stsatcl - An Single Threaded Software Architecture for Tcl

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	July 6, 2013	Initial coding.	GAM
1.0	August 11, 2013	Finished code for Revision 1.0.	GAM

Contents

List of Figures

Abstract

The `stsatcl` package is a software execution architecture intended as the target for translating Executable UML (XUML) domain models into a running program using Tcl as the implementation language. The package is build upon a TclOO foundation and will on casual inspection appear as a higher order object system. Classes may be defined with attributes, relationships and state models. The execution semantics of XUML are provided in a form to make translation of a model onto `stsatcl` a direct and understandable operation. State machine event dispatch and state machine event tracing are provided to support implementing and testing the dynamic aspects of an XUML model. The documentation includes a complete running example showing how the process of translating the model to a running implementation happens.

Introduction

By analogy to STSA, which is a software architecture for micro-controller based implementations that are coded in "C", the **stsatcl** package is a single threaded software architecture for Tcl based applications.

The concepts here are similar to those used by `STSA`. Specifically we are defining an implementation based scheme that supports the execution model of XUML and is to be used as the target for translating XUML models using Tcl as an implementation language.

There are, of course, many differences in the way a "C" based architecture and a Tcl based one will handle the implementation aspects. For example, STSA uses pointers to traverse relationships. There are no such things in Tcl. However, command names of TclOO objects can be used to serve the same purpose. In STSA, "C" data types must be given for attributes. In the Tcl world, everything is a string, Tcl variables can hold data of any type and type annotation is unnecessary. State models in Tcl have already been done in many ways. Here, we will follow the pattern of the `oomoore`, package and indeed a large amount of that code is reused in this package.

In addition to the explanation and code for the `stsatcl` package, we will also include an example and endeavor to show how an XUML model is translated into a Tcl implementation using this package. The example will, of necessity, be small and somewhat contrived. The use of small, contrived examples is always unfortunate but this document is large enough with just the package and its test cases that any *real-world* example would over burden the entire undertaking. Realistic models and their translation easily become books in themselves.

Reading This Document

There are several different topics threaded together in this document. We cover the Tcl code for the software architecture and it test cases as well as dicussion of the design decisions that make up the architecture. There is also an example that uses the package to implement an XUML model. The reader should *not* feel compelled to read this document from beginning to end in order. Skipping around is encouraged. It is hoped that the table of contents, index and hyperlinks within the article will help finding the section of interest.

Design Concepts

The main design consideration for a XUML software architecture is how to map XUML execution rules onto the implementation language. That mapping is not unique. We could build a Tcl-based XUML architecture strictly from data structures and procedures or as namespace ensembles or with other implementation technology that is available in Tcl. We have chose to base the `stsatcl` package on TclOO.

The translation of XUML model classes to TclOO classes is, to a first approximation, one-to-one. To support the XUML execution model, we will create a meta-class called, `STSAClass`. Instances of `STSAClass` will correspond to the XUML classes of the implementation. The constructor of `STSAClass` will allow for the appropriate configuration of the resulting XUML implemenation class to be specified. Instances can then be created to generate a population of the model. This is shown in the diagram below.

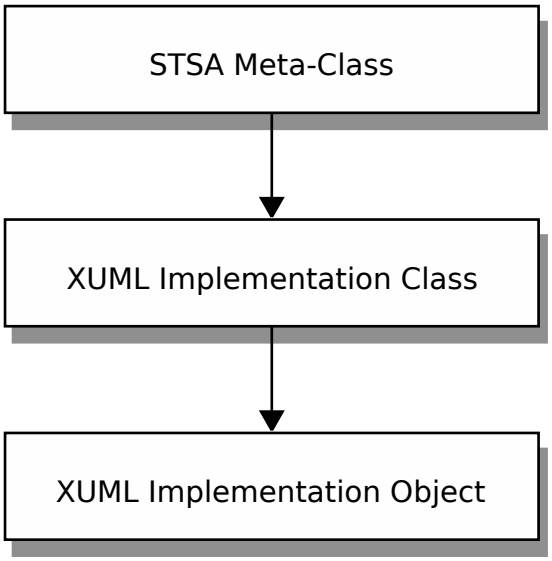

Figure 1: STSA Tcl Block Diagram

This package uses the common technique of defining a small domain specific language (DSL) that is used in the construction of objects to specify the required properties. In this case, we are constructing classes that have XUML qualities, namely:

- Attributes.
- Relationship linkage.
- State models.
- Common processing.

Document Conventions

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk defintion 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 chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to the Tcl interpreter.

We will actually include root chunks for the following items:

- Tcl source for the package.
- Test cases that run under `tcltest`.
- The `pkgIndex.tcl` package index file.
- A runnable example.

Since we also include test cases in this document, the usual order will be to show a **method** or **proc** and follow it by the tests that exercise the method. From a development and maintenance point of view, is is beneficial to keep the code and tests lexically near each other. However, you may wish to skip over the testing to keep the narrative flow more consistent.

Example Overview

In this section we start the discussion of the example that is used to illustrate the usage of `stsatcl`. Unfortunately, there is rather a lot of background material that we will not cover here. We do not explain how to create an XUML model nor will we spend much time explaining why the example model was designed the way it was. There are many good books that explain XUML in detail and will teach you the basics of modeling[1]. We suggest you read at least one of them. It is also the case that this package represents one particular piece of a larger workflow and this means that readers who may not be completely familar with model-driven translation oriented development may have many questions that remain.

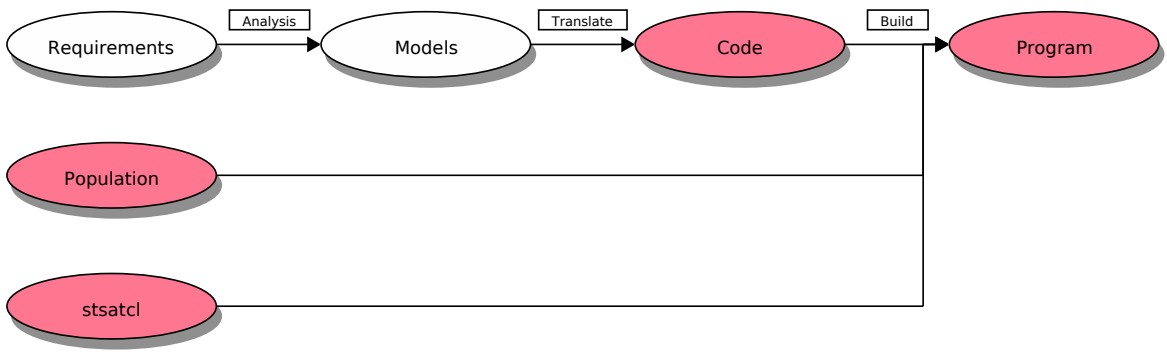

Figure 2: Simplified Translation Workflow

In the above figure, we will cover those portions that are colored. Specifically, we will not deal with how one creates models from Requirements. We will start with the Model and show how that becomes Code via translation. By combining the code with a population and the `stsatcl` package, we will produce a running example program.

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

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

Class Diagram

The figure below shows a class diagram for the washing machine control domain in UML graphical notation.

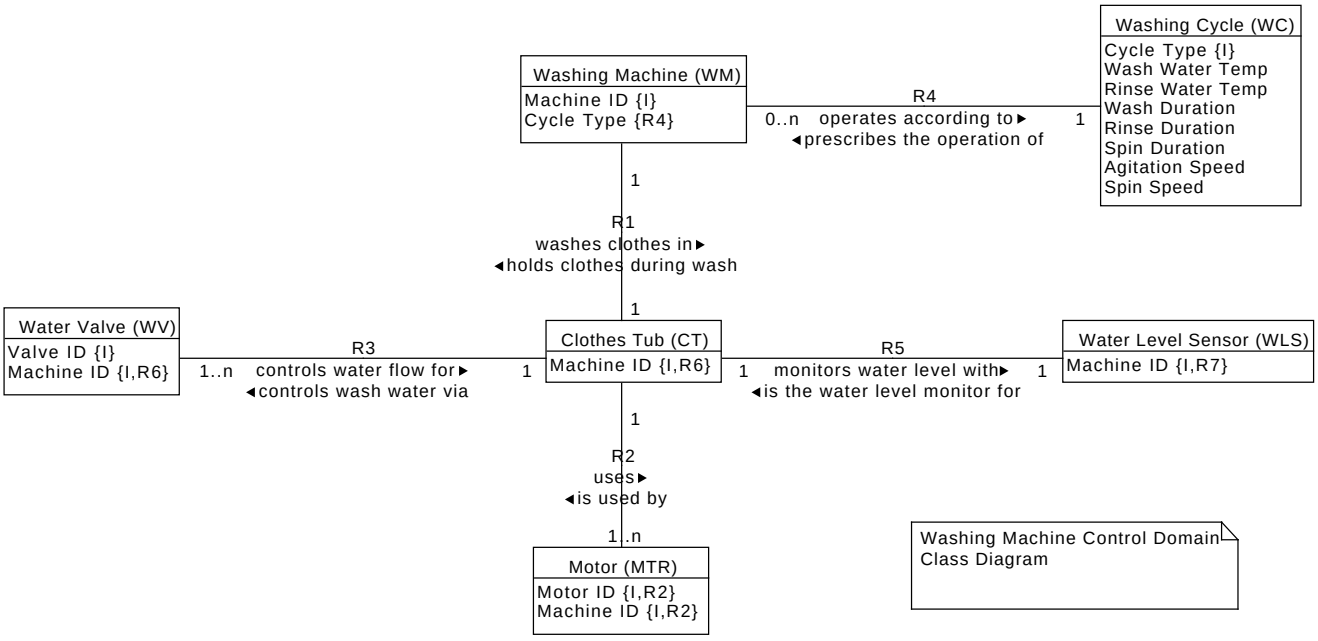

Figure 3: Washing Machine Class Diagram

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

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

The class diagram shows the static aspects of our domain and is always the first aspect of the model that must be considered when deriving the implementation. The classes hold the parameters of the domain and the relationships state how the components are associated with respect to each other. The class diagram facet of the model is static in the sense that at any point in time, the statements you can infer from the diagram will be true. Later, we visit the dynamic and algorithmic facets of the domain but, for now, we will endeavor to represent the classes and relationships in terms of the `stsatcl` package constructs. Of course, first we must explain what those constructs are and we begin that in a section below. First we get a discussion of error reporting out of the way.

