument |
        relation extend % ptup Declaration string {\
            [typeCheck composeDeclaration\
                [tuple extract $ptup DataType]\
                [tuple extract $ptup Name]]}
    } {} |%]
}
```

```
<<generation helper commands>>=
proc GenValueAssignment {domain dest src datatype} {
    set asgnType [typeCheck assignmentType [UnaliasType $domain $datatype]]
    switch -exact -- [dict get $asgnType type] {
        scalar {
            append result "$dest = $src ;\n"
        }
        string {
            set dimension [dict get $asgnType dimension]
            set maxchars [expr {$dimension - 1}]
            append result\
                "strncpy($dest, $src, $maxchars) ;\n"\
                "$dest\[$maxchars\] = '\\0' ;\n"
        }
        array {
            append result "memcpy($dest, $src, sizeof($dest)) ;\n"
        }
        default {
            error "unknown assignment type, \"[dict get $asgnType type]\""
        }
    }
    return $result
}

proc GenParamAssignment {domain param} {
    relation assign $param Name Declaration DataType

    set asgnType [typeCheck assignmentType [UnaliasType $domain $DataType]]
    switch -exact -- [dict get $asgnType type] {
        scalar {
            set result [string cat\
                "$Declaration ;\n"\
                "$Name = pp__PARAMS->$Name ;\n"\
            ]
        }
        string -
        array {
            set result "[dict get $asgnType base] const *const $Name =\
                    pp__PARAMS->$Name ;\n"
        }
        default {
            error "unknown assignment type, \"[dict get $asgnType type]\""
        }
    }
    return $result
}
```

Domain Constructor Definition

```
<<generation code commands>>=
proc domainCtorDefinition {} {
```

```
    variable domain
    set result [comment "Definition of Function to Construct Initial Instances"]

    set ctorinsts [pipe {
        Constructor findWhere {$Domain eq $domain} |
        deRef ~ |
        relation eliminate ~ Body |
        relation join ~ $::micca::ClassInstance |
        relation group ~ Instances Instance Number
    }]

    if {[relation isnotempty $ctorinsts]} {
        relation foreach ctorinst $ctorinsts {
            relation assign $ctorinst
            relation foreach instance $Instances -ascending Number {
                relation assign $instance
                append invocations\
                    "${Class}__CTOR\("\
                    [GenInstanceAddress $domain $Class $Instance]\
                    "\) ;\n"
            }
        }
        append result\
            "static void\n${domain}__INIT\(void\)\n"\
            "\{\n"\
            [indentCode $invocations]\
            "\}\n"
    }

    return $result
}
```

Domain Operation Definitions

```
<<generation code commands>>=
proc domainOpDefinitions {} {
    variable domain
    set result [comment "Domain Operation Definitions"]
    set extern {}

    set ops [pipe {
        DomainOperation findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $::micca::DomainOperationParameter Parameters
    }]

    relation foreach op $ops {
        relation assign $op
        set code "$ReturnDataType\n${domain}_${Operation}"
        append result "static ${code}__SS"
        append extern $code

        if {$ReturnDataType eq "void"} {
            set invocation {}
            set retstmt {}
        } else {
            set invocation \
                "[typeCheck composeDeclaration $ReturnDataType result] = "
```

```
            set retstmt "return result ;\n"
        }
        append invocation "${domain}_${Operation}__SS\("

        if {[relation isempty $Parameters]} {
            set decl "\(void\)\n"
            append result $decl
            append extern $decl
            append invocation "\) ;\n"
            set syms [relation create {
                Name string Ctype string Type string Class string
            }]
        } else {
            set pdecls {}
            set pinvoke {}
            relation foreach param $Parameters -ascending Number {
                relation assign $param
                append pdecls\
                    [typeCheck composeDeclaration $DataType $Name]\
                    ,\n
                append pinvoke $Name ", "
            }
            set pdecls [string trimright $pdecls ",\n"]\)\n
            set pinvoke [string trimright $pinvoke ", "]
            set decl [string cat\
                "\(\n"\
                [indentCode $pdecls]\
            ]
            append result $decl
            append extern $decl
            append invocation "$pinvoke\) ;\n"
            set syms [pipe {
                relation project $Parameters Name DataType |
                relation rename ~ DataType Ctype |
                relation extend ~ stup\
                    Type string {{}}\
                    Class string {{}}
            }]
        }

        append result\
            [blockcomment $Body]\
            "\{\n"\
            "    MRT_INSTRUMENT_ENTRY\n"\
            [linedirective $Line $File]\
            [ExpandActivity "$Operation domain operation" $Body $syms]\
            "\}\n"
        append extern\
            "\{\n"\
            "    mrt_BeginSyncService() ;\n"\
            "    $invocation"\
            "    mrt_EndSyncService() ;\n"\
            [expr {$retstmt eq {} ? {} : "    $retstmt"}]\
            "\}\n"
    }
    append result $extern

    return $result
}
```

External Operation Definitions

```
<<generation code commands>>=
proc externalOpDefinitions {} {
    upvar #0 [namespace parent]::options options
    if {![dict get $options stubexternalops]} {
        return
    }

    variable domain
    set result [comment "External Operation Stub Definitions"]

    set ops [pipe {
        ExternalOperation findWhere {$Domain eq $domain} |
        deRef ~ |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $::micca::ExternalOperationParameter Parameters
    }]

    relation foreach op $ops {
        relation assign $op
        append result\
            "$ReturnDataType\n${domain}_${Entity}_${Operation}__EOP"

        if {[relation isempty $Parameters]} {
            append result "\(void\)\n"
            set syms [relation create {
                Name string Ctype string Type string Class string
            }]
        } else {
            set pdecls {}
            relation foreach param $Parameters -ascending Number {
                relation assign $param
                append pdecls\
                    [typeCheck composeDeclaration $DataType $Name]\
                    ,\n
            }
            set pdecls [string trimright $pdecls ",\n"]\)\n
            append result\
                "\(\n"\
                [indentCode $pdecls]
            set syms [pipe {
                relation project $Parameters Name DataType |
                relation rename ~ DataType Ctype |
                relation extend ~ stup\
                    Type string {{}}\
                    Class string {{}}
            }]
        }

        append result\
            [blockcomment $Body]\
            "\{\n"\
            "    MRT_INSTRUMENT_ENTRY\n"\
            [linedirective $Line $File]\
            [ExpandActivity "$Operation external operation" $Body $syms]\
            "\}\n"
    }

    return $result
}
```

Portal Data Definitions

```
<<generation code commands>>=
proc portalDefinition {} {
    variable domain

    set classRefs [Class findWhere {$Domain eq $domain}]
    set assignRefs [AssignerStateModel findWhere {$Domain eq $domain}]
    set acount [refMultiplicity $assignRefs]
    set aptr [expr {$acount == 0 ? "NULL" : "${domain}__ASSIGNERS"}]
    append result\
        [comment "Domain Portal Definition"]\
        "MRT_DomainPortal const ${domain}__PORTAL = \{\n"\
        "    .classCount = [refMultiplicity $classRefs],\n"\
        "    .classes = ${domain}__CLASSES,\n"\
        "    .assignerCount = $acount,\n"\
        "    .assigners = $aptr,\n"\
        "        #ifndef MRT_NO_NAMES\n"\
        "    .name = \"$domain\"\n"\
        "        #endif /* MRT_NO_NAMES */\n"\
        "\} ;\n"
}
```

Epilogue Declarations

```
<<generation code commands>>=
proc epilogueDeclarations {} {
    variable epilogue

    return [string cat\
        [comment "Domain Epilogue"]\
        [indentCode [string trim $epilogue] 0]\
    ]
}
```

Chapter 40

Generating Activity Code

In this section we discuss the macro commands that are available to a translator for easing the burden of translating model level execution into the required "C" code. When using `micca` to translate a model, state activities are coded in "C" and passed along to the generated code file. To make coding model level actions easier, the "C" code supplied by the translator for state activities and other operations can contain embedded macros which are further expanded by the code generator. The macros are simple commands with arguments that are enclosed in the special delimiters **<%** and **%>**. For example, the macro to access the **Color** attribute of an instance could be written as:

```
ColorType c = <%my attr Color%> ;
```

For this example, the code generator expands the commands contained between the **<%** and **%>** delimiters and replaces the embedded macro with "C" code required to access the value of the **Color** attribute of the instance referenced by the value of the **self** variable. This is a very simple case, but even here the code generator performs a number of significant checks in an attempt to discover errors before code is sent to the compiler. In this case, the code generator insures that the attribute named **Color** exists for the class to which the `self` instance refers.

There are many of these macro commands to handle common model level processing. State activities are then a mix of "C" code and embedded macros commands to accomplish the purpose of the activity. The straight "C" contains the algorithmic processing and the expanded macros handle the model level actions such as navigating relationships, finding instances and dealing with attribute updates. These types of model level actions must account for the data structures and naming conventions that the run-time code uses. Since the code generator has access to the populated platform model, it performs many consistency checks that would otherwise result in compiler errors. Discovering errors earlier makes them much easier to correct since relating compiler error messages back to the `micca` source becomes more difficult in the face of another level of translation.

The macro commands expand to one of three types of "C" constructs:

1. A macro command may expand to a statement in "C". For statement macros, the expansion yields one or more lines of code and the code generator takes care of the semicolon punctuation.
2. A macro command may expand to an expression in "C". For expression macros, the code generator just replaces the macro and does not add any punctuation. This enables a expression macro to be used inside of another "C" expression.
3. A macro command may expand to a construct in "C" requiring a compound statement. For macros generating compound statements, additional "C" code is included in the compound statement body and the end of the loop is given by invoking the `<%end%>` macro. Typically, the these macros allow iteration over sets of instances and the compound statement body is executed once for each instance.

In the descriptions below we state the expansion type of each macro.

Expanding the Embedded Activity Commands

Just as we have been using the `textutil::expander` command to perform macro expansion for generating the header and code files, we will use it to expand the embedded macros in the state activities. We will use a separate instance of the the expander

and will direct this expansion to happen in its own namespace. The namespace will allow us to define the macro commands so that the commands will resolve with unqualified names. We will also use a namespace to handle the details of the expansion code itself. Following our usual pattern, we define the namespace for the generation of the embedded macros.

```
<<generation support namespace>>=
namespace eval GenSupport {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>
    namespace import ::ral::relvar

    namespace export ExpandActivity

    namespace path {
        ::micca
        ::micca::@Gen@::Helpers
        ::micca::@Config@::Helpers
        ::rosea::InstCmds
    }
    upvar #0 ::micca::@Gen@::errcount errcount

    textutil::expander actexpand
    actexpand setbrackets <% %> ; # ❶
    actexpand evalcmd "namespace eval [namespace parent]::GenActivity"
    actexpand errmode fail
    actexpand textcmd [namespace current]::ProcessCodeLines

    <<generation support data>>
    <<generation support commands>>
}
```

❶ The default way to embed commands in the template is with the conventional Tcl brackets ([ ]). Those characters are inconvenient in "C" so we use bracket strings that create less interference.

Creating the Embedded Commands

The commands available for embedding into "C" activity code are organized into groups.

- A command to invoke external operations.
- Commands related to classes.
- Commands related to relationships.
- Commands related to instances.
- Commands related to instance sets.

Each of these different command groups are discussed in sections below. Since we are using a template expander to expand the commands, in this section we show how the expansion commands are created.

For classes and relationships, embedded macro commands named after the classes and relationships themselves are created. For example, if the domain contains a class named `WashingMachine`, then there will be an embedded command named `WashingMachine` that can be used to access class level functions specific to the `WashingMachine` class. The same holds true for a relationship. If `R27` exists in the domain, then there will be an embedded command named `R27` that can be used to relate and unrelate instances across the relationship. To insure that there is no conflict in names, we will create the embedded commands in their own namespace, `GenActivity` as a child namespace of `GenSupport`. Since all classes (or relationships) have the same set of subcommands available to them, we will namespace ensembles to obtain the "object oriented" Tcl style for the embedded commands.

```
<<generation commands>>=
proc CreateActivityCommands {domain} {
    namespace eval GenActivity {
        ::logger::import -all -force -namespace log micca
    }

    set currns [namespace current]
    set actns ${currns}::GenActivity
    set suppns ${currns}::GenSupport

    <<CreateActivityCommands: classes>>
    <<CreateActivityCommands: relationships>>
    <<CreateActivityCommands: instances>>
    <<CreateActivityCommands: instance sets>>
    <<CreateActivityCommands: external operations>>

    interp alias {} ${actns}::end {} ${suppns}::End
    interp alias {} ${actns}::subclass {} ${suppns}::SubclassCase
    interp alias {} ${actns}::default {} ${suppns}::DefaultSubclass
}
```

We see that the code is sectioned into four chunks based on what the namespace ensemble command refers to.

First, we query some information about the classes in the domain we are generating. The class commands will vary depending upon: \(1) the class having a defined state model and \(2) if a class with a state model has creation events.

```
<<CreateActivityCommands: classes>>=
set classInfo [pipe {
    Class findWhere {$Domain eq $domain} |
    deRef ~ |
    ralutil::rvajoin ~ [relation rename $::micca::StateModel Model Name]\
        StateModels |
    relation extend ~ itup HasStateModel boolean {
        [relation isnotempty [tuple extract $itup StateModels]]} |
    ralutil::rvajoin ~\
        [relation rename $::micca::CreationState Name State Model Name]\
        CreationStates |
    relation extend ~ ctup HasCreationState boolean {
        [relation isnotempty [tuple extract $ctup CreationStates]]} |
    relation project ~ Name HasStateModel HasCreationState
}]
```

We first need to find out all the union subclass names. As usual, union subclasses have to be treated a bit differently. A union subclass cannot be created either synchronously or asynchronously. A union subclass can only be migrated as part of a generalization.

```
<<CreateActivityCommands: classes>>=
set usubnames [pipe {
    UnionSubclass findWhere {$Domain eq $domain} |
    deRef ~ |
    relation list ~ Class
}]
```

We now iterate over the classes for the domain and construct the namespace ensemble command map. Command maps are the key to ensemble commands and allow you to forward the command invocation with additional information (in this case the name of the class).

```
<<CreateActivityCommands: classes>>=
set cmdmap [dict create]
relation foreach info $classInfo {
    relation assign $info
```

```
    set cmdmap [dict create\
        foreachInstance [list ${suppns}::ClassForeachInstance $Name]\
        foreachWhere [list ${suppns}::ClassForeachWhere $Name]\
        findWhere [list ${suppns}::ClassFindWhere $Name]\
        selectWhere [list ${suppns}::ClassSelectWhere $Name]\
        refvar [list ${suppns}::ClassInstanceReference $Name]\
        idtoref [list ${suppns}::ClassIdToRef $Name]\
        instset [list ${suppns}::ClassInstanceSet $Name]\
        findByName [list ${suppns}::ClassFindByName $Name]\
        operation [list ${suppns}::ClassOperation $Name]\
    ]
    if {$Name ni $usubnames} {
        dict set cmdmap create [list ${suppns}::ClassCreate $Name]
        if {$HasStateModel} {
            dict set cmdmap createin [list ${suppns}::ClassCreateIn $Name]
        }
        if {$HasCreationState} {
            dict set cmdmap createasync [list ${suppns}::ClassCreateAsync $Name]
        }
    } else {
        dict set cmdmap create [list ${suppns}::USubClassCreate $Name]
        if {$HasStateModel} {
            dict set cmdmap createin [list ${suppns}::USubClassCreateIn $Name]
        }
        if {$HasCreationState} {
            dict set cmdmap createasync\
                [list ${suppns}::USubClassCreateAsync $Name]
        }
    }
    namespace ensemble create\
        -command ${actns}::$Name\
        -map $cmdmap
}
```

For relationships, we must consider each type of relationships separately. As shown in the platform model, the top level specialization of relationships is between associations and generalizations.

```
<<CreateActivityCommands: relationships>>=
set relRefs [Relationship findWhere {$Domain eq $domain}]

<<CreateActivityCommands: associations>>
<<CreateActivityCommands: generalizations>>
```

The functions for association commands deal primarily with updating references to instances across the association. Associations can also have an assigner defined for them. In that case they need to be able to signal an event to the assigner. In the case of a multiple assigner, the relationship command also need to support creating an instance of the assigner. Associations are further specialized into simple associations and class based associations. All the permutations of these commands are considered below.

```
<<CreateActivityCommands: associations>>=
set assocs [findRelatedWhere $relRefs {{~R30 Association}} {!$IsStatic}]

<<CreateActivityCommands: simple associations>>
<<CreateActivityCommands: class based associations>>
```

```
<<CreateActivityCommands: simple associations>>=
set simpassocs [findRelated $assocs {~R31 SimpleAssociation}]
forAllRefs simpassoc $simpassocs {
    set simpname [readAttribute $simpassoc Name]
    set cmdmap [dict create\
        reference [list ${suppns}::RelSimpleAssocSwap $simpname]\
    ]
```

```
    set assigner [findRelated $simpassoc R31 ~R52]
    if {[isNotEmptyRef $assigner]} {
        if {[isNotEmptyRef [findRelated $assigner {~R53 SingleAssigner}]]} {
            dict set cmdmap signal\
                [list ${suppns}::SingleAssignerSignal $simpname]
        } else {
            dict set cmdmap findByIdInstance\
                    [list ${suppns}::MultiAssignerFindIdInstance $simpname]
            dict set cmdmap create [list ${suppns}::MultiAssignerCreate $simpname]
        }
    }
    namespace ensemble create\
        -command ${actns}::$simpname\
        -map $cmdmap
}
```

```
<<CreateActivityCommands: class based associations>>=
set classassocs [findRelated $assocs {~R31 ClassBasedAssociation}]
forAllRefs classassoc $classassocs {
    set cbname [readAttribute $classassoc Name]
    set cmdmap [dict create\
        reference [list ${suppns}::RelClassAssocSwap $cbname]\
    ]
    set assigner [findRelated $classassoc R31 ~R52]
    if {[isNotEmptyRef $assigner]} {
        if {[isNotEmptyRef [findRelated $assigner {~R53 SingleAssigner}]]} {
            dict set cmdmap signal\
                [list ${suppns}::SingleAssignerSignal $cbname]
        } else {
            dict set cmdmap findByIdInstance\
                    [list ${suppns}::MultiAssignerFindIdInstance $cbname]
            dict set cmdmap create [list ${suppns}::MultiAssignerCreate $cbname]
        }
    }
    namespace ensemble create\
        -command ${actns}::$cbname\
        -map $cmdmap
}
```

Generalization relationships are specialized into those using references and those using unions. The command distinction is that union generalizations only support a single subcommand, namely `migrate`. Given the storage arrangement for union generalizations, relating and unrelating instances does not make sense since subclass instances are bound to the storage of their superclass instances..

```
<<CreateActivityCommands: generalizations>>=
set gens [findRelated $relRefs {~R30 Generalization}]

<<CreateActivityCommands: reference generalizations>>
<<CreateActivityCommands: union generalizations>>
```

```
<<CreateActivityCommands: reference generalizations>>=
set refgens [findRelated $gens {~R43 ReferenceGeneralization}]
foreach rgname [relation list [deRef $refgens] Name] {
    set cmdmap [dict create\
        classify [list ${suppns}::GenClassify $rgname]\
        reclassify [list ${suppns}::RelRefGenReclassify $rgname]\
    ]
    namespace ensemble create\
        -command ${actns}::$rgname\
        -map $cmdmap
}
```

```
<<CreateActivityCommands: union generalizations>>=
set uniongens [findRelated $gens {~R43 UnionGeneralization}]
foreach ugname [relation list [deRef $uniongens] Name] {
    set cmdmap [dict create\
        classify [list ${suppns}::GenClassify $ugname]\
        reclassify [list ${suppns}::RelUnionGenReclassify $ugname]\
    ]
    namespace ensemble create\
        -command ${actns}::$ugname\
        -map $cmdmap
}
```

The ensemble command for class instances is much simpler. All instances have the same set of subcommands and they are simply enumerated in the command map of the ensemble.

```
<<CreateActivityCommands: instances>>=
set cmdmap [dict create\
    attr ${suppns}::InstanceAttrRead\
    update ${suppns}::InstanceAttrUpdate\
    assign ${suppns}::InstanceAssign\
    signal ${suppns}::InstanceSignal\
    delaysignal ${suppns}::InstanceDelaySignal\
    periodicsignal ${suppns}::InstancePeriodicSignal\
    canceldelayed ${suppns}::InstanceCancelSignal\
    delayremaining ${suppns}::InstanceRemainingTime\
    delete ${suppns}::InstanceDelete\
    operation ${suppns}::InstanceOperation\
    foreachRelated ${suppns}::InstanceForeachRelated\
    foreachRelatedWhere ${suppns}::InstanceForeachRelatedWhere\
    findRelatedWhere ${suppns}::InstanceFindRelatedWhere\
    findOneRelated ${suppns}::InstanceFindOneRelated\
    selectRelated ${suppns}::InstanceSetSelectRelated\
    selectRelatedWhere ${suppns}::InstanceSetSelectRelatedWhere\
    instid ${suppns}::InstanceRefToId\
]
namespace ensemble create\
    -command ${actns}::instance\
    -parameters instref\
    -map $cmdmap
```

To save some typing, we create an ensemble command named `my` which is just a shorthand for `instance self`.

```
<<CreateActivityCommands: instances>>=
set cmdmap [dict create\
    attr [list ${suppns}::InstanceAttrRead self]\
    update [list ${suppns}::InstanceAttrUpdate self]\
    assign [list ${suppns}::InstanceAssign self]\
    signal [list ${suppns}::InstanceSignal self]\
    delaysignal [list ${suppns}::InstanceDelaySignal self]\
    periodicsignal [list ${suppns}::InstancePeriodicSignal self]\
    canceldelayed [list ${suppns}::InstanceCancelSignal self]\
    delayremaining [list ${suppns}::InstanceRemainingTime self]\
    delete [list ${suppns}::InstanceDelete self]\
    operation [list ${suppns}::InstanceOperation self]\
    foreachRelated [list ${suppns}::InstanceForeachRelated self]\
    foreachRelatedWhere [list ${suppns}::InstanceForeachRelatedWhere self]\
    findRelatedWhere [list ${suppns}::InstanceFindRelatedWhere self]\
    findOneRelated [list ${suppns}::InstanceFindOneRelated self]\
    selectRelated [list ${suppns}::InstanceSetSelectRelated self]\
    selectRelatedWhere [list ${suppns}::InstanceSetSelectRelatedWhere self]\
    instid [list ${suppns}::InstanceRefToId self]\
```

```
]
namespace ensemble create\
    -command ${actns}::my\
    -map $cmdmap
```

Instance sets also have a fixed set of subcommand operations.

