miccautil Package

Analysis Procedures for micca

Andrew Mangogna

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	August 1, 2020	Start of coding.	GAM
1.0	August 9, 2020	Initial release.	GAM
1.1	June 13, 2021	Corrected missing final state transition for constructed graphs. Added "creationevents" method to return a list of creation events.	GAM
1.2	March 10, 2023	Corrected a problem where state machine trace log events were being swallowed by miccautil. Now, the log messages are forwarded to the previous message handler.	GAM
1.3	April 2, 2023	Changes needed to operate with micca version 1.4.0.	GAM

Contents

List of Figures

List of Examples

Chapter 1

Introduction

This document describes the `miccautil` package. `Miccautil` is a pure Tcl package that is intended to accompany the `micca` XUML translation tool.

Background

Micca is a program to aid in the translation of Executable UML domain models into “C” code. During the translation process, information about a platform specific model is input to `micca` via a domain specific language. Executing the domain specific language then populates the platform specific model. The platform specific model itself consists of a relationally normalized schema. The populated platform model is then used to generate the translation code.

Upon request, `micca` will serialize the populated platform model of a domain. The `miccautil` package is used to access the platform specific model population and query common aspects of it. Note that `miccautil` is not *required* to access the domain model. It simply provides a set of more common, and sometimes more complex, queries on the model population. It is quite feasable to deserialize the domain model population (either using the TclRAL native format or SQLite format) and write custom queries against it. The relational schema of the platform model is given in the micca literate program document.

Overview

The `miccautil` package consists of one TclOO command and several ordinary procedures which together form the namespace ensemble command, `::miccautil`

The TclOO command is named, `model`, and represents the platform specific population for a single domain.

Object instances of `model` have methods which support the following types of processing.

- Obtaining basic domain information such as class names.
- Obtaining the state transition matrix of a class state model.
- Recording state transitions for a class to track the state and transition coverage of test cases.
- Reporting information on the inital instance population.
- Exporting the state model of a class as a directed graph.
- Exporting the state model of a class as a graphviz drawing.

The other procedures in `miccautil` perform calculations on the directed graph of a state model such as a depth first search (DFS) or finding a spanning tree.

How to Read This Document

This document is a literate program document. As such it includes a complete description of both the design and implementation of the `miccautil`. Further information on the particular literal programming syntax used here is given in Appendix A.

Readers are not expected to read the document in sequence from beginning to end. Skipping around is encouraged. The document file is hyperlinked with both a Table of Contents and Index to help direct you to a specific topic.

Chapter 2

Preliminaries

We will have need of a number of supporting packages.

```
<<required packages>>=
package require logger
package require logger::utils
package require logger::appender
package require ral
package require ralutil
```

The ral and ralutil packages are essential and hold the relational schema that is the platform specific model. Several procedures in the package return relation values which can be further manipulated using ral procedures.

We also need to configure the logger output.

```
<<logger setup>>=
set logger [::logger::init miccautil]
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 miccautil
```

Chapter 3

The Model Class

The `model` class represents the population of the `micca` platform specific model for a single domain.

```
<<miccautil commands>>=
::oo::class create ::miccautil::model {
    <<model class definition>>
}
```

The class command name is exported and the exported commands of the `miccautil` namespace are used to create an ensemble command of the same name.

```
<<package exports>>=
namespace export model
```

Constructor

`::miccautil model create` *objname savefile*

`::miccautil model new` *savefile*

objname
: The name of the command to be created which represents the model. The `new` version of the constructor creates a name automatically.

savefile
: The name of a file saved from a micca run. This file should be saved in TclRAL native serialization format.

The `create` and `new` methods create an instance of a `model` object. The *savefile* argument is required and is the name of a file produced by a run of `micca` with the `-save` option. The return value of the function is a fully qualified command that may be used with the methods given below.

Implementation

```
<<model class definition>>=
constructor {savefile} {
    ::logger::import -all -namespace log miccautil

    namespace import ::ral::*
    namespace import ::ralutil::*
```

```
    ral deserializeFromFile $savefile [self namespace]

    set domains [pipe {
        relvar set Domain |
        relation project ~ Name |
        relation list ~
    }]

    if {[llength $domains] > 1} {
        error "micca save file must contain only one domain:\
                found, \"[join $domains ,]\""
    }

    my variable domain_name
    set domain_name [lindex $domains 0]

    my variable recording_state
    set recording_state off

    my variable previous_callback
    set previous_callback [list]
}
```

Tests

```
<<constructor tests>>=
test constructor-1.0 {
    Create a model object for the sio domain
} -body {
    miccautil model create sio sio.ral

    relation cardinality [relvar set [info object namespace sio]::Domain]
} -result 1
```

```
<<constructor tests>>=
test constructor-2.0 {
    Create a model object for the aggrmgmt domain
} -body {
    miccautil model create aggrmgmt aggrmgmt.ral

    relation cardinality [relvar set [info object namespace aggrmgmt]::Domain]
} -result 1
```

Destructor

modelobj destroy
The `destroy` method is used to delete *modelobj* when it is no longer needed.

Implementation

```
<<model class definition>>=
destructor {
    relvar constraint delete {*}[relvar constraint names [self namespace]::*]
    relvar unset {*}[relvar names [self namespace]::*]
}
```

Tests

```
<<constructor tests>>=
test destructor-1.0 {
    Create a model object for the sio domain
} -setup {
    miccautil model create sio2 sio.ral
} -body {
    set ns [info object namespace sio2]
    sio2 destroy
    return [namespace exists $ns]
} -result 0
```

Domain name method

modelobj domainName
The `domainName` method returns the name of the domain represented by *modelobj*.

Implementation

```
<<model class definition>>=
method domainName {} {
    my variable domain_name

    return $domain_name
}
```

Tests

```
<<method tests>>=
test domainName-1.0 {
    Get the domain name
} -body {
    sio domainName
} -result sio
```

Class names

modelobj classes
The `classes` method returns a list of the names of the classes defined in the domain represented by *modelobj*.

Implementation

```
<<model class definition>>=
method classes {} {
    return [pipe {
        relvar set Class |
        relation project ~ Name |
        relation list
    }]
}
```

Tests

```
<<method tests>>=
test classes-1.0 {
    Get the list of classes
} -body {
    llength [sio classes]
} -result 24
```

Class attributes

modelobj attributes *class*	
class	The name of a class in the domain represented by *modelobj*.

The `attributes` method returns a dictionary of the attributes of *class*. The keys to the dictionary are the names of the attributes. The values associated to the keys are the data type of the attribute. If *class* does not exist in the domain, an empty dictionary is returned.

Implementation

```
<<model class definition>>=
method attributes {class} {
    return [pipe {
        relvar set Attribute |
        relation restrictwith ~ {$Class eq $class} |
        relation dict ~ Name DataType
    }]
}
```

Tests

```
<<method tests>>=
test attributes-1.0 {
    Get the Point Threshold attributes
} -body {
    set attrs [sio attributes Point_Threshold]
    return [dict size $attrs]
} -result 4
```

```
<<method tests>>=
test attributes-1.1 {
    Get the attributes for non-existent class
} -body {
    set attrs [sio attributes foobar]
    return [dict size $attrs]
} -result 0
```

State names

modelobj states *class*

class The name of a class or assigner in the domain represented by *modelobj*.

The `states` method returns a list of state names for *class*. If *class* does not exist in the domain or if class does not have a state model, an empty list is returned.

A state name of, `@`, indicates the pseudo-initial state out from which creation events transition.

Implementation

```
<<model class definition>>=
method states {class} {
    return [pipe {
        relvar set StatePlace |
        relation restrictwith ~ {$Model eq $class} |
        relation project ~ Name |
        relation list
    }]
}
```

Tests

```
<<method tests>>=
test states-1.0 {
    Get the states for the Sample_Set class
} -body {
    llength [sio states Sample_Set]
} -result 4
```

```
<<method tests>>=
test states-1.1 {
    Get non-existent states
} -body {
    llength [sio states foobar]
} -result 0
```

Event names

modelobj events *class*

class The name of a class or assigner in the domain represented by *modelobj*.

The `events` method returns a list of event names for *class*. If *class* does not exist in the domain or if *class* does not have a state model or any polymorphic events, an empty list is returned. The returned list includes the event names for any type of event that the class may have. For example, a superclass may not have a state model, but could have polymorphic events and these names would be returned.

Implementation

```
<<model class definition>>=
method events {class} {
    return [pipe {
        relvar set Event |
```

```
        relation restrictwith ~ {$Model eq $class} |
        relation project ~ Event |
        relation list
    }]
}
```

Tests

```
<<method tests>>=
test events-1.0 {
    Get the events for the Sample_Set class
} -body {
    llength [sio events Sample_Set]
} -result 4
```

```
<<method tests>>=
test events-1.1 {
    Get events for the Remote_Sensor class
} -body {
    llength [aggrmgmt events Remote_Sensor]
} -result 4
```

Creation event names

modelobj creationevents *class*

class	The name of a class or assigner in the domain represented by *modelobj*.

The `creationevents` method returns a list of event names for *class* which cause a transition out of the pseudo-initial creation state (represented here as the state named, “@”). If *class* does not exist in the domain or if *class* does not have a state model or any creation events, an empty list is returned.