Error Reporting

Often any discussion of errors is pushed to the end as if it were unimportant or an afterthought (which, sadly, in many cases it is). Here we bring error reporting up front to emphasize its importance.

We will consistently use the `throw` command to produce errors in an effort to insure that reasonable error codes are produced. Good error codes are key to programatic recovery from errors. The format of the error code will be a list of the following elements:

1. The package name in upper case, *i.e.* `STSATCL`.
2. An error code word. The error code word is a simple string in upper case and they are listed in the index.
3. Parameters that are dependent upon the error code word.
4. A human readable error message.

Although the size of the error code list varies since the number of parameters for the error code word (item 3 above) varies, the first two elements and the last element have easily computed locations.

We factor into common code the manipulations required to `throw` errors.

```
<<stsatcl commands>>=
proc ::stsatcl::DeclError {errcode args} {
    variable errFormats
    set errmsg [format [dict get $errFormats $errcode] {*}$args]
    tailcall throw [list STSACLASS $errcode {*}$args $errmsg] $errmsg
}
```

The mapping of `errcode` values to format strings is held as package data in the namespace of the `stsatcl` package.

```
<<stsatcl data>>=
variable errFormats
set errFormats [dict create {*}{
    <<error code formats>>
}]
```

It is also important to realize that programs built upon this package will perform most of their operations in event callbacks. This means that the majority of errors encountered during run time, for example during a state activity, will happen as a background error. It will be important to install a background error handler[2]. Attention to the error codes thrown by this package can make coding a background error handler much easier but the consequence of errors must be determined by the application.

STSA Class

The **stsatcl** package defines the `STSAClass` class. `STSAClass` is a meta-class, *i.e.* it is a class for defining other classes.

For each XUML class in the class diagram, we will create an implementation counterpart that is an instance of `STSAClass`. Since instances of `STSAClass` are themselves classes, we will then be able to create object instances of the XUML implementation class created by `STSAClass`. The XUML implemenation classes are all different because we will configure them differently when they are constructed.

```
<<stsatcl commands>>=
::oo::class create ::stsatcl::STSAClass {
    superclass ::oo::class      ; # ❶
    unexport new                ; # ❷
    <<stsaclass configuration>>
}

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

[2] See `bgerror` and `interp bgerror` commands

❶ Deriving from `::oo::class` makes us a meta-class.

❷ We insist that all STSA classes be named commands.

Constructor

The constructor of the `STSAClass` class takes a script that consists of configuration commands. Those commands will validate and store away the configuration information. Using the configuation information, a new class is created with the properties described in the construction configuration script. Objects created from the class then have all the characteristics specified in the configuration script.

```
<<stsaclass configuration>>=
constructor {{config {}}} {
    ::oo::objdefine [self] export new ; # ❶
    <<stsaclass constructor>>
}
```

❶ We now want the `new` method to be available to create object instances of the resulting XUML class.

Configuration DSL

The configuration language is really just a series of commands that are implemented as methods of the meta-class. The design uses the `my eval` command to evaluate the configuration script in the context of the class being created. The commands in the configuration script need to resolve correctly in that context. We would like *not* to be forced to prefix the DSL commands with the `my` command as that clutters the configuration script. The `oo::util` package in `tcllib` provides the `link` command to make the DSL command definitions easier.

```
<<required packages>>=
package require oo::util
# The mixin of "oo::class.Delegate", interacts badly with meta-classes that
# have constructor arguments. So we eliminate the oo::class mixins here. This
# will mean that you can't define class methods.
::oo::define oo::class self mixin
```

```
<<stsaclass constructor>>=
link\
    {attribute Attribute}\
    {reference Reference}\
    {partition Partition}\
    {instop InstOp}\
    {classop ClassOp}\
    {constructor Constructor}\
    {destructor Destructor}\
    {polymorphic Polymorphic}\
    {statemodel Statemodel}
```

The `link` command arranges, for example, that invoking `attribute` in the namespace of the class being created actually invokes the `Attribute` method of the `STSAClass` class. The trickery to accomplish can be found in the source to the `oo::util` package. We follow the naming convention that the methods implementing the DSL commands are unexported and the corresponding linked names begin with lower case.

So there will be nine commands in the configuration DSL. The `statemodel` command will itself take another configuration script to describe states and transitions. The state model configuration commands will be described in a section of their own. Each of the top level commands is described in a section below and the method that implements the command is given.

The methods that implement the DSL commands follow the same general pattern. They store the data given in the command arguments, possibly with some validation, into data variable associated with the created class. That data is then available to the class methods to implement the semantics of XUML attribute access, relationship navigation and state machine dispatch. The

design approach is data-driven with the DSL commands supplying the values of the data that determine the specific semantics that occur during execution.

We will encounter situations where we wish to insure that collections form a mathematical *set* (*i.e.* there are no duplicated elements). Again, `tcllib` can supply the required package.

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

Attribute Method

The `attribute` command is used to define the attributes of a XUML class. It is invoked as:

```
attribute attr-value-pair1 attr-value-pair2 ...
```

`attr-value-pair ...`
: Each `attr-value-pair` argument is treated as a one or two element list. The first element gives the name of the attribute. If the second element of the pair is present, then it gives the default value for the attribute. If the second element is missing then the default value for the attribute is the empty string. Default attributes values will be used when objects of the class are created if no other values is supplied at construction time.

The command may be invoked an arbitary number of times in the configuration script. When objects of the resulting class are created, the attributes will be present and have their default values after construction. The constructor constructor of the newly created class will also allows us to override the defaults at creation time.

Attribute names form a set and so duplicates are not allowed. We also do not allow the empty string as an attribute name on the rational that the empty string cannot convey any semantics. Attempts to define the empty string as an attribute name or to define a duplicate attribute name cause an error to be thrown.

Attribute names will be stored away in a dictionary. The keys to the dictionary will be the attribute names themselves and the value of the corresponding key will be the default value of the attribute.

```
<<stsaclass constructor>>=
my variable attrInfo
set attrInfo [dict create]
```

The implementation of the `Attribute` method consists of an interation over the arguments, validating the attribute names and assigning the default values into the `attrInfo` dictionary.

```
<<stsaclass configuration>>=
method Attribute {args} {
    my variable attrInfo

    foreach attr $args {
        lassign $attr name dfltvalue
        if {$name eq {}} {
            tailcall ::stsatcl::DeclError BADATTRIBUTENAME $name
        }
        if {[dict exists $attrInfo $name]} {
            tailcall ::stsatcl::DeclError DUPATTRIBUTENAME $name
        } else {
            dict set attrInfo $name $dfltvalue
        }
    }
}
```

```
<<error code formats>>=
BADATTRIBUTENAME        {invalid attribute name, "%s"}
DUPATTRIBUTENAME        {duplicate attribute name, "%s"}
```

Attribute Method Tests

We can test the `Attribute` method in isolation, but this requires a bit of namespace trickery to invoke the unexported method. However, in Tcl nothing is really hidden.

```
<<meta constructor tests>>=
test Attribute-1.0 {
    configure multiple attributes
} -setup {
    ::stsatcl STSAClass create at1
} -cleanup {
    at1 destroy
} -body {
    [info object namespace at1]::my Attribute\
        Count {Maximum 20}
    dict size [set [info object namespace at1]::attrInfo]
} -result {2}
```

```
<<meta constructor tests>>=
test Attribute-2.0 {
    attempt to name an attribute the empty string
} -setup {
    ::stsatcl STSAClass create at2
} -cleanup {
    at2 destroy
} -body {
    [info object namespace at2]::my Attribute {}
} -result {invalid attribute name, ""} -returnCodes error
```

```
<<meta constructor tests>>=
test Attribute-3.0 {
    attempt to have duplicate attributes
} -setup {
    ::stsatcl STSAClass create at3
} -cleanup {
    at3 destroy
} -body {
    [info object namespace at3]::my Attribute Count Count
} -result {duplicate attribute name, "Count"} -returnCodes error
```

Attribute Example

We now show how the `attribute` command in the DSL is used. In this section we will define the attributes of the Washing Cycle class. These attributes specify the parameters of the washing that we can control and change.

