dbusclient

A Tcl package for DBus Client Interactions

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	January 23, 2019	Initial coding.	GAM
1.0	March 1, 2019	Initial release.	GAM

Contents

List of Figures

Introduction

This document is about a Tcl package named `dbusclient`. `Dbusclient` is a Tcl package that facilitates interacting with a DBus daemon to obtain services from other programs on a DBus.

DBus is an inter-process communications mechanism that supports remote procedure call and asynchronous notifications. It is typically used in Linux systems to provide a means for various system services to be organized. The DBus specification gives the details of the protocol.

This document is also a literate program and contains all the design information and code for the `dbusclient` package. Readers unfamiliar with literate programs should consult the section below for more details on the syntax of literate programming chunks used in this document.

Package Overview

The `dbusclient` package is implemented as two classes using TclOO. This means we need a relatively new version of Tcl. Since we also tend to use the latest Tcl commands, we'll insist upon the latest Tcl version (as of this writing).

```
<<required packages>>=
package require Tcl 8.6
```

The `dbusclient` package builds upon the Tcl bindings to DBus contained in the `dbus` package by Schelte Bron. The `dbus` package provides the fundamental interface to the DBus "C" library used to access the DBus daemon. Since the `dbus` package is the fundamental interface to the DBus, we require that package to operate.

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

There are two TclOO class commands in the packdage and they are exported from the namespace and a namespace ensemble command is also created with them.

```
<<package exports>>=
namespace export Connection
namespace export Service
```

Connection class

The `Connection` class represents the communications connection to a DBus daemon. It's primary purpose is the encapsulate the calls to `dbus` package commands, supplying the required bus identifier for those calls. The methods in the class do not provide a comprehensive interface to the `dbus` package. Only those `dbus` commands that are useful for a client are represented.

```
<<connection class>>=
::oo::class create ::dbusclient::Connection {
    variable busId ;                # ❶

    <<connection methods>>
}
```

❶ We use the bus identifier in all the methods, so just declare it a variable and have it automatically imported into each method.

Connection constructor

Construction of a `Connection` is a thin wrapper on the underlying `dbus` package command.

```
<<connection methods>>=
constructor {address} {
    ::logger::import -all -force -namespace log dbusclient ;        # ❶

    set busId [::dbus connect $address]
    log::info "connected to dbus version = [::dbus info $busId version]"
}
```

❶ Bring in all the logger apparatus to the local object instance namespace. Logging setup is shown below.

Tests

```
<<connection tests>>=
test Dbus-create-1.0 {
    Create a DBus object
} -setup {
} -cleanup {
    sysBus destroy
} -body {
    Connection create sysBus system
    llength [info commands sysBus]
} -result {1}
```

Connection destructor

When we destroy a connection, we want to make sure to destroy any services that may be using this connection. So we have to iterate across any connected services.

```
<<connection methods>>=
destructor {
    try {
        foreach service [my FindConnectedServices] {
            $service destroy
        }
    } finally {
        ::dbus close $busId
    }
}
```

To find the connected bus clients, we look for all the instances of the `Service` class and determine if they are using this particular connection.

```
<<connection methods>>=
method FindConnectedServices {} {
    set connected [list]
    set me [self]

    set services [info class instances ::dbusclient::Service]
    foreach service $services {
        if {[$service connectedTo] eq $me} {
            lappend connected $service
        }
    }

    log::debug "services connected to $me : \"[join $connected {, }]\""
```

```
    return $connected
}
```

Connection call method

In DBus terms, a method is a member of an interface that is supported by an object instance. Calling the method is a request to perform the function of the method.

The only complication here deals with the signature of the method. The `dbus` package documentation describes the details of signatures in DBus, but, as you might suspect, it is a means of dealing with DBus data typing in the face of Tcl insisting that *everything is a string*. We interpret an empty string signature as not wishing to pass in the signature to the `dbus` command. Later we will see that internally, `Service` objects always supply the signature because it is available from the service introspection.

```
<<connection methods>>=
method call {target path interface method sig args} {
    set cmd [list ::dbus call $busId -dest $target]
    if {$sig ne {}} {
        lappend cmd -signature $sig
    }
    lappend cmd $path $interface $method

    log::debug "calling method $cmd $args"
    tailcall {*}$cmd {*}$args
}
```

Connection listen method

The DBus daemon performs the work required to multi-cast bus messages to subscribers. For DBus signals, you need register an event handler to receive the message. The `listen` method accomplishes that. Some of the contortions in the implementation of the method arise from the way the `dbus listen` command deals with adding, removing or querying the listen information.

```
<<connection methods>>=
method listen {path member args} {
    log::debug "setting listen for \"$path\" $member $args"
    set nargs [llength $args]
    if {$nargs > 1} {
        set msg "wrong # of args: expected listen path member ?script?"
        throw [list CONNECTION WRONGARGS $msg] $msg
    }

    set cmd [list ::dbus listen $busId $path $member]
    if {$nargs == 1} {
        lappend cmd [lindex $args 0]
    }
    tailcall {*}$cmd
}
```

Connection filter method

DBus filters provide a means of limiting what the DBus daemon sends asynchronously.

```
<<connection methods>>=
method filter {subcmd args} {
    log::debug "filter $subcmd $args"
    tailcall ::dbus filter $busId $subcmd {*}$args
}
```

Service class

The `Service` class represents a service provider on a DBus. Server programs may connect to the DBus daemon and request a name for the connection. This name typically follows an inverted URI host name (*e.g.* com.example). In DBus terms, these are bus names and give the target to which service requests may be directed. You can think of bus names as being similar to *well known* ports in Internet protocols. You know the name of the service in which you are interested *a priori*, although it is possible to get a list of all the bus names on a DBus.

```
<<service class>>=
::oo::class create ::dbusclient::Service {
    <<service methods>>
}
```

Service metadata

A service on the DBus exposes its capabilities by introspection. To access a service on the DBus, it is necessary to direct requests to a particular service, an object contained in that service, an interface implemented by the object, and a member of the interface. To know what objects and interfaces are supported by a service, the objects of the service support the standard interface, `org.freedesktop.DBus.Introspectable`.

By using the `Introspect` method in this interface it is possible to uncover all the objects and interfaces supported by a service. The `dbusclient` package caches this information to make various tasks easier, such as immediate access to property values and keeping track of property values when they change. In this section, we show a class diagram of the how the service metadata is held.

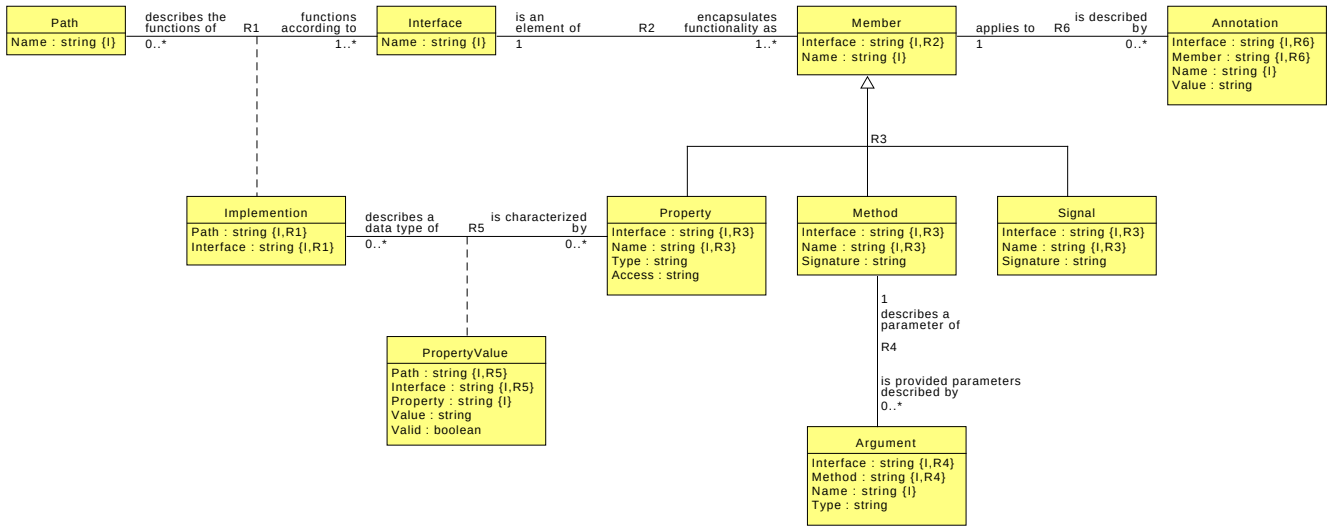

Figure 1: Class Diagram for DBus Service Metadata

This class diagram shows that a **Path** implements at least one **Interface** (R1) which describes how an object (aka **Path**) functions. An **Interface** has at least one **Member** (R2) to define the functionality of the **Interface**. A **Member** can be optionally have **Annotations** (R6). **Members** are of three types (R3), **Property**, **Method**, or **Signal**. **Methods** can optionally have **Arguments** (R4). Each **Implementation** of an **Interface** by a **Path** has distinct **PropertyValues** for **Properties** of the **Interface** (R5). Note that R5 is conditional on both sides. This implies that eventhough a **Path** implements an **Interface** that is described by a set of **Property**, a particular **Path** may still not have a value for the **Property**. This seems to be the way the DBus daemon works, however much the conditionality of R5 is weak.