Implementation

```
<<model class definition>>=
method creationevents {class} {
    # log::debug "\n[relformat [relvar set TransitionPlace] TransitionPlace]"
    return [pipe {
        relvar set TransitionPlace |
        relation restrictwith ~ {$Model eq $class && $State eq "@"} |
        relation project ~ Event |
        relation list
    }]
}
```

Tests

```
<<method tests>>=
test creationevents-1.0 {
    Get the events for the Report Record Sensor class
} -body {
    llength [aggrmgmt creationevents Report_Record]
} -result 1
```

State model transitions

modelobj transitions *class*

class The name of a class or assigner in the domain represented by *modelobj*.

The `transitions` method returns a relation value that contains the state transitions for *class*. The heading of the returned relation value is:

Domain string	Model string	State string	Event string	NewState string	Params Relation

where:

Domain is the name of the domain.

Model is the name of the class or assigner.

State is the name of a state. A *State* name of, `@`, indicates the pseudo-initial state out of which creation events transition.

Event is the name of an event which causes a transition out of *State*.

NewState is the name of the state entered by the transition caused when *Event* is received in *State*. A *NewState* name of `IG` indicates the *Event* is ignored when it is received in *State*. A *NewState* name of `CH` indicates it is logically impossible to receive *Event* in *State* (*i.e. can't happen*) and at run time will cause a *panic* condition.

Params is a relation valued attribute giving the parameters of *Event* (and hence the arguments to **NewState**). The cardinality of the *Params* attribute is zero if the event carries no supplemental event data. The *Params* attribute has the heading:

Name string	Position int	DataType string

where:

Name is the name of the parameter.

Position is the order of the parameter carried in *Event*. *Position* values start at zero and sequentially increase for each tuple in *Params*.

DataType is the “C” type name for the parameter.

The cardinality of the returned relation is *states* times *events* where *states* (including the pseudo-initial state for a creation event, if present in the model) is the number of states in the model and *events* is the number of events. The cardinality of the returned relation is zero if the class has no state model.

Each tuple in the returned relation represents a cell in a conceptual *states* by *events* transition matrix with *NewState* as the cell value.

Implementation

```
<<model class definition>>=
method transitions {class_name} {
    set transitions [pipe {
        my GetTransitionCells |
        relation restrictwith ~ {$Model eq $class_name} |
        relation join ~ [relvar set Event] |
        relation eliminate ~ Number |
        rvajoin ~ [relvar set EventSignature] EventSig |
        relation extend ~ p_tup Params {Relation {Name string Position int DataType string ↩
            }} {
            [relation isempty [tuple extract $p_tup EventSig]] ?\
                [relation create {Name string Position int DataType string}] :\
                [relation project\
                    [relation join\
                        [relation semijoin\
                            [relvar restrictone ParameterSignature\
                                Domain [tuple extract $p_tup Domain]\
                                PSigID [relation extract [tuple extract $p_tup EventSig]  ↩
                                    PSigID]]\
                            [relvar set Parameter] -using {Domain Domain PSigID PSigID}]\
                        [relvar set Argument]]\
                    Name Position DataType]} |
        relation eliminate ~ EventSig
    }] ;                                                                  # ❶
    #log::debug \n[relformat $transitions transitions]

    return $transitions
}
```

❶ Ok, this is complicated when it comes to adding the **Params** attribute. We first *rvajoin* to EventSignature to get the PSigID into a relation that is empty if there is no EventSignature and otherwise has a cardinality of one. To get the value of the **Params** attribute we extend the relation value. If there is no event signature, then the **Params** attribute is an empty relation. If there is, then we obtain the values for the **Params** attribute by selecting the correct ParameterSignature instance and joining over the relationships to Arguments and projected out the required data. Unfortunately, this has to be done as an expression that is fed to `expr`, hence the nested required for the several steps involved in obtaining the **Params** value.

Tests

```
<<method tests>>=
test transitions-1.0 {
    Get the transitions for the Sample_Set class
} -body {
    set tm [sio transitions Sample_Set]
    csvToFile [relation eliminate $tm Params] Sample_Set_trans.csv
    return [relation cardinality $tm]
} -result {16}
```

The following table shows the transitions for the Sample_Set class (minus the event parameters, of which there are none for this class).

Domain	**Model**	**State**	**Event**	**NewState**
string	string	string	string	string
sio	Sample_Set	IDLE	Start	RUNNING
sio	Sample_Set	RUNNING	Point_ready	READYING
sio	Sample_Set	READYING	Point_ready	READYING
sio	Sample_Set	READYING	Point_sampled	SAMPLING

Domain	Model	State	Event	NewState
sio	Sample_Set	SAMPLING	Done	IDLE
sio	Sample_Set	SAMPLING	Point_sampled	SAMPLING
sio	Sample_Set	RUNNING	Start	IG
sio	Sample_Set	READYING	Start	IG
sio	Sample_Set	SAMPLING	Start	IG
sio	Sample_Set	IDLE	Point_ready	CH
sio	Sample_Set	IDLE	Point_sampled	CH
sio	Sample_Set	IDLE	Done	CH
sio	Sample_Set	RUNNING	Point_sampled	CH
sio	Sample_Set	RUNNING	Done	CH
sio	Sample_Set	READYING	Done	CH
sio	Sample_Set	SAMPLING	Point_ready	CH

```
<<method tests>>=
test transitions-1.1 {
    Get the transitions for a class with no state model
} -body {
    set tm [sio transitions Point_Threshold]
    relation cardinality $tm
} -result {0}
```

```
<<method tests>>=
test transitions-2.0 {
    Get the transitions for the Report Record class
} -body {
    set tm [aggrmgmt transitions Report_Record]
    relation cardinality $tm
} -result {72}
```

```
<<method tests>>=
test transitions-2.1 {
    Get the transition matrix for the Reporting Session class
} -body {
    set tm [aggrmgmt transitions Reporting_Session]
    relation cardinality $tm
} -result {66}
```

Computing the transition matrix

Computing the transition matrix is the heart of the `transitions` method and several other methods in this package. The following method queries the `micca` platform model to determine the state transitions. The result produced is the transition matrix for the entire domain. Each method that uses this computation then adjusts the relation to suit its needs.

In `micca`, there is a concept of default transitions. This can be either `IG` or `CH` and the default transition is used as the target for a transition which is not otherwise specified explicitly. The strategy for the queries below is to compute all possible transitions. Then the transitions which were specified are subtracted from the possible set. The difference is then the unspecified transitions which are given the default transition target.

```
<<model class definition>>=
method GetTransitionCells {} {
    set state_places [pipe {
        relvar set StatePlace |
        relation eliminate ~ Number |
        relation rename ~ Name State
    }]
    set possible_trans [pipe {
```

```
        relvar set TransitioningEvent |
        relation join $state_places ~
    }]
    #log::debug \n[relformat $possible_trans possible_trans]

    set trans [relvar set StateTransition]
    #log::debug \n[relformat $trans trans]
    set non_trans [pipe {
        relvar set NonStateTransition |
        relation rename ~ TransRule NewState
    }]
    #log::debug \n[relformat $non_trans non_trans]

    set spec_trans [relation union $trans $non_trans]
    #log::debug \n[relformat $spec_trans spec_trans]

    set defrule [pipe {
        relvar set StateModel |
        relation project ~ Domain Model DefaultTrans |
        relation rename ~ DefaultTrans NewState
    }]
    # log::debug \n[relformat $defrule defrule]

    set non_spec_trans [pipe {
        relation project $spec_trans Domain Model State Event |
        relation minus $possible_trans ~ |
        relation join ~ $defrule |
        relation update ~ ftup {[tuple extract $ftup State] eq "@"}\
            {tuple update $ftup NewState CH}
    }]
    # log::debug \n[relformat $non_spec_trans non_spec_trans]

    set trans_records [relation union $spec_trans $non_spec_trans]
    #log::debug \n[relformat $trans_records trans_records]

    return $trans_records
}
```

Recording event dispatch

`Micca` generated domains can be wrapped in an automatically generated test harness by using the `bosal` program. When the resulting test harness is operated using the `mecate` package, event transitions can be captured. The methods in this section facilitate counting transitions and creating summaries of the state and transition coverage.

The following methods create a transition recording session. The semantics are similar to files, *i.e.* you first start the recording, then call a method to record each state transition, and finally you can stop the recording. At any time after starting a recording, you may request the event dispatch information.

Starting a transition recording session

modelobj startTransitionRecording
The `startTransitionRecording` method initializes internal data structures in preparation for recording event transitions in the domain represented by *modelobj*. In particular, any previous event transition counts are reset back to zero. Attempting to start an already running session is silently ignored. The method returns the empty string.

```
<<model class definition>>=
```

```
method startTransitionRecording {} {
    my variable recording_state
    if {$recording_state eq "on"} {
        return
    }

    if {![relvar exists __Event_Record__]} {                          # ❶
        set trans [pipe {
            my GetTransitionCells |
            relation extend ~ sptup TransCount int 0
        }]
        relvar create __Event_Record__ [relation heading $trans]\
            {Domain Model State Event}
        relvar set __Event_Record__ $trans
    } else {
        relvar update __Event_Record__ ertup true {
            tuple update $ertup TransCount 0
        }
    }

    set recording_state on
    return
}
```

❶ We record the event transition information in a relvar which consists of the transition matrix cell values and a new column to hold the counts. First time through, we must create the relvar to hold the counts. Next time through, we can just zero out the counts.

Stopping a transition recording session

modelobj stopTransitionRecording The `stopTransitionRecording` method closes an ongoing event transition recording session. The information gathered during the session is not modified. Attempting to stop an already stopped session is silently ignored. The method returns the empty string.

```
<<model class definition>>=
method stopTransitionRecording {} {
    my variable recording_state
    set recording_state off

    return
}
```

Recording an event transition

modelobj recordTransition *class currstate event*

class The name of a class or assigner in the domain represented by *modelobj*.

currstate The name of the state in *class* which is the source state in a transition.

event The name of an event in *class* which caused a transition from *currstate*.