```
<<WC class>>=
STSAClass create WashingCycle {
    attribute CycleType
    attribute WashWaterTemp RinseWaterTemp
    attribute WashDuration RinseDuration SpinDuration
    attribute AgitationSpeed SpinSpeed
}
```

A few items to note.

- We could have defined all the attributes in one invocation of `attribute`. We have chosen to group them into functional categories.
- We haven't yet said what the attributes mean or control, however, the names should give a good indication of the semantic intent.

- We haven't specified any default values and, consequently, will have to specify values when instances are created.
- We haven't specified the valid value range of the attributes or the physical units of the values.

So we know how to configure attributes into a class but there is still more work to be done. For now, we move on to the class relationships.

Reference Method

In XUML, a relationship between two classes is bi-directional.

The relationship defines a function (or partial function) between the two instance sets of the classes and the idea is firmly grounded in referential integrity ideas from the Relational Model of Data.

In this architecture, a relationship is implemented by decomposing it into *links*. A link is uni-directional and is included in a class if any of the activities of the class requires *navigating* the link to obtain instances of the related class.

XUML relationships also carry the notion of multiplicity and conditionality. So a relationship may navigate to multiple instances in one direction and may also allow for the case that no instances are linked. We need a way to specify whether a link reference is *singular* or *multiple* and whether it is *conditional*.

The `reference` command is invoked in the XUML class configuration script to define a linkage where one class refers to another. The invocation synopsis is:

```
reference rname spec dstclass
```

`rname`
: is the name of the reference. Conventionally, XUML references are named with a capital "R" followed by a number (*e.g.* `R42`), but any string may be used.

`spec`
: is a string that designates the multiplicity and conditionality of the reference. It consists of a hyphen (-) followed by one or two greater than signs (>) followed by an optional letter "c". A *spec* argument of "->" indicates the multiplicity of the reference is singular. A *spec* of "->>" indicates a multiplicity of *many*. Either type of multiplicity may have a "c" suffix (*e.g.* "->>c") to indicate that the reference may refer to zero instances.

`dstclass`
: is the name of a class command to which instances refer. The class need not exist at the time the `reference` command is executed. Unqualified class commands are resolved into the namespace of the creator of the class.

In this architecture, linkage is achieved by storing the command name of the object. We will make sure to store the fully qualified command name even if we take unqualified names as *dstclass* arguments. So one aspect of linkage is to perform the object command name qualification.

```
<<stsaclass configuration>>=
method Qualify {cmdname {level 4}} {
    return [expr {[string range $cmdname 0 1] ne "::" ?\
        "[string trimright [uplevel $level namespace current] :]::$cmdname" :\
        $cmdname}] ; # ❶
}
```

❶ This is a bit boiled down as the ternary expression is not the easiest thing in the world to read. The test is to determine if the leading characters of the *cmdname* are "::". If not, we climb up the call stack using `uplevel` to get the current namespace. The default number of levels to climb up is 4 so we will be rather nested when this method is invoked. The `string trimright` will discard the "::" that is returned if we hit the global namespace in our `uplevel` command. Since we are going to tack on another "::" namespace separator anyway, we don't want the case of the global namespace to end up with ":::", which is incorrect syntax. There are probably a number of other ways to code this. If the leading characters of *cmdname* are "::", then we already have a fully qualified name and will just return it.

Just as for attributes, we hold the linkage information in a dictionary. The dictionary key is the name of the linkage and the corresponding value is also a dictionary (yielding a nested dictionary). We will see the structure of the nested portion of the dictionary below.

```
<<stsaclass constructor>>=
my variable linkInfo
set linkInfo [dict create]
```

The implementation of the `Reference` method uses a regular expression to parse the *spec* argument syntax. A dictionary is then created to hold the component parts of the spec.

```
<<stsaclass configuration>>=
method Reference {rname spec dstclass} {
    my variable linkInfo
    if {[dict exists $linkInfo $rname]} {
        tailcall ::stsatcl::DeclError DUPREFERENCE $rname
    }
    set dstclass [my Qualify $dstclass]

    if {![regexp -- {\A-(>{1,2})(c?)\Z} $spec match mult cond]} { # ❶
        tailcall ::stsatcl::DeclError BADREFSPEC $spec
    }

    dict set linkInfo $rname [dict create\
        type reference\
        dest $dstclass\
        mult [string equal $mult ">>"]\
        cond [string equal $cond "c"]\
    ]
}
```

❶ In `regexp` speak: starting at the beginning of the string, find a hyphen followed by 1 or 2 greater than signs followed by an optional "c" character that is the last character of the string. The parentheses surround the subcomponents so that we can capture the matched portions into variables.

```
<<error code formats>>=
DUPREFERENCE        {linkage, "%s", already exists}
BADREFSPEC          {bad reference spec, "%s"}
```

Reference Method Tests

Using the same techniques as for the `Attribute` method, we can test the `Reference` method in isolation. However, we must be careful to give fully qualified class names. Otherwise since the `Qualify` method is not being invoked at the correct level, it will throw an error.

```
<<meta constructor tests>>=
test Reference-1.0 {
    configure a simple reference
} -setup {
    ::stsatcl STSAClass create ref1
} -cleanup {
    ref1 destroy
} -body {
    [info object namespace ref1]::my Reference R1 -> [namespace current]::foo
    dict get [set [info object namespace ref1]::linkInfo] R1
} -result {type reference dest ::stsatcl::test::foo mult 0 cond 0}
```

```
<<meta constructor tests>>=
test Reference-2.0 {
    define reference with bad syntax
} -setup {
    ::stsatcl STSAClass create ref2
} -cleanup {
    ref2 destroy
} -body {
    [info object namespace ref2]::my Reference R1 ->>x [namespace current]::foo
} -result {bad reference spec, "->>x"} -returnCodes error
```

```
<<meta constructor tests>>=
test Reference-3.0 {
    duplicated reference
} -setup {
    ::stsatcl STSAClass create ref3
} -cleanup {
    ref3 destroy
} -body {
    [info object namespace ref3]::my Reference R1 -> [namespace current]::foo
    [info object namespace ref3]::my Reference R1 -> [namespace current]::foo
} -result {linkage, "R1", already exists} -returnCodes error
```

Reference Example

Our example has many cases where references are used and will present only one of them here. Rest assured that all the references will be dealt with fully, as we will ultimately have a running example. To illustate the `reference` command, we will show the Clothes Tub class. Here we only show the `reference` definitions and we defer the state model associated with the Clothes Tub until later.

```
<<CT class>>=
STSAClass create ClothesTub {
    reference R1 -> WashingMachine
    reference R2 ->> Motor
    reference R3 ->> WaterValve
    reference R5 -> WaterLevelSensor
    <<CT state model>>
}
```

In the class diagram, relationship R2 represents the set of Motor instances that are available to a Clothes Tub instance. It is defined as a one-to-many association between Clothes Tub and Motor, respectively, and codifies the physical arrangement of how the mechanics of a Washing Machine were designed to operate[3]. In the implementation, that relationship is decomposed into two links, a one-to-many link from ClothesTub to Motor and a one-to-one link from Motor to Clothes Tub. The above shows the link from the Clothes Tub perspective. A similar set of reasoning can be applied to the other relationships in the class diagram.

Partition Method

There is one other type of linkage that must be supported. In XUML the generalization relationship of ordinary UML is interpreted in terms of a set partition[4]. A set partion of the super class instances implies that each instance of the super class is linked to exactly one instance of a subclass from among all the subclasses that participate in the generalization. You can think of the generalization in this usage as an equivalence relation.

```
partition rname sub1 sub2 ...
```

[3] Real washing machines probably don't use multiple motors. One motor and a simple transmission would make more sense.
[4] *Cf.* conventional UML that usually interprets the general relationship as some type of *inheritance*.

rname
: is the name of the partition. Conventionally, partitions and references are named with a capital "R" followed by a number (*e.g.* `R42`), but any string may be used. A partition and a reference may not have the same name.

sub1 sub2 ...
: are the names of subclasses of the partition. The subclasses do not have to exist when the partition is declared. At least two subclasses must be given.

We will store the partition information in the same `linkInfo` dictionary. Together the references and partitions form the set of linkages for the class. For the case of a partition, the dictionary value will have a different value for the `type` key and the value of the `subclasses` key holds the set of subclasses.

We will also find it convenient to have the set of paritions defined for a class accessible outside of the `linkInfo` dictionary (*i.e.* we don't want to have to search the `linkInfo` dictionary for all the `partition` type links).

```
<<stsaclass constructor>>=
my variable partitions
set partitions [list]
```

The `Partition` method follows the familiar pattern of validating the input and adding an entry to the `linkInfo` dictionary and the `partitions` list.

```
<<stsaclass configuration>>=
method Partition {rname args} {
    my variable linkInfo
    if {[dict exists $linkInfo $rname]} {
        tailcall ::stsatcl::DeclError DUPREFERENCE $rname
    }
    if {[llength $args] < 2} { # ❶
        tailcall ::stsatcl::DeclError PARTITION $rname
    }

    set subs [list]
    foreach sub $args {
        ::struct::set include subs [my Qualify $sub]
    }
    dict set linkInfo $rname [dict create\
        type partition\
        subclasses $subs\
    ]
    my variable partitions
    lappend partitions $rname
}
```

```
<<error code formats>>=
PARTITION       {partition, "%s", must have at least two subclasses}
```

❶ We insists that a partition have at least two subclasses. Although, one can conceive of an improper subset as a partition, it does not carry any interesting semantics (*i.e.* it is a "distinction without a difference") and so is interpreted as a mistake.

Partition Method Tests

`Partition` method testing follows the same pattern as for the `Attribute` and `Reference` methods.