To implement this data schema, we use the `ral` package to hold the data in relvars within the `Service` instances.

```
<<required packages>>=
package require ral
package require ralutil
```

The previous class diagram can be transcribed to `ral` commands as follows.

```
<<service meta-data definitions>>=
relvar create Path {
    Name string
} Name

relvar create Interface {
    Name string
} Name

relvar create Implementation {
    Path string
    Interface string
} {Path Interface}

relvar create Member {
    Interface string
    Name string
} {Interface Name}

relvar create Method {
    Interface string
    Name string
    Signature string
} {Interface Name}

relvar create Property {
    Interface string
    Name string
    Type string
    Access string
} {Interface Name}

relvar create Signal {
    Interface string
    Name string
    Signature string
} {Interface Name}

relvar create Argument {
    Interface string
    Method string
    Name string
    Type string
} {Interface Method Name}

relvar create PropertyValue {
    Path string
    Interface string
    Property string
    Value string
    Valid boolean
} {Path Interface Property}

relvar create Annotation {
    Interface string
```

```
    Member string
    Name string
    Value string
} {Member Name}

relvar correlation R1 Implementation\
        Path * Path Name\
        Interface * Interface Name
relvar association R2\
        Member Interface +\
        Interface Name 1
relvar partition R3 Member {Interface Name}\
        Method {Interface Name}\
        Property {Interface Name}\
        Signal {Interface Name}
relvar association R4\
        Argument {Interface Method} *\
        Method {Interface Name} 1
relvar correlation R5 PropertyValue\
        {Path Interface} * Implementation {Path Interface}\
        {Interface Property} * Property {Interface Name}
relvar association R6\
        Annotation {Interface Member} *\
        Member {Interface Name} 1
```

Service constructor

Constructing a `Service` instance is quite complicated. The complications arise from the need to use introspection to discover the object instances and interfaces they support. Objects are given file system path-like names and we use the term *path* here to mean an object instance. During the introspection, we store the data obtained in the `relvars` shown previously, effectively caching the introspection data in a form that is easier to query and manipulate.

```
<<service methods>>=
constructor {svc conn} {
    namespace import ::ral::*
    namespace import ::ralutil::*

    ::logger::import -all -force -namespace log dbusclient

    my variable svcName
    set svcName $svc
    if {![::dbus validate interface $svcName]} {
        set msg "Invalid service name syntax"
        throw [list CONSTRUCTOR BADSERVICE $msg] $msg
    }

    my variable connId
    set connId $conn

    my variable traceNumber ;                   # ❶
    set traceNumber 0

    <<service meta-data definitions>>

    relvar eval {
        my Introspect /     ;                   # ❷
        my CreatePropertyValues
    }

    my InitializeProperties ;                   # ❸
```

```
    my SetupStdSignals ;                        # ❹

    # It can be useful to see the service metadata when exploring
    # new servers.
    foreach rv [lsort [relvar names [namespace current]::*]] {
        log::info "\n[relformat [relvar set $rv] [namespace tail $rv]]"
    }
}
```

❶ Used for identifying trace requests.

❷ Recursively introspect the service starting at the root path. Note the introspection is done in a relvar transaction so we can defer enforcing the referential integrity until all the introspection is finished.

❸ Fetch the property values for the discovered object.

❹ Set up signal handlers for the internal mechanisms used to keep the local cache of properties and interfaces up to date. The DBus standard defines standard interfaces which contain signal members that can be used to insure that internal data structures are keep up to date.

Each DBus has a service, `org.freedesktop.DBus`, that represents the bus itself. The following test uses this service

Tests

```
<<service tests>>=
test Service-create-1.0 {
    Create an instance representing the bus itself
} -setup {
    set conn [Connection create sysBus system]
} -cleanup {
    sysBus destroy
} -body {
    Service create DBus org.freedesktop.DBus $conn
    llength [info commands DBus]
} -result {1}
```

Introspection of objects

When the `Introspect` method of the `org.freedesktop.DBus.Introspectable` interface is executed, the return is a chunk of XML containing the description of the objects of the service and the interfaces they implement. We use the `tdom` package to parse the XML.

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

The XML schema is defined by the DBus spec and we do not repeat it here. The essential elements of the schema are the interfaces implemented by the object instance and a list of child nodes that give the path to child object instances of the service. So the logic of storing the metadata from the introspection first creates the path represented by the object instance followed by the interfaces it implements and finally creates all the child nodes. As we see below, creating the child nodes causes the introspection to be invoked recursively.

```
<<service methods>>=
method Introspect {path}  {
    my variable connId svcName

    set svcxml [$connId call $svcName $path\
            org.freedesktop.DBus.Introspectable Introspect {}]
    log::debug "xml for $path:\n$svcxml"
```

```
    set svcdoc [dom parse -simple $svcxml]
    set svcroot [$svcdoc documentElement]

    try {
        relvar insert Path [list Name $path]
        my CreateInterfaces $path $svcroot
        my CreateNodes $path $svcroot
    } finally {
        $svcdoc delete
    }

    return
}
```

The only complication in creating interfaces has to do with the redundant information sent in the introspection XML. Interface names are unique, yet the complete interface description is sent along with every object that implements a particular interface. So, we check to make sure we haven't already seen the interface definition. Otherwise, interfaces consist of methods, properties and signals and they must be inserted into the service metadata.

```
<<service methods>>=
method CreateInterfaces {path root} {
    set intfnodes [$root getElementsByTagName interface]

    foreach intfnode $intfnodes {
        set intfname [$intfnode getAttribute name]

        set intfMissing [relation isempty \
                [relvar restrictone Interface Name $intfname]\
        ]

        if {$intfMissing} {
            relvar insert Interface [list Name $intfname]

            my CreateMethods $intfnode
            my CreateProperties $intfnode
            my CreateSignals $intfnode
        }
        relvar insert Implementation [list\
            Path $path\
            Interface $intfname\
        ] ;                                                    # ❶
    }

    return
}
```

❶ Whether the interface has been seen previously or not, we need an instance of **Implementation** to indicate that the given **Path** implements the **Interface**.

Methods have arguments. The XML for arguments has optional parts for the attribute name and the direction of the argument, either *in* or *out*. For missing names, we make one up. For missing direction, the default is that the argument is an input. The concatenation of the argument types for the method arguments forms the method signature. That is a characteristic of the way types are specified in DBus.

```
<<service methods>>=
method CreateMethods {intfnode} {
    set intfName [$intfnode getAttribute name]
    set methNodes [$intfnode getElementsByTagName method]
    foreach methNode $methNodes {
```

```
        set methName [$methNode getAttribute name]
        set haveMeth [pipe {
            relvar restrictone Method Interface $intfName Name $methName |
            relation isnotempty ~
        }]
        if {$haveMeth} {  #                                    ❶
            continue
        }
        set methSig {}
        set argCounter -1

        set argNodes [$methNode getElementsByTagName arg]
        foreach argNode $argNodes {
            set argName [expr {[$argNode hasAttribute name] ?\
                    [$argNode getAttribute name] :\
                    "arg_[incr argCounter]"}]
            set argDir [expr {[$argNode hasAttribute direction] ?\
                    [$argNode getAttribute direction] : "in"}]
            set argType [$argNode getAttribute type]
            if {$argDir eq "in"} {
                append methSig $argType
                set ainserted [relvar insert Argument [list\
                    Interface $intfName\
                    Method $methName\
                    Name $argName\
                    Type $argType\
                ]]
            }
        }

        set memTuple [list Interface $intfName Name $methName]
        relvar insert Member $memTuple
        relvar insert Method [list\
            {*}$memTuple\
            Signature $methSig\
        ]

        my CreateAnnotations $methNode $intfName $methName
    }

    return
}
```

❶ Yes, sometimes method names are duplicated within the interface. Don't know why. Looks like a bug. But we'll accomodate it here.