The `recordTransition` method counts the transition which occurred when *class* was in *currstate* and received *event*. It is necessary to start the event transition recording session by invoking the `startTransitionRecording` method before invoking this method. The method returns a boolean value indicating if the counting occurred, *i.e.* if *currstate* and *event* form a valid transition in *class*.

```
<<model class definition>>=
method recordTransition {class currstate event} {
    my variable recording_state
    if {$recording_state eq "off"} {
        error "event transition recording is stopped"
    }

    my variable domain_name
    set updated [relvar updateone __Event_Record__ ertup\
            [list Domain $domain_name Model $class State $currstate\
            Event $event] {
        tuple update $ertup TransCount\
                [expr {[tuple extract $ertup TransCount] + 1}]
    }]

    return [relation isnotempty $updated]
}
```

Reporting transitions

modelobj reportTransitions *pattern*

pattern A pattern of the format used by the `string match` command for the names of classes or assigners in the domain represented by *modelobj*.

The `reportTransitions` method returns a relation value containing the transition counts for all classes whose names match *pattern*. The heading of the returned relation value is:

Domain string	Model string	State string	Event string	NewState string	TransCount int

where:

Domain is the name of the domain.

Model is the name of the class or assigner.

State is the name of a state. A *State* name of, `@`, indicates the pseudo-initial state out of which creation events transition.

Event is the name of an event which causes a transition out of *State*.

NewState The name of the state entered by the transition caused when *Event* is received in *State*. A *NewState* name of `IG` indicates the *Event* is ignored when it is received in *State*. A *NewState* name of `CH` indicates it is logically impossible to receive *Event* in *State* (*i.e. can't happen*) and at run time will cause a *panic* condition.

TransCount The number of times recorded when **Model** was in a given **State** and **Event** was received.

```
<<model class definition>>=
method reportTransitions {pattern} {
    return [pipe {
        relvar set __Event_Record__ |
        relation restrictwith ~ {[string match $pattern $Model]}
    }]
}
```

Example 3.1 Recording transitions

Here we show a simple transition session and record three transitions

```
<<method tests>>=
test reportTransitions-1.0 {
    report event transitions
} -body {
    sio startTransitionRecording
    sio recordTransition Sample_Set IDLE Start
    sio recordTransition Sample_Set RUNNING Point_ready
    sio recordTransition Sample_Set RUNNING Point_ready
    sio stopTransitionRecording

    set trans [sio reportTransitions Sample_Set]
    ral::csvToFile $trans Sample_Set.csv
    return [relation cardinality $trans]
} -result {16}
```

The following table shows the output of the test case.

Domain	Model	State	Event	NewState	TransCount
string	string	string	string	string	int
sio	Sample_Set	IDLE	Start	RUNNING	1
sio	Sample_Set	RUNNING	Point_ready	READYING	2
sio	Sample_Set	READYING	Point_ready	READYING	0
sio	Sample_Set	READYING	Point_sampled	SAMPLING	0
sio	Sample_Set	SAMPLING	Done	IDLE	0
sio	Sample_Set	SAMPLING	Point_sampled	SAMPLING	0
sio	Sample_Set	RUNNING	Start	IG	0
sio	Sample_Set	READYING	Start	IG	0
sio	Sample_Set	SAMPLING	Start	IG	0
sio	Sample_Set	IDLE	Point_ready	CH	0
sio	Sample_Set	IDLE	Point_sampled	CH	0
sio	Sample_Set	IDLE	Done	CH	0
sio	Sample_Set	RUNNING	Point_sampled	CH	0
sio	Sample_Set	RUNNING	Done	CH	0
sio	Sample_Set	READYING	Done	CH	0
sio	Sample_Set	SAMPLING	Point_ready	CH	0

Examining the **TransCount** column show the three recorded transitions.

Example 3.2 Transitions not ignored or impossible

The relation value returned from `reportTransitions` can be further processed to yield more refined results. For example, if we are only interested in transitions which result in a state change, *i.e.* are **not** `IG` or `CH`, we can restrict the output to exclude tuples where **NewState** is `IG` or `CH`.

```
<<method tests>>=
test reportTransitions-1.1 {
    report event dispatch which are not IG or CH
} -body {
    set trans [pipe {
        sio reportTransitions Sample_Set |
        relation restrictwith ~ {$NewState ne "IG" && $NewState ne "CH"}
    }]
    ral::csvToFile $trans Sample_Set_noigch.csv
    return [relation cardinality $trans]
} -result {6}
```

The following table shows the reduced output.

Domain	Model	State	Event	NewState	TransCount
string	string	string	string	string	int
sio	Sample_Set	IDLE	Start	RUNNING	1
sio	Sample_Set	RUNNING	Point_ready	READYING	2
sio	Sample_Set	READYING	Point_ready	READYING	0
sio	Sample_Set	READYING	Point_sampled	SAMPLING	0
sio	Sample_Set	SAMPLING	Done	IDLE	0
sio	Sample_Set	SAMPLING	Point_sampled	SAMPLING	0

Example 3.3 Transitions not taken

We can further refine the transition information to yield those transitions which were not taken. This information can be used to evaluate the effect of test scenarios in covering the execution of state activities.

```
<<method tests>>=
test reportTransitions-1.2 {
    report event dispatch which are not IG or CH
} -body {
```

```
    set trans [pipe {
        sio reportTransitions Sample_Set |
        relation restrictwith ~\
            {$NewState ne "IG" && $NewState ne "CH" && $TransCount == 0}
    }]
    ral::csvToFile $trans Sample_Set_missed.csv
    return [relation cardinality $trans]
} -result {4}
```

The following table shows only those transitions in the state model which were never taken.

Domain	Model	State	Event	NewState	TransCount
string	string	string	string	string	int
sio	Sample_Set	READYING	Point_ready	READYING	0
sio	Sample_Set	READYING	Point_sampled	SAMPLING	0
sio	Sample_Set	SAMPLING	Done	IDLE	0
sio	Sample_Set	SAMPLING	Point_sampled	SAMPLING	0

Example 3.4 Executed state activities

Since the state machines produced by `micca` are Moore type machines, each time the **TransCount** of a transition is non-zero, we know the activity for the **NewState** was executed. Additional processing shows how to compute the number of times a given state activity is executed.

```
<<method tests>>=
test reportTransitions-2.0 {
    report state activity execution
} -body {
    set trans [pipe {
        sio reportTransitions Sample_Set |
        relation restrictwith ~ {$NewState ne "IG" && $NewState ne "CH"} |
        relation summarizeby ~ {Domain Model NewState} sa_rel\
                Executed int {rsum($sa_rel, "TransCount")} |
        relation rename ~ NewState State
    }]
    ral::csvToFile $trans Sample_Set_act.csv
    return [relation cardinality $trans]
} -result {4}
```

The following table shows the number times each state activity was executed.

Domain	Model	State	Executed
string	string	string	int
sio	Sample_Set	RUNNING	1
sio	Sample_Set	READYING	2
sio	Sample_Set	SAMPLING	0
sio	Sample_Set	IDLE	0

Usually in a testing scenario, we are most interested in those state activities that are *not* executed by the test suite, indicating a potential lack of coverage. Restricting the above relation to those tuples where **Executed** is zero, gives that result.

Counting transitions with `mecate`

When `bosal` generated test harnesses are operated using the `mecate` package, `mecate` has the capability of invoking a command each time an event trace arrives from the test harness. The following methods serve as *glue* code between the `mecate` interface and the `miccautil` transition recording.

modelobj startMecateTransitionCount *reinobj*

reinobj
: An object command as returned from the `rein` command of the `mecate` package. A *reinobj* represents a `bosal` generated test harness and methods of the object allow for operations on the test harness.

The `startMecateTransitionCount` method starts capturing event transitions as they arrive from a `bosal` generated test harness. This method uses the `traceNotify` method of the *reinobj* to install a callback handler for when event traces arrive.

Note this method does **not** turn on event tracing in the test harness. That is done with the *reinobj* `trace on` command which must be executed before any events will be received and counted.

Implementation

```
<<model class definition>>=
method startMecateTransitionCount {reinobj} {
    my startTransitionRecording

    my variable previous_callback
    set previous_callback [$reinobj traceNotify]
    $reinobj traceNotify [mymethod RecordMecateTransition]
}
```

modelobj stopMecateTransitionCount *reinobj*

reinobj
: An object command as returned from the `rein` command of the `mecate` package. A *reinobj* represents a `bosal` generated test harness and methods of the object allow for operations on the test harness.

The `stopMecateTransitionCount` method stops capturing event transitions as they arrive from a `bosal` generated test harness. The previous *reinobj* callback handler is re-instated.

Implementation

```
<<model class definition>>=

method stopMecateTransitionCount {reinobj} {
    my variable previous_callback
    $reinobj traceNotify $previous_callback

    my stopTransitionRecording
}
```

modelobj RecordMecateTransition *trace*

trace
: A dictionary of the form generated by a `bosal` test harness containing an event dispatch trace. See the `mecate` man pages for a detailed description of a *trace* dictionary contents.

The `RecordMecateTransition` method examines the information in *trace* and uses it to count transition of the state machines in *modelobj*.