```
<<meta constructor tests>>=
test Partition-1.0 {
    define partition
} -setup {
    ::stsatcl STSAClass create pt1
```

```
} -cleanup {
    pt1 destroy
} -body {
    [info object namespace pt1]::my Partition R1\
            [namespace current]::sub1 [namespace current]::sub2
    dict get [set [info object namespace pt1]::linkInfo] R1
} -result {type partition subclasses {::stsatcl::test::sub1 ::stsatcl::test::sub2}}
```

```
<<meta constructor tests>>=
test Partition-2.0 {
    partition with one subclass
} -setup {
    ::stsatcl STSAClass create pt2
} -cleanup {
    pt2 destroy
} -body {
    [info object namespace pt2]::my Partition R1\
            [namespace current]::sub1
} -result {partition, "R1", must have at least two subclasses}\
-returnCodes error
```

```
<<meta constructor tests>>=
test Partition-3.0 {
    define duplicate partitions
} -setup {
    ::stsatcl STSAClass create pt3
} -cleanup {
    pt3 destroy
} -body {
    [info object namespace pt3]::my Partition R1\
            [namespace current]::sub1 [namespace current]::sub2
    [info object namespace pt3]::my Partition R1\
            [namespace current]::sub3 [namespace current]::sub4
} -result {linkage, "R1", already exists} -returnCodes error
```

Statemodel Method

In XUML, a state model may be associated with a class to describe its dynamic behavior. Not all XUML classes have state behavior. The analysis model will only define state models for classes that have non-trivial and interesting dynamic behavior. We would like to keep the commands that are required to define a state model grouped together. Although not strictly necessary, grouping the state model commands together into a script will help make the semantic distinction between dynamic behavior implied by the state model and the static characteristics specified by the attributes and links.

```
statemodel script
```

`script`
A script invoking commands used to specify a Moore style state model.

We will want to execute `STSAClass` methods to handle the state model specification just as we did for attributes and references. However, we want to insure that the state model configuration script does not execute any other configuration commands (*e.g.* we don't want `attribute` commands imbedded in the state model configuration script). To do this, we will put the state model configuration commands in a namespace that is a child of the class namespace and evaluate the state model configuration script in that namespace.

The bit of trickery required for this is similar to that used in the `link` command. Indeed the code below is a simple variation on the `tcllib` source for the `oo::util link` procedure. Here we define a `linkTo` helper procedure that specifies a child namespace where we wish to evaluate the linked commands.

```
<<helper commands>>=
proc ::oo::Helpers::linkTo {childns args} {   # ❶
    set childns [string trim $childns :]    ; # ❷
    namespace eval $childns {}              ; # ❸
    set ns [uplevel 1 {namespace current}]
    foreach link $args {
        if {[llength $link] == 2} {
            lassign $link src dst
        } else {
            lassign $link src
            set dst $src
        }
        interp alias {} ${ns}::${childns}::$src {} ${ns}::my $dst ; # ❹
    }
    return
}
```

❶ *N.B.* we put this procedure into a namespace that is accessible to all TclOO objects.

❷ Clean up any extraneous colon characters. We definitely don't want any leading colons as that would imply a fully-qualified namespace name.

❸ Make sure to create the namespace.

❹ Use an `alias` to link the command in the child namespace back to a class method.

With the ability to link commands to a child namespace, we can now define the state model configuration commands.

```
<<stsaclass constructor>>=
linkTo smdsl\
    {state State}\
    {transition Transition}\
    {defaultTrans DefaultTrans}\
    {initialState InitialState}\
    {terminal Terminal}
```

With all these preliminaries out of the way, the `Statemodel` method simply consists of evaluating a `script` in the proper child namespace that resolves the commands required to specify the state behavior. We will discuss those commands in detail below.

```
<<stsaclass configuration>>=
method Statemodel {script} {
    namespace eval [namespace current]::smdsl $script
}
```

Statemodel Method Tests

Here we will only test that the statemodel script is evaluated in the proper child namespace. After presenting the code for the individual state model definition commands we will have more extensive tests.

```
<<meta constructor tests>>=
test Statemodel-1.0 {
    insure the statemodel script executes in the correct namespace
} -setup {
    ::stsatcl STSAClass create sm1
} -cleanup {
    sm1 destroy
} -body {
```

```
    [info object namespace sm1]::my Statemodel {
        puts -nonewline [namespace tail [namespace current]]
        attribute Foo Bar
    }
} -result {invalid command name "attribute"} -output {smdsl} -returnCodes error
```

Polymorphic Method

The semantics of execution model in XUML supports only one type of polymorphism and it takes the form of events that are directed at a super class. Super class events that are declared as polymorphic are propagated, at runtime, to the subclass instance to which the target super class is currently related. This idea is only valid for those classes that are the super class in a generalization relationship. From a methodology point of view, there is much to say about polymorphic events. However from an implemementation architecure point of view, they amount to little more than factoring some common code into the event signalling operation.

The `polymorphic` command is used to declare those events that are intended to be treated as polymorphic with respect to their subclasses.

```
polymorphic event1 event2 ...
```

`event1 event2 ...`
: the names of events that are to be treated polymorphically.

It is sufficient simply to store the event names as a set.

```
<<stsaclass constructor>>=
my variable polyEvents
set polyEvents [list]
```

```
<<stsaclass configuration>>=
method Polymorphic {args} {
    my variable polyEvents
    ::struct::set add polyEvents $args
}
```

Polymorphic Method Tests

```
<<meta constructor tests>>=
test Polymorphic-1.0 {
    define polymorphic events
} -setup {
    ::stsatcl STSAClass create pm1
} -cleanup {
    pm1 destroy
} -body {
    [info object namespace pm1]::my Polymorphic e1 e2
    llength [set [info object namespace pm1]::polyEvents]
} -result {2}
```

InstOp Method

Instance based operations in XUML allow code common to all instances to be factored into one place.

```
instop name arglist body
```

name
The name of the instance operation.

arglist
A list of formal parameters to the instance operation in the style of the `proc` command.

body
A Tcl script that executes when the instance operation is invoked.

The direct implementation of an instance operation as a `method` is used. TclOO provides all the machinery here and we just need to pass the parameter along.

```
<<stsaclass configuration>>=
method InstOp {name arglist body} {
    ::oo::define [self] method $name $arglist $body ; # ❶
    ::oo::define [self] export $name                ; # ❷
}
```

❶ Note here that `::oo::define` is used to declare the instance operation. Because we are constructing a class, we want the instance operation to be realized as part of the configuration of the class being constructed. Configuring the method into the class will insure that the method is available to created instances. This is just one of the mind-binding aspects of meta-classes.

❷ All instance operations are exported regardless of whether the first letter is capitalized.

InstOp Method Tests

```
<<meta constructor tests>>=
test InstOp-1.0 {
    define instance operation
} -setup {
    ::stsatcl STSAClass create io1
} -cleanup {
    io1 destroy
} -body {
    [info object namespace io1]::my InstOp print {a} {puts -nonewline $a}
    io1 create inst-io1
    inst-io1 print alpha
    expr {"print" in [info object methods inst-io1 -all]}
} -result {1} -output {alpha}
```

ClassOp Method

By analogy to instance based operations, XUML allows for class based operations. This is code that common to the class and independent of the instances. These types of operations are not common, but serve a role providing, for example, improved access to instance queries.

```
classop name arglist body
```

name
The name of the class based operation.

arglist
A list of formal parameters to the instance operation in the style of the `proc` command.

body
A Tcl script that executes when the operation is invoked.

The direct implementation of a class based operation as a `method` is used. Again, TclOO provides all the machinery here.

```
<<stsaclass configuration>>=
method ClassOp {name arglist body} {
    ::oo::objdefine [self] method $name $arglist $body  ; # ❶
    ::oo::objdefine [self] export $name                 ; # ❷
}
```

❶ Compare this with the implementation of the `InstOp` method. In this case we want the class based operation to be available to the class that is under construction. As classes are objects themselves under TclOO, we then need to define the method on the class object itself using `::oo::objedefine`

❷ All class operations are exported regardless of whether the first letter is capitalized.

ClassOp Method Tests

```
<<meta constructor tests>>=
test ClassOp-1.0 {
    define class based operation
} -setup {
    ::stsatcl STSAClass create co1
} -cleanup {
    co1 destroy
} -body {
    [info object namespace co1]::my ClassOp print {a} {puts -nonewline $a}
    co1 print beta
    expr {"print" in [info object methods co1]}
} -result {1} -output {beta}
```

Constructor Method

Under some circumstances it is necessary to have more direct access to the underlying TclOO constructor. That is provided by the `constructor` command. The primary purpose of this is to allow other system resources to be acquired at instance creation time. It is a capablity subject to abuse, but the occasional need to control the construction process is very real. The primary restriction is that there is no way to pass any arguments to this code.

```
constructor body
```

body
: A Tcl script that is executed when an instance is created.

```
<<stsaclass constructor>>=
my variable cscript
set cscript {}
```

```
<<stsaclass configuration>>=
method Constructor {body} {
    my variable cscript
    set cscript $body
}
```

Destructor Method

Resources acquired during instance construction may need to be release at instance deletion time. So we provide a hook into the destructor.

```
destructor body
```

body
A Tcl script that is executed when an instance is deleted.

```
<<stsaclass constructor>>=
my variable dscript
set dscript {}
```

```
<<stsaclass configuration>>=
method Destructor {body} {
    my variable dscript
    set dscript $body
}
```

State Model Configuration DSL

Now that we have covered all the first level class configuration commands, we can turn our attention to the commands used to configure a state model. Recall that the `statemodel` command takes a script and that script should invoke the commands discussed here to specify the state model for a class.

The state model defined by these commands is of the Moore type. This means that the processing associated with the state is executed *when the state is entered*. XUML has a long history of using this formulation of state models and specifically eschews hierarchical state models. We follow the syntax and conventions of the `oomoore` package. Indeed, most of the code for configurating and dispatching events to state models was taken from that package and adapted here.

The commands to specify a state model are:

- state → define a state and the activity executed when it is entered.
- transition → define the transition between states that an event causes.
- defaultTrans → specify the default transition.
- initialState → specify the state where a machine starts.
- terminal → specify states where an object is deleted.

The state model script may contain invocations of these commands in any order. In our examples, we tend to declare a state and then list the out-bound transitions for that state. However, this is just a convention and there is no inherent order dependency in how states and transitions are declared. The invocation synposis for each command is given in the sections below.

State Method

The `State` method is used to specify the states of the state model. States may have arguments and will execute the provided Tcl script when the state is entered.