```
<<CreateActivityCommands: instance sets>>=
set cmdmap [dict create\
    foreachInstance ${suppns}::InstanceSetForeachInstance\
    selectOneInstance ${suppns}::InstanceSetSelectOneInstance\
    empty ${suppns}::InstanceSetEmpty\
    cardinality ${suppns}::InstanceSetCardinality\
    notempty ${suppns}::InstanceSetNotEmpty\
    equal ${suppns}::InstanceSetEqual\
    notequal ${suppns}::InstanceSetNotEqual\
    add ${suppns}::InstanceSetAdd\
    remove ${suppns}::InstanceSetRemove\
    contains ${suppns}::InstanceSetContains\
    union ${suppns}::InstanceSetUnion\
    intersect ${suppns}::InstanceSetIntersect\
    minus ${suppns}::InstanceSetMinus\
]
namespace ensemble create\
    -command ${actns}::instset\
    -parameters instset\
    -map $cmdmap
```

```
<<CreateActivityCommands: external operations>>=
set ops [pipe {
    ExternalOperation findWhere {$Domain eq $domain} |
    deRef ~ |
    relation project ~ Domain Entity Name
}]
set eops [pipe {
    ExternalEntity findWhere {$Domain eq $domain} |
    deRef ~ |
    relation rename ~ Name Entity |
    ralutil::rvajoin ~ $ops Operations
}]

relation foreach eop $eops {
    relation assign $eop

    set cmdmap [dict create]
    relation foreach op $Operations {
        relation assign $op Name
        dict set cmdmap $Name\
                [list ${suppns}::InvokeExternalOp $Entity $Name]
    }

    namespace ensemble create\
        -command ${actns}::$Entity\
        -map $cmdmap
}
```

Symbol Table

The embedded macro commands have need to keep track of “C” variables. So we provide some rudamentary symbol table support. We must also account for the scoping rules for identifiers in “C”. In modern dialects of “C”, automatic variables may

be declared almost anywhere in a function. Each compound statement introduces a new variable scope and variables declared within that scope go out of scope at the end of the compound statement. Further, a variable declared in an interior scope may *hide* a variable of the same name in an outer scope. Although this language feature may have some uses, we will avoid it here. Embedded commands that declare variables will not reuse a variable name and thus risk hiding a variable by the same name in some outer scope. In the case where a variable name is reused and there is a conflict in the type of the variable, an error is generated.

We also have to track "C" language scope in order to remove variables from the symbol table as they go out of scope. Otherwise, the generated code would elicit compiler errors by referring to variables that a no longer in scope.

We hold the symbol table information in a relation variable.

```
<<generation support data>>=
relvar create Symbol {
    Name string
    Ctype string
    Type string
    Class string
    Block int
} Name
```

Name
: The name of a variable.

Ctype
: The "C" language type name for the variable.

Type
: The type of the variable as it is used in the code generation. This may be the empty string if we do not know its usage from the context.

Class
: The name of the class to which the variable pertains, if any.

Block
: The block number in which the variable was declared.

We will also be creating temporary variables during the code generation and will use a simple counter to make the names unique. The counter is reset back to zero after each activity's code is generated in order to have consistent symbol names in the generated code.

```
<<generation support data>>=
variable symcounter 0
variable labelcounter 0
```

Looking up a symbol returns a dictionary of the symbol attributes.

```
<<generation support commands>>=
proc LookUpSymbol {name} {
    set sym [relvar restrictone Symbol Name $name]
    if {[relation isnotempty $sym]} {
        return [tuple get [relation tuple $sym]]
    }

    return
}
```

Symbols can also be deleted.

```
<<generation support commands>>=
proc DeleteSymbol {name} {
    relvar deleteone Symbol Name $name
}
```

Inserting a symbol also returns the dictionary of its attributes.

```
<<generation support commands>>=
proc InsertSymbol {args} {
    variable block
    dict set args Block $block
    try {
        set sym [relvar insert Symbol $args]
        return [tuple get [relation tuple $sym]]
    } trap {RAL relvar insert DUPLICATE_TUPLE*} {result} {
        error "duplicate symbol, [dict get $args Name], $result" ; # ❶
    } on error {result opts} {
        return -options $opts $result
    }
}
```

❶ We trap duplicates to give a better error message.

Verifying Symbols

The code generator needs to verify that symbols exist and have the correct data types and other properties.

```
<<generation support commands>>=
proc CheckSymbol {name args} {
    set sym [LookUpSymbol $name]
    if {[dict size $sym] == 0} {
        error "unknown symbol, $name"
    }
    if {[llength $args] != 0} {
        CheckSymProperties $sym {*}$args
    }
    return $sym
}
```

```
<<generation support commands>>=
proc CheckSymProperties {sym args} {
    foreach {prop value} $args {
        # If the value of the property is the empty string, then we will assume
        # the value that is checked first. This lets us pass parameters where
        # we don't quite know the data type needs.
        if {[dict get $sym $prop] eq {}} {
            dict set sym $prop $value
            relvar update Symbol stup {
                    [tuple extract $stup Name] eq [dict get $sym Name] &&\
                    [tuple extract $stup $prop] eq {}
                } {
                    tuple update $stup $prop $value
                }
        }
        if {[dict get $sym $prop] ne $value} {
            error "for variable, [dict get $sym Name], expected $prop to be,\
                    $value: got, [dict get $sym $prop], instead"
        }
    }
```

```
    return
}
```

We can specialize the symbol verification to the types of symbols most commonly used by the code generation.

```
<<generation support commands>>=
proc CheckInstRefSymbol {varName args} {
    tailcall CheckSymbol $varName Type Reference {*}$args
}
```

```
<<generation support commands>>=
proc CheckInstSetSymbol {varName args} {
    tailcall CheckSymbol $varName Type InstanceSet {*}$args
}
```

The macro commands create temporary variables to hold intermediary results.

```
<<generation support commands>>=
proc CreateTempSymbolName {} {
    variable symcounter
    return t__T[incr symcounter]
}

proc CreateTempSymbol {args} {
    set name [CreateTempSymbolName]
    InsertSymbol Name $name {*}$args
    return $name
}

proc CreateLabelName {} {
    variable labelcounter
    return l__L[incr labelcounter]
}
```

Again, we specialize the creation procedure for the variable types commonly used in code generation.

```
<<generation support commands>>=
proc CreateTempRefSymbol {className} {
    set reftype [GetClassProperty $className Reference]
    set symName [CreateTempSymbol Ctype $reftype Type Reference Class $className]
    return [list $reftype $symName]
}
```

Sometimes we know the name of a variable and want a declaration for it. However, if the variable has already been declared, then we don't want a duplicate declaration. This idea is again specialized for the common types of variable created during code generation.

```
<<generation support commands>>=
proc CreateInstRefSymbol {className varName} {
    set sym [LookUpSymbol $varName]
    if {$sym ne {}} {
        CheckSymProperties $sym Class $className Type Reference
        return
    }
    set creftype [GetClassProperty $className Reference]
    InsertSymbol Name $varName Ctype $creftype Type Reference Class $className
    return "$creftype$varName ;\n"
}
```

```
<<generation support commands>>=
proc CreateInstSetSymbol {className varName} {
    set sym [LookUpSymbol $varName]
    if {$sym ne {}} {
        CheckSymProperties $sym Class $className Type InstanceSet
        return
    }
    set ctype MRT_InstSet
    InsertSymbol Name $varName Ctype $ctype Type InstanceSet Class $className
    return "$ctype $varName ;\n"
}
```

Tracking Code Blocks

Code blocks are tracked with a simple variable and are treated with stack type semantics.

```
<<generation support commands>>=
proc PushBlock {{by 1}} {
    variable block
    incr block $by
}
```

```
<<generation support commands>>=
proc GetBlock {} {
    variable block
    return $block
}
```

To pop a code block has the side effect of deleting any symbols defined with the block.

```
<<generation support commands>>=
proc PopBlock {{by -1}} {
    variable block
    relvar delete Symbol stup {[tuple extract $stup Block] == $block}
    incr block $by
}
```

In order to format the generated code neatly, and to keep track of variable scopes introduced by "C" statements in an activity, we examine all the text as it is processed to deduce blocks going in and out of scope and to format passed through code. In the end the format of the generated code is not perfect, but is much better than code generators usually produce. Although the "C" compiler does not care about whitespace and code format, the generated files from `micca` do show up during debugging and a consistent format is worthwhile.

```
<<generation support commands>>=
proc ProcessCodeLines {text} {
    variable block

    # puts "text = \"$text\""
    set newlines [list]

    foreach line [split $text \n] {
        if {[string is space $line]} {
            if {[llength $newlines] != 0 && [lindex $newlines end] ne {}} {
                lappend newlines {}
            }
        } else {
            if {[string first \} $line] != -1} {
                incr block -1
            }
```

```
            if {[regsub -- {\A\s{4,}} $line {} newline] != 0} {
                set line [string repeat { } [expr {$block * 4}]]$newline
            }
            lappend newlines $line
            if {[string first \{ $line] != -1} {
                incr block
            }
        }
    }
    set newtext [join $newlines \n]
    # puts "newtext = \"$newtext\""

    return $newtext
}
```

To maintain the generated code formatting, we supply a procedure to indent to the current block level.

```
<<generation support commands>>=
proc IndentToBlock {code} {
    variable block
    # indent seems to swallow a trailing new line
    return [::textutil::adjust::indent $code\
        [string repeat { } [expr {$block * 4}]]]\n
}
```

Expansion Context

The template expansion library supports the notion of a *context*. A context is a temporary diversion of processing that can have its own separate environment. In that environment, one can store variables that can be accessed later. The expander code insures that all contexts are popped by the end. The use we have here for the contexts is to implement the `<%end%>` macro command. The work that must be done is to close out any open compound statements in the generated "C" code. We will store one variable in the context called *depth* which will be the number of open blocks that need to be closed off.

```
<<generation support commands>>=
proc PushContext {context startblock} {
    actexpand cpush $context

    variable block
    actexpand cset depth [expr {$block - $startblock}]
    return
}
```

In our usage of expansion contexts, some commands allow other code to be given that is executed when certain conditions are met (*e.g.* a code body that is part of an iteration). At the end of the code block, we must close off the open contexts.

```
<<generation support commands>>=
proc End {} {
    set result {}

    set context [actexpand cname] ; # ❶
    switch -exact -- $context {
        InstanceSetForeachInstance -
        InstanceForeachRelated -
        InstanceForeachRelatedWhere -
        ClassForeachInstance -
        ClassForeachWhere {
            set result [CloseBlocks]
        }
        GenClassify {
            set subclasses [actexpand cget subclasses]
```

```
            set foundsubs [actexpand cget foundsubs]
            set gotdefault [actexpand cget gotdefault]

            set unused [::struct::set difference $subclasses $foundsubs]
            if {!([::struct::set empty $unused] || $gotdefault)} {
                set relname [actexpand cget relname]
                log::warn "subclass(es), \"[join $unused {, }]\", were\
                    not present in the \"$relname classify\" statement\
                    and no default case was given"
            }

            set result [CloseBlocks]
        }
        Subclass {
            set result [CloseBlocks]
            append result [IndentToBlock "break ;\n"]
        }
        default {
            error "unknown context, \"$context\""
        }
    }
    append result [IndentToBlock [linecomment end]]

    return $result
}
```

```
<<generation support commands>>=
proc CloseBlocks {} {
    set depth [actexpand cget depth]
    set result [actexpand cpop [actexpand cname]] ; # ❶
    for {set i 0} {$i < $depth} {incr i} {
        PopBlock
        append result [IndentToBlock "\}\n"]
    }
    return $result
}
```

❶ We check the context name to be one of the allowed ones just to trap any errant calls to `PushContext`.

❶ When the expander pops a context, it returns what has been accumulated in that context.

Template Expansion of Activity Code

```
<<generation support commands>>=
proc ExpandActivity {name body parameters} {
    # parameters is a relation value with heading:
    # Name string Ctype string Type string Class string

    variable block 1
    try {
        foreach param [relation body $parameters] {
            InsertSymbol {*}$param
        }
        return [actexpand expand $body]
    } on error {result} {
        # puts $::errorInfo
        log::error "$name: $result"
        variable errcount
```

```
        incr errcount
        set msg [pipe {
            split $result \n |
            lrange ~ 1 2 |
            string map [list \" {}] ~ |
            join ~
        }] ; # ❶
        return "#error \"$msg\"\n" ; ❷
    } finally {
        relvar set Symbol [relation emptyof [relvar set Symbol]]
        variable symcounter 0
        variable labelcounter 0
    }
}
```

❶ Preparing a message suitable for `#error` from the error result.

❷ We place a `#error` statement in the code to make sure it will not compile.

Invoking External Operations

<% extentity opname ?name1 value1 name2 value2 ... ? %>

extentity
The name of the external entity.

opname
The name of the external operation of the entity.

nameN / valueN
The arguments to the external operation are given as a set of name / value pairs.

The *opname* subcommand of the *extentity* command generates a "C" expression to invoke the external operation given by, *opname*. Arguments passed to the operation are given as name / value pairs. The ordering of the pairs is arbitrary but all arguments of the operation must be given.

```
<<generation support commands>>=
proc InvokeExternalOp {entity opName args} {
    if {[llength $args] % 2 != 0} {
        error "operation parameters must be given as name / value pairs"
    }
    variable domain

    set opRef [ExternalOperation findWhere {$Domain eq $domain &&\
            $Entity eq $entity && $Name eq $opName}]
    if {[isEmptyRef $opRef]} {
        error "unknown external operation, $opName, for domain, $domain\
                and external entity, $entity"
    }

    set provided [dict keys $args]
    set params [deRef [findRelated $opRef ~R11]]
    set required [relation list $params Name -ascending Number]
    CheckSuppliedParams $opName $provided $required

    set pset {}
    foreach pname $required {
```

```
        append pset "[dict get $args $pname], "
    }
    set pset [string trimright $pset {, }]

    return "${domain}_${entity}_${opName}__EOP\($pset\)"
}
```

Instance Macro Commands

The macro command, `instance` has a number of subcommands used to access various aspects of a class instance. The instance commands follow a pattern:

`<%instance` *instvar* `subcommand` *?argument1 argument2 . . . ?* `%>`

The operation of the `subcommand` is performed on class instance whose reference is held in the *instvar* variable. The `subcommand` argument names the specific instance operation to be performed. Each possible `subcommand` is given in a section below describing the operation. Difference `subcommand` operation may require additional arguments as described.

Reading Instance Attribute Values

It is possible and often desirable to access attributes of an instance directly using "C" pointer indirection. For example, the `Color` attribute of the instance referenced by a `self` variable can be obtained using `self->Color`.

However, `micca` also provides an embedded macro command to generate the pointer indirection expression. The command has the advantage of checking that the attribute name is valid for the class and thus catching potential errors before compilation.

`<%instance` *instvar* `attr` *attrname* `%>`

instvar
The name of a "C" variable that holds a reference to a class instance.

attrname
The name of the attribute to be retrieved.

The `attr` subccommand generates a "C" expression to read *attrname* for the instance whose reference is contained in the variable named, *instvar*. The generated code is suitable for use in either an rvalue or lvalue context. The class whose instance is held in *instvar* must contain an attribute whose name is *attrname* or an error is generated.

```
<<generation support commands>>=
proc InstanceAttrRead {instref attr} {
    variable domain

    set sym [CheckInstRefSymbol $instref]
    set className [dict get $sym Class]
    set attrRef [Attribute findById Domain $domain Class $className Name $attr]
    if {[isNotEmptyRef $attrRef]} {
        set depRef [findRelated $attrRef {~R29 DependentAttribute}]
        if {[isNotEmptyRef $depRef]} {
            error "cannot use \"attr\" command for dependent attribute:\
                use \"assign\" command instead"
        }
        return "$instref->$attr" ; # ❶
    } else {
        set massRef [MultipleAssigner findById Domain $domain\
            Association $className]
```

```
        if {[isNotEmptyRef $massRef] && $attr eq "idinstance"} {
            return "$instref->$attr"
        }
        error "instance reference, $instref, refers to an instance of\
            class, $className, which does not have an attribute\
            named, $attr"
    }
}
```

❶ Normal return is just the pointer indirection expression.

Updating Instance Attribute Values

<%instance *instvar* update ? *attrname1 value1* ... ? %>

instvar
The name of a "C" variable that holds a reference to a class instance.

attrnameN
The name of the attribute to be updated.

valueN
A valid "C" rvalue expression to be used as the new value for the attribute

The update subccommand generates a "C" statements to update zero or more attribute values for the instance whose reference is contained in the variable named, *instvar*. Arguments are given in *attrname* / *value* pairs. The class whose instance is held in *instvar* must contain an attribute whose name is *attrname* or an error is generated. Updating dependent attributes is not allowed.

```
<<generation support commands>>=
proc InstanceAttrUpdate {instref args} {
    variable domain

    if {[llength $args] % 2 != 0} {
        error "attribute name / values must be given in pairs, got:\
            \"$args\""
    }

    set sym [CheckInstRefSymbol $instref]
    set className [dict get $sym Class]

    set result [linecomment "instance $instref update $args"]
    foreach {attr value} $args {
        set attrRef [Attribute findById Domain $domain Class $className Name $attr]
        if {[isEmptyRef $attrRef]} {
            error "instance reference, $instref, refers to an instance of\
                class, $className, which does not have an attribute\
                named, $attr"
        }

        set depRef [findRelated $attrRef {~R29 DependentAttribute}] ; # ❶
        if {[isNotEmptyRef $depRef]} {
            error "cannot update dependent attribute"
        }

        append result [GenValueAssignment $domain $instref->$attr\
                $value [readAttribute $attrRef DataType]]
```

```
    }

    return [IndentToBlock $result]
}
```

❶ Dependent attributes are read only.

```
<<generation helper commands>>=
proc UnaliasType {domain typename} {
    set aliasRef [TypeAlias findById Domain $domain TypeName $typename]
    return [expr {[isEmptyRef $aliasRef] ? $typename :\
        [readAttribute $aliasRef TypeDefinition]}]
}
```

Assigning Instance Attributes to Variables

A variation of the the `attr` command is the `assign` command. This command assigns instance attributes to local "C" variables. Note that dependent attributes can only be accessed in this way (because in "C" an array cannot be treated as an ordinary value).

`<%instance` *instvar* `assign` *?assignspec1 assignspec2 ... ?* `%>`

instvar
The name of a "C" variable that holds a reference to a class instance.

assignspecN
A set of attribute name / variable name pairs. Each *assignspec* argument consists of a list of one or two elements. The first element is the name of the attribute of *instvar* to assign. The second element is the name of a local variable into which the assignment is made. If the second element is absent, then the assignment is made to a local variable with the same name as the attribute.

The `assign` subccommand generates "C" statements to assign a set of attributes of *instvar* to local variables according the the *assignspec* arguments. All local variables are declared as necessary. The class whose instance is held in *instvar* must contain an attribute whose name matches the attribute name in the *assignspec* or an error is generated. If no *assignspecN* arguments are given, the all the attributes of *instvar* are assigned to variables whose names are the same as the attribute name.

```
<<generation support commands>>=
proc InstanceAssign {instref args} {
    variable domain

    set sym [CheckInstRefSymbol $instref]
    set className [dict get $sym Class]

    set attrProps [pipe {
        Attribute findWhere {$Domain eq $domain && $Class eq $className} |
        deRef ~ |
        relation project ~ Name DataType
    }]

    set attrNames [relation list $attrProps Name]
    set attrTypes [relation dict $attrProps Name DataType]

    if {[llength $args] == 0} {
        set attrspecs $attrNames
    } else {
        foreach spec $args {
```

```
            if {[lindex $spec 0] ni $attrNames} {
                error "instance reference, $instref, refers to an instance of\
                    class, $className, which does not have an attribute\
                    named, [lindex $spec 0]"
            }
        }
        set attrspecs $args
    }

    set result [linecomment "instance $instref assign [list $args]"]
    foreach attrspec $attrspecs {
        if {[llength $attrspec] == 1} {
            set attrname [lindex $attrspec 0]
            set varname $attrname
        } elseif {[llength $attrspec] == 2} {
            lassign $attrspec attrname varname
            if {![isIdentifier $varname]} {
                error "\"$varname\" is not a valid \"C\" identifier"
            }
        } else {
            error "bad attribute specification, \"$attrspec\":\
                expected 1 or 2 element list"
        }
        set attrtype [dict get $attrTypes $attrname]

        set varsym [LookUpSymbol $varname]
        if {[dict size $varsym] != 0} {
            CheckSymProperties $varsym Ctype $attrtype
        } else {
            InsertSymbol Name $varname Ctype $attrtype Type {} Class {}
            append result "[typeCheck composeDeclaration $attrtype $varname] ;\n"
        }

        set indepRef [IndependentAttribute findById\
                Domain $domain Class $className Name $attrname]
        if {[isNotEmptyRef $indepRef]} {
            append result [GenValueAssignment $domain $varname\
                    $instref->$attrname $attrtype]
        } else {
            set asgnType [typeCheck assignmentType\
                    [UnaliasType $domain $attrtype]]
            set varref [expr {[dict get $asgnType type] ne "scalar" ?\
                    "$varname" : "&$varname"}]
            append result\
                "${className}_${attrname}__FORMULA($instref, $varref,\
                sizeof($varname)) ;\n"
        }
    }

    return [IndentToBlock $result]
}
```

Deleting Instances

`<%instance` *instvar* `delete%>`

instvar
The name of a "C" variable that holds a reference to a class instance.

The `delete` subccommand generates a "C" statement to delete the class instance whose reference is contained in *instvar*.

```
<<generation support commands>>=
proc InstanceDelete {instref} {
    CheckInstRefSymbol $instref
    return [IndentToBlock [string cat\
        [linecomment "instance $instref delete"]\
        "mrt_DeleteInstance($instref) ;\n"\
    ]]
}
```

Invoking Instance Based Operations

`<%instance` *instvar* `operation` *opname param1 value1 param2 value2 ...* `%>`

instvar
The name of a "C" variable that holds a reference to a class instance.

opname
The name of the instance operation to be invoked.

paramN valueN
A set of named parameter values to the operation.

The `operation` subccommand generates a "C" expression to invoke the instance based operation on the class instance whose reference is contained in *instvar*. The *opname* argument must be a valid instance based operation defined on the class of the *instvar* instance. Parameter arguments are parameter name / parameter value pairs, where the parameter name must match that provided in the definition of the operation. Parameter values may be any valid "C" value expression such as a variable name or a constant. Parameters may be given in any order since they are named. It is an error not to provide values for all the parameters of the operation.

```
<<generation support commands>>=
proc InstanceOperation {instref opName args} {
    set instsym [CheckInstRefSymbol $instref]
    if {[llength $args] % 2 != 0} {
        error "operation parameters must be given as name / value pairs"
    }
    variable domain

    set className [dict get $instsym Class]
    set opRef [Operation findWhere {$Domain eq $domain && $Class eq $className\
        && $Name eq $opName && $IsInstance}]
    if {[isEmptyRef $opRef]} {
        error "unknown instance operation, $opName, for class, $className"
    }

    dict set args self $instref ; # ❶
    set provided [dict keys $args]
    set params [deRef [findRelated $opRef ~R4]]
    set required [relation list $params Name -ascending Number] ; # ❷
    CheckSuppliedParams $opName $provided $required
```

```
    set pset {}
    foreach pname $required {
        append pset "[dict get $args $pname], "
    }
    set pset [string trimright $pset {, }]

    return "${className}_$opName\($pset\)"
}
```

❶ The value of the implicit `self` argument is just the instance reference upon which the macro command was invoked.

❷ The order is important here since "C" passes parameters by order. We use the parameter numbering to translate from pass by name to pass by order.

We factor out some validation code which will be reused when generating code for invoking other types of operations.

```
<<generation support commands>>=
proc CheckSuppliedParams {name provided required} {
    if {![struct::set equal $provided $required]} {
        lassign [struct::set intersect3 $provided $required]\
            p_inter_r p_minus_r r_minus_p
        if {![struct::set empty $p_minus_r]} {
            error "provided parameter(s), \"[join $p_minus_r ,]\", which are\
                not parameters to, $name"
        }
        if {![struct::set empty $r_minus_p]} {
            error "parameter(s), \"[join $r_minus_p ,]\", are required for,\
                $name, but were not provided"
        }
    }
}
```

Iterating Across Related Instances

Navigating a chain of relationships is one of the more complicated model level actions in terms of the required code. A relationship chain starts at a given class instance and then traverses relationships to obtain a related set of class instances. Often, it is not necessary to actually accumulate the set of related instance references. For simple operations on the related instances, it is sufficient to iterate across them and perform the operation as each related instance is visited.

The different "C" level storage strategies for relationship pointers is what makes navigation complicated. Sometimes the navigation is accomplished by a single pointer, sometimes a linked list and sometimes a counted array. Some relationships are conditional and traversal through a conditional relationship must insure that we have actually found an instance. All of these considerations make is essential to provide an embedded macro command to generate the required code.

Previously, we explained the conventions used to specify relationship traversal. Those same conventions will be used here to specify, as a set of arguments, the relationship chain to be navigated.

`<%instance` *start* `foreachRelated` *instvar rel1 ?rel2? ...* `%>`
Loop Body
`<%end%>`

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instvar
The name of a "C" variable that holds a reference to a related class instance.

relN
A set of relationship navigation specifications.

The `foreachRelated` subccommand generates "C" statements to iterate across the set of instances obtained by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. A reference to each instance found in the navigation is assigned to the *instvar* variable and then *Loop Body* is executed. Statements in the *Loop Body* terminate at the `<%end%>` macro command. The scope of the *instvar* variable is confined to the statements in the *Loop Body* unless it was declared previously.

The relationship traversal is given as a set of relationship navigation specifications of the form:

- `Rdd` — traversing relationship *Rdd* in the forward direction.
- `~Rdd` — traversing relationship *Rdd* in the backward direction.
- `{Rdd class}` — traversing relationship *Rdd* in the forward direction and stopping at `class`.
- `{~Rdd class}` — traversing relationship *Rdd* in the backward direction and stopping at `class`.