Implementation

```
<<model class definition>>=
method RecordMecateTransition {trace} {
    my variable previous_callback
    if {[llength $previous_callback] != 0} {
        eval [linsert $previous_callback end $trace]
    }

    if {[dict get $trace type] eq "transition"} {
        set target_class [lindex [split [dict get $trace target] .] 0] ; # ❶
        my recordTransition $target_class\
            [dict get $trace currstate] [dict get $trace event]
    }

    return
}
export RecordMecateTransition
```

❶ In the trace data, the *target* of the event is given in the form: <*class*>.<*instance*>. Here we only want the class name part.

Tests

```
<<method tests>>=
test RecordMecateTransition-1.0 {
    record event dispatch using mecate trace data
} -cleanup {
    sio stopTransitionRecording
} -body {
    sio startTransitionRecording

    set trace_info [dict create\
        type transition\
        target Sample_Set.0\
        currstate RUNNING\
        event Point_ready\
    ]
    sio RecordMecateTransition $trace_info

    dict set trace_info currstate RUNNING
    dict set trace_info event Point_ready
    sio RecordMecateTransition $trace_info
    sio RecordMecateTransition $trace_info

    set report [sio reportTransitions Sample_Set]
    return [pipe {
        sio reportTransitions Sample_Set |
        relation summarize ~ $::ralutil::DEE rpt_rel\
                TotalCount int {rsum($rpt_rel, "TransCount")} |
        relation extract ~ TotalCount
    }]
} -result {3}
```

Default attribute values

modelobj defaultAttributeValues

The `defaultAttributeValues` returns a relation value giving the default values that attributes in the domain represented by *modelobj* are given if not otherwise specified.

The heading of the returned relation is:

Domain string	Class string	Defaults Relation

where:

Domain is the name of the domain.

Class is the name of a class in **Domain**.

Defaults is a relation valued attribute containing the attribute names and values for **Class**.

The heading of the **Defaults** attribute is:

Attribute string	Value string	DataType string

where:

Attribute is the name of the attribute of the instance.

Value is the value of the attribute in the instance.

DataType is the “C” type name for the attribute.

Implementation

```
<<model class definition>>=
method defaultAttributeValues {} {
    return [pipe {
        relvar set DefaultValue |
        relation join ~ [relvar set Attribute]\
                -using {Domain Domain Class Class Attribute Name} |
        relation group ~ Defaults Attribute Value DataType
    }]
}
```

Tests

```
<<method tests>>=
test defaultAttributeValues-1.0 {
    list default attributes
} -body {
    set def_attr [sio defaultAttributeValues]
    log::debug \n[relformat $def_attr]

    return [relation cardinality $def_attr]
} -result {11}
```

Initial instance population

modelobj initialInstancePopulation

The `initialInstancePopulation` method returns a relation value containing the initial instance population of the domain represented by *modelobj*. The heading of the returned relation is:

Domain string	Class string	Instances Relation

where:

Domain is the name of the domain.

Class is the name of a class in **Domain**.

Instances is a relation valued attribute containing the initial instances of **Class**.

The heading of the **Instances** attribute is:

Instance string	ID int	Attributes Relation

where:

Instance is the name given to the inital instance in the `micca` population.

ID is the numeric identifier of the instance. This number is the same as the array index of the instance in the storage pool for the class.

Attributes is a relation valued attribute giving the attribute names and values of the initial instance.

The heading of the **Attributes** attribute is:

Attribute string	Value string

where:

Attribute is the name of the attribute of the instance.

Value is the value of the attribute in the instance.

Example 3.5 Example initial instance population tuple

An example tuple (*i.e.* one row) of the initial instance population relation might appear in tabular form as:

<table>
<tr><th>Domain</th><th>Class</th><th colspan="4">Instances</th></tr>
<tr><th>string</th><th>string</th><th colspan="4">Relation</th></tr>
<tr><td rowspan="12">sio</td><td rowspan="12">Signaled_Point</td><th>Instance</th><th>ID</th><th colspan="2">Attributes</th></tr>
<tr><th>string</th><th>int</th><th colspan="2">Relation</th></tr>
<tr><td rowspan="2"></td><td rowspan="2"></td><th>Attribute</th><th>Value</th></tr>
<tr><th>string</th><th>string</th></tr>
<tr><td rowspan="4">sigpt1</td><td rowspan="4">6</td><td>Trigger</td><td>ed_Active</td></tr>
<tr><td>Active_high</td><td>true</td></tr>
<tr><td>Settle_interval</td><td>100</td></tr>
<tr><td>R3</td><td>sigpt1</td></tr>
<tr><td rowspan="4">sigpt2</td><td rowspan="4">7</td><td>Trigger</td><td>ed_BothActive</td></tr>
<tr><td>Active_high</td><td>false</td></tr>
<tr><td>Settle_interval</td><td>100</td></tr>
<tr><td>R3</td><td>sigpt2</td></tr>
</table>

Implementation

```
<<model class definition>>=
method initialInstancePopulation {} {
    return [pipe {
        relvar set PopulatedComponent |
        relation semijoin ~\
            [relvar set ClassComponent]\
            [relvar set ClassComponentValue]\
                -using {Domain Domain Class Class Name Component}\
            [relvar set SpecifiedComponentValue] |
        relation join ~ [relvar set ClassInstance] |
        relation rename ~ Component Attribute Number ID |
        relation group ~ Attributes Attribute Value |
        relation group ~ Instances Instance ID Attributes
    }]
}
```

Tests

```
<<method tests>>=
test initialInstancePopulation-1.0 {
    list initial instance values
} -body {
    set init_inst [sio initialInstancePopulation]

    log::debug \n[relformat $init_inst]

    return [relation cardinality $init_inst]
} -result {22}
```

State model as a graph

modelobj stateModelGraph *class*

class The name of a class or assigner in the domain represented by *modelobj*.

The `stateModelGraph` method returns a *graph* command from the `struct::graph` package in Tcllib that represents the state model for *class* as a graph. It is the responsibility of the caller to insure that the returned graph command is disposed of properly by invoking *graph* `destroy` when no longer needed. If *class* does not not have a state model, the returned *graph* has no nodes or arcs.

The returned graph is annotated by the following **key** / value attributes:

domain the name of the domain.

class the name of the class or assigner.

initialstate the name of the default initial state.

defaulttrans the name of the default transition, *i.e.* `IG` or `CH`.

Nodes in the graph represent states in the state model and are named the same as the state name. Nodes are annotated by the following **key** / value attributes:

activity the state activity code.

final a boolean value indicating if the state is a final state.

Arcs in the graph represent the directed transitions from a source state to a target state. Note that `IG` and `CH` transitions are *not* represented by arcs since as target states they do not cause an actual transition. Arcs are annotated by the following **key** / value attributes:

event the name of the event causing the transition.

params a list of event parameter names giving the additional values carried by the event.

We need the `struct::graph` package from Tcllib and we want to make sure that it is at least version 2 or higher.

```
<<required packages>>=
package require struct::graph 2
```

Implementation

```
<<model class definition>>=
method stateModelGraph {class_name} {
    my variable domain_name
    set gr [::struct::graph]

    try {
        $gr set domain $domain_name
        $gr set class $class_name

        set smodel [relvar restrictone StateModel\
                Domain $domain_name Model $class_name]
        if {[relation isempty $smodel]} {
            $gr set initialstate {}
```

```
            $gr set defaulttrans {}
            return $gr
        }

        $gr set defaulttrans [relation extract $smodel DefaultTrans]

        set cr_state [pipe {
            relvar set CreationState |
            relation semijoin $smodel ~ |
            relation extend ~ cstup\
                    Activity string {{}}\
                    IsFinal boolean {false} |
            relation project ~ Name Activity IsFinal
        }]

        $gr set initialstate [expr {[relation isnotempty $cr_state] ?\
                "@" : [relation extract $smodel InitialState]}]

        set states [pipe {
            relvar set State |
            relation semijoin $smodel ~ |
            relation project ~ Name Activity IsFinal |
            relation union ~ $cr_state
        }]
        # log::debug \n[relformat $states states]

        relation foreach state $states {
            relation assign $state Name Activity IsFinal
            $gr node insert $Name
            $gr node set $Name activity $Activity
            $gr node set $Name final $IsFinal
        }

        set finals [relation restrictwith $states {$IsFinal}]
        if {[relation isnotempty $finals]} {
            $gr node insert __x__
            $gr node set __x__ activity {}
            $gr node set __x__ final false
        }
        relation foreach final_state $finals {
            relation assign $final_state Name
            $gr arc insert $Name __x__
        }

        set trans [pipe {
            my transitions $class_name |
            relation restrictwith ~ {$NewState ne "IG" && $NewState ne "CH"} |
            relation eliminate ~ Domain Model
        }]

        relation foreach tran $trans {
            relation assign $tran State Event NewState Params
            set arc [$gr arc insert $State $NewState]
            $gr arc set $arc event $Event
            $gr arc set $arc params\
                    [relation list $Params Name -ascending Position]
        }

        return $gr
    } on error {result opts} {
        $gr destroy
        return -options $opts $result
```

```
    }
}
```

Tests

```
<<method tests>>=
test stateModelGraph-1.0 {
    Get the state model graph for the Sample_Set class
} -body {
    set sample_set_graph [sio stateModelGraph Sample_Set]
    set nnodes [llength [$sample_set_graph nodes]]
    $sample_set_graph destroy
    return $nnodes
} -result {4}
```

```
<<method tests>>=
test stateModelGraph-1.1 {
    State model graph for class with not state model
} -body {
    set pt_graph [sio stateModelGraph Point_Threshold]
    set nnodes [llength [$pt_graph nodes]]
    $pt_graph destroy
    return $nnodes
} -result {0}
```

```
<<method tests>>=
test stateModelGraph-2.0 {
    Get the state model graph for the Report Record class
} -body {
    set rrec_graph [aggrmgmt stateModelGraph Report_Record]
    set nnodes [llength [$rrec_graph nodes]]
    $rrec_graph destroy
    return $nnodes
} -result {9}
```

```
<<method tests>>=
test stateModelGraph-2.1 {
    Check the state model graph for the Report Record class
} -body {
    set rrec_graph [aggrmgmt stateModelGraph Report_Record]
    set create_trans [$rrec_graph arcs -out @]
    set cr_event [$rrec_graph arc get $create_trans event]
    $rrec_graph destroy
    return $cr_event
} -result {Report}
```

State model with Tcldot

modelobj stateModelDot *class*

class	The name of a class or assigner in the domain represented by *modelobj*.