```
state name arglist body
```

name
the name of the state. The names, “IG”, “CH” and “@” are reserved. The reserved state names are discussed as part of the `transition` command.

arglist
: a list of parameters in the style of the `proc` command. Events that cause a transition into the state must supply actual argument values that correspond to the parameters.

body
: a script that is executed when the state is entered.

We use a variable to hold the set of defined states. Order is important here because we will choose to use the first defined state as the default initial state if one is not explicitly declared.

```
<<stsaclass constructor>>=
my variable states
set states [list]
```

```
<<stsaclass configuration>>=
method State {name argList body} {
    my variable states
    if {$name in {IG CH @}} {
        tailcall ::stsatcl::DeclError RESERVED_STATE $name
    } elseif {$name in $states} {
        tailcall ::stsatcl::DeclError DUPLICATE_STATE $name
    } else {
        lappend states $name  ; # ❶
        ::oo::define [self] method ${name}__STATE__ $argList $body ; # ❷
        ::oo::define [self] unexport ${name}__STATE__              ; # ❸
    }
}
```

❶ We don't use the `::struct::set` operations so as to guarantee the order of the state name list.

❷ State activities just become methods of the class being created. Again, we just use the existing TclOO facilities. However, we do want to insure that the method names for state activities do not interfer with any other method definitions that might arise from instance operations.

❸ State activity methods should **not** be invoked from outside of the object. They also should **not** be invoked directly as methods from other methods. It is not possible to hide things completely in Tcl, but an unexported, contrived name is intended to make sure you have to work a little harder to circumvent the design intent.

```
<<error code formats>>=
RESERVED_STATE      {states may not be named by the reserved name, "%s"}
DUPLICATE_STATE     {duplicate state, "%s"}
```

State Method Tests

```
<<meta constructor tests>>=
test State-1.0 {
    define state
} -setup {
    ::stsatcl STSAClass create st1
} -cleanup {
    st1 destroy
} -body {
    [info object namespace st1]::my Statemodel {
        state s1 {} {
            puts "in state s1"
        }
    }
    set isdef [expr {"s1" in [set [info object namespace st1]::states]}]
```

```
    set hasmeth [expr {"s1__STATE__" in [info class methods st1 -private]}]
    expr {$isdef && $hasmeth}
} -result {1}
```

```
<<meta constructor tests>>=
test State-2.0 {
    define state with reserved name
} -setup {
    ::stsatcl STSAClass create st2
} -cleanup {
    st2 destroy
} -body {
    [info object namespace st2]::my Statemodel {
        state IG {} {}
    }
} -result {states may not be named by the reserved name, "IG"}\
-returnCodes error
```

```
<<meta constructor tests>>=
test State-3.0 {
    define state with reserved name
} -setup {
    ::stsatcl STSAClass create st3
} -cleanup {
    st3 destroy
} -body {
    [info object namespace st3]::my Statemodel {
        state Idle {} {}
        state Idle {a b} {puts "$a $b"}
    }
} -result {duplicate state, "Idle"} -returnCodes error
```

Transition Method

The `Transition` method is used to specify how the state model transitions when it receives events. It is useful to consider the possible transitions that a state model may make to be a matrix where the rows of the matrix are labeled by state names and columns are labeled by event names. The matrix entry at the intersection of each row and column is the name of the new state to which the state machine will enter. It is not necessary to specify the entire *states* by *events* matrix. A default entry may be specified for the missing entries.

There are a few more rules about state model transitions. The "IG" state denotes that an event is to be ignored. The "CH" state asserts that the event can never happen in a state and if it does then it is an error. Neither of these state cause a transition, despite their being used as the target of a transition. The "@" state denotes the initial pseudo-state into which a new asynchronously created state machine is placed. There is never an activity associated with the "@" state (hence it is called a pseudo-state).

```
transition current - event -> target
```

`current`
: The state name from which the transition arises. The special states, **IG** and **CH** are not allowed for this argument.

`event`
: The event name that causes the transition.

`target`
: The state to which the transition is made. The special state @ is not allowed as a target.

The "-" and "->" arguments are literally required but only serve as syntactic sugar. The command asserts that when the state model is in the *current* state and receives the *event* event it is to transition to the *target* state. *N.B.* that *current* and *target* may be the same state. In that case, the *event* causes the the state to be reentered and its activity is executed again.

This configuration command will fill in two pieces of data. There is no special configuration command to declare events. They are simply picked up from the transition command. The transitions themselves are stored in a dictionary keyed by the values of both the *current* and *event* arguments.

```
<<stsaclass constructor>>=
my variable events
set events [list]
my variable transitions
set transitions [dict create]
```

```
<<stsaclass configuration>>=
method Transition {current - event -> target} {
    if {$current in {IG CH}} {
        tailcall ::stsatcl::DeclError TRANS_SRC $current
    } elseif {$current eq "@"} { # ❶
        my variable states
        ::struct::set include states $current
    }

    if {$target eq "@"} {
        tailcall ::stsatcl::DeclError TRANS_DST
    }
    my variable events
    ::struct::set include events $event ; # ❷

    my variable transitions
    if {[dict exists $transitions $current $event]} {
        tailcall ::stsatcl::DeclError DUPLICATE_TRANS $current $event
    } else {
        dict set transitions $current $event $target
    }
}
```

❶ The only valid place were the "@" state may be specified is as the *current* state in a `transition` command. If it shows up we need to record that fact.

❷ Event names are collected from all the transition statements but they form a set.

```
<<error code formats>>=
TRANS_SRC           {a transition cannot originate from the "%s" pseudo-state}
TRANS_DST           {the initial pseudo-state, "@", cannot be the\
                     target of a transition}
DUPLICATE_TRANS     {duplicate transition, "%s - %s"}
```

Transition Method Tests

```
<<meta constructor tests>>=
test Transition-1.0 {
    define transition
} -setup {
    ::stsatcl STSAClass create tr1 {}
} -cleanup {
    tr1 destroy
} -body {
    [info object namespace tr1]::my Statemodel {
        transition s1 - e1 -> s1
    }
```

```
    namespace upvar [info object namespace tr1] transitions t
    namespace upvar [info object namespace tr1] events e

    set hasevent [expr {"e1" in $e}]
    set hastrans [expr {[dict get $t s1 e1] eq "s1"}]
    expr {$hasevent && $hastrans}
} -result {1}
```

```
<<meta constructor tests>>=
test Transition-2.0 {
    define transition -- bad source state
} -setup {
    ::stsatcl STSAClass create tr2 {}
} -cleanup {
    tr2 destroy
} -body {
    [info object namespace tr2]::my Statemodel {
        transition IG - e1 -> s1
    }
} -result {a transition cannot originate from the "IG" pseudo-state}\
-returnCodes error
```

```
<<meta constructor tests>>=
test Transition-3.0 {
    define transition -- bad source state
} -setup {
    ::stsatcl STSAClass create tr3 {}
} -cleanup {
    tr3 destroy
} -body {
    [info object namespace tr3]::my Statemodel {
        transition s1 - e1 -> @
    }
} -result {the initial pseudo-state, "@", cannot be the target of a transition}\
-returnCodes error
```

```
<<meta constructor tests>>=
test Transition-4.0 {
    define transition -- duplicate transition
} -setup {
    ::stsatcl STSAClass create tr4 {}
} -cleanup {
    tr4 destroy
} -body {
    [info object namespace tr4]::my Statemodel {
        transition s1 - e1 -> s1
        transition s1 - e1 -> s2
    }
} -result {duplicate transition, "s1 - e1"} -returnCodes error
```

DefaultTrans Method

It is not necessary to specify all the entries of the conceptual transition matrix for a state model. Any unspecified entries will default to “CH”. However, sometimes it is more convenient to specify “IG” as the default transition. The `DefaultTrans` method allows you to set the default transition to either “CH” or “IG”.

```
defaultTrans trans
```

trans
Either “IG” or “CH” may be specified as the default transition.

A simple variable is used to hold the default transition. That variable is initialized to “CH”.

```
<<stsaclass constructor>>=
my variable defaulttrans
set defaulttrans CH
```

```
<<stsaclass configuration>>=
method DefaultTrans {trans} {
    if {$trans in {IG CH}} {
        my variable defaulttrans
        set defaulttrans $trans
    } else {
        tailcall ::stsatcl::DeclError BAD_DEFAULT_TRANS $trans
    }
}
```

```
<<error code formats>>=
BAD_DEFAULT_TRANS       {bad default transition name, "%s": must be one of\
                         "IG" or "CH"}
```

InitialState Method

```
initialState state
```

state
The state name into which newly create instances are placed.

```
<<stsaclass constructor>>=
my variable initialstate
set initialstate {}
```

```
<<stsaclass configuration>>=
method InitialState {state} {
    if {$state in {IG CH @ {}}} {
        tailcall ::stsatcl::DeclError BAD_INIT_STATE $state
    }
    my variable initialstate
    set initialstate $state
}
```

```
<<error code formats>>=
BAD_INIT_STATE      {initial state, "%s", cannot be "IG", "CH", "@" or empty}
```

Terminal Method

```
terminal state1 state2 ...
```

state
The name of a state that is to be deemed a terminal state. When an instance transitions into a terminal state, it is destroyed at the completion of the state activity.