```
<<generation support commands>>=
proc InstanceForeachRelated {startref instref args} {
    set startblock [GetBlock]
    set result [IndentToBlock\
        [linecomment "instance $startref foreachRelated $instref $args"]\
    ]

    append result [TraverseRelChain $startref $instref $args]

    PushContext InstanceForeachRelated $startblock

    return $result
}
```

The difficult part of the code generation for traversing a relationship chain is factored in the to procedure below. The `TraversRelChai` procedure generates the code to access the pointer in the instance structures associated with a given set of relationships. Since there are many different pointer arrangements, the traversal code will be specific to each type of relationships being traversed. The return value of the function is a three element list giving the generated code to traverse the relationship chain, the target class of the end of the chain and a variable name containing the target instance reference of the end.

```
<<generation support commands>>=
proc TraverseRelChain {startref endref relspecs} {
    variable domain

    if {[llength $relspecs] == 0} {
        error "empty relationship chain specification"
    }

    set result {}
    set startsym [CheckInstRefSymbol $startref]
    set startClass [dict get $startsym Class]
```

```
    foreach relspec $relspecs {
        set relinfo [LookUpRelationship $startClass $relspec]
        while {[llength $relinfo] != 0} {
            # puts "relinfo = \"$relinfo\""
            set relinfo [lassign $relinfo sourceclass targetclass reftype cond\
                comp]
            switch -exact -- $reftype {
                reference {
                    <<TraverseRelChain: single reference>>
                }
                array {
                    <<TraverseRelChain: array reference>>
                }
                linked {
                    <<TraverseRelChain: linked reference>>
                }
                reftosuper {
                    <<TraverseRelChain: reference to superclass>>
                }
                reftosub {
                    <<TraverseRelChain: reference to subclass>>
                }
                uniontosuper {
                    <<TraverseRelChain: union subclass to superclass reference>>
                }
                uniontosub {
                    <<TraverseRelChain: union superclass to subclass reference>>
                }
            }
            set startClass $targetclass
            set startref $targetref
        }
    }
    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol $targetclass $endref]\
        "$endref = $targetref ;\n"]]

    return $result
}
```

Instance references stored as a single pointer value generate code to assign the pointer value into a temporary variable that is suitable as an instance reference to the target of the navigation.

```
<<TraverseRelChain: single reference>>=
lassign [CreateTempRefSymbol $targetclass] targettype targetref
set refcode "$targettype$targetref = $startref->$comp ; // $relspec \n"
if {$cond} { # ❶
    append refcode "if ($targetref != NULL) \{\n"
    append result [IndentToBlock $refcode]
    PushBlock
} else {
    append result [IndentToBlock $refcode]
}
```

❶ If the relationship is conditional, then we must generate code to test if we obtained an instance pointer.

When we encounter a set of instance references stored in an array, the generated code must set up a loop construct to visit all the members of the set.

```
<<TraverseRelChain: array reference>>=
```

```
set itervar [CreateTempSymbol\
    Ctype "struct $targetclass *const *"\
    Type "ReferenceArray" Class $targetclass]
set cntvar [CreateTempSymbol Ctype size_t\
    Type ArrayCounter Class $targetclass]
set refcode "struct $targetclass *const *$itervar =\
        $startref->$comp.links ; // $relspec\n"
append refcode\
    "for (size_t $cntvar = $startref->$comp.count ;\
        $cntvar != 0 ; $cntvar--, $itervar++) \{" ; # ❶
append result [IndentToBlock $refcode]
PushBlock
lassign [CreateTempRefSymbol $targetclass] targettype targetref
append result [IndentToBlock "$targettype$targetref = *$itervar ;"]
```

❶ For the array case there is no special handling in case the set of related instance is empty. In the empty case, the code generator will have set the count value to 0 and the loop will not be executed.

For references stored in linked lists, we again have generate code to iterate across the links.

```
<<TraverseRelChain: linked reference>>=
set itervar [CreateTempSymbol Ctype "MRT_LinkRef *"\
    Type "ReferenceLink" Class $targetclass]
lassign $comp termcomp linkcomp
append result [IndentToBlock\
    "for (MRT_LinkRef *$itervar =\
        mrtLinkRefBegin(&$startref->$termcomp) ;\
        $itervar != mrtLinkRefEnd(&$startref->$termcomp) ;)\
        \{\n"] ; # ❶
PushBlock
lassign [CreateTempRefSymbol $targetclass] targettype targetref
append result [IndentToBlock [string cat\
    "$targettype$targetref =\
        ($targettype)((uintptr_t)$itervar -\
        offsetof(struct $targetclass, $linkcomp)) ;\n"\
    "$itervar = $itervar->next ;\n"\
]] ; # ❷
```

❶ Again, there is no special empty case for linked lists. If there are no related instances linked together, the linked list iterator will be at the end and the loop will not be entered.

❷ Two complications arise using linked list. Since a class instance may be threaded onto several linked lists, we must do some pointer arithmetic to transform the pointer value stored in the linked list into a pointer to the beginning of the instance. So, we must subtract off the offset to the link pointers in the instance. Secondly, we advance the linked list iterator at the beginning of the loop rather than at the end. This is to make sure that any user supplied code that might unlink the instance from the linked list does not invalidate our iterator.

References from subclasses to superclasses are just a single pointer value that is unconditional.

```
<<TraverseRelChain: reference to superclass>>=
lassign [CreateTempRefSymbol $targetclass] targettype targetref
append result [IndentToBlock\
    "$targettype$targetref = $startref->$comp ; // $relspec \n"\
]
```

References from a superclass to a subclass are also just a single pointer value. However, the traversal is conditional. When specifying a traversal to a subclass, the subclass class name must be given and the generated code has to insure that we are currently related to an instance of the designated subclass.

```
<<TraverseRelChain: reference to subclass>>=
lassign [CreateTempRefSymbol $targetclass] targettype targetref
set classDesc [GetClassDescriptor $domain $targetclass]
set refcode "$targettype$targetref = $startref->$comp ; // $relspec\n"
append refcode "if ($targetref->base__INST.classDesc == &$classDesc) \{\n"
append result [IndentToBlock $refcode]
PushBlock
```

When a generalization is stored in a union, then pointer arithmetic alone is sufficient to *up cast* to the supeclass instance.

```
<<TraverseRelChain: union subclass to superclass reference>>=
lassign [CreateTempRefSymbol $targetclass] targettype targetref
append result [IndentToBlock\
    "$targettype$targetref = ($targettype)((uintptr_t)$startref -\
    offsetof(struct $targetclass, $comp.$startClass)) ;\
    // $relspec \n"]
```

Going from the superclass to the subclass with a union storage arrangement has the same complication as when pointers are used, namely we must make sure that the instance is currently related to an instance of the given target subclass. The addressing expression is much easier as the compiler will do the address arithmetic for us.

```
<<TraverseRelChain: union superclass to subclass reference>>=
lassign [CreateTempRefSymbol $targetclass] targettype targetref
set classDesc [GetClassDescriptor $domain $targetclass]
set refcode "$targettype$targetref = &$startref->$comp.$targetclass ;\
        // $relspec\n"
append refcode\
    "if ($targetref->base__INST.classDesc == &$classDesc) \{\n"
append result [IndentToBlock $refcode]
PushBlock
```

To simplify the code of `TraverseRelChain`, we use the `LookUpRelationship` procedure to summarize the relationship characteristics that are needed to navigate the relationship.

The Relationship Subsystem of the platform model discusses the various arrangements for reference pointer storage and how for class based associations traversal is decomposed into two steps. The structure of `LookUpRelationship` follows directly from the platform model for relationships, searching for associations and generalizations and the various types of each.

```
<<generation support commands>>=
proc LookUpRelationship {startclass relspec} {
    variable domain

    lassign $relspec relname destclass
    if {[string index $relname 0] eq "~"} {
        set dir back
        set relname [string range $relname 1 end]
    } else {
        set dir forw
    }

    set relRef [Relationship findById Domain $domain Name $relname]
    set assocRef [findRelated $relRef {~R30 Association}]
    if {[isNotEmptyRef $assocRef]} {
        <<LookUpRelationship: associations>>
    } else {
        <<LookUpRelationship: generalizations>>
    }
}
```

Association type relationships are partitioned into either simple associations or class based associations. We follow that division in the code below.

```
<<LookUpRelationship: associations>>=
set isstatic [readAttribute $assocRef IsStatic]
set sassocRef [findRelated $assocRef {~R31 SimpleAssociation}]
if {[isNotEmptyRef $sassocRef]} {
    <<LookUpRelationship: simple associations>>
} else {
    <<LookUpRelationship: class based associations>>
}
```

For simple associations, it is a matter of making we are specifying things correctly with respect to the direction and then dealing with the correct type of storage for the back links.

```
<<LookUpRelationship: simple associations>>=
if {$destclass ne {}} {
    error "simple association, $relname, cannot have a\
        destination specifier"
}
assignAttribute [findRelated $sassocRef ~R32]\
    {Class srcclass} {Conditionality srccond} {Multiplicity srcmult}
assignAttribute [findRelated $sassocRef ~R33] {Class trgclass}
if {$dir eq "forw"} {
    if {$startclass ne $srcclass} {
        error "relationship, $relname, is from\
            $srcclass to $trgclass: got, $startclass,\
            as the traversal start"
    }
    return [list $srcclass $trgclass reference false $relname]
} else {
    if {$startclass ne $trgclass} {
        error "relationship, ~$relname, is from\
            $trgclass to $srcclass: got, $startclass,\
            as the traversal start"
    }
    if {!$srcmult} {
        set type reference
        set comp ${relname}__BACK
    } else {
        if {$isstatic} {
            set type array
            set comp ${relname}__BACK
        } else {
            set type linked
            set comp [list ${relname}__BACK ${relname}__BLINKS]
        }
    }
    return [list $trgclass $srcclass $type $srccond $comp]
}
```

For class based associations, we have the added complication of traversal to the associative class or across the entire association.

```
<<LookUpRelationship: class based associations>>=

<<LookUpRelationship: lookup participant properties>>
<<LookUpRelationship: determine target properties>>
<<LookUpRelationship: determine source properties>>

if {$destclass eq {}} {
    <<LookUpRelationship: full traversal>>
} else {
    <<LookUpRelationship: partial traversal>>
}
```

```
<<LookUpRelationship: lookup participant properties>>=
set cassocRef [findRelated $assocRef {~R31 ClassBasedAssociation}]
assignAttribute [findRelated $cassocRef ~R34]\
    {Class srcclass} {Conditionality srccond} {Multiplicity srcmult}
assignAttribute [findRelated $cassocRef ~R35]\
    {Class trgclass} {Conditionality trgcond} {Multiplicity trgmult}
assignAttribute [findRelated $cassocRef ~R42]\
    {Class assocclass}
```

```
<<LookUpRelationship: determine target properties>>=
if {!$trgmult} {
    set srctype reference
    set srccomp ${relname}__FORW
} else {
    if {$isstatic} {
        set srctype array
        set srccomp ${relname}__FORW
    } else {
        set srctype linked
        set srccomp [list ${relname}__FORW ${relname}__FLINKS]
    }
}
```

```
<<LookUpRelationship: determine source properties>>=
if {!$srcmult} {
    set trgtype reference
    set trgcomp ${relname}__BACK
} else {
    if {$isstatic} {
        set trgtype array
        set trgcomp ${relname}__BACK
    } else {
        set trgtype linked
        set trgcomp [list ${relname}__BACK ${relname}__BLINKS]
    }
}
```

```
<<LookUpRelationship: full traversal>>=
if {$dir eq "forw"} {
    if {$startclass eq $srcclass} {
        return [list\
            $srcclass $assocclass $srctype $trgcond $srccomp\
            $assocclass $trgclass reference false ${relname}.forward\
        ]
    } elseif {$startclass eq $assocclass} {
        return [list\
            $assocclass $trgclass reference false ${relname}.forward\
        ]
    } else {
        error "relationship, $relname, is from class based from\
            $srcclass to $trgclass via $assocclass: got, $startclass,\
            as the traversal start"
    }
} else {
    if {$startclass eq $trgclass} {
        return [list\
            $trgclass $assocclass $trgtype $srccond $trgcomp\
            $assocclass $srcclass reference false ${relname}.backward\
        ]
    } elseif {$startclass eq $assocclass} {
```

```
        return [list\
            $assocclass $srcclass reference false ${relname}.backward\
        ]
    } else {
        error "relationship, ~$relname, is from class based from\
            $trgclass to $srcclass via $assocclass: got, $startclass,\
            as the traversal start"
    }
}
```

```
<<LookUpRelationship: partial traversal>>=
if {$destclass eq $assocclass} {
    if {$dir eq "forw"} {
        return [list $srcclass $assocclass $srctype $trgcond $srccomp]
    } else {
        return [list $trgclass $assocclass $trgtype $srccond $trgcomp]
    }
} elseif {$destclass eq $trgclass} {
    if {$dir eq "forw"} {
        return [list $assocclass $trgclass reference false ${relname}.forward]
    } else {
        error "navigating forward from $assocclass arrives at\
            $trgclass: got $destclass"
    }
} elseif {$destclass eq $srcclass} {
    if {$dir eq "back"} {
        return [list $assocclass $srcclass reference false ${relname}.backward]
    } else {
        error "navigating backward from $assocclass arrives at\
            $srcclass: got $destclass"
    }
} else {
    error "$destclass does not participate in $relname"
}
```

Generalizations are divided into those implemented using references and those using a union.

```
<<LookUpRelationship: generalizations>>=
set genRef [findRelated $relRef {~R30 Generalization}]
set refGenRef [findRelated $genRef {~R43 ReferenceGeneralization}]
if {[isNotEmptyRef $refGenRef]} {
    set super [readAttribute [findRelated $refGenRef ~R36] Class]
    set subs [findRelated $refGenRef ~R37]
    set subnames [relation list [deRef $subs] Class]
    set isRefGen true
} else {
    set uGenRef [findRelated $genRef {~R43 UnionGeneralization}]
    set super [readAttribute [findRelated $uGenRef ~R44] Class]
    set subs [findRelated $uGenRef ~R45]
    set subnames [relation list [deRef $subs] Class]
    set isRefGen false
}
```

The code can be the same since we have parameterized it for the different types of generalization storage.

```
<<LookUpRelationship: generalizations>>=
if {$dir eq "forw"} {
    if {$destclass ne {}} {
        error "generalization, $relname, cannot have a\
            destination specifier when traversing to the superclass"
    } elseif {$startclass ni $subnames} {
        error "relationship, $relname, is from\
```

```
            \"[join $subnames {, }]\", to $super,\
            got, $startclass, as the traversal start"
    }
    set reftype [expr {$isRefGen ? "reftosuper" : "uniontosuper"}]
    return [list $startclass $super $reftype false $relname]
} else {
    if {$destclass eq {}} {
        error "generalization, ~$relname, must specify a\
            destination when traversing to a subclass"
    } elseif {$startclass ne $super} {
        error "relationship, ~$relname, is from\
            $super to \"[join $subnames {, }]\", got, $startclass,\
            as the traversal start"
    } elseif {$destclass ni $subnames} {
        error "generalization, ~$relname, cannot traverse to\
            class, $destclass: should be one of:\
            \"[join $subnames {, }]\""
    }
    set reftype [expr {$isRefGen ? "reftosub" : "uniontosub"}]
    return [list $super $destclass $reftype false $relname]
}
```

Conditional Iteration Across Related Instances

`<%instance` *start* `foreachRelatedWhere` *instvar clause rel1 ?rel2? ...* `%>`
Loop Body
`<%end%>`

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instvar
The name of a "C" variable that holds a reference to a class instance.

clause
A "C" expression that evaluates to a value that can be interpreted as a boolean.

relN
A set of relationship navigation specifications.

The `foreachRelatedWhere` subccommand generates "C" statements to iterate across the set of instances obtained by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. A reference to each instance found in the navigation is assigned to the *instvar* variable and then `clause` is executed. If the result of executing `clause` is non-zero, then *Loop Body* is executed. Otherwise, *Loop Body* is skipped. Statements in the *Loop Body* terminate at the `<%end%>` macro command. The scope of the *instvar* variable is confined to the statements in the *Loop Body* unless it was declared previously. The relationship traversal is given as a set of relationship navigation specifications of the same form as for the `foreachRelated` subcommand.

```
<<generation support commands>>=
proc InstanceForeachRelatedWhere {startref instref where args} {
    set startblock [GetBlock]
    set where [string trim $where]
    set result [IndentToBlock\
        [linecomment "instance $startref foreachRelatedWhere\
                $instref [list $where] $args"]\
    ]
```

```
    append result [TraverseRelChain $startref $instref $args]
    append result [IndentToBlock "if ($where) \{\n"]
    PushBlock

    PushContext InstanceForeachRelatedWhere $startblock

    return $result
}
```

Conditionally Find a Single Related Instance

`<%instance` *start* `findRelatedWhere` *instvar clause rel1 ?rel2? ...* `%>`

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instvar
The name of a "C" variable that holds a reference to a class instance.

clause
A "C" expression that evaluates to a value that can be interpreted as a boolean.

relN
A set of relationship navigation specifications.

The `findRelatedWhere` subccommand generates "C" statements to iterate across the set of instances obtained by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. A reference to each instance found in the navigation is assigned to the *instvar* variable and then `clause` is executed. If the result of executing `clause` is non-zero, then the iteration stops and *instvar* contains a reference to the matching instance. If no match is found, then *instvar* will be set to `NULL`. The command finds the first matching related instance. If there is more than one matching related instance, the one found is arbitrary.

```
<<generation support commands>>=
proc InstanceFindRelatedWhere {startref instref where args} {
    set startlevel [GetBlock]
    set where [string trim $where]
    set endref [CreateTempSymbolName]
    set endlabel [CreateLabelName]

    set chaincode [TraverseRelChain $startref $endref $args]
    set endsym [LookUpSymbol $endref]

    append chaincode [IndentToBlock [string cat\
        "$instref = $endref ;\n"\
        "if ($where) \{\n"\
    ]]
    PushBlock

    append chaincode [IndentToBlock "goto $endlabel ;\n"] ; # ❶
    PopBlock

    append chaincode [IndentToBlock "\} else \{\n"]
    PushBlock

    append chaincode [IndentToBlock "$instref = NULL ;\n"]

    set depth [expr {[GetBlock] - $startlevel}]
    for {set i 0} {$i < $depth} {incr i} {
```

```
        PopBlock
        append chaincode [IndentToBlock "\}\n"]
    }
    append chaincode [IndentToBlock "${endlabel}: ;\n"] ;    # ❷

    set result [IndentToBlock [string cat\
        [linecomment "instance $startref findRelatedWhere\
                $instref [list [string trim $where]] $args"]\
        [CreateInstRefSymbol [dict get $endsym Class] $instref]\
        "$instref = NULL ;\n"\
    ]]

    return [append result $chaincode]
}
```

❶ We use a naked `goto` here because we may be many levels deep in a loop to accomplish the relationships chain traversal. Alternatively, we might be traversing a singular relationships. The `goto` works for all the cases.

❷ The semi-colon is necessary because a `goto` must label a statement.

Finding a Single Related Instance

`<%instance` *start* `findOneRelated` *instvar rel1 ?rel2? ...* `%>`

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instvar
The name of a "C" variable that will hold the related instance.

relN
A set of relationship navigation specifications.

The `findOneRelated` subccommand generates "C" statements obtain a single instance reference by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. A reference to the related instance is assigned into the *instvar* variable. The *instvar* variable is declared if necessary. If the traversal of the relationship chain would yield more than a single instance, then an error is thrown.

```
<<generation support commands>>=
proc InstanceFindOneRelated {startref instref args} {
    variable domain

    if {[llength $args] == 0} {
        error "empty relationship chain specification"
    }

    set startsym [CheckInstRefSymbol $startref]
    set startClass [dict get $startsym Class]
    set heading [IndentToBlock\
        [linecomment "instance $startref findOneRelated $instref $args"]\
    ]
    set startlevel [GetBlock]
    set isCond false

    foreach relspec $args {
        set relinfo [LookUpRelationship $startClass $relspec]
        while {[llength $relinfo] != 0} {
```

```
            set relinfo [lassign $relinfo sourceclass targetclass reftype cond\
                comp]
            switch -exact -- $reftype {
                reference {
                    if {$startClass ne $sourceclass} {
                        error "relationship, [lindex $relspec 0], is from\
                            $sourceclass to $targetclass, got, $startClass,\
                            as the traversal start"
                    }
                    lassign [CreateTempRefSymbol $targetclass]\
                        targettype targetref
                    set refcode "$targettype$targetref = $startref->$comp ;\
                    // $relspec \n"
                    if {$cond} {
                        set isCond true
                        append refcode "if ($targetref != NULL) \{\n"
                        append result [IndentToBlock $refcode]
                        PushBlock
                    } else {
                        append result [IndentToBlock $refcode]
                    }
                }
                array -
                linked {
                    error "relationship, [lindex $relspec 0], is\
                        multiple from $sourceclass to $targetclass"
                }
                reftosuper {
                    if {$startClass ni $sourceclass} {
                        error "relationship, [lindex $relspec 0], is from\
                            \"[join $sourceclass {, }]\", to $targetclass,\
                            got, $startClass, as the traversal start"
                    }
                    lassign [CreateTempRefSymbol $targetclass] targettype\
                            targetref
                    append result [IndentToBlock\
                        "$targettype$targetref = $startref->$comp ;\
                        // $relspec \n"]
                }
                reftosub {
                    set isCond true
                    lassign [CreateTempRefSymbol $targetclass] targettype\
                            targetref
                    set classDesc [GetClassDescriptor $domain $targetclass]
                    set refcode "$targettype$targetref = $startref->$comp ;\
                            // $relspec\n"
                    append refcode\
                        "if ($targetref->base__INST.classDesc == &$classDesc) \{\n"
                    append result [IndentToBlock $refcode]
                    PushBlock
                }
                uniontosuper {
                    if {$startClass ni $sourceclass} {
                        error "relationship, [lindex $relspec 0], is from\
                            \"[join $sourceclass {, }]\", to $targetclass,\
                            got, $startClass, as the traversal start"
                    }
                    lassign [CreateTempRefSymbol $targetclass] targettype\
                            targetref
                    append result [IndentToBlock\
                        "$targettype$targetref =\
                        ($targettype)((uintptr_t)$startref -\
```

```
                        offsetof(struct $targetclass, $comp.$startClass)) ;\
                        // $relspec \n"]
                }
                uniontosub {
                    set isCond true
                    lassign [CreateTempRefSymbol $targetclass] targettype\
                            targetref
                    set classDesc [GetClassDescriptor $domain $targetclass]
                    set refcode\
                        "$targettype$targetref = &$startref->$comp.$targetclass ;\
                            // $relspec\n"
                    append refcode\
                        "if ($targetref->base__INST.classDesc == &$classDesc) \{\n"
                    append result [IndentToBlock $refcode]
                    PushBlock
                }
            }
            set startClass $targetclass
            set startref $targetref
        }
    }
    append result [IndentToBlock "$instref = $targetref ;\n"]

    for {set i [expr {[GetBlock] - $startlevel}]} {$i > 0} {incr i -1} {
        PopBlock
        append result [IndentToBlock "\}\n"]
    }
    append heading [IndentToBlock [CreateInstRefSymbol $targetclass $instref]]
    if {$isCond} {
        append heading [IndentToBlock "$instref = NULL ;\n"]
    }

    return [string cat $heading $result]
}
```

Selecting Related Instances

`<%instance` *start* `selectRelated` *instsetvar rel1 ?rel2? ...* `%>`

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instsetvar
The name of a "C" variable which holds an instance set.

relN
A set of relationship navigation specifications.

The `selectRelated` subccommand generates "C" statements to iterate across the set of instances obtained by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. Each instance found in the navigation is added to the instance set contained in the *instsetvar* variable. The *instsetvar* variable is declared if necessary.