The `stateModelDot` method returns a `Tcldot` command handle to the state model of *class*. The command handle can be used to render an image of the state model graph (along with many other uses).

Implementation

```
<<model class definition>>=
method stateModelDot {class_name} {
    package require Tcldot ;                                        # ❶

    my variable domain_name

    set dot [dotnew digraph]
    try {
        set smodel [relvar restrictone StateModel\
                Domain $domain_name Model $class_name]
        if {[relation isempty $smodel]} {
            return $dot
        }

        $dot setnodeattributes shape box
        $dot setnodeattributes style filled
        $dot setnodeattributes fillcolor yellow

        set cr_state [pipe {
            relvar set CreationState |
            relation semijoin $smodel ~ |
            relation extend ~ cstup IsFinal boolean {false} |
            relation project ~ Name IsFinal
        }]
        set states [pipe {
            relvar set State |
            relation semijoin $smodel ~ |
            relation project ~ Name IsFinal |
            relation union ~ $cr_state
        }]
        #log::debug \n[relformat $states states]

        set node(@) [$dot addnode @ {*}{
            shape point
            fillcolor black
            label {}
            width 0.15
            fixedsize true
        }]

        set finals [relation restrictwith $states {$IsFinal}]
        if {[relation isnotempty $finals]} {
            set node(__x__) [$dot addnode __x__ {*}{
                shape doublecircle
                fillcolor black
                label {}
                width 0.15
                fixedsize true
            }]
        }

        relation foreach state $states {
            relation assign $state
            if {$Name eq "@"} {
                continue
            }
            set node($Name) [$dot addnode $Name\
                label [string map {_ { }} $Name]\
            ]
            if {$IsFinal} {
```

```
                set edge($Name,__x__) [$dot addedge $node($Name) $node(__x__)]
            }
        }

        if {[relation isempty $cr_state]} {
            set initialstate [relation extract $smodel InitialState]
            set edge(@,$initialstate)\
                    [$dot addedge $node(@) $node($initialstate)]
        } ;                                                                 # ❷

        set statetrans [pipe {
            relvar set StateTransition |
            relation semijoin $smodel ~ |
            relation rename ~ State SrcState NewState State |
            rvajoin ~ [relvar set StateSignature] Signature |
            relation project ~ SrcState Event State Signature
        }]
        #puts [relformat $statetrans statetrans]

        relation foreach statetran $statetrans {
            relation assign $statetran
            set evt_label [string map {_ { }} $Event]
            if {[relation isnotempty $Signature]} {
                set params [pipe {
                    relvar restrictone ParameterSignature Domain $domain_name\
                        PSigID [relation extract $Signature PSigID] |
                    relation semijoin ~ [relvar set Parameter] |
                    relation list ~ Name -ascending Position |
                    join ~ ,
                }]
                append evt_label "(" $params ")"
            }
            set edge($SrcState,$State) [$dot addedge\
                    $node($SrcState) $node($State)\
                    label $evt_label
            ]
        }

        return $dot
    } on error {result opts} {
        rename $dot {}
        return -options $opts $result
    }
}
```

1. Since `Tcldot` is not a common package, we do the `package require` here to minimize the dependency upon `Tcldot`. Other commands and methods can be used without having to have `Tcldot` installed.

2. If there is no creation state, we connect the pseudo-initial state to the default initial state with no event label. This is a convenient indication of the default initial state.

Tests

```
<<method tests>>=
test stateModelDot-1.0 {
    Get the dot graph for the Sample_Set class
} -cleanup {
    chan close $ss_file
    chan close $gv_file
    rename $sample_set_dot {}
} -body {
```

```
    set sample_set_dot [sio stateModelDot Sample_Set]

    set ss_file [open Sample_Set.pdf w]
    $sample_set_dot write $ss_file pdf

    set gv_file [open Sample_Set.gv w]
    $sample_set_dot write $gv_file dot

    $sample_set_dot countnodes
} -result {5}
```

The following figure is the rendered state model for the Sample_Set class.

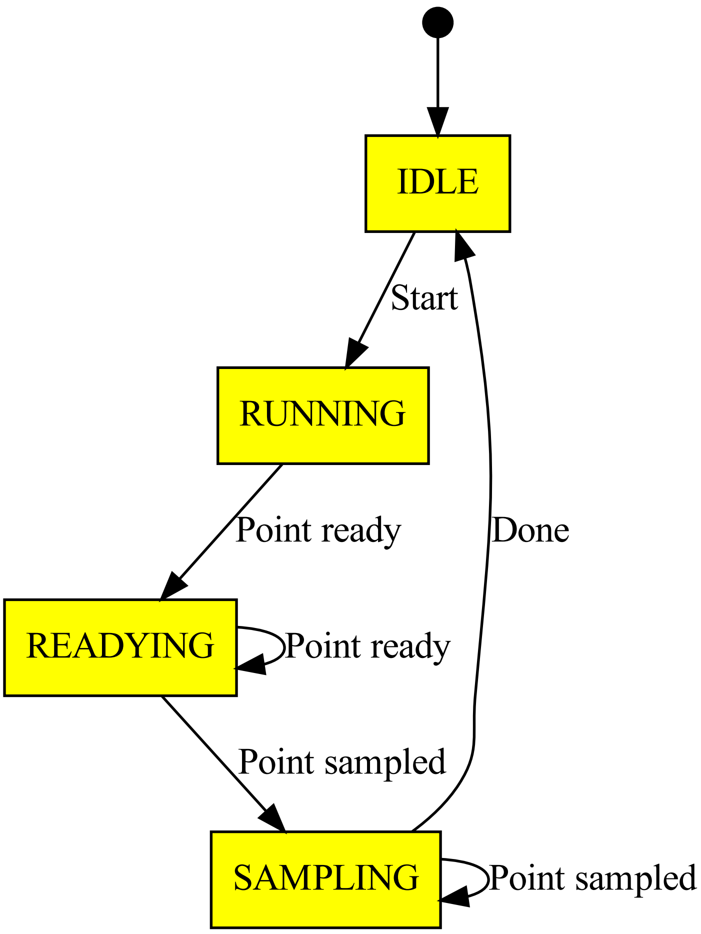

Figure 3.1: Sample Set state model rendered by dot

```
<<method tests>>=
test stateModelDot-2.0 {
    Get the dot graph for the Report Record class
} -cleanup {
    chan close $rrec_file
```

```
    chan close $gv_file
    rename $rrec_dot {}
} -body {
    set rrec_dot [aggrmgmt stateModelDot Report_Record]

    set rrec_file [open Report_Record.pdf w]
    $rrec_dot write $rrec_file pdf

    set gv_file [open Report_Record.gv w]
    $rrec_dot write $gv_file dot

    $rrec_dot countnodes
} -result {9}
```

The following figure is the rendered state model for the Report Record class.