```
<<stsaclass constructor>>=
my variable terminals
set terminals [list]
```

```
<<stsaclass configuration>>=
method Terminal {args} {
    foreach state $args {
        if {$state in {IG CH @ {}}} {
            tailcall ::stsatcl::DeclError BAD_TERM_STATE $state
        }
    }
    my variable terminals
    ::struct::set add terminals $args
}
```

```
<<error code formats>>=
BAD_TERM_STATE  {terminal state, "%s", cannot be "IG", "CH", "@" or empty}
```

State Model Example

With the state model configuration commands in hand, we can now show to define one of the state models for our example. Here we will consider the Washing Machine class. The state model for that class is shown below.

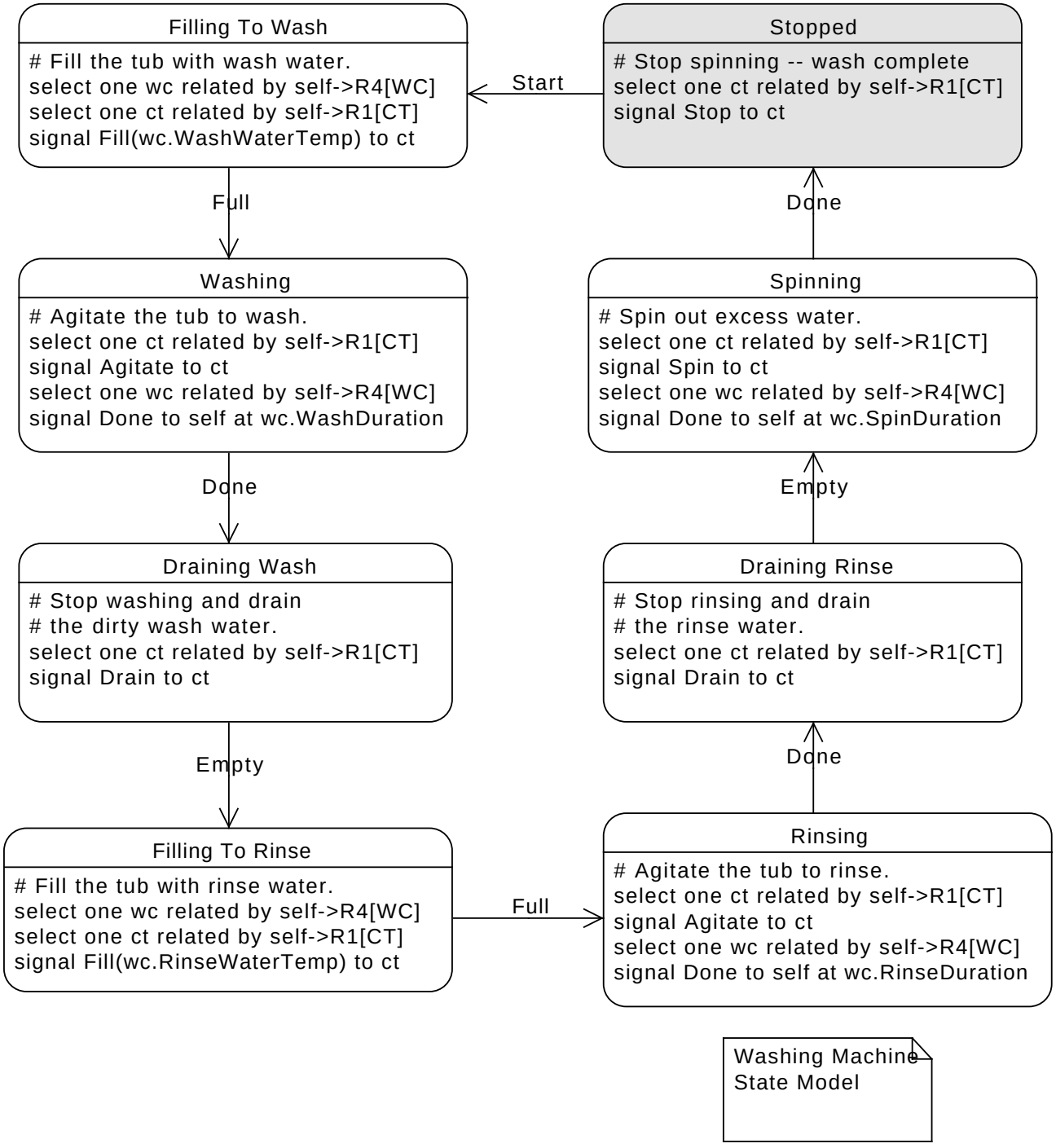

Figure 4: Washing Machine State Model

The initial state is Stopped. The transitions in the model show how dirty clothes are cleaned. Events drive the state machine through its lifecycle filling, washing, draining, filling again, rinsing, draining and finally spinning out the excess water. The responsiblities of the Washing Machine class are rather narrow. It restricts its scope to controlling the Clothes Tub and timing the various parts of the overall cycle.

You will notice that the state activities contained in each state are coded in a pseudo-code with which you may not be familiar. There are many ways to express the algorithmic processing and we have chosen one here that is very similar to that supported

proposed by others. We don't have to be overly precise since we don't intend to parse the action language. Rather the intent is to state the overall intent and then show how, in terms of the *model level constructs*, that intent is realized.

Below we show the definition of the Washing Machine class along with its state model configuration. For now, we are not showing the how the state activities end up as Tcl code. We will have more to say on that subject later. What we want to emphasize here is the correspondence between the state model graphic and its representation as a set of declarative commands.

```
<<WM class>>=
STSAClass create WashingMachine {
    attribute MachineID
    reference R1 -> ClothesTub
    reference R4 -> WashingCycle
    statemodel {
        initialState Stopped
        defaultTrans CH

        state Stopped {} {
            <<WM stopped activity>>
        }
        transition Stopped - Start -> FillingToWash

        state FillingToWash {} {
            <<WM filling to wash activity>>
        }
        transition FillingToWash - Full -> Washing

        state Washing {} {
            <<WM washing activity>>
        }
        transition Washing - Done -> DrainingWash

        state DrainingWash {} {
            <<WM draining wash activity>>
        }
        transition DrainingWash - Empty -> FillingToRinse

        state FillingToRinse {} {
            <<WM filling to rinse activity>>
        }
        transition FillingToRinse - Full -> Rinsing

        state Rinsing {} {
            <<WM rinsing activity>>
        }
        transition Rinsing - Done -> DrainingRinse

        state DrainingRinse {} {
            <<WM draining rinse activity>>
        }
        transition DrainingRinse - Empty -> Spinning

        state Spinning {} {
            <<WM spinning activity>>
        }
        transition Spinning - Done -> Stopped
    }
}
```

Completing XUML Class Construction

At this point we have now covered all the commands that are used to construct a configuration script used to create a new XUML class. These commands all follow the same pattern, namely they validate their inputs as possible and store away argument values into appropriate data structures.

Now we consider what needs to be done to evaluate the configuration script and complete constructing the new class.

First, we evaluate the configuration script in the namespace of the newly created class.

```
<<stsaclass constructor>>=
my eval $config
```

This will cause the various configuration command methods to be invoked. The next step is to perform some semantic validation. It is still possible to incorrectly specify parts of the configuration and a correct configuration cannot be determined on a command by command basis. Like any language, in our DSL it is quite possible to make syntactically correct statements that are still meaningless.

We divide the remaining evaluation into parts that parallel the configuration, *i.e.* attributes, linkage, state models and polymorphic events. We will also encounter some common code that we have factored into some utility procedures.

```
<<stsaclass constructor>>=
<<attribute semantics>>
<<linkage semantics>>
<<state model semantics>>
<<polymorphic semantics>>
::oo::define [self] {
    <<select related methods>>
    <<map related methods>>
    <<utility methods>>
}
::oo::objdefine [self] {
    <<select instance methods>>
    <<map instance methods>>
}
```

Each of these categories is discussed below.

Attribute Semantics

If the XUML class declares any attributes, we want to make available a set of attribute access methods so that other classes may read and update attributes.

```
<<attribute semantics>>=
if {[dict size $attrInfo] != 0} {
    ::oo::define [self] {
        <<attribute access methods>>
    }
}
```

We will discuss the class methods that will be provided to access attributes below. Here we note that if no attributes are declared, then the methods will not be part of the interface for objects created from the class.

Linkage Semantics

For `reference` and `partition` definitions, we want to include the methods used to create and *traverse* the linkage into the created class. These methods are also described below.

```
<<linkage semantics>>=
if {[dict size $linkInfo] != 0} {
    ::oo::define [self] {
        <<linkage access methods>>
    }
}
```

State Model Semantics

For state models, there is a bit more work to do to insure that a consistent model has been defined. One of the goals of the DSL was *not* to place a rigid order on the statements. Consequently, it is possible to refer to entities during the configuration that are not yet defined (*e.g.* a `transition` command may refer to states not yet defined). So there is some additional validation that must be done now that the state model configuration is fully known.

```
<<state model semantics>>=
if {[llength $states] != 0 || [llength $polyEvents] != 0} {
    <<validate transitions>>
    <<compute transition matrix>>
    <<validate initial state>>
    <<validate terminal states>>
    <<creation events>>
    ::oo::objdefine [self] {
        <<in state creation methods>>
    }
    ::oo::define [self] {
        <<state model methods>>
    }
} elseif {$dscript ne {}} {
    ::oo::define [self] destructor {
        classvariable dscript
        my eval $dscript
    }
}
```

We first insure that no isolated states were defined. Isolated state are those that have no incoming or outgoing transitions. Such state are not reachable and indicate a specification error.

The strategy to compute isolated states starts with finding all the states that have out-bound transitions and all the states that have in-bound transitions. This can be determined by looking at the values of `transitions` dictionary.