```
<<generation support commands>>=
proc InstanceSetSelectRelated {startref set args} {
    variable domain

    set startlevel [GetBlock]
```

```
    set endref [CreateTempSymbolName]
    set chaincode [TraverseRelChain $startref $endref $args]
    set targetclass [dict get [LookUpSymbol $endref] Class]
    set targetclassDesc [GetClassDescriptor $domain $targetclass]

    append chaincode\
        [IndentToBlock "mrt_InstSetAddInstance(&$set, $endref) ;\n"]

    set depth [expr {[GetBlock] - $startlevel}]
    for {set i 0} {$i < $depth} {incr i} {
        PopBlock
        append chaincode [IndentToBlock "\}\n"]
    }

    set result [IndentToBlock [string cat\
        [linecomment "instance $startref selectedRelated $set $args"]\
        [CreateInstSetSymbol $targetclass $set]\
        "mrt_InstSetInitialize(&$set, &$targetclassDesc) ;\n"]]

    return [append result $chaincode]
}
```

Conditionally Selecting Related Instances

<%instance *start* selectRelatedWhere *instsetvar instref clause rel1 ?rel2? ...* %>

start
The name of a "C" variable containing the instance reference where the relationship navigation will begin.

instsetvar
The name of a "C" variable which holds an instance set.

instref
The name of a variable into which a reference to a class instance is placed.

clause
A "C" expression that evaluates to a value that can be interpreted as a boolean.

relN
A set of relationship navigation specifications.

The selectRelatedWhere subccommand generates "C" statements to iterate across the set of instances obtained by navigating the relationship chain which starts at the *start* instance and is given by the *relN* arguments. A reference to each instance found in the navigation is assigned to the *instvar* variable and *clause* is evaluated. If *clause* evaluates to non-zero, then the instance is added to the instance set contained in the *instsetvar* variable. The *instsetvar* and *instvar* variables are declared if necessary.

```
<<generation support commands>>=
proc InstanceSetSelectRelatedWhere {startref set instref where args} {
    variable domain

    set startlevel [GetBlock]
    set where [string trim $where]

    set chaincode [TraverseRelChain $startref $instref $args]
    set targetclass [dict get [LookUpSymbol $instref] Class]
    set targetclassDesc [GetClassDescriptor $domain $targetclass]
    append chaincode [IndentToBlock "if ($where) \{\n"]
```

```
    PushBlock
    append chaincode [IndentToBlock\
        "mrt_InstSetAddInstance(&$set, $instref) ;\n"]

    set depth [expr {[GetBlock] - $startlevel}]
    for {set i 0} {$i < $depth} {incr i} {
        PopBlock
        append chaincode [IndentToBlock "\}\n"]
    }

    set result [IndentToBlock [string cat\
        [linecomment "instance $startref selectRelatedWhere $set $instref\
                [list $where] $args"]\
        [CreateInstSetSymbol $targetclass $set]\
        "mrt_InstSetInitialize(&$set, &$targetclassDesc) ;\n"]]

    return [append result $chaincode]
}
```

Signaling an Event

`<%instance` *instvar* `signal` *event ?name1 value1 name2 value2 ... ?* `%>`

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

event
The name of an event in the state model for the class to which the instance belongs.

nameN / valueN
A set of event parameter name / event parameter value pairs for the parameters that match the parameter signature of *event*.

The `signal` subccommand generates "C" statements to arrange for *event* to be signaled to the instance referenced by the *instvar* variable. Any given name / value pairs are passed as the parameters of the event. All event parameters must be supplied and parameter names must match the event signature of the *event* being signaled. If *event* is not an event for the state model of *instvar* then an error is thrown.

```
<<generation support commands>>=
proc InstanceSignal {instref event args} {
    set result [linecomment "instance $instref signal $event [list $args]"]
    set eventRef [VerifyEvent $event $instref]
    set params [VerifyEventParams $eventRef $args]
    if {[relation isempty $params]} {                    ; # ❶
        set eventNum [readAttribute $eventRef Number]
        set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]
        append result "mrt_SignalEvent($eventNum, $instref, $sourceinst,\
                NULL, 0) ; // $event\n"
    } else {
        set ecbvar [ECBAllocAndFill $instref $eventRef $params $args]
        append result "mrt_PostEvent($ecbvar) ;\n"
    }
    return [IndentToBlock $result]
}
```

❶ For the common case where there are no event parameters, we can use the function that does the ECB allocation and posting in one step.

First, we verify that the event is being signaled in the correct context, *i.e.* the class of the target instance actually defines such an event.

```
<<generation support commands>>=
proc VerifyEvent {event targetinst} {
    variable domain

    set targetsym [CheckInstRefSymbol $targetinst]
    set className [dict get $targetsym Class]
    set eventRef [Event findById Domain $domain Model $className Event $event]
    if {[isEmptyRef $eventRef]} {
        error "event, $event, is not a known event for class, $className"
    }
    return $eventRef
}
```

We also have to make sure that the supplied parameters match those required for the event signature.

```
<<generation support commands>>=
proc VerifyEventParams {eventRef supplied} {
    if {[llength $supplied] % 2 != 0} {
        error "event parameters must be given as name / value pairs, got:\
            \"$supplied\""
    }
    set psig [findRelated $eventRef R69]
    set params [FindParamsFromSig $psig]
    set suppliedNames [dict keys $supplied]
    set requiredNames [relation list $params Name -ascending Position]
    CheckSuppliedParams [readAttribute $eventRef Event]\
            $suppliedNames $requiredNames
    return $params
}
```

Since we will be signaling events in a number of different contexts, we have factored out the code to generate the “C” required to obtain an ECB from the run-time and fill in any event parameters.

```
<<generation support commands>>=
proc ECBAllocAndFill {targetinst eventRef params arglist {resultRef result}} {
    upvar 1 $resultRef result
    variable domain

    set className [readAttribute $eventRef Model]
    set eventName [readAttribute $eventRef Event]
    set eventNum [readAttribute $eventRef Number]
    set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]

    set ecbvar [CreateTempSymbol Ctype {MRT_ecb *} Type ECB Class $className]
    append result "MRT_ecb *$ecbvar =\
            mrt_NewEvent($eventNum, $targetinst, $sourceinst) ; // $eventName\n"
    if {[relation isnotempty $params]} {
        set eparamsname ${domain}_${className}_${eventName}__EPARAMS
        set ptrparams [CreateTempSymbol Ctype "struct *$eparamsname"\
            Type EventParams Class $className]
        append result\
            "struct $eparamsname *const $ptrparams =\n"\
            "    (struct $eparamsname *)$ecbvar->eventParameters ;\n"
        relation foreach param $params -ascending Position {
            relation assign $param {Name pname} {DataType datatype}
            append result [GenValueAssignment $domain $ptrparams->$pname\
                    [dict get $arglist $pname] $datatype]
        }
    }
```

```
    return $ecbvar
}
```

Delayed Signaling of an Event

<%instance *instvar* delaysignal *time event ?name1 value1 name2 value2 ... ?* %>

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

time
The minimum number of milliseconds that are to elapse before the event is signaled.

event
The name of an event in the state model for the class to which the instance belongs.

nameN / valueN
A set of event parameter name / event parameter value pairs for the parameters that match the parameter signature of *event*.

The delaysignal subccommand generates "C" statements to arrange for *event* to be signaled to the instance referenced by the *instvar* variable after *time* number of milliseconds has elapsed. The value of *time* may be zero, in which case the event is signaled immediately. Any given name / value pairs are passed as the parameters of the event. All event parameters must be supplied and parameter names must match the event signature of the *event* being signaled. If *event* is not an event for the state model of *instvar* then an error is thrown.

```
<<generation support commands>>=
proc InstanceDelaySignal {instref time event args} {
    set result\
        [linecomment "instance $instref delaysignal $time $event [list $args]"]
    set eventRef [VerifyEvent $event $instref]
    set params [VerifyEventParams $eventRef $args]
    if {[relation isempty $params]} {
        set eventNum [readAttribute $eventRef Number]
        set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]
        append result "mrt_SignalDelayedEvent($time, $eventNum, $instref,\
                $sourceinst, NULL, 0) ; // $event\n"
    } else {
        set ecbvar [ECBAllocAndFill $instref $eventRef $params $args]
        append result "mrt_PostDelayedEvent($ecbvar, $time) ;\n"
    }
    return [IndentToBlock $result]
}
```

Periodic Signaling of an Event

`<%instance` *instvar* `periodicsignal` *initial rollover event* `%>`

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

initial
The minimum number of milliseconds that are to elapse before the event is signaled the first time.

rollover
The minimum number of milliseconds that are to elapse before the event is signaled the second and all subsequent times.

The `periodicsignal` subccommand generates "C" statements to arrange for *event* to be signaled repeatedly to the instance referenced by the *instvar* variable. The *event* is signaled the first time after *initial* milliseconds elapses. The *event* is signaled after *rollover* milliseconds on the second and all subsequent times. The value of *initial* may be zero, in which case the event is signaled immediately. The value of *rollover* may be zero, in which case the event is signaled only once and has the same behavior as a delayed event. Any given name / value pairs are passed as the parameters of the event.

Note that periodic events are not allowed to have parameters. The rational is that since there is no reliable way to change the pararameter values between event dispatches, any data that might be a parameter is still available in the domain when the event is received.

```
<<generation support commands>>=
proc InstancePeriodicSignal {instref initial rollover event} {
    set result\
        [linecomment "instance $instref periodicsignal $initial $rollover $event"]
    set eventRef [VerifyEvent $event $instref]
    set params [VerifyEventParams $eventRef {}]
    if {[relation isempty $params]} {
        set eventNum [readAttribute $eventRef Number]
        set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]
        append result "mrt_SignalPeriodicEvent($initial, $rollover,\
                $eventNum, $instref, $sourceinst) ; // $event\n"
    } else {
        error "event, $event, requires parameters, [relation list $params Name],\
                and cannot be used periodically"
    }
    return [IndentToBlock $result]
}
```

Cancel a Delayed Signal

`<%instance` *instvar* `canceldelayed` *event ?sourceinst?* `%>`

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

event
The name of an event in the state model for the class to which the instance belongs.

sourceinst
The name of a variable whose value is a reference to the instance which originally signaled the delayed event.

The `canceldelay` subccommand generates "C" statements to arrange the delayed event, *event*, not be delivered to the instance referenced by *instvar*. The *sourceinst* argument is the name of a variable containing a reference to the instance that signaled the event. If *sourceinst* is not supplied, then the signaling instance is taken as `self` if invoked within as state activity and `NULL` otherwise. After it is canceled, *event* will not be dispatched. It is not an error to cancel an event that does not exist or has already been dispatched. Only events that were signaled as delayed events may be canceled.

```
<<generation support commands>>=
proc InstanceCancelSignal {instref event {sourceref {}}} {
    variable domain

    append result [linecomment "instance $instref canceldelayed $event\
        [list $sourceref]"]
    set target [CheckInstRefSymbol $instref]
    set eventNum [FindEventNumber [dict get $target Class] $event]
    if {$sourceref ne {}} {
        CheckInstRefSymbol $sourceref
        set sourceinst $sourceref
    } else {
        try {
            CheckInstRefSymbol self
            set sourceinst self
        } on error {} {
            set sourceinst NULL
        }
    }
    append result\
        "mrt_CancelDelayedEvent($eventNum, $instref, $sourceinst) ;\n"
    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc FindEventNumber {class event} {
    variable domain
    set eventRef [Event findById Domain $domain Model $class Event $event]
    if {[isEmptyRef $eventRef]} {
        error "event, $event, is not a known event for class, $class"
    }
    return [readAttribute $eventRef Number]
}
```

Time Remaining for a Delayed Event

`<%instance` *instvar* `delayremaining` *event ?sourceinst?* `%>`

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

event
The name of an event in the state model for the class to which the instance belongs.

sourceinst
The name of a variable whose value is a reference to the instance which originally signaled the delayed event.

The `delayremaining` subccommand generates a "C" expression containing the number of milliseconds that remain before *event* is signaled to the instance referenced by *instvar*. The *sourceinst* argument is the name of a variable containing a reference to the instance that signaled the event. If *sourceinst* is not supplied, then the signaling instance is taken as `self` if invoked within as state activity and `NULL` otherwise. A return value of zero implies that the event has already been dispatched or was never signaled.

```
<<generation support commands>>=
proc InstanceRemainingTime {instref event {sourceref {}}} {
    variable domain

    set target [CheckInstRefSymbol $instref]
    set eventNum [FindEventNumber [dict get $target Class] $event]
    if {$sourceref ne {}} {
        CheckInstRefSymbol $sourceref
        set sourceinst $sourceref
    } else {
        try {
            CheckInstRefSymbol self
            set sourceinst self
        } on error {} {
            set sourceinst NULL
        }
    }
    return "mrt_RemainingDelayTime($eventNum, $instref, $sourceinst)"
}
```

Obtaining an Instance Identifier

It is often useful for external operation to be given a token that can be used to identify an instance of a class. Directly passing instance reference pointers outside of a domain is **strongly discouraged**. Since class instances are stored in an array, the index into the array provides a convenient identifier.

`<%instance` *instvar* `instid %>`

instvar
The name of a "C" variable containing an instance reference which is to be signaled.

The `instvar` subccommand generates a "C" expression for an identifier which corresponds to the instance reference of *instvar*. The identifier values range from 0 to the number of instance of the class of *instvar* minus one.

```
<<generation support commands>>=
proc InstanceRefToId {instref} {
    CheckInstRefSymbol $instref
    return "mrt_InstanceIndex($instref)"
}
```

Instance Set Commands

The run-time code provides functions for instance sets and so we also provide embedded macro commands to access the instance set functions.

All instance set operations are accessed via the `instset` command

Iterating Over an Instance Set

`<%instset` *instsetvar* `foreachInstance` *instvar* `%>`
Loop Body
`<%end%>`

instsetvar
The name of a "C" variable which holds an instance set.

instvar
The name of a "C" variable into which a reference to a class instance is assigned.

The `foreachInstance` subcommand generates "C" statements to iterate over the instances contained in the instance reference set held in the variable named, *instsetvar*. For each instance reference in the set, *instvar* is assigned the value of the instance reference and *Loop Body* is executed.

```
<<generation support commands>>=
proc InstanceSetForeachInstance {set inst} {
    set setsym [CheckInstSetSymbol $set]

    set startlevel [GetBlock]
    set iter [CreateTempSymbol Ctype MRT_InstSetIterator Type SetIterator\
            Class [dict get $setsym Class]]
    append result [IndentToBlock [string cat\
        [linecomment "instset $set foreachInstance $inst"]\
        "MRT_InstSetIterator $iter ;\n"\
        "for (mrt_InstSetIterBegin(&$set, &$iter) ;\
        mrt_InstSetIterMore(&$iter) ; mrt_InstSetIterNext(&$iter)) \{\n"\
    ]]
    PushBlock

    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol [dict get $setsym Class] $inst]\
        "$inst = mrt_InstSetIterGet(&$iter) ;\n"\
    ]]

    PushContext InstanceSetForeachInstance $startlevel

    return $result
}
```

Selecting an Arbitrary Instance from an Instance Set

`<%instset` *instsetvar* `selectOneInstance` *instvar* `%>`

instsetvar
The name of a “C” variable which holds an instance set.

instvar
The name of a “C” variable into which a reference to a class instance is assigned.

The `selectOneInstance` subcommand generates “C” statements to select an arbitrary instance from the instance set held in the variable named, *instsetvar*. A reference to the selected instance is assigned to *instvar*. If *instsetvar* is empty, then *instvar* is set to `NULL`. If *instsetvar* contains more than one instance, the instance assigned to *instvar* is chosen arbitrarily from the set.

```
<<generation support commands>>=
proc InstanceSetSelectOneInstance {set inst} {
    set setsym [CheckInstSetSymbol $set]
    set iter [CreateTempSymbol Ctype MRT_InstSetIterator Type SetIterator\
            Class [dict get $setsym Class]]

    set result [IndentToBlock [string cat\
        [linecomment "instset $set selectOneInstance $inst"]\
        "MRT_InstSetIterator $iter ;\n"\
        "mrt_InstSetIterBegin(&$set, &$iter) ;\n"\
        [CreateInstRefSymbol [dict get $setsym Class] $inst]\
        "$inst = mrt_InstSetIterMore(&$iter) ? mrt_InstSetIterGet(&$iter) :\
                NULL ;\n"\
    ]]

    return $result
}
```

Testing for an Empty Set

`<%instset` *instsetvar* `empty%>`

instsetvar
The name of a “C” variable which holds an instance set.

The `empty` subcommand generates a “C” expression to test whether the instance set contained in the *instsetvar* variable contains zero elements.

```
<<generation support commands>>=
proc InstanceSetEmpty {set} {
    CheckInstSetSymbol $set
    return "mrt_InstSetEmpty(&$set)"
}
```

Testing for an Non-Empty Set

`<%instset` *instsetvar* `notempty%>`

instsetvar
The name of a "C" variable which holds an instance set.

The `notempty` subcommand generates a "C" expression to test whether the instance set contained in the *instsetvar* variable contains at least one element.

```
<<generation support commands>>=
proc InstanceSetNotEmpty {set} {
    return (![InstanceSetEmpty $set])
}
```

Number of Instances in a Set

`<%instset` *instsetvar* `cardinality%>`

instsetvar
The name of a "C" variable which holds an instance set.

The `cardinality` subcommand generates a "C" expression containing the number instances in the instance set contained in the *instsetvar* variable.

```
<<generation support commands>>=
proc InstanceSetCardinality {set} {
    CheckInstSetSymbol $set
    return "mrt_InstSetCardinality(&$set)"
}
```

Instance Set Equality

`<%instset` *instsetvar* `equal` *instsetvar2* `%>`

instsetvar
The name of a "C" variable which holds an instance set.

instsetvar2
The name of a "C" variable which holds an instance set.

The `equal` subcommand generates a "C" expression that evaluates to non-zero if the instance set held in *instsetvar* is equal to the instance set held in *instsetvar2*.

```
<<generation support commands>>=
proc InstanceSetEqual {set1 set2} {
    set sym1 [CheckInstSetSymbol $set1]
    CheckInstSetSymbol $set2 Class [dict get $sym1 Class]
    return "mrt_InstSetEqual(&$set1, &$set2)"
}
```

Instance Set Inequality

`<%instset` *instsetvar* `notequal` *instsetvar2* `%>`

instsetvar
The name of a "C" variable which holds an instance set.

instsetvar2
The name of a "C" variable which holds an instance set.

The `notequal` subcommand generates a "C" expression that evaluates to non-zero if the instance set held in *instsetvar* is **not** equal to the instance set held in *instsetvar2*.

```
<<generation support commands>>=
proc InstanceSetNotEqual {set1 set2} {
    return (![InstanceSetEqual $set1 $set2])
}
```

Adding to an Instance Set

`<%instset` *instsetvar* `add` *instref* `%>`

instsetvar
The name of a "C" variable which holds an instance set.

instref
The name of a "C" variable which holds an instance reference.

The `add` subcommand generates a "C" statement that adds the instance held in the *instref* variable to the instance set held in the *instservar* variable. It is not an error if *instref* is already contained in *instsetvar*.

```
<<generation support commands>>=
proc InstanceSetAdd {set inst} {
    set setsym [CheckInstSetSymbol $set]
    CheckInstRefSymbol $inst Class [dict get $setsym Class]
    return [IndentToBlock [string cat\
        [linecomment "instset $set add $inst"]\
        "mrt_InstSetAddInstance(&$set, $inst) ;\n"\
    ]]
}
```

Removing from an Instance Set

`<%instset` *instsetvar* `remove` *instref* `%>`

instsetvar
The name of a “C” variable which holds an instance set.

instref
The name of a “C” variable which holds an instance reference.

The `remove` subcommand generates a “C” statement that removes the instance held in the *instref* variable to the instance set held in the *instservar* variable. It is not an error if *instref* is not present in *instsetvar*.

```
<<generation support commands>>=
proc InstanceSetRemove {set inst} {
    set setsym [CheckInstSetSymbol $set]
    CheckInstRefSymbol $inst Class [dict get $setsym Class]
    return [IndentToBlock [string cat\
        [linecomment "instset $set remove $inst"]\
        "mrt_InstSetRemoveInstance(&$set, $inst) ;\n"\
    ]]
}
```

Instance Set Membership

`<%instset` *instsetvar* `contains` *instsetvar2* `%>`

instsetvar
The name of a “C” variable which holds an instance set.

instref
The name of a “C” variable which holds an instance reference.

The `contains` subcommand generates a “C” expression that evaluates to non-zero if the instance held in *instvar* is a member of the instance set held in *instsetvar*.

```
<<generation support commands>>=
proc InstanceSetContains {set inst} {
    set setsym [CheckInstSetSymbol $set]
    CheckInstRefSymbol $inst Class [dict get $setsym Class]
    return "mrt_InstSetMember(&$set, $inst)"
}
```

Instance Set Union

`<%instset` *resultsetvar* `union` *instsetvar1 instsetvar2 ?... ?* `%>`

instsetvarN
The name of a “C” variable which holds an instance set.

The `union` subcommand generates “C” statements that computes the set union of the *instsetvarN* instance sets placing the result in *resultsetvar*. The *instsetvarN* instance sets must contain instances of the same class.

```
<<generation support commands>>=
proc InstanceSetUnion {opset set1 set2 args} {
```

```
    tailcall InstanceSetOperation union mrt_InstSetUnion $opset $set1 $set2\
            {*}$args
}
```

Instance Set Intersection

<%instset *resultsetvar* intersect *instsetvar1 instsetvar2 ?... ?* %>

instsetvarN
The name of a "C" variable which holds an instance set.

The intersect subcommand generates "C" statements that computes the set intersection of the *instsetvarN* instance sets placing the result in *resultsetvar*. The *instsetvarN* instance sets must contain instances of the same class.

```
<<generation support commands>>=
proc InstanceSetIntersect {opset set1 set2 args} {
    tailcall InstanceSetOperation intersect mrt_InstSetIntersect $opset\
            $set1 $set2 {*}$args
}
```

We can factor the basic set operation code generation into a common procedure.