All interface members may optionally have an annotation attached to them. Usually this is used to denote a deprecated method in an interface.

```
<<service methods>>=
method CreateAnnotations {node interface member} {
    set noteNodes [$node getElementsByTagName annotation]
    foreach noteNode $noteNodes {
        set noteName [$noteNode getAttribute name]
        set noteValue [$noteNode getAttribute value]
        relvar insert Annotation [list\
            Interface $interface\
            Member $member\
            Name $noteName\
            Value $noteValue\
        ]
```

```
    }
}
```

Properties are data values supported by an interface. They have types and access permissions.

```
<<service methods>>=
method CreateProperties {intfnode} {
    set intfName [$intfnode getAttribute name]
    set propNodes [$intfnode getElementsByTagName property]
    foreach propNode $propNodes {
        set propName [$propNode getAttribute name]
        set haveProp [pipe {
            relvar restrictone Property Interface $intfName Name $propName |
            relation isnotempty ~
        }]
        if {$haveProp} {
            continue
        }
        set propType [$propNode getAttribute type]
        set propAccess [$propNode getAttribute access]

        set memTuple [list Interface $intfName Name $propName]
        relvar insert Member $memTuple
        relvar insert Property [list\
            {*}$memTuple\
            Type $propType\
            Access $propAccess\
        ]

        my CreateAnnotations $propNode $intfName $propName
    }

    return
}
```

Interfaces may also have signals associated with them. Signals are sent asychronously. As we have seen already, there are a set of standard signals used to maintain the population of properties and interfaces. These are handled internally by the package. Other signals may be of interest to the client application.

```
<<service methods>>=
method CreateSignals {intfnode} {
    set intfName [$intfnode getAttribute name]

    set sigNodes [$intfnode getElementsByTagName signal]
    foreach sigNode $sigNodes {
        set sigName [$sigNode getAttribute name]
        set haveSig [pipe {
            relvar restrictone Signal Interface $intfName Name $sigName |
            relation isnotempty ~
        }]
        if {$haveSig} {
            continue
        }
        set sigSig [list]

        set argCounter -1
        set argNodes [$sigNode getElementsByTagName arg]
        foreach argNode $argNodes {
            set argName [expr {[$argNode hasAttribute name] ?\
                    [$argNode getAttribute name] :\
                    "arg_[incr argCounter]"}]
            set argType [$argNode getAttribute type]
```

```
            append sigSig $argType
        }

        set memTuple [list Interface $intfName Name $sigName]
        relvar insert Member $memTuple
        relvar insert Signal [list\
            {*}$memTuple\
            Signature $sigSig\
        ]

        my CreateAnnotations $sigNode $intfName $sigName
    }
}
```

The final part of introspection of a given node is to create any child nodes. Object instances and their path names are strictly hierarchical and use file system-like naming for the paths.

```
<<service methods>>=
method CreateNodes {path root} {
    set childNodes [$root getElementsByTagName node]
    foreach childNode $childNodes {
        set nodeName [$childNode getAttribute name]
        set fullpath [file join $path $nodeName] ;      # ❶
        my Introspect $fullpath ;                       # ❷
    }
}
```

❶ File join does the right thing here, particularly if `path` is set to "/". Better than just sandwiching a "/" between the path and its child node name using sting concatenation.

❷ Here is the recursive introspection to obtain the information for child nodes.

Once we know all the paths and interfaces in a service and know which interfaces are implemented by which paths, then we can create **PropertyValue** tuples for the properties that are associated with a given path. At this point, we don't know the values of the properties, so each property value is set to the empty string and marked as invalid.

```
<<service methods>>=
method CreatePropertyValues {} {
    pipe {
        relvar set Implementation |
        relation join ~ [relvar set Property] |
        relation project ~ Path Interface Name |
        relation rename ~ Name Property |
        relation extend ~ exTup Value string {""} Valid boolean {"false"} |
        relvar set PropertyValue ~
    } ;                                                                 # ❶

    return
}
```

❶ The strategy behind this query starts with realizing that we intend to set a new value for the **PropertyValue** relvar. So, we need all the tuples in that value. Since **PropertyValue** is a correlation between **Property** and **Implementation**, we can derive a new value by joining the two, projecting the identifiers and then adding the attributes, by extension, required for **PropertyValue**. As long as we get the correct header, then we can set the value of the relvar just like any other variable. In this way, the new value is generated as a complete set and no explicit iteration is required.

After introspection is completed, we can then request the property values and place those values in the metadata cache.

```
<<service methods>>=
method InitializeProperties {} {
    my variable connId svcName

    # check if there is an object manager, we can get the properties
    # from it and set up listening for future changes.
    set objMgrPaths [pipe {
        relvar set Implementation |
        relation restrictwith ~\
            {$Interface eq "org.freedesktop.DBus.ObjectManager"} |
        relation list ~ Path
    }]

    foreach path $objMgrPaths {
        set managed [$connId call $svcName $path\
                org.freedesktop.DBus.ObjectManager GetManagedObjects {}]
        log::debug "managed objects = $managed"

        dict for {objPath objDesc} $managed {
            dict for {intf props} $objDesc {
                dict for {propName propValue} $props {
                    my UpdatePropertyValue $objPath $intf $propName $propValue
                }
            }
        }

        # Add signal traces for the signals in ObjectManager that
        # let us know when interfaces have changed.
        my trace add signal org.freedesktop.DBus.ObjectManager\
                InterfacesAdded ${path}* [mymethod InterfacesAdded]
        my trace add signal org.freedesktop.DBus.ObjectManager\
                InterfacesRemoved ${path}* [mymethod InterfacesRemoved]
    }

    # Check for paths that implement org.freedesktop.DBus.Properties
    # We can then invoke the GetAll method using the property
    # interface to get the set of valid values.
    set propPaths [pipe {
        relvar set Implementation |
        relation restrictwith ~\
                {$Interface eq "org.freedesktop.DBus.Properties"} |
        relation project ~ Path |
        relation join ~ [relvar set PropertyValue] |
        relation project ~ Path Interface
    }]

    relation foreach propPath $propPaths {
        relation assign $propPath
        set values [$connId call $svcName $Path org.freedesktop.DBus.Properties\
                GetAll {} $Interface]
        dict for {propName propValue} $values {
            my UpdatePropertyValue $Path $Interface $propName $propValue
        }
    }
    # Add signal traces for the signals in Properties that
    # let us know when property values have changed.
    if {[relation isnotempty $propPaths]} {
        set svcroot /[string map {. /} $svcName]
        my trace add signal org.freedesktop.DBus.Properties\
                PropertiesChanged ${svcroot}* [mymethod PropertiesChanged]
    }
```

```
}
```

We update property values from several locations, so that code is factored into a method.

```
<<service methods>>=
method UpdatePropertyValue {path interface property value} {
    log::debug "updating property value: $path $interface.$property $value"
    relvar updateone PropertyValue propTup [list\
        Path $path Interface $interface Property $property] {
            tuple update $propTup Value $value Valid true
        }
}
```

Invalidating a property value is also an update operation on an instance of the **PropertyValue** relvar.

```
<<service methods>>=
method InvalidatePropertyValue {path interface property} {
    log::debug "invalidating property value: $path $interface.$property"
    relvar updateone PropertyValue propTup [list\
        Path $path Interface $interface Property $property] {
            tuple update $propTup Valid false
        }
}
```

Finally, we have to set up the DBus to tell us when the standard signals have occurred. This configures the DBus daemon's listening and filtering to route any signals from our service to us.

```
<<service methods>>=
method SetupStdSignals {} {
    my variable connId svcName

    set stdSignals [my QueryStdSignals]
    foreach stdSignal $stdSignals {
        $connId listen {} $stdSignal [mymethod HandleSignals]
    }
    if {[llength $stdSignals] != 0} {
        $connId filter add -type signal -sender $svcName
    }

    return
}
```

Notice we used the `mymethod` command to specify a callback that is a method of the `Service` class. This is a utilty function available from tcllib.

```
<<required packages>>=
package require oo::util
```

```
<<service methods>>=
method QueryStdSignals {} {
    set stdInterfaces {
        PropertiesChanged
        InterfacesAdded
        InterfacesRemoved
    }

    return [pipe {
        relvar set Signal |
        relation restrictwith ~ {$Name in $stdInterfaces} |
        relation list ~ Name
    }]
}
```

```
<<service methods>>=
method TakeDownStdSignals {} {
    my variable connId svcName

    set stdSignals [my QueryStdSignals]
    foreach stdSignal $stdSignals {
        $connId listen {} $stdSignal {}
    }
    if {[llength $stdSignals] != 0} {
        $connId filter remove -type signal -sender $svcName
    }

    return
}
```