```
<<validate transitions>>=
set outstates [list]
set instates [list]
dict for {src trans} $transitions {
    dict for {event dst} $trans {
        if {$src ni $states} { # ❶
            tailcall ::stsatcl::DeclError UNKNOWN_SRC_STATE $src $event $dst
        }
        ::struct::set include outstates $src
        if {$dst ni {IG CH}} { # ❷
            if {$dst ni $states} {
                tailcall ::stsatcl::DeclError UNKNOWN_DST_STATE $src $event $dst
            } else {
                ::struct::set include instates $dst
            }
        }
    }
}
```

❶ While we are iterating across the transitions, check that they are consistent with the set of defined states. We have to defer this until here so there will not be any order dependency in the configuration language.

❷ Skip the non-transitioning pseudo-states of IG and CH.

```
<<error code formats>>=
UNKNOWN_SRC_STATE       {unknown source state in transition, "%s - %s -> %s"}
UNKNOWN_DST_STATE       {unknown target state in transition, "%s - %s -> %s"}
```

Next, we compute the set of states with no in-bound transitions and the set with no out-bound transitions. The set of states with no in-bound transitions is just the difference between the set of all states and those that do have in-bound transitions. Similarly the set of states with no out-bound transitions is computed.

```
<<validate transitions>>=
set noincoming [::struct::set difference $states $instates]
set nooutgoing [::struct::set difference $states $outstates]
```

Finally, isolated states are those which have neither in-bound nor out-bound transitions. This is just the intersection of the two sets we just computed.

```
<<validate transitions>>=
set isostates [::struct::set intersect $noincoming $nooutgoing]
if {![::struct::set empty $isostates]} {
    tailcall ::stsatcl::DeclError ISOLATED $isostates
}
```

```
<<error code formats>>=
ISOLATED        {state model has isolated state(s): "%s"}
```

During state machine event dispatch, we will find it much more convenient to find a new target state if the entire transition matrix is completely populated. We use this opportunity to fill in any transitions not explicity mentioned in a `transition` command with the default transition.

```
<<compute transition matrix>>=
foreach s $states {
    foreach e $events {
        set id [list $s $e]
        if {![dict exists $transitions $s $e]} {
            dict set transitions $s $e $defaulttrans
        }
    }
}
```

We also have to validate that the initial state is reasonable. This is another situation where we must defer the check until after the configuration script is executed to avoid imposing some arbitrary order onto the configuration commands in a script. We can also now determine if the initial state was not explicitly defined and implement the default behavior of taking the first defined state as the default initial state.

```
<<validate initial state>>=
if {$initialstate eq {}} {
    set initialstate [lindex $states 0]
} elseif {$initialstate ni $states} {
    tailcall ::stsatcl::DeclError UNKNOWN_INIT_STATE $initialstate
}
```

```
<<error code formats>>=
UNKNOWN_INIT_STATE      {unknown initial state, "%s"}
```

We must insure that the set of terminal states is a subset of the set of all states. This will make sure that no undefined states are included in the terminal state set.

```
<<validate terminal states>>=
if {![::struct::set subsetof $terminals $states]} {
    set unknown [::struct::set difference $terminals $states]
    tailcall ::stsatcl::DeclError UNKNOWN_TERM_STATE $unknown
}
```

```
<<error code formats>>=
UNKNOWN_TERM_STATE      {unknown terminal states, "%s"}
```

Finally, we deal with **creation** events. Creation events allow class instances to be created asynchronously. In this architecture, a creation event implies creating a new instance, placing that instance in the "@" initial pseudo-state and signalling the creation event to the instance. Creation events are consequently directed at the class as it will be responsible for creating the instance. Here, we test to see if the "@" initial pseudo-state is present and if so, we then need to have a method to handle the creation events.

```
<<creation events>>=
if {"@" in $states} {
    ::oo::objdefine [self] method signal {event args} {
        try {
            set src [lindex [uplevel 1 {self caller}] 1]
        } on error {} {
            set src {}
        }

        set inst [my newin @]
        ::stsatcl::TraceCreation $src $event $inst
        $inst signal $event {*}$args
        return $inst
    }
}
```

Note that the creation event generated by a method named, `signal`, just as for other events. The difference here is that the method is defined on the XUML class not on one of its objects.

```
<<exported tests>>=
test creation-event-1.0 {
    send a creation event
} -setup {
    stsatcl STSAClass create cre1 {
        attribute Count {Maximum 27}
        statemodel {
            transition @ - e1 -> s1

            state s1 {} {
                my variable Maximum
                incr Maximum 10
                ::stsatcl::test::syncToTest $Maximum
            }
        }
    }
} -cleanup {
    cre1 destroy
} -body {
    set inst [cre1 signal e1]
    ::stsatcl::test::waitForSync
    $inst readAttributes Maximum
} -result {37}
```

Polymorphic Event Semantics

Polymorphic events are kept entirely separate from transitioning events. It is possible that a superclass has its own state model and has polymorphic events passed along to its subclasses. So we cannot have the same event name for both a polymorphic and transitioning event.

```
<<polymorphic semantics>>=
set commonEvents [::struct::set intersect $polyEvents $events]
if {![::struct::set empty $commonEvents]} {
    tailcall ::stsatcl::DeclError COMMON_EVENTS $commonEvents
}
```

```
<<error code formats>>=
COMMON_EVENTS   {polymorphic events and ordinary events cannot have\
                 the same name, "%s"}
```

Polymorphism in events is only defined across a generalization relationship. So defining polymorphic events only makes sense if there is at least one partition defined for the class.

```
<<polymorphic semantics>>=
if {[llength $polyEvents] != 0 && [llength $partitions] == 0} {
    tailcall ::stsatcl::DeclError BAD_POLY_EVENTS $polyEvents
}
```

```
<<error code formats>>=
BAD_POLY_EVENTS     {polymorphic events, "%s", are defined,\
                     but there are no defined partition linkages}
```

Completing the Example Classes

We have now completed the mini-DSL that is used configure an XUML class. To recap, the `STSAClass` is a meta-class that, as part of its construction, accepts a configuration script. That configuration script invokes commands that together form a small domain specific language (DSL) to describe the data and dynamics the newly create class is to have. We refer, collectively, to those newly created classes as *XUML classes*.

In this section we complete the XUML class defintions for the remaining classes in our example model. We have already seen the definitions for the Washing Cycle class and the Washing Machine class. We have also seen the references defined for the Clothes Tub class. The remaining three classes are given below along with the state model for the Clothes Tub class. Again we will defer the discussion of the state activities. To be able to code the state activities we need to know about some class methods that we will discuss after we finish with the class and state diagrams.

We start with the state model for the Clothes Tub class. It is shown below.

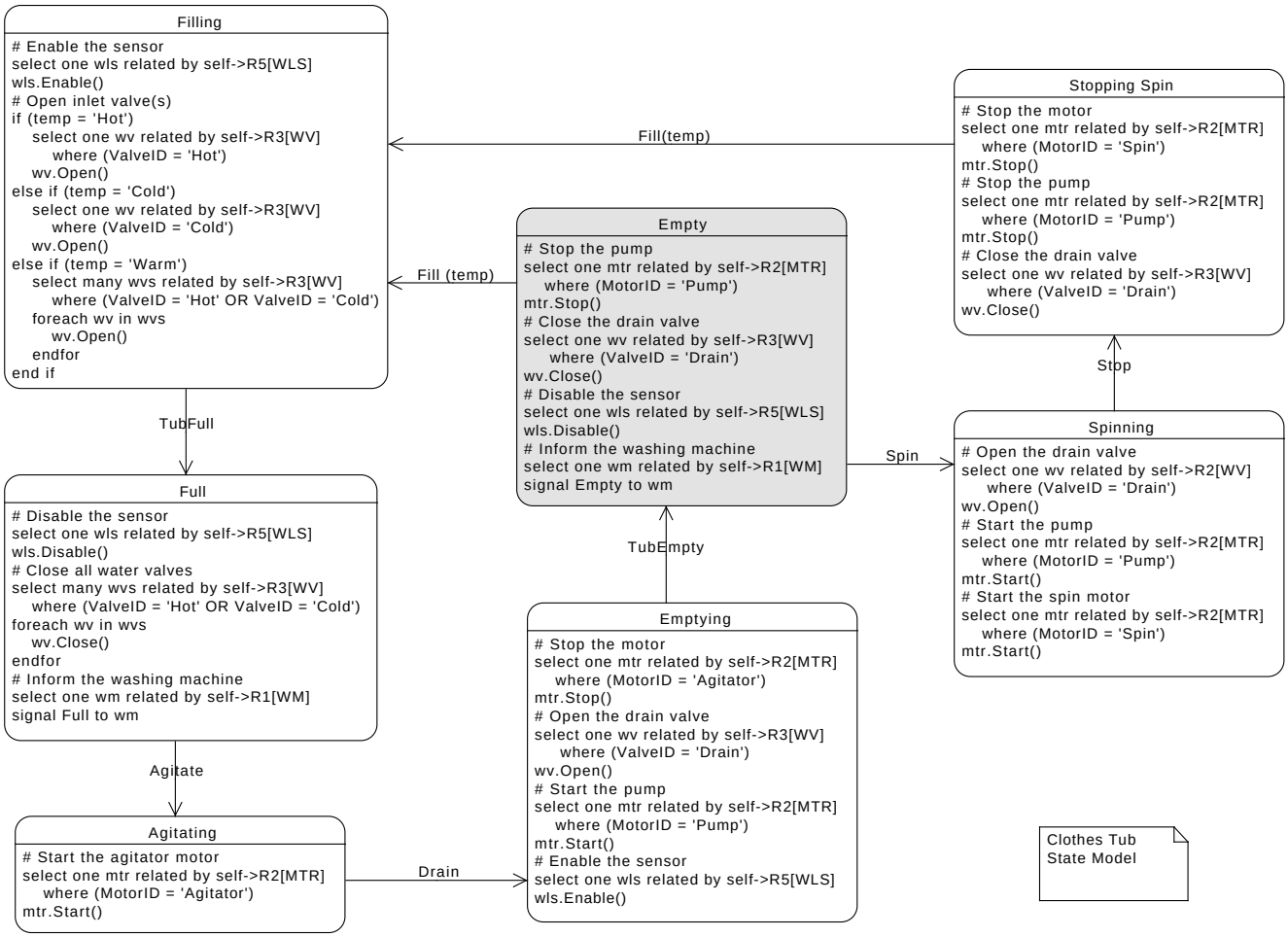

Figure 5: Clothes Tub State Model

Like the state model for Washing Machine, we code the diagram into the DSL command syntax. The code sequences for the activities are defined later.