```
<<generation support commands>>=
proc InstanceSetOperation {label op opset set1 set2 args} {
    variable domain

    set sym1 [CheckInstSetSymbol $set1]
    set className [dict get $sym1 Class]
    CheckInstSetSymbol $set2 Class $className

    if {$opset in [concat [list $set1 $set2] $args]} {
        error "result set, \"$opset\", may not be one of the operation\
            arguments"
    }

    set classDesc [GetClassDescriptor $domain $className]
    append result\
        [linecomment "instset $opset $label $set1 [list $args]"]\
        [CreateInstSetSymbol $className $opset]\
        "mrt_InstSetInitialize(&$opset, &$classDesc) ;\n"\
        "$op\(&$set1, &$set2, &$opset\) ;\n"
    foreach set $args {
        CheckInstSetSymbol $set Class $className
        append result "$op\(&$opset, &$set, &$opset\) ;\n"
    }
    return [IndentToBlock $result]
}
```

Instance Set Difference

`<%instset` *resultsetvar* `minus` *instsetvar1 instsetvar2* `%>`

instsetvarN
The name of a "C" variable which holds an instance set.

The `minus` subcommand generates "C" statements that computes the set difference of the *instsetvar1* and *instvarset2* (in that order) placing the result in *resultsetvar*. The *instsetvarN* instance sets must contain instances of the same class.

```
<<generation support commands>>=
proc InstanceSetMinus {diffset set1 set2} {
    variable domain

    if {$diffset in [list $set1 $set2]} {
        error "result set, \"$diffset\", may not be one of the minus\
            operation arguments"
    }

    set sym1 [CheckInstSetSymbol $set1]
    set className [dict get $sym1 Class]
    set sym2 [CheckInstSetSymbol $set2 Class $className]

    set classDesc [GetClassDescriptor $domain $className]
    append result\
        [linecomment "instset $diffset minus $set1 $set2"]\
        [CreateInstSetSymbol $className $diffset]\
        "mrt_InstSetInitialize(&$diffset, &$classDesc) ;\n"\
        "mrt_InstSetMinus(&$set1, &$set2, &$diffset) ;\n"
    return [IndentToBlock $result]
}
```

Class Commands

For each class in a domain, the code generator creates a macro command that has the same name as the class. For example, if a domain contains a class named, **WashingMachine**, the there will be an embedded macro command named, `WashingMachine`, which can be invoked as:

`<%WashingMachine subcommand` ?*arg1 arg2* . . . ? `%>`

where `subcommand` is a class model level operation command.

In this section we describe the embedded macro commands that are available for each class.

Synchronous Instance Creation

<% *class* `create` *instref ?attr1 value1 att2 value2 ... ?* %>

class
The name of a domain class for which an instance is synchronously created.

instref
The name of a variable into which a reference to the newly created class instance is placed.

attrN / valueN
A set of attribute name / attribute values which are used to set the attribute values of the newly created instance.

The `create` subcommand for a class creates "C" statements to synchronously create a new instance of the class. A reference to the instance is placed in the variable called, *instref*. If *instref* has not been previously declared, it is declared automatically. The attributes of the newly created instance are set to the values given by the *attrN / valueN* pairs. All attributes of the instance must either be supplied a value as part of the invocation of the `create` subcommand or have default values declared for them as part of the class definition. Any value given as an argument overrides any default value that might have been defined for an attribute. If *class* has an associated state model, then the newly created instance is placed in the default initial state that was defined for the state model. No state activity is executed since the instance creation is done synchronously.

```
<<generation support commands>>=
proc ClassCreate {className instref args} {
    variable domain
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }

    set classDesc [GetClassDescriptor $domain $className]
    append result\
        [linecomment "$className create $instref [list $args]"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_CreateInstance(&$classDesc, MRT_StateCode_IG) ;\n"\
        [AssignComponentValues $className $instref $args]

    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc USubClassCreate {className instref args} {
    variable domain
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }

    # Find the relationship for which the class is a union subclass.
    set usubclass [UnionSubclass findWhere\
            {$Domain eq $domain && $Class eq $className && $Role eq "source"}]
    if {[isEmptyRef $usubclass]} {
        error "\"$className\" is not a union subclass"
    }

    # Get the superclass name.
    set supername [readAttribute [findRelated $usubclass R45 ~R44] Class]
    set relName [readAttribute $usubclass Relationship]
    set relDesc [GetRelationshipDescriptor $domain $relName]

    # Find the super class instance reference among the arguments.
    if {![dict exists $args $relName]} {
        error "no value supplied for superclass reference, \"$relName\""
    }
```

```
    set superinst [dict get $args $relName]
    CheckInstRefSymbol $superinst Class $supername

    set classDesc [GetClassDescriptor $domain $className]
    append result\
        [linecomment "$className create $instref [list $args]"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_CreateUnionInstance(&$classDesc, MRT_StateCode_IG, "\
        "$relDesc, $superinst) ;\n"\
        [AssignComponentValues $className $instref $args]

    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc AssignComponentValues {className instref suppliedvalues} {
    variable domain
    set result {}

    set popcomps [GetPopComponents $className]
    relation foreach popcomp $popcomps {
        relation assign $popcomp
        switch -exact -- $Type {
        association {
            if {![dict exists $suppliedvalues $Name]} {
                error "no value supplied for association reference, \"$Name\""
            } else {
                set assocrefvar [dict get $suppliedvalues $Name]
            }
            set refclass [pipe {
                AssociationReference findById Domain $Domain Class $Class\
                        Name $Name |
                findRelated % ~R90 R32 ~R33 |
                readAttribute % Class
            } {} |%]
            CheckInstRefSymbol $assocrefvar Class $refclass

            set relDesc [GetRelationshipDescriptor $Domain $Name]
            append result "mrt_CreateSimpleLinks($relDesc, $instref,\
                    $assocrefvar, true) ;\n"
        }
        associator {
            if {![dict exists $suppliedvalues $Name]} {
                error "no value supplied for associator reference, \"$Name\""
            } else {
                set atorrefs [dict get $suppliedvalues $Name]
            }
            set atorclass [pipe {
                AssociatorReference findById Domain $Domain Class $Class\
                        Name $Name |
                findRelated % ~R93
            } {} |%]
            set srcclass [pipe {
                findRelated $atorclass R42 ~R34 |
                readAttribute % Class
            } {} |%]
            set trgclass [pipe {
                findRelated $atorclass R42 ~R35 |
                readAttribute % Class
            } {} |%]

            # Reflexive case
```

```
            if {$srcclass eq $trgclass} {
                if {[dict exists $atorrefs backward]} {
                    set srcinstref [dict get $atorrefs backward]
                } else {
                    error "for reflexive associator references, expected\
                        \"backward\" reference: got \"$atorrefs\""
                }
                if {[dict exists $atorrefs forward]} {
                    set trginstref [dict get $atorrefs forward]
                } else {
                    error "for reflexive associator references, expected\
                        \"forward\" reference: got \"$atorrefs\""
                }
            } else {
                if {[dict exists $atorrefs $srcclass]} {
                    set srcinstref [dict get $atorrefs $srcclass]
                } else {
                    error "for associator references, expected\
                        \"$srcclass\" reference: got \"$atorrefs\""
                }
                if {[dict exists $atorrefs $trgclass]} {
                    set trginstref [dict get $atorrefs $trgclass]
                } else {
                    error "for associator references, expected\
                        \"$trgclass\" reference: got \"$atorrefs\""
                }
            }
            CheckInstRefSymbol $srcinstref Class $srcclass
            CheckInstRefSymbol $trginstref Class $trgclass

            set relDesc [GetRelationshipDescriptor $Domain $Name]
            append result "mrt_CreateAssociatorLinks($relDesc, $instref,\
                    $srcinstref, $trginstref) ;\n"
        }
        superclass {
            if {![dict exists $suppliedvalues $Name]} {
                error "no value supplied for superclass reference, \"$Name\""
            } else {
                set superrefvar [dict get $suppliedvalues $Name]
                if {$superrefvar eq {}} {
                    dict unset suppliedvalues $Name
                    continue
                }
            }
            set superref [pipe {
                SuperclassReference findById Domain $Domain Class $Class\
                        Name $Name |
                findRelated % ~R91 {~R47 ReferringSubclass} R37 ~R36
            } {} |%]
            if {[isNotEmptyRef $superref]} {
                CheckInstRefSymbol $superrefvar\
                        Class [readAttribute $superref Class]

                set relDesc [GetRelationshipDescriptor $Domain $Name]
                append result "mrt_CreateSimpleLinks($relDesc, $instref,\
                        $superrefvar, true) ;\n"
            }
        }
        attribute {
            if {![dict exists $suppliedvalues $Name]} {
                set defvalueref [DefaultValue findById Domain $Domain\
                        Class $Class Attribute $Name]
```

```
                if {[isEmptyRef $defvalueref]} {
                    error "attribute, \"$Name\", has no supplied value and\
                        no default value"
                }
                set attrValue [readAttribute $defvalueref Value]
            } else {
                set attrValue [dict get $suppliedvalues $Name]
            }
            set attrref [Attribute findById Domain $Domain Class $Class\
                    Name $Name]
            assignAttribute $attrref {DataType datatype}
            append result [GenValueAssignment $domain $instref->$Name\
                    $attrValue $datatype]
        }
        }
        dict unset suppliedvalues $Name
    }

    if {[dict size $suppliedvalues] != 0} {
        error "class, $Class, has no attribute(s) named,\
            \"[join [dict keys $suppliedvalues] {, }]\"\
            or the attribute(s) is zero initialized"
    }

    return $result
}

proc GetPopComponents {className} {
    variable domain

    set popcomps [PopulatedComponent findWhere\
            {$Domain eq $domain && $Class eq $className}]
    set typedcomps [pipe {
        findRelated $popcomps {~R21 Reference} {~R23 AssociationReference} |
        deRef % |
        relation extend % artup Type string {"association"}
    } {} |%]

    set typedcomps [pipe {
        findRelated $popcomps {~R21 Reference} {~R23 AssociatorReference} |
        deRef % |
        relation extend % artup Type string {"associator"} |
        relation union $typedcomps %
    } {} |%]

    set typedcomps [pipe {
        findRelated $popcomps {~R21 Reference} {~R23 SuperclassReference} |
        deRef % |
        relation extend % artup Type string {"superclass"} |
        relation union $typedcomps %
    } {} |%]

    set typedcomps [pipe {
        findRelated $popcomps {~R21 Attribute} {~R29 IndependentAttribute}\
                {~R19 ValueInitializedAttribute} |
        deRef % |
        relation extend % artup Type string {"attribute"} |
        relation union $typedcomps %
    } {} |%]
    # puts [relformat $typedcomps typedcomps]

    return $typedcomps
```

```
}

proc GetRelationshipDescriptor {domain name} {
    set relnum [pipe {
        Relationship findById Domain $domain Name $name |
        readAttribute ~ Number
    }]
    return &${domain}__RSHIPS\[$relnum\]
}
```

Synchronous Instance Creation in a Given State

<% *class* `createin` *instref state ?attr1 value1 att2 value2 ... ?* %>

class
The name of a domain class for which an instance is synchronously created.

instref
The name of a variable into which a reference to the newly created class instance is placed.

state
The name of the state into which the newly created instance is placed.

attrN / valueN
A set of attribute name / attribute values which are used to set the attribute values of the newly created instance.

The `createin` subcommand for a class generates "C" statements to synchronously create a new instance of the class in a given state. This subcommand is defined only for classes that have a state model. A reference to the instance is placed in the variable called, *instref*. If *instref* has not been previously declared, it is declared automatically. The attributes of the newly created instance are set to the values given by the *attrN / valueN* pairs. All attributes of the instance must either be supplied a value as part of the invocation of the `createin` subcommand or have default values declared for them as part of the class definition. Any value given as an argument overrides any default value that might have been defined for an attribute. The instance will start in the *state* state and **no** activity is executed since the instance creation is done synchronously.

```
<<generation support commands>>=
proc ClassCreateIn {className instref state args} {
    variable domain
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }

    if {$state eq "@"} {
        error "instances may not be synchronously created\
                in the pseudo-initial state, \"@\""
    }

    set statenum [GetStateNumber $domain $className $state]
    set classDesc [GetClassDescriptor $domain $className]

    append result\
        [linecomment "$className createin $instref [list $args]"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_CreateInstance(&$classDesc, $statenum) ;\n"\
        [AssignComponentValues $className $instref $args]

    return [IndentToBlock $result]
```

```
}
```

```
<<generation support commands>>=
proc USubClassCreateIn {className instref state args} {
    variable domain
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }

    if {$state eq "@"} {
        error "instances may not be synchronously created\
                in the pseudo-initial state, \"@\""
    }

    # Find the relationship for which the class is a union subclass.
    set usubclass [UnionSubclass findWhere\
            {$Domain eq $domain && $Class eq $className && $Role eq "source"}]
    if {[isEmptyRef $usubclass]} {
        error "\"$className\" is not a union subclass"
    }

    # Get the superclass name.
    set supername [readAttribute [findRelated $usubclass R45 ~R44] Class]
    set relName [readAttribute $usubclass Relationship]
    set relDesc [GetRelationshipDescriptor $domain $relName]

    # Find the super class instance reference among the arguments.
    if {![dict exists $args $relName]} {
        error "no value supplied for superclass reference, \"$relName\""
    }
    set superinst [dict get $args $relName]
    CheckInstRefSymbol $superinst Class $supername

    set statenum [GetStateNumber $domain $className $state]
    set classDesc [GetClassDescriptor $domain $className]

    append result\
        [linecomment "$className createin $instref [list $args]"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_CreateUnionInstance(&$classDesc, $statenum, "\
        "$relDesc, $superinst) ;\n"\
        [AssignComponentValues $className $instref $args]

    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc GetStateNumber {domain className state} {
    set stateRef [StatePlace findById Domain $domain Model $className\
            Name $state]
    if {[isEmptyRef $stateRef]} {
        error "class, $className, has no state named, $state"
    }
    return [readAttribute $stateRef Number]
}
```

Asynchronous Instance Creation

<% *class* createasync *event* \{ *?eparams?* } *?attr1 value1 att2 value2 . . . ?* %>

class
The name of a domain class for which an instance is asynchronously created.

event
The name of the creation event that is to be signaled to the newly created class instance.

eparams
A set of parameter name / value pairs that are the parameters of the creation event. If there are no parameters, is it sufficent to use the empty list ({}) as the command argument.

attrN / valueN
A set of attribute name / attribute values which are used to set the attribute values of the newly created instance.

The createasync subcommand for a class generates "C" statements to asynchronously create a new instance of the class by signaling *event* to it. Any parameters of the creation event are given by name / value pairs specified as the *eparams* list. The attributes of the newly created instance are set to the values given by the *attrN / valueN* pairs. All attributes of the instance must either be supplied a value as part of the invocation of the createasync subcommand or have default values declared for them as part of the class definition. Any value given as an argument overrides any default value that might have been defined for an attribute. This command only defined for classes that have a state model for which creation events have been specified. The instance is created and placed in it creation state awaiting delivery of *event*, whereupon it will transition and execute a state activity.

```
<<generation support commands>>=
proc ClassCreateAsync {className event eventparams args} {
    variable domain

    if {[llength $eventparams] % 2 != 0} {
        error "event parameters must be given as name / value pairs, got:\
            \"$eventparams\""
    }
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }
    set eventRef [Event findById Domain $domain Model $className Event $event]
    if {[isEmptyRef $eventRef]} {
        error "class, $className, has no event named, $event"
    }
    set eventNum [readAttribute $eventRef Number]

    append result\
        [linecomment "$className createasync $event [list $eventparams]\
            [list $args]"]

    lassign [GenCreationEventParams $className $eventRef $eventparams]\
        paramptr paramsize

    set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]

    lassign [CreateTempRefSymbol $className] reftype refvar
    set classDesc [GetClassDescriptor $domain $className]
    append result\
        "$reftype$refvar = mrt_CreateInstanceAsync(&$classDesc, $eventNum,\
            $paramptr, $paramsize, $sourceinst) ;\n"\
        [AssignComponentValues $className $refvar $args]

    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc GenCreationEventParams {className eventRef eventparams {resultRef result}} {
    variable domain
    upvar 1 $resultRef result

    set psig [findRelated $eventRef R69]
    set params [FindParamsFromSig $psig]
    if {[relation isnotempty $params]} {
        set required [relation list $params Name]
        set supplied [dict keys $eventparams]
        if {![::struct::set equal $required $supplied]} {
            error "event, $event, requires parameters,\
                \"[join $required {, }]\", got:\
                \"[join $supplied {, }]\""
        }

        set event [readAttribute $eventRef Event]
        set eparamsname ${domain}_${className}_${event}__EPARAMS
        set paramvar [CreateTempSymbol Ctype "struct *$eparamsname"\
            Type EventParams Class $className]

        append result\
            "struct $eparamsname $paramvar = \{\n"
        relation foreach param $params -ascending Position {
            relation assign $param Name
            append result "    .$Name = [dict get $eventparams $Name],\n"
        }
        append result "\} ;\n"

        set paramptr &$paramvar
        set paramsize sizeof($paramvar)
    } else {
        set paramptr NULL
        set paramsize 0
    }

    return [list $paramptr $paramsize]
}
```

```
<<generation support commands>>=
proc USubClassCreateAsync {className event eventparams args} {
    variable domain

    if {[llength $eventparams] % 2 != 0} {
        error "event parameters must be given as name / value pairs, got:\
            \"$eventparams\""
    }
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }

    # Find the relationship for which the class is a union subclass.
    set usubclass [UnionSubclass findWhere\
            {$Domain eq $domain && $Class eq $className && $Role eq "source"}]
    if {[isEmptyRef $usubclass]} {
        error "\"$className\" is not a union subclass"
    }

    # Get the superclass name.
    set supername [readAttribute [findRelated $usubclass R45 ~R44] Class]
    set relName [readAttribute $usubclass Relationship]
    set relDesc [GetRelationshipDescriptor $domain $relName]
```

```
    # Find the super class instance reference among the arguments.
    if {![dict exists $args $relName]} {
        error "no value supplied for superclass reference, \"$relName\""
    }
    set superinst [dict get $args $relName]
    CheckInstRefSymbol $superinst Class $supername

    set eventRef [Event findById Domain $domain Model $className Event $event]
    if {[isEmptyRef $eventRef]} {
        error "class, $className, has no event named, $event"
    }
    set eventNum [readAttribute $eventRef Number]

    append result\
        [linecomment "$className createasync $event [list $eventparams]\
            [list $args]"]

    lassign [GenCreationEventParams $className $eventRef $eventparams]\
        paramptr paramsize

    set sourceinst [expr {[LookUpSymbol self] eq {} ? "NULL" : "self"}]

    lassign [CreateTempRefSymbol $className] reftype refvar
    set classDesc [GetClassDescriptor $domain $className]
    append result\
        "$reftype$refvar = mrt_CreateUnionInstanceAsync(&$classDesc, $eventNum,\
            $paramptr, $paramsize, $sourceinst, $relDesc, $superinst) ;\n"\
        [AssignComponentValues $className $refvar $args]

    return [IndentToBlock $result]
}
```

Iterating Over Class Instances

<% *class* foreachInstance *instref* %> *Loop Body* <%end%>

class
The name of a domain class.

instref
The name of a variable into which a reference to a class instance is placed.

The foreachInstance subcommand for a class generates "C" statements to visit the instances of *class*. For each allocated instance of *class*, a reference to the instance is assigned to the *instref* variable and *Loop Body* is executed.

```
<<generation support commands>>=
proc ClassForeachInstance {className instref} {
    variable domain

    set iter [CreateTempSymbol Ctype {MRT_InstIterator}\
            Type InstanceIterator Class $className]

    set startlevel [GetBlock]

    set classDesc [GetClassDescriptor $domain $className]
    set result [IndentToBlock [string cat\
        [linecomment "$className foreachInstance $instref"]\
```

```
        "MRT_InstIterator $iter ;\n"\
        "for (mrt_InstIteratorStart(&$iter, &$classDesc) ;\
            mrt_InstIteratorMore(&$iter) ;\
            mrt_InstIteratorNext(&$iter)) \{\n"\
    ]]
    PushBlock
    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_InstIteratorGet(&$iter) ;\n"\
    ]]

    PushContext ClassForeachInstance $startlevel

    return $result
}
```

Conditional Iteration Over Class Instances

<% *class* foreachWhere *instref clause*%> *Loop Body* <%end%>

class
: The name of a domain class.

instref
: The name of a variable into which a reference to a class instance is placed.

where
: "C" code expression that can be evaluated to yield a boolean value.

The foreachWhere subcommand for a class generates "C" statements to visit conditionally the instances of *class*. For each allocated instance of *class*, a reference to the instance is assigned to the *instref* variable and *clause* is evaluated. If *clause* evaluates to non-zero, then *Loop Body* is executed. Otherwise, *Loop Body* is skipped.

```
<<generation support commands>>=
proc ClassForeachWhere {className instref where} {
    variable domain

    set startlevel [GetBlock]
    set where [string trim $where]

    set classDesc [GetClassDescriptor $domain $className]
    set iter [CreateTempSymbol Ctype {MRT_InstIterator}\
            Type InstanceIterator Class $className]
    set result [IndentToBlock [string cat\
        [linecomment "$className foreachWhere $instref [list $where]"]\
        "MRT_InstIterator $iter ;\n"\
        "for (mrt_InstIteratorStart(&$iter, &$classDesc) ;\
            mrt_InstIteratorMore(&$iter) ;\
            mrt_InstIteratorNext(&$iter)) \{\n"\
    ]]
    PushBlock
    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_InstIteratorGet(&$iter) ;\n"\
        "if ($where) \{\n"\
    ]]
    PushBlock
```

```
    PushContext ClassForeachWhere $startlevel

    return $result
}
```

Searching Class Instances

<% *class* findWhere *instref clause*%>

class
The name of a domain class.

instref
The name of a variable into which a reference to a class instance is placed.

where
"C" code expression that can be evaluated to yield a boolean value.

The findWhere subcommand for a class generates "C" statements to search the instances of *class*. A reference to the first allocated instance of *class* for which *clause* evaluates to true is assigned to the *instref* variable. If *clause* never evaluates to true then *instref* will have the value of NULL.

```
<<generation support commands>>=
proc ClassFindWhere {className instref where} {
    variable domain
    set where [string trim $where]

    set classDesc [GetClassDescriptor $domain $className]
    set iter [CreateTempSymbol Ctype {MRT_InstIterator}\
            Type InstanceIterator Class $className]
    set result [IndentToBlock [string cat\
        [linecomment "$className findWhere $instref [list $where]"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = NULL ;\n"\
        "MRT_InstIterator $iter ;\n"\
        "for (mrt_InstIteratorStart(&$iter, &$classDesc) ;\
            mrt_InstIteratorMore(&$iter) ;\
            mrt_InstIteratorNext(&$iter)) \{\n"\
    ]]
    PushBlock
    append result [IndentToBlock [string cat\
        "$instref = mrt_InstIteratorGet(&$iter) ;\n"\
        "if ($where) \{\n"\
    ]]
    PushBlock

    append result [IndentToBlock "break ;\n"]
    PopBlock

    append result [IndentToBlock "\} else \{\n"]
    PushBlock

    append result [IndentToBlock "$instref = NULL ;\n"]
    PopBlock

    append result [IndentToBlock "\}\n"]
    PopBlock
```

```
    append result [IndentToBlock "\}\n"]

    return $result
}
```

Conditionally Selecting Class Instances

<% *class* selectWhere *instref clause instset*%>

class
The name of a domain class.

instref
The name of a variable into which a reference to a class instance is placed.

where
"C" code expression that can be evaluated to yield a boolean value.

instset
The name of a variable which will contain an instance set.

The selectWhere subcommand for a class generates "C" statements to conditionally add instances of *class* to an instance set. For each allocated instance of *class*, a reference to the instance is assigned to the *instref* variable and *clause* is evaluated. If *clause* evaluates to non-zero, then the instance is added to the set being accumulated in the *instset* variable.