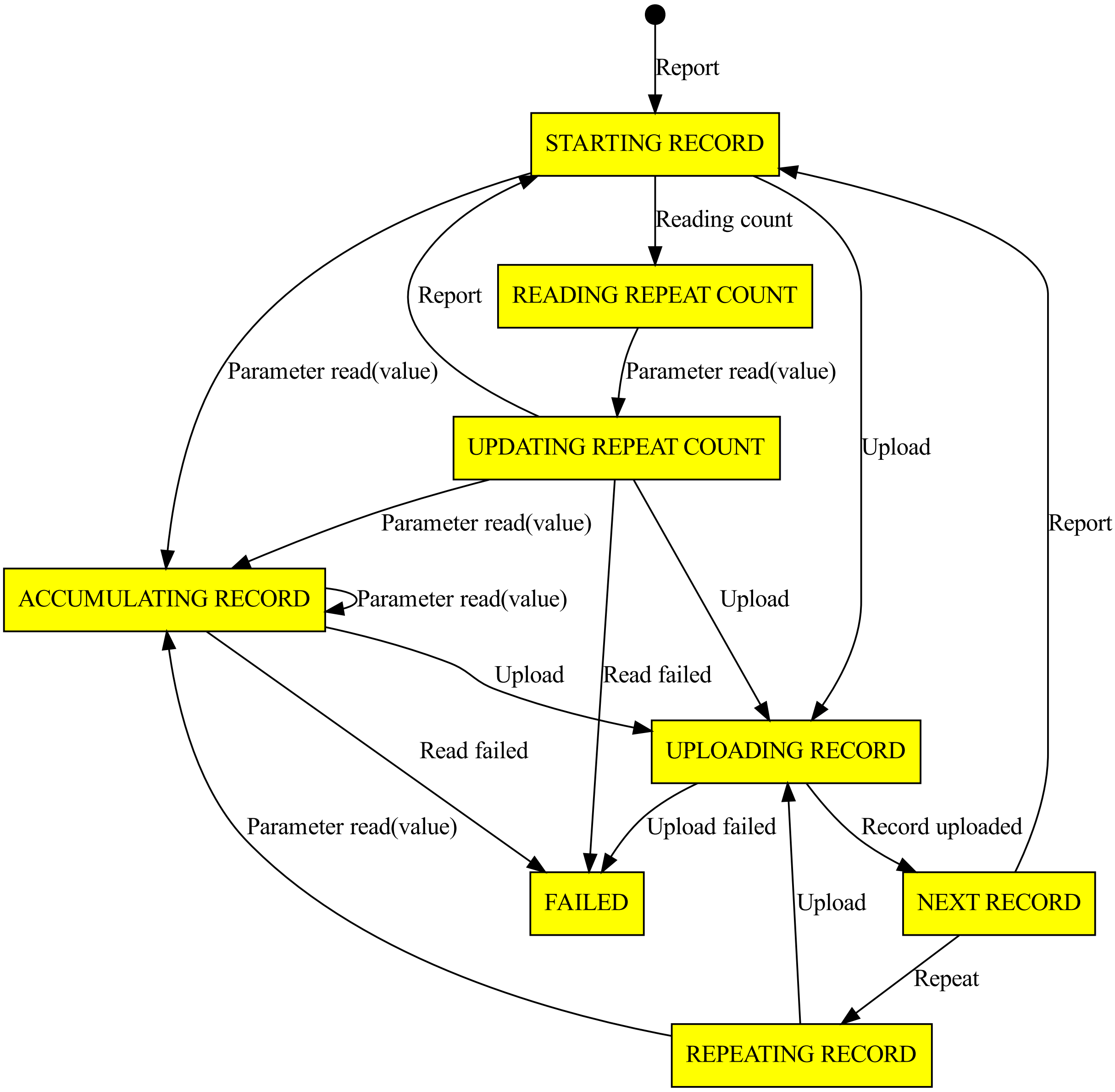

Figure 3.2: Report Record state model rendered by dot

As a comparison, the following figure shows the original layout of the Report Record state model drawn manually during the analysis effort. The Umlet program was used to draw the state model.

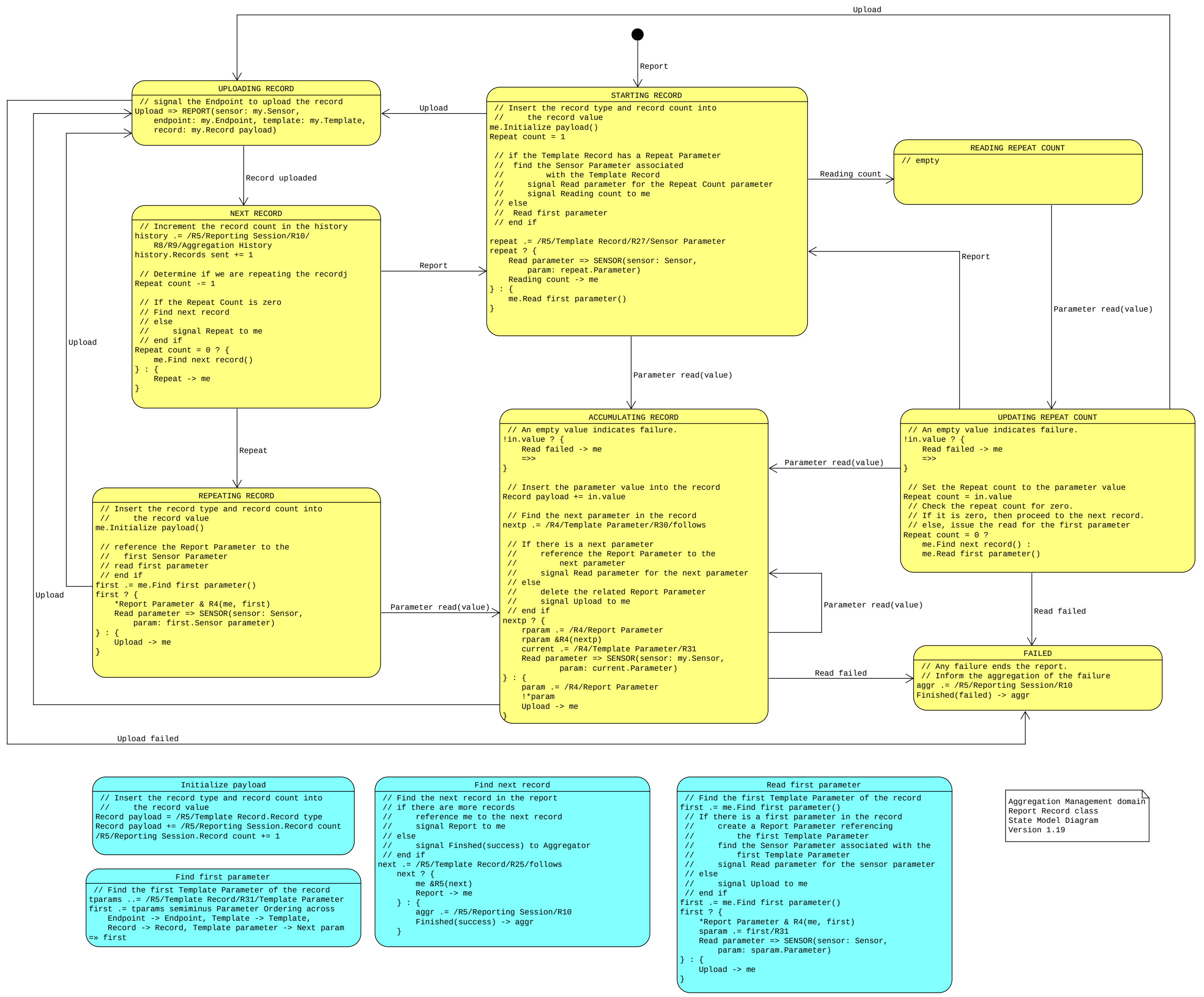

Figure 3.3: Original Report Record state model from Umlet

```
<<method tests>>=
test stateModelDot-2.1 {
    Get the dot graph for the Remote Sensor class
} -cleanup {
    chan close $rs_file
    chan close $gv_file
    rename $rs_dot {}
} -body {
    set rs_dot [aggrmgmt stateModelDot Remote_Sensor]

    set rs_file [open Remote_Sensor.pdf w]
    $rs_dot write $rs_file pdf

    set gv_file [open Remote_Sensor.gv w]
    $rs_dot write $gv_file dot

    $rs_dot countnodes
} -result {8}
```

The following figure is the rendered state model for the Discovered_Sensor class.

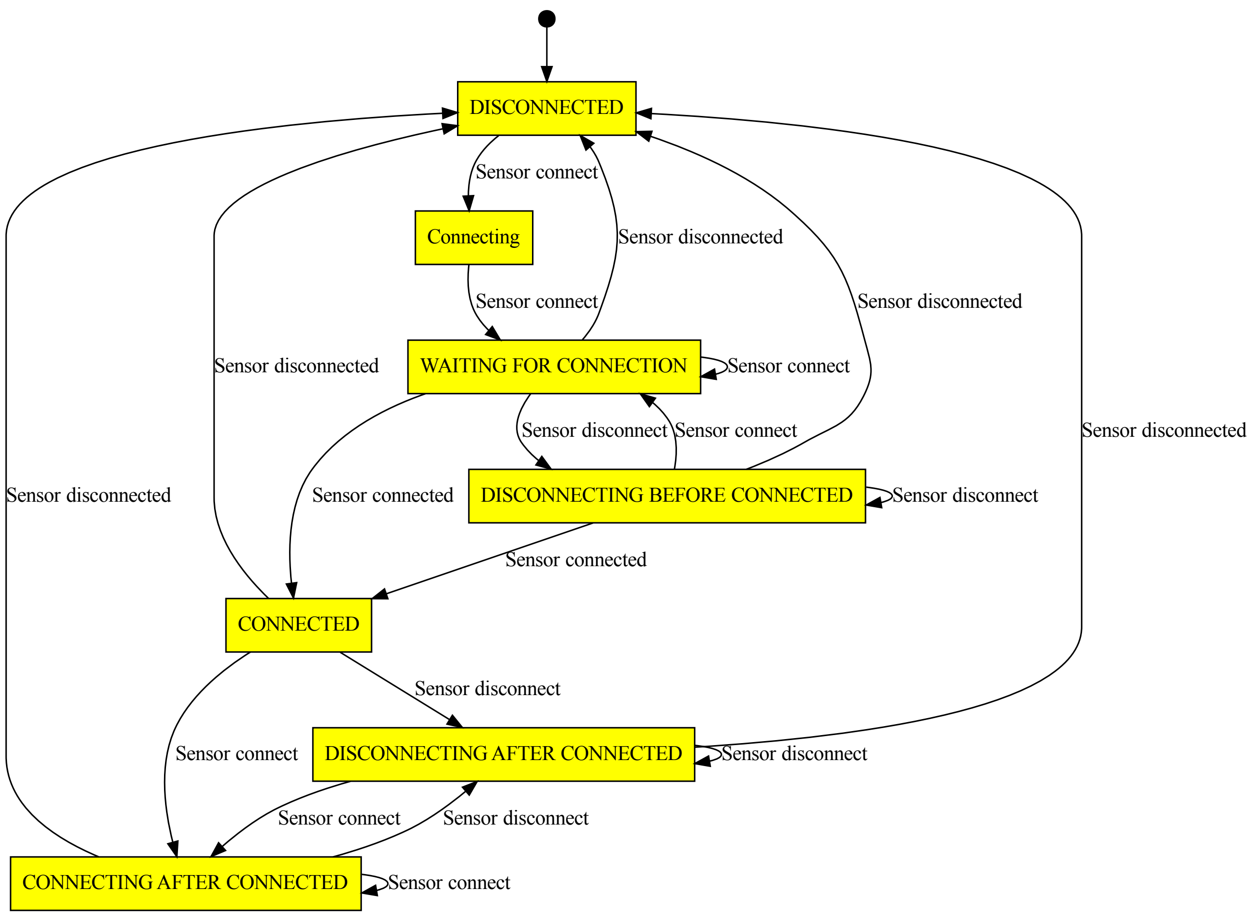

Figure 3.4: Remote Sensor state model rendered by dot

Chapter 4

Functions on state model graphs

In this section, we show `miccautil` ensemble subcommands which operate on graphs. These are convenience commands and are provided as procedures which take a `struct::graph` command handle. Invoking the `stateModelGraph` method yields a suitable graph command.