Destructor

Destroying an service instance means we must clean up our interactions with the DBus and discard the service metadata.

```
<<service methods>>=
destructor {
    my TakeDownStdSignals

    relvar constraint delete {*}[relvar constraint names [namespace current]::*]
    relvar unset {*}[relvar names [namespace current]::*]
}
```

Service Introspection

Once we have connected to a service and gone through all the DBus introspection, the following methods can be used to obtain access to some of the service metadata accumulated during construction.

connectedTo method

```
<<service methods>>=
method connectedTo {} {
    my variable connId
    return $connId
}
```

Tests

```
<<service tests>>=
test Service-introspection-1.0 {
    which connection
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus connectedTo
} -result $conn
```

pathList method

```
<<service methods>>=
method pathList {} {
    return [relation list [relvar set Path] Name] ;          # ❶
}
```

❶ Since `Name` is an identifier for **Path**, we are assured that the returned list is actually a set.

Tests

```
<<service tests>>=
test Service-introspection-2.0 {
    object paths
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus pathList
} -result {/ /org/freedesktop/DBus} -match set
```

findPathsByPropertyValue method

```
<<service methods>>=
method findPathsByPropertyValue {interface property expression} {
    set matches [pipe {
        relvar set PropertyValue |
        relation restrictwith ~ {$Interface eq $interface &&\
                $Property eq $property && $Valid}
    }]
    set qualified [uplevel 1\
            [list ::ral relation restrictwith $matches $expression]]
    return [relation list $qualified Path]
}
```

Tests

```
<<service tests>>=
test Service-introspection-3.0 {
    selecting paths based on property value
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus findPathsByPropertyValue org.freedesktop.DBus Interfaces\
            {"org.freedesktop.DBus.Monitoring" in $Value}
} -result {/org/freedesktop/DBus}
```

pathInterfaces method

```
<<service methods>>=
method pathInterfaces {path} {
    return [pipe {
```

```
        relvar set Implementation |
        relation restrictwith ~ {$Path eq $path} |
        relation list ~ Interface
    }]
}
```

For these tests, we want to be able to compare sets for equality. Fortunately, `tcllib` provides all the needed commands and `tcltest` allows us to define custom matching procedures.

Tests

```
<<required packages for test>>=
package require struct::set
```

```
<<test utilities>>=
customMatch set {::struct::set equal}
```

```
<<service tests>>=
test Service-introspection-4.0 {
    querying the interfaces supported by a path
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus pathInterfaces /org/freedesktop/DBus
} -result {org.freedesktop.DBus org.freedesktop.DBus.Properties\
    org.freedesktop.DBus.Introspectable org.freedesktop.DBus.Monitoring\
    org.freedesktop.DBus.Debug.Stats org.freedesktop.DBus.Peer}\
    -match set
```

pathProperties method

```
<<service methods>>=
method pathProperties {path interface} {
    return [pipe {
        relvar set PropertyValue |
        relation restrictwith ~\
            {$Path eq $path && $Interface eq $interface && $Valid} |
        relation list ~ Property
    }]
}
```

interfaceMethods method

```
<<service methods>>=
method interfaceMethods {interface} {
    return [pipe {
        relvar set Method |
        relation restrictwith ~ {$Interface eq $interface} |
        relation list ~ Name
    }]
}
```

Tests

```
<<service tests>>=
test Service-introspection-5.0 {
    querying the methods supported by an interface
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus interfaceMethods org.freedesktop.DBus.Properties
} -result {Get GetAll Set} -match set
```

methodSignature method

```
<<service methods>>=
method methodSignature {interface method} {
    return [pipe {
        relvar restrictone Method Interface $interface Name $method |
        relation extract ~ Signature
    }]
}
```

Tests

```
<<service tests>>=
test Service-introspection-6.0 {
    querying the signature of a method
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus methodSignature org.freedesktop.DBus.Properties Get
} -result {ss}
```

interfaceProperties method

```
<<service methods>>=
method interfaceProperties {interface} {
    return [pipe {
        relvar set Property |
        relation restrictwith ~ {$Interface eq $interface} |
        relation list ~ Name
    }]
}
```

Tests

```
<<service tests>>=
test Service-introspection-7.0 {
    querying the properties for an interface
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
```

```
    DBus interfaceProperties org.freedesktop.DBus
} -result {Features Interfaces} -match set
```

interfaceSignals method

```
<<service methods>>=
method interfaceSignals {interface} {
    return [pipe {
        relvar set Signal |
        relation restrictwith ~ {$Interface eq $interface} |
        relation list ~ Name
    }]
}
```

Tests

```
<<service tests>>=
test Service-introspection-8.0 {
    querying the signals for an interface
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus interfaceSignals org.freedesktop.DBus.Properties
} -result {PropertiesChanged}
```

A test server

It is convenient to put up our own service against which we can test. Fortunately, the [package dbif] package, also by Schelte Bron, provides some convenient commands to do that. In this section, we show a test server build using the `dbif` package. This server is then used for testing other parts of this package.

```
<<test-server.tcl>>=
#!/usr/bin/env tclsh
package require dbif

puts stderr "test server starting"

dbif connect -bus session -noqueue -replace com.modelrealization.test

<<test server methods>>
<<test server properties>>
<<test server signals>>

vwait forever
```

We include some methods that can be invoked on the server. In particular, we want to be able to exit programmaticallly.

```
<<test server methods>>=
dbif method /com/modelrealization/test Quit {
    dbif return $msgid {}
    puts stderr "test server exiting"
    exit
}

dbif method /com/modelrealization/test AddToCounter {cnt} {i} {
```

```
    incr ::Counter $cnt
    dbif return $msgid $::Counter
}

dbif method /com/modelrealization/test Trigger {
    dbif generate $::AttnSigId
}
```

We also add some properties that can be accessed.

```
<<test server properties>>=
dbif property -attributes {Property.EmitsChangedSignal true}\
        /com/modelrealization/test Counter:i Counter
set ::Counter 0

dbif property /com/modelrealization/test Name:s Name
set ::Name "Test Server"

dbif property -access read\
        /com/modelrealization/test Source:s Source
set ::Source "Model Realization"
```

And finally, a signal that can be sent to a client.

```
<<test server signals>>=
set ::AttnSigId [dbif signal /com/modelrealization/test Attention\
        {Count:i Identity:s} {} {
    return [list $::Counter $::Source]
}]
```

The test service is started by the setup portion of the test script.

```
<<test server setup>>=
exec ./test-server.tcl &
```

To clean up the server, we can call its `Quit` method to force an exit.

```
<<test utilities>>=
proc killServer {} {
    set conn [Connection create sessionBus session]
    Service create tserver com.modelrealization.test $conn
    tserver call /com/modelrealization/test com.modelrealization.test\
            Quit

    $conn destroy
}
```

```
<<test server cleanup>>=
killServer
```

Calling methods

One of the primary activities to perform on a service is to call one of its methods.

```
<<service methods>>=
method call {path interface method args} {
    my ValidateCall $path $interface $method $args
    my variable connId svcName
    tailcall $connId call $svcName $path $interface $method\
        [my methodSignature $interface $method] {*}$args
}
```

Tests

```
<<test utilities>>=
proc isInSet {expected actual} {
    return [::struct::set contains $actual $expected]
}
```

```
<<test utilities>>=
customMatch contains [namespace code isInSet]
```

```
<<service tests>>=
test Service-call-1.0 {
    Invoke a method
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus call /org/freedesktop/DBus\
            org.freedesktop.DBus ListNames
} -result {org.freedesktop.DBus} -match contains
```