```
<<generation support commands>>=
proc ClassSelectWhere {className instref where instset} {
    variable domain
    set where [string trim $where]
    set result [IndentToBlock\
        [linecomment "$className selectWhere $instref [list $where] $instset"]\
    ]

    set startlevel [GetBlock]

    set classDesc [GetClassDescriptor $domain $className]
    append result [IndentToBlock [string cat\
        [CreateInstSetSymbol $className $instset]\
        "mrt_InstSetInitialize(&$instset, &$classDesc) ;\n"]\
    ]

    set iter [CreateTempSymbol Ctype {MRT_InstIterator}\
            Type InstanceIterator Class $className]
    append result [IndentToBlock [string cat\
        "MRT_InstIterator $iter ;\n"\
        "for (mrt_InstIteratorStart(&$iter, &$classDesc) ;\
            mrt_InstIteratorMore(&$iter) ;\
            mrt_InstIteratorNext(&$iter)) \{\n"\
    ]]
    PushBlock

    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_InstIteratorGet(&$iter) ;\n"\
        "if ($where) \{\n"\
    ]]
    PushBlock
```

```
    append result [IndentToBlock\
        "mrt_InstSetAddInstance(&$instset, $instref) ;\n"\
    ]

    set depth [expr {[GetBlock] - $startlevel}]
    for {set i 0} {$i < $depth} {incr i} {
        PopBlock
        append result [IndentToBlock "\}\n"]
    }

    return $result
}
```

Declaring Instance Reference Variables

<% *class* refvar *varname* %>

class
The name of a domain class.

varname
The name of a variable which will be declared as an instance reference to an instance of *class*.

The refvar subcommand for a class generates "C" statements declare *varname* to be a variable whose type is suitable to hold an instance reference to instances of *class*.

```
<<generation support commands>>=
proc ClassInstanceReference {className varName} {
    return [IndentToBlock [string cat\
        [linecomment "$className instref $varName"]\
        "[CreateInstRefSymbol $className $varName]"\
        "$varName = NULL ;\n"\
    ]]
}
```

Converting an ID to an Instance Reference

<% *class* idtoref *instid instref* %>

class
The name of a domain class.

instid
An instance identifier value. This is a small non-negative integer value.

instref
The name of a variable which will be set to a reference to an instance of *class*.

The idtoref subcommand for a class generates "C" statements to set the value of the *instref* variable to be an instance reference to an instance of *class*. The *instid* argument is a small integer value that is equivalent to the index into the storage pool array for *class*. It is a fatal system error to attempt to obtain an instance reference to an unallocated slot in the storage pool or if *instid* is not a valid instance identifier number.

```
<<generation support commands>>=
proc ClassIdToRef {className instid instref} {
    variable domain

    set classDesc [GetClassDescriptor $domain $className]

    return [IndentToBlock [string cat\
        [linecomment "$className idtoref $instref"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_InstanceReference(&$classDesc, $instid) ;\n"\
    ]]
}
```

Finding an Initial Instance by Name

`<% class findByName instname instref %>`

class
The name of a domain class.

instname
The name given to an instance as part of an initial instance population.

instref
The name of a variable which will be set to a reference to an instance of *class*.

The `findByName` subcommand for a class generates "C" statements to set the value of the *instref* variable to be an instance reference to an instance of *class* that was given the name, *name*, as part of an initial instance population. This subcommand provides a way to directly obtain a reference to a well know instance. It is a fatal system error to attempt to obtain an instance reference to the named instance if it has been deleted.

```
<<generation support commands>>=
proc ClassFindByName {className instName instref} {
    variable domain
    set instRef [ClassInstance findById Domain $domain Class $className\
        Instance $instName]
    if {[isEmptyRef $instRef]} {
        error "class, $className, has no instance named, $instName"
    }
    set classDesc [GetClassDescriptor $domain $className]
    set instNum [readAttribute $instRef Number]

    return [IndentToBlock [string cat\
        [linecomment "$className findByName $instName $instref"]\
        [CreateInstRefSymbol $className $instref]\
        "$instref = mrt_InstanceReference(&$classDesc, $instNum) ;\n"\
    ]]
}
```

Invoking a Class Operation

```
<<generation support commands>>=
proc ClassOperation {className opName args} {
    if {[llength $args] % 2 != 0} {
        error "operation parameters must be given as name / value pairs"
```

```
    }
    variable domain

    set opRef [Operation findWhere {$Domain eq $domain && $Class eq $className\
        && $Name eq $opName && !$IsInstance}]
    if {[isEmptyRef $opRef]} {
        error "unknown class operation, $opName, for class, $className"
    }

    set provided [dict keys $args]
    set params [deRef [findRelated $opRef ~R4]]
    set required [relation list $params Name -ascending Number]
    CheckSuppliedParams $opName $provided $required

    set pset {}
    foreach pname $required {
        append pset "[dict get $args $pname], "
    }
    set pset [string trimright $pset {, }]

    return "${className}_$opName\($pset\)"
}
```

Creating an Instance Set

```
<<generation support commands>>=
proc ClassInstanceSet {className varName} {
    variable domain
    set classDesc [GetClassDescriptor $domain $className]
    return [IndentToBlock [string cat\
        [linecomment "$className instset $varName"]\
        [CreateInstSetSymbol $className $varName]\
        "mrt_InstSetInitialize(&$varName, &$classDesc) ;\n"\
    ]]
}
```

Relationship Commands

Similar to classes, each relationship in the domain model will have an embedded macro command of the same name as the relationship.

Association Commands

<% *assoc* reference *sourceref targetref* %>

assoc
The name of a simple association.

sourceref
The name of a variable which contains an instance reference to a class instance of the class that serves the role of a referring class in *assoc*.

targetref
The name of a variable which contains an instance reference to a class instance of the class that serves the role of a referenced class in *assoc*.

When applied to a simple association, the `reference` command replaces the references for the association in the instance contained in `sourceref` so that it now refers to the instance contained in `targetref`.

```
<<generation support commands>>=
proc RelSimpleAssocSwap {relName sourceref targetref} {
    variable domain
    set relRef [SimpleAssociation findById Domain $domain Name $relName]
    set rship [GetRelationshipDescriptor $domain $relName]

    set sourceClass [readAttribute [findRelated $relRef ~R32] Class]
    CheckInstRefSymbol $sourceref Class $sourceClass

    set targetClass [readAttribute [findRelated $relRef ~R33] Class]
    CheckInstRefSymbol $targetref Class $targetClass

    return [IndentToBlock [string cat\
        [linecomment "$relName reference $sourceref $targetref"]\
        "mrt_CreateSimpleLinks($rship, $sourceref, $targetref, true) ;\n"\
    ]]
}
```

<% *assoc* reference *assocref partref* %>

assoc
The name of a simple association.

assocref
The name of a variable which contains an instance reference to a class instance of the class that serves the role of the associator class in *assoc*.

partref
The name of a variable which contains an instance reference to a class instance of the class that serves the role of a participant class in *assoc*.

When applied to a class based association, the `reference` command replaces one of the references in the instance contained in `assocref` so that it now refers to the instance contained in `partref`.

```
<<generation support commands>>=
proc RelClassAssocSwap {relName assocref partref {dir {}}} {
    variable domain
    set relRef [ClassBasedAssociation findById Domain $domain Name $relName]
    set rship [GetRelationshipDescriptor $domain $relName]
```

```
    set assocClass [readAttribute [findRelated $relRef ~R42] Class]
    set assocsym [CheckInstRefSymbol $assocref Class $assocClass]

    set sourceClass [readAttribute [findRelated $relRef ~R34] Class]
    set targetClass [readAttribute [findRelated $relRef ~R35] Class]

    set partrefSym [LookUpSymbol $partref]
    if {$partrefSym eq {}} {
        error "unknown participant reference symbol, \"$partref\""
    }
    set partClass [dict get $partrefSym Class]
    if {!($partClass eq $sourceClass || $partClass eq $targetClass)} {
        error "$partref is a reference to an instance of class, $partClass,\
            which does not participate in relationship, $relName"
    }

    set result [linecomment "$relName reference $assocref $partref $dir"]

    # Check for reflexive case
    if {$sourceClass eq $targetClass} {
        if {$dir eq {}} {
            error "for reflexive association, \"$relName\", the direction\
                of the reference must be specified"
        }
        switch -exact -- $dir {
            forward {
                set isForward true
            }
            backward {
                set isForward false
            }
            default {
                error "unknown association direction, \"$dir\""
            }
        }
    } else {
        set isForward true
    }
    append result\
        "mrt_CreateSimpleLinks($rship, $assocref, $partref, $isForward) ;\n"

    return [IndentToBlock $result]
}
```

```
<<generation support commands>>=
proc RelRefGenReclassify {relName subref newSubclass newref args} {
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }
    set result [linecomment "$relName reclassify $subref $newSubclass\
            $newref [list $args]"]

    variable domain
    set relRef [ReferenceGeneralization findById Domain $domain Name $relName]

    set subclasses [findRelated $relRef ~R37]
    CheckSubClassName $newSubclass $subclasses

    set subrefClass [dict get [CheckSymbol $subref] Class]
    CheckSubClassName $subrefClass $subclasses
```

```
    set classDesc [GetClassDescriptor $domain $newSubclass]
    set relDesc [GetRelationshipDescriptor $domain $relName]
    lappend args $relName {}
    append result\
        [CreateInstRefSymbol $newSubclass $newref]\
        "$newref = mrt_Reclassify\($relDesc, $subref, &$classDesc\) ;\n"\
        [AssignComponentValues $newSubclass $newref $args]

    if {$subref ne "self"} {
        append result "$subref = NULL ;\n" ;                # ❶
    }

    return [IndentToBlock $result]
}
```

❶ After reclassification, the instance reference is no longer valid. For references other than `self`, we assign `NULL` to the reference pointer value to indicate this. However, `self` is defined as a constant pointer, so it cannot be assigned.

```
<<generation support commands>>=
proc CheckSubClassName {subclass subRefs} {
    set subclassnames [pipe {
        deRef $subRefs |
        relation list ~ Class
    }]
    if {$subclass ni $subclassnames} {
        error "bad subclass, $subclass, should be one of:\
                [join $subclassnames {, }]"
    }
}
```

```
<<generation support commands>>=
proc RelUnionGenReclassify {relName subref newSubclass newref args} {
    if {[llength $args] % 2 != 0} {
        error "initializers must be given as name / value pairs"
    }
    set result [linecomment "$relName reclassify $subref $newSubclass\
            $newref [list $args]"]

    variable domain
    set relRef [UnionGeneralization findById Domain $domain Name $relName]

    set subclasses [findRelated $relRef ~R45]
    CheckSubClassName $newSubclass $subclasses

    set subrefClass [dict get [CheckSymbol $subref] Class]
    CheckSubClassName $subrefClass $subclasses

    set classDesc [GetClassDescriptor $domain $newSubclass]
    set relDesc [GetRelationshipDescriptor $domain $relName]
    lappend args $relName {}
    append result\
        [CreateInstRefSymbol $newSubclass $newref]\
        "$newref = mrt_Reclassify\($relDesc, $subref, &$classDesc\) ;\n"\
        [AssignComponentValues $newSubclass $newref $args]

    if {$subref ne "self"} {
        append result "$subref = NULL ;\n" ;                # ❶
    }

    return [IndentToBlock $result]
```

```
}
```

❶ See comment under `RelRefGenReclassify`.

```
<<generation support commands>>=
proc GenClassify {relName superref subref} {
    variable domain

    if {[llength [LookUpSymbol $subref]] != 0} {
        error "subclass reference symbol, \"$subref\", already exists:\
            this variable name must not already exist"
    }

    set startblock [GetBlock]

    set result [linecomment "$relName classify $superref $subref"]

    set relRef [Generalization findById Domain $domain Name $relName]

    set superclassref [Superclass findWhere {$Domain eq $domain &&\
            $Relationship eq $relName && $Role eq "target"}]
    set supername [readAttribute $superclassref Class]
    CheckInstRefSymbol $superref Class $supername

    set subclassref [Subclass findWhere {$Domain eq $domain &&\
            $Relationship eq $relName && $Role eq "source"}]
    set subclasses [pipe {
        deRef $subclassref |
        relation project ~ Class |
        relation list ~
    }]

    # calculate the address to the subclass,
    # calculation depends upon whether we are reference generalization
    # or union generalization
    set tinst [CreateTempSymbol Ctype {MRT_Instance *} Type Reference Class {}]
    append result "MRT_Instance *$tinst = "

    set rgref [findRelated $relRef {~R43 ReferenceGeneralization}]
    if {[isNotEmptyRef $rgref]} {
        set gentype reference
        append result "$superref->$relName ;\n"
    } else {
        set gentype union
        append result "(MRT_Instance *)&$superref->$relName ;\n"
    }
    set cindex [CreateTempSymbol Ctype ptrdiff_t Type Index Class {}]
    append result\
        "ptrdiff_t $cindex = $tinst->classDesc - ${domain}__CLASSES ;\n"\
        "switch ($cindex) \{\n"

    set result [IndentToBlock $result]

    PushBlock

    PushContext GenClassify $startblock
    actexpand cset subclasses $subclasses
    actexpand cset foundsubs [list]
    actexpand cset superref $superref
    actexpand cset subref $subref
```

```
    actexpand cset relname $relName
    actexpand cset gentype $gentype
    actexpand cset gotdefault false

    return $result
}
```

```
<<generation support commands>>=
proc SubclassCase {subclassname} {
    variable domain
    set startblock [GetBlock]

    set context [actexpand cname]
    if {$context ne "GenClassify"} {
        error "invoked subclass command outside of a classify context"
    }
    set subclasses [actexpand cget subclasses]
    if {![::struct::set contains $subclasses $subclassname]} {
        error "unknown subclass, \"$subclassname\""
    }

    set foundsubs [actexpand cget foundsubs]
    if {[::struct::set contains $foundsubs $subclassname]} {
        error "duplicate subclass, \"$subclassname\""
    }
    ::struct::set include foundsubs $subclassname
    actexpand cset foundsubs $foundsubs

    set classnum [GetClassProperty $subclassname Number]

    set result [IndentToBlock [string cat\
        [linecomment "subclass $subclassname"]\
        "case ${classnum}: \{\n"\
    ]]

    PushBlock

    set superref [actexpand cget superref]
    set subref [actexpand cget subref]
    set gentype [actexpand cget gentype]
    set relname [actexpand cget relname]
    set refResult [CreateInstRefSymbol $subclassname $subref]
    if {$gentype eq "reference"} {
        append refResult "$subref = $superref->$relname ;\n"
    } else {
        append refResult "$subref = &$superref->$relname.$subclassname ;\n"
    }

    append result [IndentToBlock $refResult]

    PushContext Subclass $startblock

    return $result
}
```

```
<<generation support commands>>=
proc DefaultSubclass {} {
    set startblock [GetBlock]

    set context [actexpand cname]
    if {$context ne "GenClassify"} {
```

```
        error "invoked default command outside of a classify context"
    }

    append result "default: \{\n"
    set result [IndentToBlock $result]

    PushBlock

    actexpand cset gotdefault true
    PushContext Subclass $startblock

    return $result
}
```

Assigner Commands

```
<<generation support commands>>=
proc SingleAssignerSignal {relName event args} {
    set reftype "struct $relName *"
    set instref [CreateTempSymbol Ctype $reftype Type Reference Class $relName]

    append result\
        [linecomment "$relName signal $event [list $args]"]\
        "$reftype$instref = &${relName}__POOL\[0\] ;\n"

    return [string cat\
        [IndentToBlock $result]\
        [InstanceSignal $instref $event {*}$args]\
    ]
}
```

```
<<generation support commands>>=
proc MultiAssignerFindIdInstance {relName idinst instref} {
    variable domain

    set maRef [MultipleAssigner findWhere {$Domain eq $domain &&\
            $Association eq $relName}]
    set idClass [readAttribute $maRef Class]
    CheckInstRefSymbol $idinst Class $idClass

    set startlevel [GetBlock]

    set classDesc [GetClassDescriptor $domain $relName]
    set iter [CreateTempSymbol Ctype {MRT_InstIterator}\
            Type InstanceIterator Class $relName]
    set asgnref [CreateTempSymbolName]
    set result [IndentToBlock [string cat\
        [linecomment "$relName findByIdInstance $idinst $instref"]\
        [CreateInstRefSymbol $relName $instref]\
        "$instref = NULL ;\n"\
        "MRT_InstIterator $iter ;\n"\
        "for (mrt_InstIteratorStart(&$iter, &$classDesc) ;\
            mrt_InstIteratorMore(&$iter) ;\
            mrt_InstIteratorNext(&$iter)) \{\n"\
    ]]
    PushBlock
    append result [IndentToBlock [string cat\
        [CreateInstRefSymbol $relName $asgnref]\
        "$asgnref = mrt_InstIteratorGet(&$iter) ;\n"\
```

```
        "if ($asgnref->idinstance == $idinst) \{\n"\
    ]]

    PushBlock
    append result [IndentToBlock [string cat\
        "$instref = $asgnref ;\n"\
        "break ;\n"\
    ]]
    PopBlock
    append result [IndentToBlock "\}\n"]
    PopBlock
    append result [IndentToBlock "\}\n"]

    return $result
}
```

```
<<generation support commands>>=
proc MultiAssignerCreate {relName instref idinst} {
    variable domain

    set maRef [MultipleAssigner findWhere {$Domain eq $domain &&\
            $Association eq $relName}]
    set idClass [readAttribute $maRef Class]
    CheckInstRefSymbol $idinst Class $idClass

    set assignerClass [GetClassDescriptor $domain $relName]
    append result\
        [linecomment "$relName create $instref $idinst"]\
        [CreateInstRefSymbol $relName $instref]\
        "$instref = mrt_CreateInstance(&$assignerClass, MRT_StateCode_IG) ;\n"\
        "$instref->idinstance = $idinst ;\n"

    return [IndentToBlock $result]
}
```

Chapter 41

Helper Commands

```
<<generation helpers namespace>>=
namespace eval Helpers {
    ::logger::import -all -force -namespace log micca

    <<tclral imports>>

    namespace path {
        ::micca
        ::rosea::InstCmds
    }
    namespace import ::micca::@Config@::Helpers::typeCheck

    <<generation helper data>>
    <<generation helper commands>>
}
```

```
<<generation helper commands>>=
proc banner {} {
    set marker [string repeat - 70]
    string cat\
        "/*\n"\
        " * $marker\n"\
        " * THIS FILE IS AUTOMATICALLY GENERATED. DO NOT EDIT.\n"\
        " * Created by: $::argv0 $::argv\n"\
        " * Created on: [clock format [clock seconds]]\n"\
        " * This is micca version $::micca::version\n"\
        " * $marker\n"\
        " */\n"
}
```

```
<<generation helper commands>>=
proc comment {args} {
    set result "/*\n"
    foreach c $args {
        append result [::textutil::adjust::indent\
            [::textutil::adjust::adjust $c] { * }]\n
    }
    append result " */\n"

    return $result
}
```

```
<<generation helper commands>>=
```

```
proc linecomment {text} {
    set result {}
    foreach line [split $text \n] {
        append result "// $line\n"
    }
    return $result
}
```

```
<<generation helper commands>>=
proc blockcomment {block} {
    return [textutil::adjust::indent [textutil::adjust::undent\
            [string trim $block \n]] {// }]\n
}
```

```
<<generation helper commands>>=
proc linedirective {line file} {
    upvar #0 ::micca::@Gen@::options options
    set result {}
    if {[dict get $options lines]} {
        append result "#line [expr {$line + 1}]"
        if {$file ne {}} {
            append result " \"$file\""
        }
        append result "\n"
    }
    return $result
}
```

```
<<generation helper commands>>=
proc indentCode {code {indent 4}} {
    return [textutil::adjust::indent [textutil::adjust::undent $code]\
            [string repeat { } $indent]]\n
}
```

```
<<generation helper commands>>=
proc GenInstanceAddress {domainName className instName} {
    set path {}
    set usubRef [UnionSubclass findWhere {$Domain eq $domainName &&\
        $Class eq $className}]
    if {[isNotEmptyRef $usubRef]} {
        # puts [relformat [deRef $usubRef] usubRef]
        while {[isNotEmptyRef $usubRef]} {
            assignAttribute $usubRef {Relationship rship} {Class subName}
            set path .$rship.${subName}$path

            set usuperRef [findRelated $usubRef R45 ~R44]
            set superClass [readAttribute $usuperRef Class]

            set usubRef [pipe {
                deRef $usuperRef |
                relation semijoin ~ $::micca::UnionSubclass\
                    -using {Domain Domain Class Class} |
                ::rosea::Helpers::ToRef ::micca::UnionSubclass ~
            }]
            # puts [relformat [deRef $usubRef] usubRef]
        }

        # puts "path = \"$path\""
        set storageClass $superClass
    } else {
```

```
        set storageClass $className
    }
    set instNumber [readAttribute\
            [ClassInstance findById Domain $domainName Class $className\
                Instance $instName]\
        Number]
    return &${storageClass}__POOL\[$instNumber\]$path
}
```

Part VII

Code Organization

In this section we show the organization of the files that can be tangled from the literate source.

Chapter 42

Legal Information

Copyright Information

This software is copyrighted. It is licensed in the same manner as Tcl itself.

```
<<copyright info>>=
# This software is copyrighted 2015 - 2023 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
```

Version Information

```
<<version info>>=
1.4.0
```

Edit Warning

```
<<edit warning>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
```

Chapter 43

Source Code

We start with some preliminaries. `Micca` is implemented in Tcl. `Micca` is also a rosea based appplication, so, we need the `rosea` package.

```
<<required packages>>=
package require rosea
```

We will find it convenient to import some of the commands from TclRAL since we will be using them repeatedly.

```
<<tclral imports>>=
namespace import\
    ::ral::relation\
    ::ral::tuple\
    ::ral::relformat\
    ::ralutil::pipe
```

The `::ralutil::pipe` command is used extensively in the code for `micca`. This command has nothing to do with relational algebra but rather is a control structure. It allows a sequence of operations to be written as a linear "pipe" where the result of one command is used as an argument to the next. It turns out that such sequencing is common in relational algebraic processing. The command rewrites a linear sequence of commands into a set of nested procedure invocations. It is a classic Tcl control structure type of procedure and makes it much easier to understand a sequence of operations without having to tease apart the procedure nesting or introduce superfluous variables.

```
<<micca.tcl>>=
<<edit warning>>
<<copyright info>>

package require Tcl 8.6
package require logger
package require logger::utils
package require logger::appender

<<required packages>>

rosea configure {
    domain micca {
        <<micca configuration>>
    }
}
rosea generate micca

namespace eval ::micca {
    variable version\
    <<version info>>
```

```
    <<logger setup>>

    <<tclral imports>>
    namespace import ::ral::relvar

    <<micca constraints>>
    <<configuration commands namespace>>
    <<generation commands namespace>>
}

rosea populate {
    domain micca {
        <<micca population>>
    }
}

package provide micca $::micca::version
```

```
<<logger setup>>=
set logger [::logger::init micca]
set appenderType [expr {[dict exist [fconfigure stdout] -mode] ?\
        "colorConsole" : "console"}]
::logger::utils::applyAppender -appender $appenderType -serviceCmd $logger\
        -appenderArgs {-conversionPattern {%c: \[%p\] '%m'}}
::logger::import -all -force -namespace log micca
```

Micca Starpack Application

Tcl applications can be packages as a single file executable with no external dependencies. This is known as a “starpack”. When `micca` is packaged as an application in a starpack, this code is the main entry point.