Depth first search of a state model graph

::miccautil dfs *graph* ?*start*?

graph
: a graph command as returned from `struct::graph`, usually obtained by invoking, *modelobj* `stateModelGraph` *class*.

start
: the name of node where the search is to start. If *start* is not given, then the search starts at node given by the **initialstate** attribute of the graph.

The `dfs` subcommand performs a depth first search (DFS) of *graph*. *graph* is a command as returned from `struct::graph`, usually obtained by invoking the `stateModelGraph` method with the desired class name. The return value of the command is the empty string.

During the DFS, each node in the graph is annotated with the following additional attributes:

pre
: the pre-order number of the node, starting at 1. This is the order in which the node was visited during the DFS.

rpost
: the reverse post-order number of the node, starting at 1. This is the order the node would be visited in a reverse post-order traversal. For graphs that do *not* contain cycles, the **rpost** numbers form a topological sort of the graph. Graphs which have no **back** edges (see following) have no cycles.

Each edge in the graph is annotated with one additional attribute:

type
: the classification of the graph edge. The **type** attribute has one of the following values:

  tree
  : the edge is part of a spanning tree for the graph, *i.e.* the target node is visited for the first time when the edge is traversed.

  forw
  : the edge is a forward directed, *i.e.* the target node is a decendent of the source node.

  back
  : the edge is a back edge, *i.e.* the target node is an ancestor of the source node.

  cross
  : the edge is a cross edge. All edges which are not classified as **tree**, **forw,** or **back** are classified as cross edges.

```
<<package exports>>=
namespace export dfs
```

Implementation

```
<<miccautil commands>>=
proc ::miccautil::dfs {graph {start {}}} {
    if {$start eq {}} {
        set start [$graph get initialstate]
    }

    set nodes [$graph nodes]
    foreach node $nodes {
```

```
        $graph node set $node pre 0
        $graph node set $node rpost 0
    }
    variable preorder 1
    variable postorder [llength $nodes]
    ClassifyNode $graph $start
    return
}
```

The classification algorithm is the convention recursive algorithm. The classification of the graph arcs is accomplished by examining the pre and post order numbering to determine when the node under consideration has been seen.

```
<<miccautil commands>>=
proc ::miccautil::ClassifyNode {graph node} {
    variable preorder
    set thisPre $preorder
    $graph node set $node pre $thisPre
    incr preorder
    set arcList [$graph arcs -out $node]

    foreach arc $arcList {
        set succ [$graph arc target $arc]
        set succPre [$graph node get $succ pre]
        if {$succPre == 0} {
            $graph arc set $arc type tree
            ClassifyNode $graph $succ
        } elseif {[$graph node get $succ rpost] == 0} {
            $graph arc set $arc type back
        } elseif {$thisPre < $succPre} {
            $graph arc set $arc type frwd
        } else {
            $graph arc set $arc type cross
        }
    }
    variable postorder
    $graph node set $node rpost $postorder
    incr postorder -1

    return
}
```

The following figure shows the `dfs` annotations applied to the Reporting Session state model.

Reporting_Session_dfs.pdf

Figure 4.1: DFS annotations for Reporting Session state model

Tests

```
<<method tests>>=
test dfs-1.0 {
    DFS on the state model graph for the Reporting Session class
} -cleanup {
    $session_graph destroy
} -body {
    set session_graph [aggrmgmt stateModelGraph Reporting_Session]
    miccautil dfs $session_graph

    set walkproc [lambda {action graph node} {
        foreach outarc [$graph arcs -out $node] {
```

```
            set target [$graph arc target $outarc]
            # log::debug "$node - [$graph arc get $outarc event] -> $target\
                ==> [$graph arc get $outarc type]"
        }
    }]
    $session_graph walk [$session_graph get initialstate]\
        -order pre -type bfs -dir forward -command $walkproc
} -result {}
```

```
<<method tests>>=
test dfs-2.0 {
    DFS on the state model graph for the Sample_Set class
} -cleanup {
    chan close $ss_file
    rename $ss_dot {}
} -body {
    set ss_graph [sio stateModelGraph Sample_Set]
    miccautil dfs $ss_graph

    set ss_dot [miccautil graphToDot $ss_graph type {pre rpost}]
    set ss_file [open Sample_Set_dfs.pdf w]
    $ss_dot write $ss_file pdf
} -result {}
```

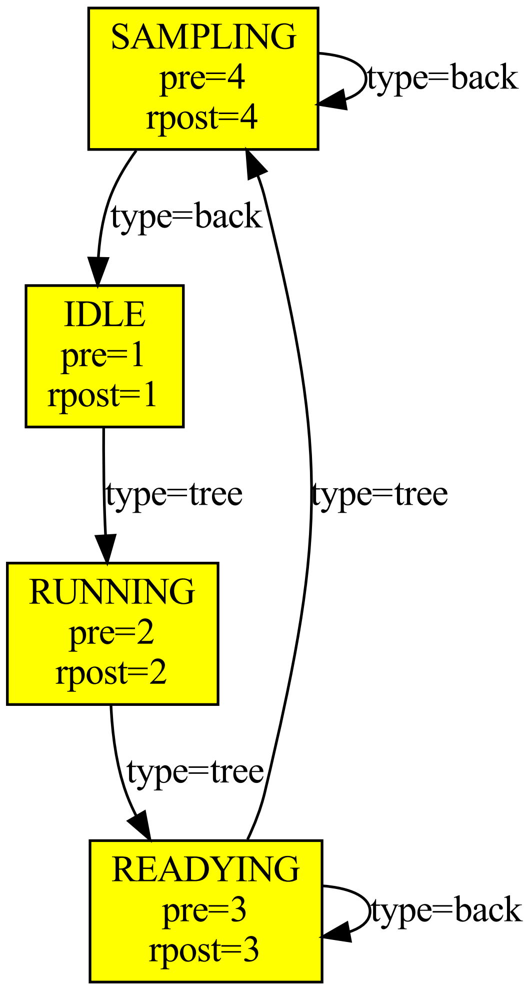

Figure 4.2: DFS annotations for Sample Set state model

Spanning tree of a state model

::miccautil spanningTree *graph start*

graph	a graph command as returned from `struct::graph`, usually obtained by invoking, *modelobj* stateModelGraph *class*.
start	the name of node where the DFS for the tree is to start. If *start* is not given, then the tree starts at node given by the **initialstate** attribute of the graph.

The `spanningTree` subcommand returns a `graph` command handle as obtained from the `struct::graph` package in Tcllib. The returned graph contains a spanning tree of the *graph* argument. The spanning tree returned is the sub-graph of `graph` where only **tree** type edges are retained. The caller is responsible for invoking the `destroy` method on the returned graph command when it is no longer needed. It is not necessary to have run the `dfs` command previously on *graph* as that will be done by `spanningTree`.

```
<<package exports>>=
namespace export spanningTree
```

```
<<required packages>>=
package require lambda
```

Implementation

```
<<miccautil commands>>=
proc ::miccautil::spanningTree {graph {start {}}} {
    set span [::struct::graph]
    $span = $graph
    dfs $span $start

    set ffunc [lambda {graph arc} {
        expr {[$graph arc get $arc type] ne "tree"}
    }]
    set non_tree [$span arcs -key type -filter $ffunc]
    $span arc delete {*}$non_tree

    return $span
}
```

The following figure shows the spanning tree of the Conduit state model.