We use the metadata that we have collected about the service through introspection to validate the method call. We do this before attempting the call itself in the hope to get better error messages on this side of the interface to the DBus.

```
<<service methods>>=
method ValidateCall {path interface method arglist} {
    my ValidatePath $path
    my ValidateInterface $path $interface
    my ValidateMethod $interface $method $arglist
    my CheckAnnotations $interface $method
    return
}
```

```
<<service methods>>=
method ValidatePath {path} {
    if {![::dbus validate path $path]} {
        set msg "invalid path name: \"$path\""
        throw [list PATH INVALID $msg] $msg
    }
    set pathFound [pipe {
        relvar restrictone Path Name $path |
        relation isnotempty
    }]
    if {!$pathFound} {
        set msg "unknown path, \"$path\""
        throw [list PATH UNKNOWN $msg] $msg
    }
    return
}
```

```
<<service tests>>=
test Service-call-2.0 {
    Invoke a method -- bad path
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
```

```
    DBus call /foo org.freedesktop.DBus ListNames
} -result {unknown path, "/foo"} -returnCodes error
```

```
<<service methods>>=
method ValidateInterface {path interface} {
    if {![::dbus validate interface $interface]} {
        set msg "invalid interface name: \"$interface\""
        throw [list INTERFACE INVALID $msg] $msg
    }
    set intfFound [pipe {
        relvar restrictone Interface Name $interface |
        relation isnotempty
    }]
    if {!$intfFound} {
        set msg "unknown interface, \"$interface\""
        throw [list INTERFACE UNKNOWN $msg] $msg
    }
    set isImplemented [pipe {
        relvar restrictone Implementation Path $path Interface $interface |
        relation isnotempty
    }]
    if {!$isImplemented} {
        set msg "path, \"$path\", does not implement\
                interface, \"$interface\""
        throw [list INTERFACE NOTIMPLEMENTED $msg] $msg
    }
    return
}
```

```
<<service tests>>=
test Service-call-2.1.0 {
    Invoke a method -- bad interface
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus call /org/freedesktop/DBus org.foo ListNames
} -result {unknown interface, "org.foo"} -returnCodes error
```

```
<<service tests>>=
test Service-call-2.1.1 {
    Invoke a method -- unimplemented on a path
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus call / org.freedesktop.DBus.Properties GetAll
} -result {path, "/", does not implement interface,\
        "org.freedesktop.DBus.Properties"} -returnCodes error
```

```
<<service methods>>=
method ValidateMethod {interface method arglist} {
    if {![::dbus validate member $method]} {
        set msg "invalid method name: \"$method\""
        throw [list METHOD INVALID $msg] $msg
    }
```

```
    set methRel [relvar restrictone Method\
            Interface $interface Name $method]
    if {[relation isempty $methRel]} {
        set msg "interface, \"$interface\", does not have a method named,\
                \"$method\""
        throw [list METHOD UNKNOWN $msg] $msg
    }

    set args [pipe {
        relvar set Argument |
        relation semijoin $methRel ~\
            -using {Interface Interface Name Method}
    }]
    if {[relation cardinality $args] != [llength $arglist]} {
        set expectArgs [relation list $args Name]
        set msg "wrong # args: expected $method [join $expectArgs ,]:\
                got: [join $arglist ,]"
        throw [list METHOD WRONGARGS $msg] $msg
    }
    return
}
```

```
<<service tests>>=
test Service-call-2.2 {
    Invoke a method -- bad method
} -setup {
    set conn [Connection create sysBus system]
    Service create DBus org.freedesktop.DBus $conn
} -cleanup {
    sysBus destroy
} -body {
    DBus call /org/freedesktop/DBus org.freedesktop.DBus Foo
} -result {interface, "org.freedesktop.DBus",\
            does not have a method named, "Foo"} -returnCodes error
```

```
<<service methods>>=
method CheckAnnotations {interface member} {
    set notes [pipe {
        relvar set Annotation |
        relation restrictwith ~\
                {$Interface eq $interface && $Member eq $member}
    }]

    relation foreach note $notes {
        relation assign $note
        log::notice "member, $Interface.$Member,\
            is annotated as, $Name = $Value"
    }
}
```

Accessing properties

Once constructed, all the property values for the paths found for the service are cached in the service metadata. This method provides access to the property values.

The `property` method interface is typical of Tcl commands which set values. If no value is supplied then the property value is returned from the metadata. If a value is supplied, then a method is called to write the property value.

Note that after a write, the cached metadata is out of sync. Later, a signal is sent to indicate the property value was changed. Note that is will be necessary to enter the event loop to process that property value change and sync the cached metadata. We update the cached metadata in this way so that tracing on properties happens when we know the write was accepted.

```
<<service methods>>=
method property {path interface property args} {
    if {[llength $args] > 1} {
        set msg "wrong # args:\
                expected, \"property path interface property ?value?\""
        throw [list PROPERTY WRONGARGS $msg] $msg
    }

    set prop [relvar restrictone Property\
            Interface $interface Name $property]
    if {[relation isempty $prop]} {
        set msg "unknown property, \"$interface.$property\""
        throw [list PROPERTY UNKNOWN $msg] $msg
    }
    relation assign $prop {Access propAccess} {Type propType}

    set propValue [relvar restrictone PropertyValue\
            Path $path Interface $interface Property $property]
    if {[relation isempty $propValue]} {
        set msg "property, \"$interface.$property\", is not implemented\
                for path, \"$path\""
        throw [list PROPERTY NOTIMPLEMENTED $msg] $msg
    }

    if {[llength $args] == 1} {
        if {![string match *write* $propAccess]} {
            set msg "property, $interface.$property cannot be updated"
            throw [list PROPERTY NOWRITE $msg] $msg
        }
        set value [lindex $args 0]
        my call $path org.freedesktop.DBus.Properties Set\
                $interface $property [list $propType $value]
    } else {
        if {![string match *read* $propAccess]} {
            set msg "property, $interface.$property cannot be read"
            throw [list PROPERTY NOREAD $msg] $msg
        }
        relation assign $propValue {Value value} Valid
        if {!$Valid} {
            my variable connId svcName
            set value [$connId call $svcName $path\
                    org.freedesktop.DBus.Properties Get {} $interface $property]
            my UpdatePropertyValue $path $interface $property $value
        }
    }
    return $value
}
```

```
<<service tests>>=
test property-1.0 {
    Get a property from the test server
} -setup {
    set conn [Connection create sessionBus session]
    Service create tserver com.modelrealization.test $conn
} -cleanup {
    sessionBus destroy
} -body {
    tserver property\
            /com/modelrealization/test com.modelrealization.test Counter
} -result {0}
```

```
<<service tests>>=
test property-2.0 {
    Set an property value
} -setup {
    set conn [Connection create sessionBus session]
    Service create tserver com.modelrealization.test $conn
} -cleanup {
    tserver waitForProperty com.modelrealization.test Counter\
        /com/modelrealization/test\
        {tserver property /com/modelrealization/test\
            com.modelrealization.test Counter 100}
    sessionBus destroy
} -body {
    set chng [tserver waitForProperty com.modelrealization.test Counter\
        /com/modelrealization/test\
        {tserver property /com/modelrealization/test\
            com.modelrealization.test Counter 100}]
    dict get $chng value
} -result {100}
```

Tracing properties and signals

Services on the DBus usually implement asynchronous notifications when a property value is changed or when new paths and interfaces are added or removed. The `dbusclient` package handles these asynchronous notifications and updates the values in the service metadata if the Tcl event loop is entered. The package also provides an interface to allow programs using [package dbusclient] to be informed of the changes. This is accomplished by *tracing* the various aspects of the DBus service.

The tracing supported by `dbusclient` has an interface modeled after the Tcl core **trace** command. For the DBus, we can trace on properties, signals or paths.

The following is a class diagram of the metadata that is held to support tracing of DBus entities.