```
<<micca_main.tcl>>=
<<edit warning>>
<<copyright info>>

set iswrapped [expr {[lindex [file system [info script]] 0] ne "native"}]
if {$iswrapped} {
    set libdir [file join $::starkit::topdir lib]
    set appdir [file join $libdir application]
    set libs [list]
    if {$::tcl_platform(os) eq "Linux"} {
        set libs [glob -nocomplain -directory $libdir P-linux*]
    } elseif {$::tcl_platform(os) eq "Darwin"} {
        set libs [glob -nocomplain -directory $libdir P-macosx*]
    }
    foreach lib $libs {
        lappend ::auto_path $lib
    }
} else {
    set appdir [file dirname [info script]]
}

source [file join $appdir micca.tcl]
package require micca
package require cmdline

set cmdOpts {
    {version {Print out version information and exit}}
    {stubexternalops {Include external operation code in output}}
```

```
    {nogenerate {Don't generate any output files}}
    {lines {Generate #line directives}}
    {save.arg {} {Save the domain configuration to a file}}
    {savesqlite.arg {} {Save the domain configuration to a SQLite database}}
    {posix {Output the POSIX run time code and exit}}
    {msp432 {Output the arm7m run time code and exit}}
    {efm32gg {Output the arm7m run time code and exit}}
    {msp430 {Output the msp430 run time code and exit}}
    {doc {Output a copy of the manual documentation and exit}}
    {expanderror.arg fail {How macro expansion errors are handled}}
    {debug.secret {Turn on debugging output}}
}

set usage "\[options] file1 file2 ...\noptions:\n"
try {
    set options [cmdline::getoptions argv $cmdOpts $usage]
} on error {result} {
    puts stderr $result
    exit 1
}

if {[dict get $options version]} {
    chan puts "micca: version $::micca::version"
    chan puts {
<<copyright info>>
}
    exit 0
} elseif {[dict get $options posix]} {
    file copy -force [file join $appdir posix micca_rt.h] ./micca_rt.h
    file copy -force [file join $appdir posix micca_rt_internal.h]\
            ./micca_rt_internal.h
    file copy -force [file join $appdir posix micca_rt.c] ./micca_rt.c
    exit 0
} elseif {[dict get $options msp432]} {
    file copy -force [file join $appdir arm-7m msp432 micca_rt.h] ./micca_rt.h
    file copy -force [file join $appdir arm-7m msp432 micca_rt_internal.h]\
            ./micca_rt_internal.h
    file copy -force [file join $appdir arm-7m msp432 micca_rt.c] ./micca_rt.c
    exit 0
} elseif {[dict get $options efm32gg]} {
    file copy -force [file join $appdir arm-7m efm32gg micca_rt.h] ./micca_rt.h
    file copy -force [file join $appdir arm-7m efm32gg micca_rt_internal.h]\
            ./micca_rt_internal.h
    file copy -force [file join $appdir arm-7m efm32gg micca_rt.c] ./micca_rt.c
    exit 0
} elseif {[dict get $options msp430]} {
    file copy -force [file join $appdir msp430 micca_rt.h] ./micca_rt.h
    file copy -force [file join $appdir msp430 micca_rt_internal.h]\
            ./micca_rt_internal.h
    file copy -force [file join $appdir msp430 micca_rt.c] ./micca_rt.c
    exit 0
} elseif {[dict get $options doc]} {
    file copy -force [file join $appdir HTML] ./miccadoc
    exit 0
}

set nerrs 0
foreach file $argv {
    try {
        micca configureFromFile $file
    } on error result {
        puts stderr $result
```

```
        incr nerrs
    }
}
if {$nerrs != 0} {
    exit $nerrs
}

if {[dict get $options save] ne {}} {
    ral serializeToFile [dict get $options save] {::micca::[A-Z]*}
}

if {[dict get $options savesqlite] ne {}} {
    ral storeToSQLite [dict get $options savesqlite] {::micca::[A-Z]*}
}

if {![dict get $options nogenerate]} {
    try {
        set genfiles [micca generate {*}$options]
    } on error {result} {
        puts stderr $result
        exit 1
    }
}