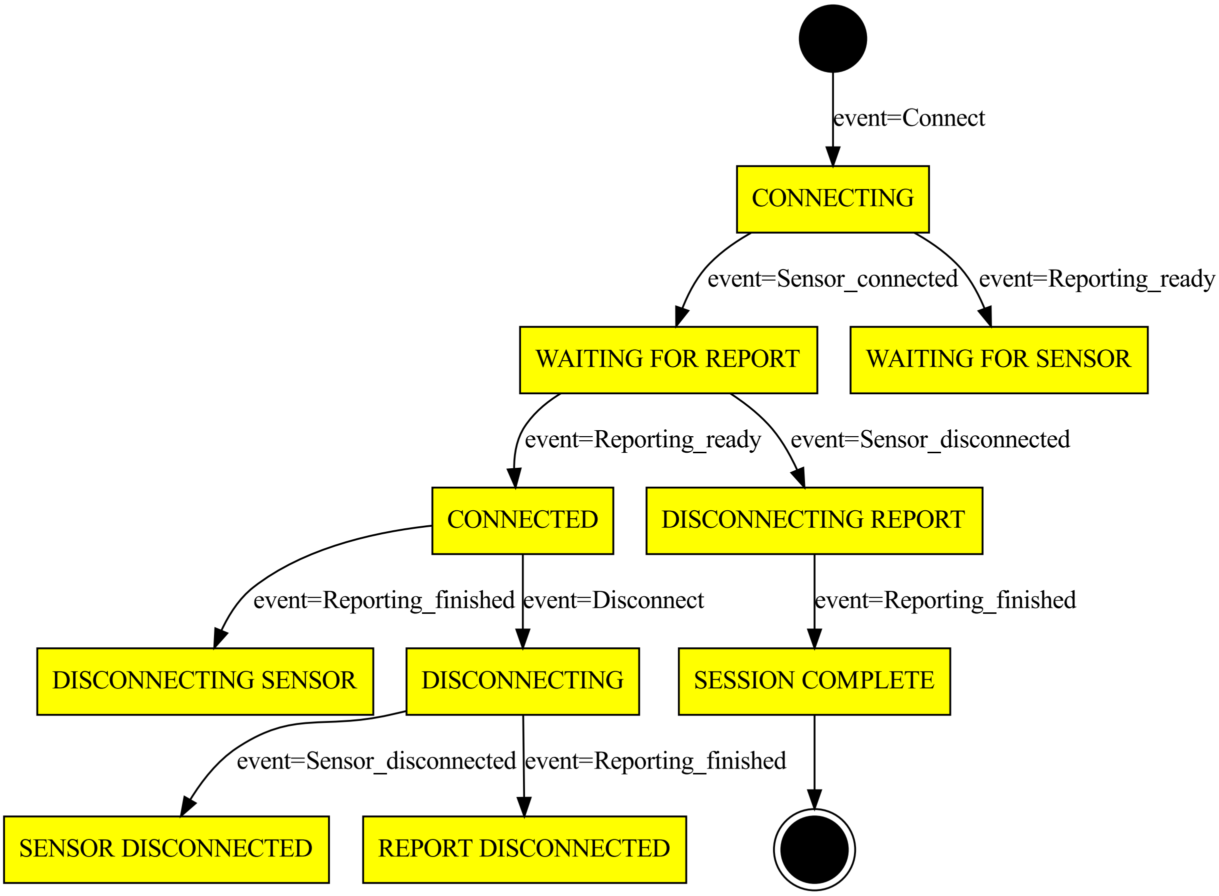

Figure 4.3: Spanning tree of Conduit state model

Tests

```
<<method tests>>=
test spanning-1.0 {
    Spanning tree for the state model graph of the Reporting Session class
} -cleanup {
    $session_graph destroy
    $span_tree destroy
} -body {
    set session_graph [aggrmgmt stateModelGraph Reporting_Session]
    miccautil dfs $session_graph
    set span_tree [miccautil spanningTree $session_graph]

    variable tree_nodes 0

    set walkproc [lambda@ [namespace current] {action graph node} {
        variable tree_nodes
        foreach outarc [$graph arcs -out $node] {
            set node_type [$graph arc get $outarc type]
            if {$node_type eq "tree"} {
                incr tree_nodes
            }
```

```
            set target [$graph arc target $outarc]
            # log::debug "$node - [$graph arc get $outarc event] -> $target\
                ==> $node_type"
        }
    }]
    $session_graph walk [$session_graph get initialstate]\
        -order pre -type bfs -dir forward -command $walkproc

    return $tree_nodes
} -result {11}
```

Graphviz view of a graph

::miccautil graphToDot *graph* ?*edgekeys*? ?*nodekeys*?

graph	a graph command as returned from `struct::graph`, usually obtained by invoking, *modelobj* stateModelGraph *class*.
edgekeys	a list of key names which will be included as the label of an edge.
nodekeys	a list of key names which will be included in the label of a node.

The `graphToDot` subcommand returns a `Tcldot` command handle which is the `dot` representation of *graph*. The edges of the dot drawing are annotated with the values given by the edge attribute keys contained in the *edgekeys* list. Similarly, the nodes of the dot drawing are annotated with the values given by the node attribute keys contained in the *nodekeys* list. The command handle can be used in the same way as those returned by the `stateModelDot` method.

Note that invoking `graphToDot` with the return value of `stateModelGraph` does *not* yield the same rendering as the `stateModelDot` method. The later method insures the rendered state model appears more in line with usual UML notation.

```
<<package exports>>=
namespace export graphToDot
```

Implementation

```
<<miccautil commands>>=
proc ::miccautil::graphToDot {graph {edgekeys {}} {nodekeys {}}} {
    package require Tcldot

    set dot_graph [dotnew digraph]
    $dot_graph setnodeattributes shape box
    $dot_graph setnodeattributes style filled
    $dot_graph setnodeattributes fillcolor yellow

    foreach node [$graph nodes] {
        set dot_node [$dot_graph addnode $node]
        if {$node eq "@"} {
            $dot_node setattributes\
                shape circle fontcolor white fillcolor black
            set label_value {}
        } elseif {$node eq "__x__"} {
            $dot_node setattributes\
                shape doublecircle fontcolor white fillcolor black
            set label_value {}
        } else {
```

```
            set label_value [string map {_ { }} $node]
        }
        set node_attrs [$graph node keys $node]
        set node_keys [lmap nodekey $nodekeys {
            expr {$nodekey in $node_attrs ?
                "$nodekey=[$graph node get $node $nodekey]" : [continue]}
        }]
        if {$label_value ne {}} {
            set label_value [concat [list $label_value] $node_keys]
        } else {
            set label_value $node_keys
        }
        set label_value [join $label_value "\\n"]
        $dot_node setattributes label $label_value
    }

    foreach arc [$graph arcs] {
        set source [$graph arc source $arc]
        set target [$graph arc target $arc]
        set dot_edge [$dot_graph addedge $source $target]
        set label_value {}
        set nl {}
        set edge_attrs [$graph arc keys $arc]
        foreach edgekey $edgekeys {
            if {$edgekey in $edge_attrs} {
                append label_value $nl\
                        ${edgekey}=[$graph arc get $arc $edgekey]
                set nl "\\n"
            }
        }
        $dot_edge setattributes label $label_value
    }

    return $dot_graph
}
```

Tests

```
<<method tests>>=
test graphToDot-1.0 {
    Draw spanning tree for the state model graph of the Reporting Session class
} -cleanup {
    $session_graph destroy
    $span_tree destroy
    chan close $span_file
    chan close $dfs_file
    rename $span_dot {}
    rename $dfs_dot {}
} -body {
    set session_graph [aggrmgmt stateModelGraph Reporting_Session]
    miccautil dfs $session_graph
    set span_tree [miccautil spanningTree $session_graph]

    set span_dot [miccautil graphToDot $span_tree event]

    set span_file [open Conduit_span.pdf w]
    $span_dot write $span_file pdf

    miccautil dfs $session_graph
    set dfs_dot [miccautil graphToDot $session_graph type {pre rpost}]
    set dfs_file [open Conduit_dfs.pdf w]
    $dfs_dot write $dfs_file pdf
```

```
} -result {}
```

Chapter 5

Code Layout

In literate programming terminology, a *chunk* is a named part of the final program. The program chunks form a tree and the root of that tree is named, *, by default. We follow the convention of naming the root the same as the output file name. The process of extracting the program tree formed by the chunks is called *tangle*. By the default the program, **atangle**, extracts the root chunk to produce the Tcl source file.

miccautil Source

```
<<miccautil.tcl>>=
<<edit warning>>
<<copyright info>>
# ++
#  Project:
#    mrtools
#
#  Module:
#    miccautil source code
# --

<<required packages>>

namespace eval ::miccautil {
    <<package exports>>
    namespace ensemble create

    <<logger setup>>

    variable version 1.3
}

<<miccautil commands>>

package provide miccautil $::miccautil::version
```

Testing Source

```
<<miccautil.test>>=
#!/usr/bin/env tclsh
#
<<edit warning>>
```

```
#
<<copyright info>>
#
# ++
#  Project:
#    mrtools
#
#  Module:
#    miccautil test code
# --

package require Tcl 8.6
package require cmdline
package require logger
package require logger::utils
package require logger::appender
package require fileutil
package require ral
package require ralutil
package require tcltest
package require lambda

# Add custom arguments here.
set optlist {
    {level.arg warn {Log debug level}}
}
array set options [::cmdline::getKnownOptions argv $optlist]

::logger::setlevel $options(level)

tcltest::configure {*}$argv

source ../code/miccautil.tcl

namespace eval ::miccautil::test {
    set logger [::logger::init miccautil::test]
    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 miccautil::test

    log::info "testing miccautil version: [package require miccautil]"

    namespace import ::tcltest::*
    namespace import ::ral::*
    namespace import ::ralutil::*

    <<test utilities>>
    <<constructor tests>>
    <<method tests>>

    cleanupTests
}
```

Edit Warning

We want to make sure to warn readers that the source code is generated and not manually written.

```
<<edit warning>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
```

Copyright Information

The following is copyright and licensing information.

```
<<copyright info>>=
#  This software is copyrighted 2020 - 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.
```

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 code which is then given to the `micca` program. 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 code in an order suitable for a language processor. 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 that are not familiar with the literate style and who are adept at reading source code directly, the chunk 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 in an acceptable order.

Index