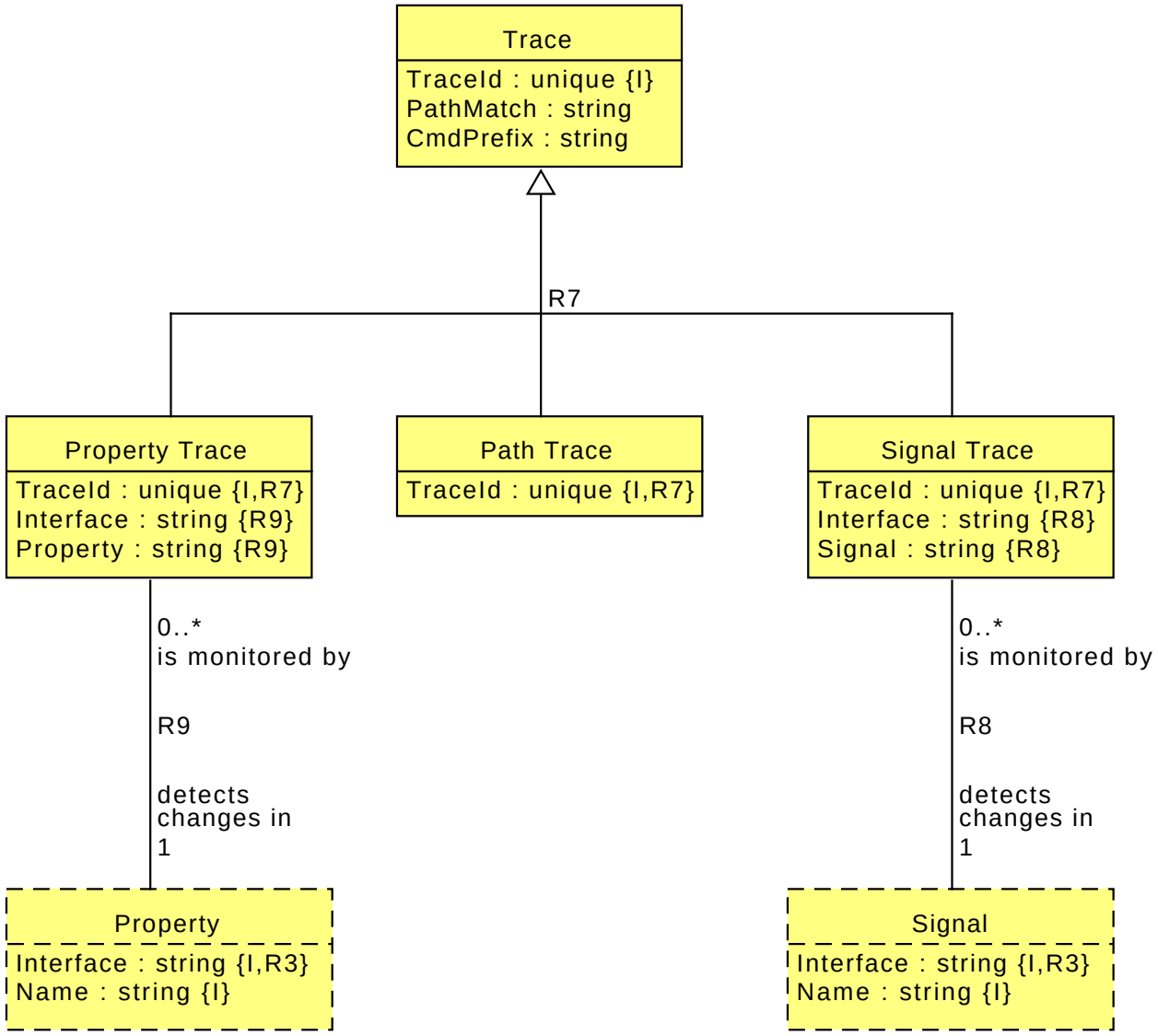

Figure 2: Class Diagram for Tracing Metadata

The diagram shows that there are three types of traces (**R7**). Each trace has an identifier, a path matching pattern and a command prefix. The path matching pattern is specified as in the **string match** command and qualifies traces to the path names they must match. The command prefix is executed when the DBus send an asynchronous notification and the conditions of the trace are matched. Property traces are further qualified by the interface and property name that is traced. Similarly, signal traces are qualified by the interface and signal name they must match. In both cases, property traces and signal traces must be related to a property or signal defined by an interface (**R8** and **R9**).

Following the pattern we used for service metadata, the previous diagram can be rendered as TclRAL `relvar` commands to establish the trace metadata schema.

```
<<service meta-data definitions>>=
relvar create Trace {
    TraceId int
    PathMatch string
    CmdPrefix string
} TraceId
```

```
relvar create PathTrace {
    TraceId int
} TraceId

relvar create SignalTrace {
    TraceId int
    Interface string
    Signal string
} TraceId

relvar create PropertyTrace {
    TraceId int
    Interface string
    Property string
} TraceId

relvar partition R7 Trace TraceId\
        PathTrace TraceId\
        SignalTrace TraceId\
        PropertyTrace TraceId

relvar association R8\
        SignalTrace {Interface Signal} *\
        Signal {Interface Name} 1

relvar association R9\
        PropertyTrace {Interface Property} *\
        Property {Interface Name} 1
```

The `trace` method determines which type of trace operation is requested and delegates the operation on to another method.

```
<<service methods>>=
method trace {operation args} {
    switch -exact $operation {
        add {
            set result [my AddTrace $args]
        }
        remove {
            set result [my RmTrace {*}$args]
        }
        info {
            set result [my InfoTrace {*}$args]
        }
        default {
            set msg "unknown trace operation, \"$operation\":\
                expected add, remove or info"
            throw [list TRACE UNKNOWNOP $msg] $msg
        }
    }
    return $result
}
```

The `AddTrace` method determines the type of trace to be added and further delegates the work onward to a trace type specific method.

```
<<service methods>>=
method AddTrace {argList} {
    if {[llength $argList] < 2} {
        set msg "wrong # of args, expected: trace add tracetype args"
        throw [list TRACE WRONGARGS $msg] $msg
    }
```

```
    set argList [lassign $argList tracetype]
    switch -exact -- $tracetype {
        path {
            set traceid [my AddPathTrace $argList]
        }
        signal {
            set traceid [my AddSignalTrace $argList]
        }
        property {
            set traceid [my AddPropertyTrace $argList]
        }
        default {
            set msg "unknown trace type, \"$tracetype\", expected:\
                path, signal, or property"
            throw [list TRACE UNKNOWNTYPE $msg] $msg
        }
    }

    return $traceid
}
```

Method for the specific types of traces simply create new tuples in the relvars for **R7**.

```
<<service methods>>=
method AddPathTrace {argList} {
    if {[llength $argList] != 2} {
        set msg "wrong # of args, expected: trace add path\
            pathpattern cmdprefix"
        throw [list TRACE WRONGARGS $msg] $msg
    }
    lassign $argList pathpattern cmdprefix
    my variable traceNumber
    incr traceNumber

    relvar eval {
        relvar insert Trace [list\
            TraceId $traceNumber\
            PathMatch $pathpattern\
            CmdPrefix $cmdprefix\
        ]
        relvar insert PathTrace [list\
            TraceId $traceNumber\
        ]
    }

    return $traceNumber
}
```

```
<<service methods>>=
method AddSignalTrace {argList} {
    if {[llength $argList] != 4} {
        set msg "wrong # of args, expected:\
            trace add signal interface signal pathpattern cmdprefix"
        throw [list TRACE WRONGARGS $msg] $msg
    }
    lassign $argList interface signal pathpattern cmdprefix
    my variable traceNumber
    incr traceNumber
    relvar eval {
        relvar insert Trace [list\
            TraceId $traceNumber\
```

```
            PathMatch $pathpattern\
            CmdPrefix $cmdprefix\
        ]
        relvar insert SignalTrace [list\
            TraceId $traceNumber\
            Interface $interface\
            Signal $signal\
        ]
    }
    my variable connId
    $connId listen {} $interface.$signal [mymethod HandleSignals]

    return $traceNumber
}
```

```
<<service methods>>=
method AddPropertyTrace {argList} {
    if {[llength $argList] != 4} {
        set msg "wrong # of args, expected:\
            trace add signal interface property pathpattern cmdprefix"
        throw [list TRACE WRONGARGS $msg] $msg
    }
    lassign $argList interface property pathpattern cmdprefix
    my variable traceNumber
    incr traceNumber
    relvar eval {
        relvar insert Trace [list\
            TraceId $traceNumber\
            PathMatch $pathpattern\
            CmdPrefix $cmdprefix\
        ]
        relvar insert PropertyTrace [list\
            TraceId $traceNumber\
            Interface $interface\
            Property $property\
        ]
    }

    return $traceNumber
}
```

Removing a trace is somewhat easier since we can simply remove the generalization instances from **R7**.

```
<<service methods>>=
method RmTrace {traceid} {
    set trace [pipe {
        relvar restrictone Trace TraceId $traceid |
        rvajoin ~ [relvar set PathTrace] Path |
        rvajoin ~ [relvar set SignalTrace] Signal |
        rvajoin ~ [relvar set PropertyTrace] Property
    }]
    if {[relation isnotempty $trace]} {
        relation assign $trace

        variable connId
        relvar eval {
            relvar deleteone Trace TraceId $traceid
            if {[relation isnotempty $Path]} {
                relvar deleteone PathTrace TraceId $traceid
            } elseif {[relation isnotempty $Signal]} {
                relation assign $Signal Interface Signal
                $connId listen {} $Interface.$Signal {} ;          # ❶
```

```
                relvar deleteone SignalTrace TraceId $traceid
            } elseif {[relation isnotempty $Property]} {
                relvar deleteone PropertyTrace TraceId $traceid
            }
        }
    }
    return
}
```

❶ Remove any bus message listening associated with the signal.