exit 0
```

Part VIII

Reference Materials

Appendix A

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers who are unfamiliar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to the Tcl interpreter.

Bibliography

Books

[1] [mb-xuml] Stephen J. Mellor and Marc J. Balcer, Executable UML: a foundation for model-driven architecture, Addison-Wesley (2002), ISBN 0-201-74804-5.

[2] [rs-xuml] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, Model Driven Architecture with Executable UML, Cambridge University Press (2004), ISBN 0-521-53771-1.

[3] [mtoc] Leon Starr, Andrew Mangogna and Stephen Mellor, Models to Code: With No Mysterious Gaps, Apress (2017), ISBN 978-1-4842-2216-4

[4] [ls-build], Leon Starr, How to Build Shlaer-Mellor Object Models, Yourdon Press (1996), ISBN 0-13-207663-2.

[5] [sm-data] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in Data, Prentice-Hall (1988), ISBN 0-13-629023-X.

[6] [sm-states] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in States, Prentice-Hall (1992), ISBN 0-13-629940-7.

Articles

[7] [ls-articulate] Leon Starr, How to Build Articulate UML Class Models, 2008, http://www.modelint.com/how-to-build-articulate-uml-class-models/

[8] [ls-time] Leon Starr, Time and Synchronization in Executable UML, 2008, http://www.modelint.com/time-and-synchronization-in-executable-uml/

Index

A
Activity Generation
add, 607
assign, 581
attr, 579
cancelsignal, 602
cardinality, 606
classify, 629
classop, 624
contains, 608
create, 611, 632
createasync, 617, 618
createin, 615, 616
default, 630
delayremaining, 603
delaysignal, 600, 601
delete, 583
empty, 605
end, 576
equal, 606
externalop, 578
findByName, 624
findOneRelated, 594
findRelatedWhere, 593
findWhere, 621
foreachInstance, 604, 619
foreachRelated, 585
foreachRelatedWhere, 592
foreachWhere, 620
idtoref, 623
instid, 603
instop, 583
instref, 623
instset, 625
intersect, 609
minus, 610
notempty, 606, 607
reclassify, 627
reference, 626
remove, 608
selectOneInstance, 605
selectRelated, 596
selectRelatedWhere, 597
selectWhere, 622
signal, 598, 631
subclass, 630
union, 608
update, 580
activityDeclarations, 527
activityDefinitions, 558
add, 607
AppendToDomainAttribute, 184
ARG_FORMAT, 202
assign, 581
assigner, 246
assignerDeclarations, 524
assignerDefinitions, 541
assignerInstanceDefinitions, 554
ASSOC_OPTIONS, 191
association, 189
attr, 579
ATTR_OPTIONS, 202
attribute, 201

B
BAD_CREATION_TARGET, 227
BAD_MULTIPLE_OPT, 192
BAD_NAME, 181
BAD_RELATIONSHIP_SPEC, 192
BAD_STATE_NAME, 220

C
cancelsignal, 602
cardinality, 606
CheckDuplicate, 283
CheckExists, 283
CheckPopAttrs, 240
Class
ClothesTub, 11
Motor, 12
WashingCycle, 11
WashingMachine, 10
WaterLevelSensor, 13
WaterValve, 12
class, 184, 236
classDeclarations, 520
classDefinitions, 538
classify, 629
classInstanceDefinitions, 548
classop, 212, 624
CleanUpCommand, 178
clear, 175
ClothesTub, 11, 17
Code Generation

activityDeclarations, 527
activityDefinitions, 558
assignerDeclarations, 524
assignerDefinitions, 541
assignerInstanceDefinitions, 554
classDeclarations, 520
classDefinitions, 538
classInstanceDefinitions, 548
ctorDeclarations, 526
ctorDefinitions, 556
DeclareClassStructure, 521
DeclareUnionSubclassStructures, 520
domainCtorDefinition, 560
domainInclude, 518
domainNameString, 518
domainOpDefinitions, 561
dtorDeclarations, 527
dtorDefinitions, 557
edbDefinitions, 532
epilogueDeclarations, 564
externalOpDefinitions, 563
FindNonUnionSubclasses, 521
FindReferencedClass, 524
formulaDeclarations, 527
formulaDefinitions, 557
forwardClassDeclarations, 519
forwardRelationshipDeclaration, 519
iabDefinitions, 530
implementationTypeAliases, 518
nameDefinitions, 529
operationDeclarations, 526
operationDefinitions, 555
pdbDefinitions, 536
portalDefinitions, 564
prologueDeclarations, 518
relationshipDefinitions, 543
stateParamDeclarations, 525
storageDeclarations, 528
COMPONENT_MISMATCH, 245
Config
AppendToDomainAttribute, 184
assigner, 246
association, 189
attribute, 201
class, 184, 236
classop, 212
ConfigEvaluate, 177
constructor, 214
defaulttrans, 228
DefineOperation, 213
destructor, 215
domain, 180
domainop, 199, 233
eentity, 197
epilogue, 183
event, 224
final, 228
generalization, 195
HandleConfigError, 274
identifyby, 231
initialstate, 227
instop, 213
interface, 183
miccaClear, 178
miccaConfigure, 176
polymorphic, 203
population, 235
prologue, 183
state, 219
statemodel, 216, 230
transition, 225
typealias, 198
CONFIG_ERRORS, 177, 500
ConfigEvaluate, 177
configure, 173
configureFromChan, 174
configureFromFile, 174
constructor, 214
contains, 608
create, 611, 632
createasync, 617, 618
CreateDefaultedValues, 241
createin, 615, 616
CreateLinkedRefs, 269
CreateNilDestBackRefs, 262
CreateNilLinkedBackRefs, 270
CreateNilLinkedForwRefs, 271
CreateNilLinkedRefs, 270
CreateNilSourceForwRefs, 265
CreateRequiredValues, 240
createWasher, 28
CreateZeroInitValues, 241
ctorDeclarations, 526
ctorDefinitions, 556

D

DeclareClassStructure, 521
DeclareUnionSubclassStructures, 520
DeclError, 279
default, 630
defaulttrans, 228
DefineOperation, 213
delayremaining, 603
delaysignal, 600, 601
delete, 583
deleteWasher, 28
destructor, 215
domain, 180
Domain Operation
createWasher, 28
deleteWasher, 28
init, 30
selectCycle, 29
startWasher, 29

domainCtorDefinition, 560
domainInclude, 518
domainNameString, 518
domainop, 199, 233
domainOpDeclarations, 510
domainOpDefinitions, 561
dtorDeclarations, 527
dtorDefinitions, 557
DUP_ELEMENT_NAME, 283
DUPLICATE_SUBCLASS, 197

E
edbDefinitions, 532
eentity, 197
empty, 605
end, 576
EntityError, 283
epilogue, 183
epilogueDeclarations, 564
equal, 606
error code
    ARG_FORMAT, 202
    ASSOC_OPTIONS, 191
    ATTR_OPTIONS, 202
    BAD_CREATION_TARGET, 227
    BAD_MULTIPLE_OPT, 192
    BAD_NAME, 181
    BAD_RELATIONSHIP_SPEC, 192
    BAD_STATE_NAME, 220
    COMPONENT_MISMATCH, 245
    CONFIG_ERRORS, 177, 500
    DUP_ELEMENT_NAME, 283
    DUPLICATE_SUBCLASS, 197
    EXPECTED_INT, 238
    EXPECTED_NONNEG, 238
    EXPECTED_NONTRANS_STATE, 228
    FINAL_OUTTRANS, 218
    MISSING_POPULATION, 248
    MISSING_VALUES, 240
    NEED_ASSOCIATOR, 192
    NO_INSTANCES, 249
    NOINIT_VALUES, 240
    REFLEXIVE_NOT_ALLOWED, 192
    SUPER_AS_SUBCLASS, 197
    TILDE_NAME, 191
    TOO_FEW_SUBCLASSES, 197
    UNION_SUBCLASS_EXISTS, 197
    UNKNOWN_INIT, 240
Error Codes, 406
event, 224
eventParamDeclarations, 511
Example
    Class
        ClothesTub, 11
        Motor, 12
        WashingCycle, 11
        WashingMachine, 10
        WaterLevelSensor, 13
        WaterValve, 12
    Domain Operation
        createWasher, 28
        deleteWasher, 28
        init, 30
        selectCycle, 29
        startWasher, 29
    Operation
        findByCycleType, 27
        findByMachineID, 26
    State Model
        ClothesTub, 17
        WashingMachine, 15
EXPECTED_INT, 238
EXPECTED_NONNEG, 238
EXPECTED_NONTRANS_STATE, 228
externalop, 578
externalOpDeclarations, 511
externalOpDefinitions, 563

F
final, 228
FINAL_OUTTRANS, 218
FindAllPopComps, 241
FindArgumentSignature, 223
findByCycleType, 27
findByMachineID, 26
findByName, 624
FindDefaultableAttrs, 242
FindInterfaceTypeAliases, 508
FindNilDestBackRefs, 262
FindNilSourceForwRefs, 265
FindNoInitializeAttrs, 243
FindNonUnionSubclasses, 521
findOneRelated, 594
FindParameterSignature, 221
FindReferencedClass, 524
findRelatedWhere, 593
FindRequiredPopComps, 242
FindSimpleBackRefs, 266
FindSimpleReferringRefs, 266
FindSourceForwRefs, 268
findSubclassesOf, 209
FindTargetBackRefs, 266
FindUltimateSuperclasses, 205
findWhere, 621
FindZeroInitAttrs, 242
foreachInstance, 604, 619
foreachRelated, 585
foreachRelatedWhere, 592
foreachWhere, 620
formulaDeclarations, 527
formulaDefinitions, 557
forwardClassDeclarations, 519
forwardRelationshipDeclaration, 519

G
GatherClassProperties, 501
Gen
    miccaGenerate, 499
generalization, 195
GenNumber, 280
GetClassDescriptor, 506
GetClassProperty, 506

H
HandleConfigError, 274
Header Generation
    domainOpDeclarations, 510
    eventParamDeclarations, 511
    externalOpDeclarations, 511
    headerFileGuard, 508
    interface, 508
    interfaceTypeAliases, 509
    portalDeclaration, 515
    portalIds, 512
headerFileGuard, 508
Helpers
    CheckDuplicate, 283
    CheckExists, 283
    CheckPopAttrs, 240
    CleanUpCommand, 178
    CreateDefaultedValues, 241
    CreateLinkedRefs, 269
    CreateNilDestBackRefs, 262
    CreateNilLinkedBackRefs, 270
    CreateNilLinkedForwRefs, 271
    CreateNilLinkedRefs, 270
    CreateNilSourceForwRefs, 265
    CreateRequiredValues, 240
    CreateZeroInitValues, 241
    DeclError, 279
    EntityError, 283
    FindAllPopComps, 241
    FindArgumentSignature, 223
    FindDefaultableAttrs, 242
    FindInterfaceTypeAliases, 508
    FindNilDestBackRefs, 262
    FindNilSourceForwRefs, 265
    FindNoInitializeAttrs, 243
    FindParameterSignature, 221
    FindRequiredPopComps, 242
    FindSimpleBackRefs, 266
    FindSimpleReferringRefs, 266
    FindSourceForwRefs, 268
    findSubclassesOf, 209
    FindTargetBackRefs, 266
    FindUltimateSuperclasses, 205
    FindZeroInitAttrs, 242
    GatherClassProperties, 501
    GenNumber, 280
    GetClassDescriptor, 506
    GetClassProperty, 506
    PropagatePolyEvents, 205
    ResolveInitialValue, 243
    typeverifier, 281

I
iabDefinitions, 530
identifyby, 231
idtoref, 623
implementationTypeAliases, 518
init, 30
initialstate, 227
instid, 603
instop, 213, 583
instref, 623
instset, 625
interface, 183, 508
interfaceTypeAliases, 509
intersect, 609

M
main, 289
micca
    Activity Generation
    add, 607
    assign, 581
    attr, 579
    cancelsignal, 602
    cardinality, 606
    classify, 629
    classop, 624
    contains, 608
    create, 611, 632
    createasync, 617, 618
    createin, 615, 616
    default, 630
    delayremaining, 603
    delaysignal, 600, 601
    delete, 583
    empty, 605
    end, 576
    equal, 606
    externalop, 578
    findByName, 624
    findOneRelated, 594
    findRelatedWhere, 593
    findWhere, 621
    foreachInstance, 604, 619
    foreachRelated, 585
    foreachRelatedWhere, 592
    foreachWhere, 620
    idtoref, 623
    instid, 603
    instop, 583
    instref, 623
    instset, 625
    intersect, 609
    minus, 610

notempty, 606, 607
reclassify, 627
reference, 626
remove, 608
selectOneInstance, 605
selectRelated, 596
selectRelatedWhere, 597
selectWhere, 622
signal, 598, 631
subclass, 630
union, 608
update, 580
Code Generation
activityDeclarations, 527
activityDefinitions, 558
assignerDeclarations, 524
assignerDefinitions, 541
assignerInstanceDefinitions, 554
classDeclarations, 520
classDefinitions, 538
classInstanceDefinitions, 548
ctorDeclarations, 526
ctorDefinitions, 556
DeclareClassStructure, 521
DeclareUnionSubclassStructures, 520
domainCtorDefinition, 560
domainInclude, 518
domainNameString, 518
domainOpDefinitions, 561
dtorDeclarations, 527
dtorDefinitions, 557
edbDefinitions, 532
epilogueDeclarations, 564
externalOpDefinitions, 563
FindNonUnionSubclasses, 521
FindReferencedClass, 524
formulaDeclarations, 527
formulaDefinitions, 557
forwardClassDeclarations, 519
forwardRelationshipDeclaration, 519
iabDefinitions, 530
implementationTypeAliases, 518
nameDefinitions, 529
operationDeclarations, 526
operationDefinitions, 555
pdbDefinitions, 536
portalDefinitions, 564
prologueDeclarations, 518
relationshipDefinitions, 543
stateParamDeclarations, 525
storageDeclarations, 528
Config
AppendToDomainAttribute, 184
assigner, 246
association, 189
attribute, 201
class, 184, 236
classop, 212
ConfigEvaluate, 177
constructor, 214
defaulttrans, 228
DefineOperation, 213
destructor, 215
domain, 180
domainop, 199, 233
eentity, 197
epilogue, 183
event, 224
final, 228
generalization, 195
HandleConfigError, 274
identifyby, 231
initialstate, 227
instop, 213
interface, 183
miccaClear, 178
miccaConfigure, 176
polymorphic, 203
population, 235
prologue, 183
state, 219
statemodel, 216, 230
transition, 225
typealias, 198
Gen
miccaGenerate, 499
Header Generation
domainOpDeclarations, 510
eventParamDeclarations, 511
externalOpDeclarations, 511
headerFileGuard, 508
interface, 508
interfaceTypeAliases, 509
portalDeclaration, 515
portalIds, 512
Helpers
CheckDuplicate, 283
CheckExists, 283
CheckPopAttrs, 240
CleanUpCommand, 178
CreateDefaultedValues, 241
CreateLinkedRefs, 269
CreateNilDestBackRefs, 262
CreateNilLinkedBackRefs, 270
CreateNilLinkedForwRefs, 271
CreateNilLinkedRefs, 270
CreateNilSourceForwRefs, 265
CreateRequiredValues, 240
CreateZeroInitValues, 241
DeclError, 279
EntityError, 283
FindAllPopComps, 241
FindArgumentSignature, 223
FindDefaultableAttrs, 242

FindInterfaceTypeAliases, 508
FindNilDestBackRefs, 262
FindNilSourceForwRefs, 265
FindNoInitializeAttrs, 243
FindParameterSignature, 221
FindRequiredPopComps, 242
FindSimpleBackRefs, 266
FindSimpleReferringRefs, 266
FindSourceForwRefs, 268
findSubclassesOf, 209
FindTargetBackRefs, 266
FindUltimateSuperclasses, 205
FindZeroInitAttrs, 242
GatherClassProperties, 501
GenNumber, 280
GetClassDescriptor, 506
GetClassProperty, 506
PropagatePolyEvents, 205
ResolveInitialValue, 243
typeverifier, 281
operation
clear, 175
configure, 173
configureFromChan, 174
configureFromFile, 174
Portal
Error Codes, 406
Portal Data
MRT_DomainPortal, 405
Portal Function
mrt_PortalAssignerCurrentState, 424
mrt_PortalCancelDelayedEvent, 416
mrt_PortalClassAttributeCount, 427
mrt_PortalClassAttributeName, 430
mrt_PortalClassAttributeSize, 431
mrt_PortalClassEventCount, 428
mrt_PortalClassEventName, 432
mrt_PortalClassInstanceCount, 427
mrt_PortalClassName, 426
mrt_PortalClassStateName, 433
mrt_PortalCreateInstance, 418
mrt_PortalCreateInstanceAsync, 419
mrt_PortalDeleteInstance, 420
mrt_PortalDomainClassCount, 425
mrt_PortalDomainName, 424
mrt_PortalErrorString, 407
mrt_PortalGetAttrRef, 409
mrt_PortalInstanceCurrentState, 423
mrt_PortalReadAttr, 410
mrt_PortalRemainingDelayTime, 417
mrt_PortalSignalDelayedEvent, 415
mrt_PortalSignalEvent, 413
mrt_PortalSignalEventToAssigner, 421
mrt_PortalStateCount, 429
mrt_PortalUpdateAttr, 412
mrtPortalGetAssignerRef, 422
mrtPortalGetInstRef, 408
mrtPortalNewECB, 414
Run Time Constant
MRT_ECB_PARAM_SIZE, 372
MRT_EVENT_POOL_SIZE, 371
MRT_INSTANCE_SET_SIZE, 316
MRT_StateCode_CH, 299
MRT_StateCode_IG, 299
MRT_SYNC_QUEUE_SIZE, 436
Run Time Data
MRT_ActivityFunction, 396
MRT_AllocStatus, 299
MRT_ArrayRef, 330
MRT_AssignerId, 405
MRT_AssociatorRole, 331
MRT_AssocRole, 330
MRT_AttrFormula, 300
MRT_Attribute, 300
MRT_AttrId, 405
MRT_AttrOffset, 399
MRT_AttrSize, 405
MRT_AttrType, 299
MRT_Cardinality, 330
MRT_Class, 301
MRT_ClassAssociation, 332
MRT_ClassId, 405
MRT_DelayTime, 371
MRT_DispatchCount, 396
MRT_ecb, 370
MRT_edb, 395
MRT_ErrorCode, 448
MRT_EventCode, 371
MRT_EventParams, 372
MRT_EventQueue, 372
MRT_EventType, 370
MRT_FatalErrorHandler, 452
MRT_FDServiceFunc, 464
MRT_FDServiceMap, 465
MRT_gdb, 399
MRT_iab, 302
MRT_Instance, 298
MRT_InstId, 405
MRT_InstIterator, 313
MRT_InstSet, 316
MRT_InstSetIterator, 324
MRT_LevelCount, 335
MRT_LinkRef, 330
MRT_pdb, 399
MRT_RefCount, 299
MRT_RefGeneralization, 333
MRT_RefStorageType, 330
MRT_RefSubClassRole, 332
MRT_Relationship, 333
MRT_RelType, 330
MRT_SimpleAssociation, 331
MRT_StateCode, 299
MRT_SubclassCode, 441
MRT_SuperClassRole, 332

MRT_SyncBlock, 437
MRT_SyncFunc, 436
MRT_SyncParams, 436
MRT_SyncQueue, 437
MRT_TraceHandler, 442
MRT_TraceInfo, 440
MRT_TransLevel, 335, 336
MRT_UnionGeneralization, 333
mrtErrHandler, 452
mrtErrorMsgs, 449
mrtTraceHandler, 442
mrtTransEntries, 336
mrtTransStorage, 336
Run Time Function
main, 289
mrt_BeginSyncService, 335
mrt_CancelDelayedEvent, 387
mrt_CanCreateInstance, 454
mrt_CanSignalEvent, 455
mrt_CreateAssociatorLinks, 358
mrt_CreateInstance, 308
mrt_CreateInstanceAsync, 379
mrt_CreateSimpleLinks, 353
mrt_CreateUnionInstance, 309
mrt_CreateUnionInstanceAsync, 380
mrt_DeleteInstance, 312
mrt_DispatchSingleEvent, 295
mrt_DispatchThreadOfControl, 294
mrt_EndSyncService, 335
mrt_EventLoop, 292
mrt_Initialize, 290
mrt_InstanceIndex, 306
mrt_InstanceReference, 306
mrt_InstIteratorGet, 314
mrt_InstIteratorMore, 314
mrt_InstIteratorNext, 315
mrt_InstIteratorStart, 313
mrt_InstSetAddInstance, 317
mrt_InstSetCardinality, 320
mrt_InstSetEmpty, 320
mrt_InstSetEqual, 321
mrt_InstSetInitialize, 317
mrt_InstSetIntersect, 323
mrt_InstSetIterBegin, 325
mrt_InstSetIterGet, 326
mrt_InstSetIterMore, 326
mrt_InstSetIterNext, 327
mrt_InstSetMember, 319
mrt_InstSetMinus, 324
mrt_InstSetRemoveInstance, 318
mrt_InstSetUnion, 322
mrt_NewEvent, 375
mrt_Panic, 455
mrt_PostDelayedEvent, 382, 383
mrt_PostEvent, 376
mrt_Reclassify, 366
mrt_RegisterTraceHandler, 442
mrt_RemainingDelayTime, 388
mrt_SetFatalErrorHandler, 452
mrt_SyncToEventLoop, 293
mrt_TimerExpireService, 390
mrt_TrySyncRequest, 455
mrtAddRelToCheck, 338
mrtCantHappenError, 449
mrtCheckAssociatorRefs, 345
mrtCheckDupAssociator, 359
mrtCheckRefCounts, 344
mrtCheckRelationship, 340
mrtCompareAtMostOne, 344
mrtCompareExactlyOne, 344
mrtCompareOneOrMore, 344
mrtCountArrayRefs, 348
mrtCountAssocRefs, 346
mrtCountClassAssocRefs, 349
mrtCountGenRefs, 350
mrtCountLinkedListRefs, 349
mrtCountSingularRefs, 347
mrtCountUnionRefs, 351
mrtDecrTransLevel, 336
mrtDefaultFatalErrorHandler, 452
mrtDeleteLinks, 361
mrtDiscardTrans, 337
mrtDispatchCreationEvent, 402
mrtDispatchEvent, 394
mrtDispatchEventFromQueue, 393
mrtDispatchPolymorphicEvent, 400
mrtDispatchTransitionEvent, 397
mrtDupAssociatorError, 451
mrtECBalloc, 374
mrtECBfree, 374
mrtECBPoolInit, 373
mrtEndTransaction, 339
mrtEventInFlightError, 450
mrtEventQueueBegin, 372
mrtEventQueueEmpty, 373
mrtEventQueueEnd, 373
mrtEventQueueInsert, 373
mrtEventQueueRemove, 373
mrtExpireDelayedEvent, 392
mrtExpirePeriodicEvent, 392
mrtFatalError, 453
mrtFindEvent, 374
mrtFindInstSlot, 303
mrtFindRefGenSubclassCode, 358
mrtFindRelEntry, 338
mrtFindUnionGenSubclassCode, 368
mrtFinishThreadOfControl, 296
mrtGetStorageProperties, 305
mrtIncrAllocCounter, 311
mrtIncrRefCount, 345
mrtIncrTransLevel, 335
mrtIndexToInstance, 307
mrtInitializeInstance, 310
mrtInsertDelayedEvent, 385

mrtInsertTrans, 337
mrtInvokeOneSyncFunction, 439
mrtLink, 355
mrtLinkRefBegin, 457
mrtLinkRefEmpty, 457
mrtLinkRefEnd, 457
mrtLinkRefInit, 457
mrtLinkRefInsert, 458
mrtLinkRefNotEmpty, 457
mrtLinkRefRemove, 458
mrtMarkRelationship, 339
mrtNextInstSlot, 304
mrtNoInstSlotError, 450
mrtPrintTraceInfo, 444
mrtQueueExpiredDelayedEvents, 389
mrtRefIntegrityError, 451
mrtRemoveDelayedEvent, 386
mrtRemoveTrans, 337
mrtRunThreadOfControl, 296
mrtStartDelayedQueueTiming, 389
mrtStopDelayedQueueTiming, 390
mrtSyncQueueEmpty, 437
mrtSyncQueueGet, 438
mrtSyncQueuePut, 437
mrtTraceCreationEvent, 443
mrtTracePolymorphicEvent, 443
mrtTraceTransitionEvent, 443
mrtTransactionsInit, 337
mrtUnallocSlotError, 450
mrtUnlinkBackref, 364
mrtZeroRefCounts, 343
mtrProcessTOCEvent, 296
Run Time POSIX Data
mrtSigMask, 459
Run Time POSIX Function
MRT_RegisterSignal, 460
mrtBeginCriticalSection, 460
mrtEndCriticalSection, 460
mrtPlatformInit, 470
static data
mrtExitEventLoop, 293
mrtInWhiteState, 295
miccaClear, 178
miccaConfigure, 176
miccaGenerate, 499
minus, 610
MISSING_POPULATION, 248
MISSING_VALUES, 240
Motor, 12
MRT_ActivityFunction, 396
MRT_AllocStatus, 299
MRT_ArrayRef, 330
MRT_AssignerId, 405
MRT_AssociatorRole, 331
MRT_AssocRole, 330
MRT_AttrFormula, 300
MRT_Attribute, 300
MRT_AttrId, 405
MRT_AttrOffset, 399
MRT_AttrSize, 405
MRT_AttrType, 299
mrt_BeginSyncService, 335
mrt_CancelDelayedEvent, 387
mrt_CanCreateInstance, 454
mrt_CanSignalEvent, 455
MRT_Cardinality, 330
MRT_Class, 301
MRT_ClassAssociation, 332
MRT_ClassId, 405
mrt_CreateAssociatorLinks, 358
mrt_CreateInstance, 308
mrt_CreateInstanceAsync, 379
mrt_CreateSimpleLinks, 353
mrt_CreateUnionInstance, 309
mrt_CreateUnionInstanceAsync, 380
MRT_DelayTime, 371
mrt_DeleteInstance, 312
MRT_DispatchCount, 396
mrt_DispatchSingleEvent, 295
mrt_DispatchThreadOfControl, 294
MRT_DomainPortal, 405
MRT_ecb, 370
MRT_ECB_PARAM_SIZE, 372
MRT_edb, 395
mrt_EndSyncService, 335
MRT_ErrorCode, 448
MRT_EVENT_POOL_SIZE, 371
MRT_EventCode, 371
mrt_EventLoop, 292
MRT_EventParams, 372
MRT_EventQueue, 372
MRT_EventType, 370
MRT_FatalErrorHandler, 452
MRT_FDServiceFunc, 464
MRT_FDServiceMap, 465
MRT_gdb, 399
MRT_iab, 302
mrt_Initialize, 290
MRT_Instance, 298
MRT_INSTANCE_SET_SIZE, 316
mrt_InstanceIndex, 306
mrt_InstanceReference, 306
MRT_InstId, 405
MRT_InstIterator, 313
mrt_InstIteratorGet, 314
mrt_InstIteratorMore, 314
mrt_InstIteratorNext, 315
mrt_InstIteratorStart, 313
MRT_InstSet, 316
mrt_InstSetAddInstance, 317
mrt_InstSetCardinality, 320
mrt_InstSetEmpty, 320
mrt_InstSetEqual, 321
mrt_InstSetInitialize, 317

mrt_InstSetIntersect, 323
MRT_InstSetIterator, 324
mrt_InstSetIterBegin, 325
mrt_InstSetIterGet, 326
mrt_InstSetIterMore, 326
mrt_InstSetIterNext, 327
mrt_InstSetMember, 319
mrt_InstSetMinus, 324
mrt_InstSetRemoveInstance, 318
mrt_InstSetUnion, 322
MRT_LevelCount, 335
MRT_LinkRef, 330
mrt_NewEvent, 375
mrt_Panic, 455
MRT_pdb, 399
mrt_PortalAssignerCurrentState, 424
mrt_PortalCancelDelayedEvent, 416
mrt_PortalClassAttributeCount, 427
mrt_PortalClassAttributeName, 430
mrt_PortalClassAttributeSize, 431
mrt_PortalClassEventCount, 428
mrt_PortalClassEventName, 432
mrt_PortalClassInstanceCount, 427
mrt_PortalClassName, 426
mrt_PortalClassStateName, 433
mrt_PortalCreateInstance, 418
mrt_PortalCreateInstanceAsync, 419
mrt_PortalDeleteInstance, 420
mrt_PortalDomainClassCount, 425
mrt_PortalDomainName, 424
mrt_PortalErrorString, 407
mrt_PortalGetAttrRef, 409
mrt_PortalInstanceCurrentState, 423
mrt_PortalReadAttr, 410
mrt_PortalRemainingDelayTime, 417
mrt_PortalSignalDelayedEvent, 415
mrt_PortalSignalEvent, 413
mrt_PortalSignalEventToAssigner, 421
mrt_PortalStateCount, 429
mrt_PortalUpdateAttr, 412
mrt_PostDelayedEvent, 382, 383
mrt_PostEvent, 376
mrt_Reclassify, 366
MRT_RefCount, 299
MRT_RefGeneralization, 333
MRT_RefStorageType, 330
MRT_RefSubClassRole, 332
MRT_RegisterSignal, 460
mrt_RegisterTraceHandler, 442
MRT_Relationship, 333
MRT_RelType, 330
mrt_RemainingDelayTime, 388
mrt_SetFatalErrorHandler, 452
MRT_SimpleAssociation, 331
MRT_StateCode, 299
MRT_StateCode_CH, 299
MRT_StateCode_IG, 299
MRT_SubclassCode, 441
MRT_SuperClassRole, 332
MRT_SYNC_QUEUE_SIZE, 436
MRT_SyncBlock, 437
MRT_SyncFunc, 436
MRT_SyncParams, 436
MRT_SyncQueue, 437
mrt_SyncToEventLoop, 293
mrt_TimerExpireService, 390
MRT_TraceHandler, 442
MRT_TraceInfo, 440
MRT_TransLevel, 335, 336
mrt_TrySyncRequest, 455
MRT_UnionGeneralization, 333
mrtAddRelToCheck, 338
mrtBeginCriticalSection, 460
mrtCantHappenError, 449
mrtCheckAssociatorRefs, 345
mrtCheckDupAssociator, 359
mrtCheckRefCounts, 344
mrtCheckRelationship, 340
mrtCompareAtMostOne, 344
mrtCompareExactlyOne, 344
mrtCompareOneOrMore, 344
mrtCountArrayRefs, 348
mrtCountAssocRefs, 346
mrtCountClassAssocRefs, 349
mrtCountGenRefs, 350
mrtCountLinkedListRefs, 349
mrtCountSingularRefs, 347
mrtCountUnionRefs, 351
mrtDecrTransLevel, 336
mrtDefaultFatalErrorHandler, 452
mrtDeleteLinks, 361
mrtDiscardTrans, 337
mrtDispatchCreationEvent, 402
mrtDispatchEvent, 394
mrtDispatchEventFromQueue, 393
mrtDispatchPolymorphicEvent, 400
mrtDispatchTransitionEvent, 397
mrtDupAssociatorError, 451
mrtECBalloc, 374
mrtECBfree, 374
mrtECBPoolInit, 373
mrtEndCriticalSection, 460
mrtEndTransaction, 339
mrtErrHandler, 452
mrtErrorMsgs, 449
mrtEventInFlightError, 450
mrtEventQueueBegin, 372
mrtEventQueueEmpty, 373
mrtEventQueueEnd, 373
mrtEventQueueInsert, 373
mrtEventQueueRemove, 373
mrtExitEventLoop, 293
mrtExpireDelayedEvent, 392
mrtExpirePeriodicEvent, 392

mrtFatalError, 453
mrtFindEvent, 374
mrtFindInstSlot, 303
mrtFindRefGenSubclassCode, 358
mrtFindRelEntry, 338
mrtFindUnionGenSubclassCode, 368
mrtFinishThreadOfControl, 296
mrtGetStorageProperties, 305
mrtIncrAllocCounter, 311
mrtIncrRefCount, 345
mrtIncrTransLevel, 335
mrtIndexToInstance, 307
mrtInitializeInstance, 310
mrtInsertDelayedEvent, 385
mrtInsertTrans, 337
mrtInvokeOneSyncFunction, 439
mrtInWhiteState, 295
mrtLink, 355
mrtLinkRefBegin, 457
mrtLinkRefEmpty, 457
mrtLinkRefEnd, 457
mrtLinkRefInit, 457
mrtLinkRefInsert, 458
mrtLinkRefNotEmpty, 457
mrtLinkRefRemove, 458
mrtMarkRelationship, 339
mrtNextInstSlot, 304
mrtNoInstSlotError, 450
mrtPlatformInit, 470
mrtPortalGetAssignerRef, 422
mrtPortalGetInstRef, 408
mrtPortalNewECB, 414
mrtPrintTraceInfo, 444
mrtQueueExpiredDelayedEvents, 389
mrtRefIntegrityError, 451
mrtRemoveDelayedEvent, 386
mrtRemoveTrans, 337
mrtRunThreadOfControl, 296
mrtSigMask, 459
mrtStartDelayedQueueTiming, 389
mrtStopDelayedQueueTiming, 390
mrtSyncQueueEmpty, 437
mrtSyncQueueGet, 438
mrtSyncQueuePut, 437
mrtTraceCreationEvent, 443
mrtTraceHandler, 442
mrtTracePolymorphicEvent, 443
mrtTraceTransitionEvent, 443
mrtTransactionsInit, 337
mrtTransEntries, 336
mrtTransStorage, 336
mrtUnallocSlotError, 450
mrtUnlinkBackref, 364
mrtZeroRefCounts, 343
mtrProcessTOCEvent, 296

N

nameDefinitions, 529
NEED_ASSOCIATOR, 192
NO_INSTANCES, 249
NOINIT_VALUES, 240
notempty, 606, 607

O

Operation
    findByCycleType, 27
    findByMachineID, 26
operation
    clear, 175
    configure, 173
    configureFromChan, 174
    configureFromFile, 174
operationDeclarations, 526
operationDefinitions, 555

P

pdbDefinitions, 536
polymorphic, 203
population, 235
Portal
    Error Codes, 406
Portal Data
    MRT_DomainPortal, 405
Portal Function
    mrt_PortalAssignerCurrentState, 424
    mrt_PortalCancelDelayedEvent, 416
    mrt_PortalClassAttributeCount, 427
    mrt_PortalClassAttributeName, 430
    mrt_PortalClassAttributeSize, 431
    mrt_PortalClassEventCount, 428
    mrt_PortalClassEventName, 432
    mrt_PortalClassInstanceCount, 427
    mrt_PortalClassName, 426
    mrt_PortalClassStateName, 433
    mrt_PortalCreateInstance, 418
    mrt_PortalCreateInstanceAsync, 419
    mrt_PortalDeleteInstance, 420
    mrt_PortalDomainClassCount, 425
    mrt_PortalDomainName, 424
    mrt_PortalErrorString, 407
    mrt_PortalGetAttrRef, 409
    mrt_PortalInstanceCurrentState, 423
    mrt_PortalReadAttr, 410
    mrt_PortalRemainingDelayTime, 417
    mrt_PortalSignalDelayedEvent, 415
    mrt_PortalSignalEvent, 413
    mrt_PortalSignalEventToAssigner, 421
    mrt_PortalStateCount, 429
    mrt_PortalUpdateAttr, 412
    mrtPortalGetAssignerRef, 422
    mrtPortalGetInstRef, 408
    mrtPortalNewECB, 414
portalDeclaration, 515
portalDefinitions, 564

portalIds, 512
prologue, 183
prologueDeclarations, 518
PropagatePolyEvents, 205

R
reclassify, 627
reference, 626
REFLEXIVE_NOT_ALLOWED, 192
relationshipDefinitions, 543
remove, 608
ResolveInitialValue, 243
Run Time Constant
MRT_ECB_PARAM_SIZE, 372
MRT_EVENT_POOL_SIZE, 371
MRT_INSTANCE_SET_SIZE, 316
MRT_StateCode_CH, 299
MRT_StateCode_IG, 299
MRT_SYNC_QUEUE_SIZE, 436
Run Time Data
MRT_ActivityFunction, 396
MRT_AllocStatus, 299
MRT_ArrayRef, 330
MRT_AssignerId, 405
MRT_AssociatorRole, 331
MRT_AssocRole, 330
MRT_AttrFormula, 300
MRT_Attribute, 300
MRT_AttrId, 405
MRT_AttrOffset, 399
MRT_AttrSize, 405
MRT_AttrType, 299
MRT_Cardinality, 330
MRT_Class, 301
MRT_ClassAssociation, 332
MRT_ClassId, 405
MRT_DelayTime, 371
MRT_DispatchCount, 396
MRT_ecb, 370
MRT_edb, 395
MRT_ErrorCode, 448
MRT_EventCode, 371
MRT_EventParams, 372
MRT_EventQueue, 372
MRT_EventType, 370
MRT_FatalErrorHandler, 452
MRT_FDServiceFunc, 464
MRT_FDServiceMap, 465
MRT_gdb, 399
MRT_iab, 302
MRT_Instance, 298
MRT_InstId, 405
MRT_InstIterator, 313
MRT_InstSet, 316
MRT_InstSetIterator, 324
MRT_LevelCount, 335
MRT_LinkRef, 330
MRT_pdb, 399
MRT_RefCount, 299
MRT_RefGeneralization, 333
MRT_RefStorageType, 330
MRT_RefSubClassRole, 332
MRT_Relationship, 333
MRT_RelType, 330
MRT_SimpleAssociation, 331
MRT_StateCode, 299
MRT_SubclassCode, 441
MRT_SuperClassRole, 332
MRT_SyncBlock, 437
MRT_SyncFunc, 436
MRT_SyncParams, 436
MRT_SyncQueue, 437
MRT_TraceHandler, 442
MRT_TraceInfo, 440
MRT_TransLevel, 335, 336
MRT_UnionGeneralization, 333
mrtErrHandler, 452
mrtErrorMsgs, 449
mrtTraceHandler, 442
mrtTransEntries, 336
mrtTransStorage, 336
Run Time Function
main, 289
mrt_BeginSyncService, 335
mrt_CancelDelayedEvent, 387
mrt_CanCreateInstance, 454
mrt_CanSignalEvent, 455
mrt_CreateAssociatorLinks, 358
mrt_CreateInstance, 308
mrt_CreateInstanceAsync, 379
mrt_CreateSimpleLinks, 353
mrt_CreateUnionInstance, 309
mrt_CreateUnionInstanceAsync, 380
mrt_DeleteInstance, 312
mrt_DispatchSingleEvent, 295
mrt_DispatchThreadOfControl, 294
mrt_EndSyncService, 335
mrt_EventLoop, 292
mrt_Initialize, 290
mrt_InstanceIndex, 306
mrt_InstanceReference, 306
mrt_InstIteratorGet, 314
mrt_InstIteratorMore, 314
mrt_InstIteratorNext, 315
mrt_InstIteratorStart, 313
mrt_InstSetAddInstance, 317
mrt_InstSetCardinality, 320
mrt_InstSetEmpty, 320
mrt_InstSetEqual, 321
mrt_InstSetInitialize, 317
mrt_InstSetIntersect, 323
mrt_InstSetIterBegin, 325
mrt_InstSetIterGet, 326
mrt_InstSetIterMore, 326

mrt_InstSetIterNext, 327
mrt_InstSetMember, 319
mrt_InstSetMinus, 324
mrt_InstSetRemoveInstance, 318
mrt_InstSetUnion, 322
mrt_NewEvent, 375
mrt_Panic, 455
mrt_PostDelayedEvent, 382, 383
mrt_PostEvent, 376
mrt_Reclassify, 366
mrt_RegisterTraceHandler, 442
mrt_RemainingDelayTime, 388
mrt_SetFatalErrorHandler, 452
mrt_SyncToEventLoop, 293
mrt_TimerExpireService, 390
mrt_TrySyncRequest, 455
mrtAddRelToCheck, 338
mrtCantHappenError, 449
mrtCheckAssociatorRefs, 345
mrtCheckDupAssociator, 359
mrtCheckRefCounts, 344
mrtCheckRelationship, 340
mrtCompareAtMostOne, 344
mrtCompareExactlyOne, 344
mrtCompareOneOrMore, 344
mrtCountArrayRefs, 348
mrtCountAssocRefs, 346
mrtCountClassAssocRefs, 349
mrtCountGenRefs, 350
mrtCountLinkedListRefs, 349
mrtCountSingularRefs, 347
mrtCountUnionRefs, 351
mrtDecrTransLevel, 336
mrtDefaultFatalErrorHandler, 452
mrtDeleteLinks, 361
mrtDiscardTrans, 337
mrtDispatchCreationEvent, 402
mrtDispatchEvent, 394
mrtDispatchEventFromQueue, 393
mrtDispatchPolymorphicEvent, 400
mrtDispatchTransitionEvent, 397
mrtDupAssociatorError, 451
mrtECBalloc, 374
mrtECBfree, 374
mrtECBPoolInit, 373
mrtEndTransaction, 339
mrtEventInFlightError, 450
mrtEventQueueBegin, 372
mrtEventQueueEmpty, 373
mrtEventQueueEnd, 373
mrtEventQueueInsert, 373
mrtEventQueueRemove, 373
mrtExpireDelayedEvent, 392
mrtExpirePeriodicEvent, 392
mrtFatalError, 453
mrtFindEvent, 374
mrtFindInstSlot, 303
mrtFindRefGenSubclassCode, 358
mrtFindRelEntry, 338
mrtFindUnionGenSubclassCode, 368
mrtFinishThreadOfControl, 296
mrtGetStorageProperties, 305
mrtIncrAllocCounter, 311
mrtIncrRefCount, 345
mrtIncrTransLevel, 335
mrtIndexToInstance, 307
mrtInitializeInstance, 310
mrtInsertDelayedEvent, 385
mrtInsertTrans, 337
mrtInvokeOneSyncFunction, 439
mrtLink, 355
mrtLinkRefBegin, 457
mrtLinkRefEmpty, 457
mrtLinkRefEnd, 457
mrtLinkRefInit, 457
mrtLinkRefInsert, 458
mrtLinkRefNotEmpty, 457
mrtLinkRefRemove, 458
mrtMarkRelationship, 339
mrtNextInstSlot, 304
mrtNoInstSlotError, 450
mrtPrintTraceInfo, 444
mrtQueueExpiredDelayedEvents, 389
mrtRefIntegrityError, 451
mrtRemoveDelayedEvent, 386
mrtRemoveTrans, 337
mrtRunThreadOfControl, 296
mrtStartDelayedQueueTiming, 389
mrtStopDelayedQueueTiming, 390
mrtSyncQueueEmpty, 437
mrtSyncQueueGet, 438
mrtSyncQueuePut, 437
mrtTraceCreationEvent, 443
mrtTracePolymorphicEvent, 443
mrtTraceTransitionEvent, 443
mrtTransactionsInit, 337
mrtUnallocSlotError, 450
mrtUnlinkBackref, 364
mrtZeroRefCounts, 343
mtrProcessTOCEvent, 296
Run Time POSIX Data
mrtSigMask, 459
Run Time POSIX Function
MRT_RegisterSignal, 460
mrtBeginCriticalSection, 460
mrtEndCriticalSection, 460
mrtPlatformInit, 470

S

selectCycle, 29
selectOneInstance, 605
selectRelated, 596
selectRelatedWhere, 597
selectWhere, 622

signal, 598, 631
startWasher, 29
state, 219
State Model
    ClothesTub, 17
    WashingMachine, 15
statemodel, 216, 230
stateParamDeclarations, 525
static data
    mrtExitEventLoop, 293
    mrtInWhiteState, 295
storageDeclarations, 528
subclass, 630
SUPER_AS_SUBCLASS, 197

T
TILDE_NAME, 191
TOO_FEW_SUBCLASSES, 197
transition, 225
typealias, 198
typeverifier, 281

U
union, 608
UNION_SUBCLASS_EXISTS, 197
UNKNOWN_INIT, 240
update, 580

W
WashingCycle, 11
WashingMachine, 10, 15
WaterLevelSensor, 13
WaterValve, 12