Finally, introspection on traces is also just querying the **R7** generalization and returning the attribute values.

```
<<service methods>>=
method InfoTrace {traceid} {
    set trace [pipe {
        relvar restrictone Trace TraceId $traceid |
        rvajoin ~ [relvar set PathTrace] Path |
        rvajoin ~ [relvar set SignalTrace] Signal |
        rvajoin ~ [relvar set PropertyTrace] Property
    }]

    if {[relation isnotempty $trace]} {
        relation assign $trace
        if {[relation isnotempty $Path]} {
            set result [pipe {
                relation project $trace PathMatch CmdPrefix Path |
                relation ungroup ~ Path
            }]
            set tracetype path
        } elseif {[relation isnotempty $Signal]} {
            set result [pipe {
                relation project $trace PathMatch CmdPrefix Signal |
                relation ungroup ~ Signal
            }]
            set tracetype signal
        } elseif {[relation isnotempty $Property]} {
            set result [pipe {
                relation project $trace PathMatch CmdPrefix Property |
                relation ungroup ~ Property
            }]
            set tracetype property
        }
        log::debug "Info Trace result:\n[relformat $result]"
        set result [lindex [relation body $result] 0]
        lappend result Type $tracetype
        return $result
    } else {
        set msg "unknown trace, \"$traceid\""
        throw [list INFO UNKNOWNTRACE $msg] $msg
    }
}
```

Synchronizing to changes

In certain coding situations, *e.g.* sequencing tests, it is convenient to keep the code path sequential. This requires performing operations and then entering the event loop so that asynchronous updates from the DBus can arrive.

The general scheme is:

1. Add a trace for the entity of interest.
2. Execute a command that triggers some asynchronous response.
3. Enter the Tcl event loop to wait for the response or potentially a timeout.
4. Remove the trace.

This set of actions has been factored into methods to ease the tedium of repeatedly coding the sequence. There are separate methods for properties, paths and signals. Note that these methods only support waiting for a single trace. More complicated logical combinations will have to be coded.

```
<<service methods>>=
method waitForProperty {interface property pathpattern trigger {timeout 5000}} {
    set timerid {}
    if {$timeout != 0} {
        set timerid [::after $timeout [mymethod PropertySyncTimeout]]
    }
    set traceId [my trace add property $interface $property $pathpattern\
            [mymethod PropertySync]]
    try {
        set result [uplevel 1 $trigger]

        vwait [my varname propSync]
        if {$timerid ne {}} {
            ::after cancel $timerid
        }
    } on error {result opts} {
        return -options $opts $result
    } finally {
        my trace remove $traceId
    }

    variable propSync
    if {$propSync eq "TIMEOUT"} {
        set msg "timed out waiting for property change: $interface $property\
                $pathpattern $trigger"
        throw [list PROPERTY TIMEOUT $msg] $msg
    }
    return $propSync
}
```

The `PropertySync` method is invoked by the property trace and stores a result dictionary into the variable used for synchronization.

```
<<service methods>>=
method PropertySync {status path interface property value} {
    set [my varname propSync] [dict create\
        status $status\
        path $path\
        interface $interface\
        property $property\
        value $value\
    ]
    return
}
```

Usually there is a timeout placed on waiting. The `PropertySyncTimeout` method handles the timeout case.

```
<<service methods>>=
method PropertySyncTimeout {} {
    set [my varname propSync] TIMEOUT
```

```
    return
}
```

Waiting for signals or paths follows the same general pattern. The differences are mainly in how the interactions are specified.

```
<<service methods>>=
method waitForSignal {interface signal pathpattern trigger {timeout 5000}} {
    set timerid {}
    if {$timeout != 0} {
        set timerid [::after $timeout [mymethod SignalSyncTimeout]]
    }
    set traceId [my trace add signal $interface $signal $pathpattern\
            [mymethod SignalSync]]
    try {
        set result [uplevel 1 $trigger]

        vwait [my varname sigSync]
        if {$timerid ne {}} {
            ::after cancel $timerid
        }
    } on error {result opts} {
        return -options $opts $result
    } finally {
        my trace remove $traceId
    }

    variable sigSync
    if {$sigSync eq "TIMEOUT"} {
        set msg "timed out waiting for signal change: $interface $signal\
                $pathpattern $trigger"
        throw [list SIGNAL TIMEOUT $msg] $msg
    }
    return $sigSync
}
```

```
<<service tests>>=
test signal-1.0 {
    Wait for a signal
} -setup {
    set conn [Connection create sessionBus session]
    Service create tserver com.modelrealization.test $conn
} -cleanup {
    sessionBus destroy
} -body {
    set chng [tserver waitForSignal com.modelrealization.test Attention\
        /com/modelrealization/test\
        {tserver call /com/modelrealization/test\
            com.modelrealization.test Trigger}]
    dict get $chng args
} -result {100 {Model Realization}}
```

```
<<service tests>>=
test signal-2.0 {
    Wait for a signal -- timeout
} -setup {
    set conn [Connection create sessionBus session]
    Service create tserver com.modelrealization.test $conn
} -cleanup {
    sessionBus destroy
} -body {
    tserver waitForSignal com.modelrealization.test Attention\
```

```
        /com/modelrealization/test {} 500
} -result {timed out waiting for signal change:*}\
    -match glob -returnCodes error
```

```
<<service methods>>=
method SignalSync {eventInfo args} {
    set [my varname sigSync] [dict create\
        path [dict get $eventInfo path]\
        interface [dict get $eventInfo interface]\
        signal [dict get $eventInfo member]\
        sender [dict get $eventInfo sender]\
        signature [dict get $eventInfo signature]\
        args $args\
    ]
    return
}
```

```
<<service methods>>=
method SignalSyncTimeout {} {
    set [my varname sigSync] TIMEOUT
    return
}
```

Path tracing is the simplest. Here we are only concerned about a patter to match a path name.

```
<<service methods>>=
method waitForPath {pathpattern trigger {timeout 5000}} {
    set timerid {}
    if {$timeout != 0} {
        set timerid [::after $timeout [mymethod PathSyncTimeout]]
    }
    set traceId [my trace add path $pathpattern [mymethod PathSync]]
    set result [uplevel 1 $trigger]

    vwait [my varname pathSync]
    if {$timerid ne {}} {
        ::after cancel $timerid
    }

    my trace remove $traceId

    variable pathSync
    if {$pathSync eq "TIMEOUT"} {
        set msg "timed out waiting for path change"
        throw [list PROPERTY TIMEOUT $msg] $msg
    }
    return $pathSync
}
```

```
<<service methods>>=
method PathSync {status path} {
    set [my varname pathSync] [dict create\
        status $status\
        path $path\
    ]
    return
}
```

```
<<service methods>>=
method PathSyncTimeout {} {
```

```
    set [my varname pathSync] TIMEOUT
    return
}
```

Callbacks for signals

The methods in this section are helpers to deal with various callbacks that come up from the DBus. These callbacks are used to find and evaluate any registered traces.

All signal callbacks result in invoking the `HandleSignals` method.

```
<<service methods>>=
method HandleSignals {eventInfo args} {
    log::debug [info level 0]
    set interface [dict get $eventInfo interface]
    set signal [dict get $eventInfo member]
    set path [dict get $eventInfo path]

    my EvalTrace signal {$Interface eq $interface &&\
            $Signal eq $signal &&\
            [string match $PathMatch $path]} $eventInfo {*}$args
}
```

Traces are looked up and evaluated in the `EvalTrace` method. Here we see the logic of executing traces from youngest to oldest and the logic associated with handling errors and early termination of a chain of traces.

```
<<service methods>>=
method EvalTrace {tracetype predicate args} {
    set typeMap [dict create\
        path PathTrace\
        signal SignalTrace\
        property PropertyTrace\
    ]
    set trace [pipe {
        relvar set Trace |
        relation join ~ [relvar set [dict get $typeMap $tracetype]]
    }]
    set matches [uplevel 1 [list ::ral relation restrictwith $trace $predicate]]

    log::debug "$tracetype traces:\n[relformat $matches MatchingTrace]"

    relation foreach trace $matches -descending TraceId {
        relation assign $trace CmdPrefix TraceId
        set cmd [list {*}$CmdPrefix {*}$args]

        try {
            eval $cmd
        } on continue {} {
            continue
        } on break {} {
            break
        } on error {result opts} {
            log::error $::errorInfo
            log::error "error executing $tracetype trace command,\
                    \"$cmd\": $result"
            my RmTrace $TraceId
            break
        }
    }
}
```

```
<<service methods>>=
method TracePropertyValue {status path interface property args} {
    my EvalTrace property {$Interface eq $interface &&\
            $Property eq $property && [string match $PathMatch $path]}\
            $status $path $interface $property {*}$args
}
```

```
<<service methods>>=
method TracePath {status path} {
    my EvalTrace path {[string match $PathMatch $path]} $status $path
}
```

The `InterfacesAdded` method is specifically used for the callback resulting from a signal from the org.freedesktop.DBus.ObjectMan
interface.

```
<<service methods>>=
method InterfacesAdded {eventInfo path interfaces} {
    relvar eval {
        dict for {interface properties} $interfaces {
            set added [my AddNewInterface $interface $path]
            if {!$added} {
                # At this point we failed to find the interface
                # in the introspection XML. The best we can do
                # is just add it with the properties we have
                # and just ignore any methods or signals.
                relvar insert Interface [list Name $interface]
                foreach propName [dict keys $properties] {
                    relvar insert Member [list\
                        Interface $interface\
                        Name $propName\
                    ]
                    relvar insert Property [list\
                        Interface $interface\
                        Name $propName\
                        Type {}\
                        Access read\
                    ]
                }
            }

            dict for {propName propValue} $properties {
                relvar insert PropertyValue [list\
                    Path $path\
                    Interface $interface\
                    Property $propName\
                    Value $propValue\
                    Valid true\
                ]
                my TracePropertyValue changed $path $interface $propName\
                        $propValue
            }
        }
        my AddNewPath $path [dict keys $interfaces]
    }

    return
}
```

```
<<service methods>>=
method AddNewInterface {interface path} {
```

```
    set intf [relvar restrictone Interface Name $interface]
    if {[relation isempty $intf]} {
        my variable connId svcName

        set svcxml [$connId call $svcName $path\
                org.freedesktop.DBus.Introspectable Introspect {}]
        log::debug "new interface xml = \"$svcxml\""
        set svcdoc [dom parse -simple $svcxml]
        set svcroot [$svcdoc documentElement]
        set intfNode [$svcroot selectNodes {/node/interface[@name=$interface]}]
        log::debug "intfNode = $intfNode"
        if {$intfNode eq {}} {
            log::notice "failed to find interface, \"$interface\",\
                in introspection XML for path, \"$path\""
            return false
        }

        try {
            set intf [relvar insert Interface [list Name $interface]]
            my CreateMethods $intfNode
            my CreateProperties $intfNode
            my CreateSignals $intfNode
        } finally {
            $svcdoc delete
        }
    }
    return true
}
```

```
<<service methods>>=
method AddNewPath {path interfaces} {
    foreach interface $interfaces {
        relvar uinsert Implementation [list Path $path Interface $interface]
    }

    set pathRel [relvar restrictone Path Name $path]
    if {[relation isempty $pathRel]} {
        relvar insert Path [list Name $path]
        my TracePath added $path
    }
}
```

Again the org.freedesktop.DBus.ObjectManager interface has a specified signal to indicate when interfaces have been removed.

```
<<service methods>>=
method InterfacesRemoved {eventInfo path interfaces} {
    relvar eval {
        foreach interface $interfaces {
            relvar deleteone Implementation Path $path Interface $interface
            relvar delete PropertyValue pvTup {
                [tuple extract $pvTup Path] eq $path &&\
                [tuple extract $pvTup Interface] eq $interface}
        }
        set noImplRemain [pipe {
            relvar set Implementation |
            relation restrictwith ~ {$Path eq $path} |
            relation isempty
        }]
        if {$noImplRemain} {
            relvar deleteone Path Name $path
            my TracePath removed $path
        }
```

```
    }
}
```

Another common interface is org.freedesktop.DBus.Properties. It also has a signal to indicate that a property on an interface has either changed value or is now invalid.

```
<<service methods>>=
method PropertiesChanged {eventInfo intfName changed invalidated} {
    set path [dict get $eventInfo path]
    dict for {propName propValue} $changed {
        my UpdatePropertyValue $path $intfName $propName $propValue
        my TracePropertyValue changed $path $intfName $propName $propValue
    }
    foreach invalProp $invalidated {
        my InvalidatePropertyValue $path $intfName $invalProp
        my TracePropertyValue invalidated $path $intfName $invalProp
    }
}
```

Package logging

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

The following incantation sets up the logger to use color to output log messages if we are connected to a terminal and plain text otherwise. Then the logger commands are imported into a child namespace for convenience.

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

Code Organization

In this section, we show how the literate program chunks defined previously are composed into the various source code files.

Package source code

The package source code is delivered in a file named, *dbusclient.tcl*. The following chunk is a root chunk for extracting the package source code.

```
<<dbusclient.tcl>>=
<<edit warning>>
<<copyright info>>

<<required packages>>

namespace eval ::dbusclient {
    <<logger setup>>

    <<package exports>>
```

```
    namespace ensemble create
}

<<connection class>>
<<service class>>

package provide dbusclient 1.0
```

Unit tests

Part of the literate program documentation is to show the test cases for the code. The package uses `tcltest` to run and tally the tests. In the document, we have included the test cases near to the code they test. The package tests are delivered in a file name, *dbusclient.test*. The following chunk is a root chunk for extracting the package unit tests.

```
<<dbusclient.test>>=
<<edit warning>>
<<copyright info>>

<<required packages for test>>

# Add custom arguments here.
set optlist {
    {level.arg warn {Logging level}}
}
array set options [::cmdline::getKnownOptions argv $optlist]
tcltest::configure {*}$argv

logger::setlevel $options(level)

source ../code/dbusclient.tcl

namespace eval ::dbusclient::test {
    <<logger setup>>

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

    namespace import ::tcltest::*
    namespace import ::dbusclient::*

    <<test utilities>>

    <<test server setup>>

    <<connection tests>>
    <<service tests>>

    <<test server cleanup>>

    cleanupTests
}
```

We collect all the additional packages required for the tests into one place.

```
<<required packages for test>>=
<<required packages>>
package require tcltest
package require cmdline
```

Reference documentation

The manpage or reference documentation has been included in the literate program source as comments. This allows the documentation to be near the code it describes, but by placing it in a comment, it does not clutter the text itself.

The manpages are formated with Tcl doctools that can be found in `tcllib` and can be transformed into HTML (among other forms) using `dtplite`. The document is place in a file named, *dbuclient.man*. The following chunk is a root chunk for extracting the package manpage document.

```
<<dbusclient.man>>=
[comment {
<<edit warning>>
<<copyright info>>
}]
[vset version 1.0]
[manpage_begin dbusclient n [vset version]]
[comment {
#
#*++
# PROJECT:
#   dbusclient
#
# MODULE:
#   dbusclient -- manual documentation for the dbusclient package
#
# ABSTRACT:
#
#*--
#
}]
[moddesc "DBus Client Commands"]
[titledesc "Client side DBus interactions"]
[require Tcl 8.6]
[require dbusclient [opt [vset version]]]
<<man description>>
<<man commands section>>
[list_begin definitions]
<<man command synopsis>>
[list_end]

[subsection "Connection object methods"]
The following methods are available for [class Connection] objects.

[list_begin definitions]
<<man connection methods>>
[list_end]

[subsection "Service object methods"]
The following methods are available for [class Service] objects.

[list_begin definitions]
<<man service methods>>
[list_end]

[section Dependencies]

The [package dbusclient] package depends upon three binary Tcl packages.

[list_begin itemized]
[item] [package dbus] is the Tcl bindings to the DBus library.
[item] [package tcom] is used to parse XML returned during introspection.
[item] [package ral] is used to manage the introspection data.
```

```
[list_end]

[keywords DBus]

[manpage_end]
```

Copyright Information

The following is copyright and licensing information.

```
<<copyright info>>=
# This software is copyrighted 2019 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.
```

Edit Warning

We want to make sure to warn readers that the source code is extracted and editing the extracted file breaks the correspondence with the ultimate source, the literate program source.

```
<<edit warning>>=
#
# DO NOT EDIT THIS FILE!
```

```
# THIS FILE IS AUTOMATICALLY EXTRACTED FROM A LITERATE PROGRAM SOURCE FILE.
#
```

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 extraction 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 definitions of 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 chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. However, there is much more information in the literate program document than in the tangled source. You can, of course, examine the tangled source output, but if you read the literate 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 a language compiler or interpreter.

Index