```
<<CT state model>>=
statemodel {
    initialState Empty
    defaultTrans CH

    state Empty {} {
        <<CT empty activity>>
    }
    transition Empty - Fill -> Filling
    transition Empty - Spin -> Spinning

    state Filling {temp} {
        <<CT filling activity>>
    }
    transition Filling - TubFull -> Full
    transition Filling - TubEmpty -> IG     ; # ❶

    state Full {} {
        <<CT full activity>>
    }
```

```
    transition Full - Agitate -> Agitating

    state Agitating {} {
        <<CT agitating activity>>
    }
    transition Agitating - Drain -> Emptying

    state Emptying {} {
        <<CT emptying activity>>
    }
    transition Emptying - TubEmpty -> Empty
    transition Emptying - TubFull -> IG

    state Spinning {} {
        <<CT spinning activity>>
    }
    transition Spinning - Stop -> StoppingSpin

    state StoppingSpin {} {
        <<CT stopping spin activity>>
    }
    transition StoppingSpin - Fill -> Filling
}
```

(1) We allow for the fact that when the water level sensor is enabled it may send events that reflect its current state. So we simply ignore those sensor events in which we are not currently interested. See also the **Emptying** state transition for the **TubFull** event.

The Water Valve and Motor classes have a single attribute and some instance operations.

```
<<WV class>>=
STSAClass create WaterValve {
    attribute ValveID
    instop Open {} {
        <<WV open operation>>
    }
    instop Close {} {
        <<WV close operation>>
    }
}
```

```
<<MTR class>>=
STSAClass create Motor {
    attribute MotorID
    instop Start {} {
        <<MTR start operation>>
    }
    instop Stop {} {
        <<MTR stop operation>>
    }
}
```

```
<<WLS class>>=
STSAClass create WaterLevelSensor {
    reference R5 -> ClothesTub
    instop Enable {} {
        <<WLS enable operation>>
    }
    instop Disable {} {
        <<WLS disable operation>>
```

```
    }
}
```

XUML Class Methods

XUML defines a set of data *accessors* that are required to support the execution semantics of the model. These accessors are independent of the nature of the class. Roughly speaking the accessors are:

- Create instances.
- Create instances in a particular state.
- Delete instances
- Select instances that meet a stated criteria.
- Read attributes of a specified instance.
- Update attributes of a specified instance.

There are also operations that support the dynamic execution model. These allow a state activity to:

- Create a link between instances.
- Destroy a link between instances.
- Signal an event.
- Signal an event delayed to some time in the future.
- Cancel a delayed event.
- Query the time remaining of a delayed event.

In the following section we describe the methods that the XUML classes have. These are methods that are common to all XUML classes and represent the implementation of the data accessors and dynamic operations that the XUML logical model requires. Recall that the `instop` and `classop` DSL commands can be used to create object methods and class methods, respectively, to a particular XUML class. The methods in this section go somewhat beyond the minimum required to support the XUML execution model as experience shows some support for common operation is useful.

First we deal with creating objects and the variations on construction that are supported. Afterwards we follow the set of topics that we have already established. There are methods to deal with attributes, linkage and state model dynamics and sections will be devoted to each of these topics. Then we explain methods that are used to obtain sets of object instances, either selecting them by some criteria or selected related instances for some criteria. Lastly, we look a methods for computing on sets of instances.

Creating Objects

In this section we deal with constructing an object from an XUML class. We need to be clear of that distinction given how much of our effort so far has been devoted to constructing the XUML class itself from the `STSAClass` meta-class.

Constructor

The constructor for an XUML class is invoked when the `create` or `new` method is used to generate an object, as in:

`STSAClass create` *xumlclass* `{` **`...`** *configuration script* **`...`**`}`

xumlclass `create` *objname attr1 value1 attr2 value2 ...*

xumlclass `new` *attr1 value1 attr2 value2 ...*

`attr value ...`
: a set of attribute name / attribute value pairs. If supplied, then the values will override any default value set for the attribute.

The implementation of the constructor deals, roughly speaking, with the same areas that the configuration DSL deals with, namely, attributes, linkages and state model.

```
<<stsaclass constructor>>=
::oo::define [self] constructor {args} {
    <<attribute initialization>>
    <<linkage initialization>>
    <<state model initialization>>

    classvariable cscript ; # ❶
    my eval $cscript
}
```

❶ At the end of construction we execute any user supplied construction script.

If the XUML class declares any attributes, we want to make them instance variables. This allows state activities to use the `my variable` command to access the attributes. It is a convenient and expected way to do things is TclOO objects. We also need to set up values for the attributes. When an object is constructed, it attributes will be given values that are either its default value (which will be the empty string if no default was supplied) or the value given in the arguments to `create` or `new` (or `createin` or `newin`).

The easy way to accomplish this is to merge the dictionary supplied by `args` onto the `attrInfo` dictionary. The `dict merge` command does the overriding as we want, if we make sure that the supplied `args` dictionary is the second argument to `dict merge`.

```
<<attribute initialization>>=
classvariable attrInfo  ; # ❶

if {[dict size $attrInfo] != 0} { # ❷
    dict for {attrName defValue} [dict merge $attrInfo $args] {
        if {[dict exists $attrInfo $attrName]} { # ❸
            my variable $attrName
            set $attrName $defValue
        } else {
            tailcall ::stsatcl::DeclError UNKNOWN_ATTRIBUTE $attrName
        }
    }
}
```

❶ From the point of view of the an object of an XUML class, the attribute information gleened from the DSL commands is stored in a variable in the class. The `oo::util` package provides the `classvariable` command to conveniently create a reference to the proper variable in the class. We will use this construct many more times and it insures that all the objects of the XUML class work off of the same configuration information.

❷ If no attributes were defined when the class was configured, then no `variables` will have been defined and we ignore any supplied arguments.

❸ We need to check if the attribute name is valid since the caller may have included an unknown attribute in the argument list and, after the dictionary merge, it would show up as part of the iteration over the dictionary.

```
<<error code formats>>=
UNKNOWN_ATTRIBUTE        {unknown attribute, "%s"}
```

Constructor Attribute Tests

```
<<constructor tests>>=
test constructor-1.0 {
    create an instance with attributes
} -setup {
    stsatcl STSAClass create test5 {
        attribute Count {Maximum 20}
    }
} -cleanup {
    test5 destroy
} -body {
    test5 create inst5
    set [info object namespace inst5]::Maximum
} -result {20}
```

```
<<constructor tests>>=
test constructor-1.1 {
    create an instance with attributes -- override values
} -setup {
    stsatcl STSAClass create test5 {
        attribute Count {Maximum 20}
    }
} -cleanup {
    test5 destroy
} -body {
    test5 create inst5 Maximum 30
    set [info object namespace inst5]::Maximum
} -result {30}
```

```
<<constructor tests>>=
test constructor-1.2 {
    error overriding attributes
} -setup {
    stsatcl STSAClass create test5 {
        attribute Count {Maximum 20}
    }
} -cleanup {
    test5 destroy
} -body {
    test5 create inst5 Foo 30
} -result {unknown attribute, "Foo"} -returnCodes error
```

The object command names that make up the linkage information are stored in variables by the same name as the linkage. So we must initialize these variables.

```
<<linkage initialization>>=
classvariable linkInfo

foreach linkName [dict keys $linkInfo] {
    my variable $linkName
    set $linkName {}
}
```

Constructor Linkage Tests

```
<<constructor tests>>=
test constructor-2.0 {
    create an instance with a reference
} -setup {
    stsatcl STSAClass create ct2 {
        reference R42 -> foo
    }
} -cleanup {
    ct2 destroy
} -body {
    ct2 create inst2
    info exists [info object namespace inst2]::R42
} -result {1}
```

One of the more significant aspects of an XUML class is its ability to dispatch events into a state machine. There is substantial machinery required to do this and we devote much text below to the details of how the state machine execution is implemented on top of Tcl. We will defer that discussion until later, but during object construction we must do some initialization of the state machine execution mechanism.

What we will discuss here is the notion of the current state of an object. All objects of an XUML class have the same state model behavior. However, each object can be in a state that is different from that of any other object. This means we need a variable to hold the current state that is part of the object

```
<<state model initialization>>=
classvariable states
if {[llength $states] != 0} {
    my variable __currentstate__
    # Set the initial state to the default
    classvariable initialstate
    set __currentstate__ $initialstate

    <<binding state machine events>>
}
```

When an object is created from an XUML class using `create` or `new` it's current state is set to be the initial state defined by the `initialState` command[5]. *N.B.* that any activity associated with the initial state is **not** executed. We will also see below that the `createin` and `newin` methods can be used to create an object in some state other than its default initial state. We will also see that it is possible to create objects asychronously and, in that case, we will be able to cause some processing to happen immediately after creation.

Constructor State Tests

```
<<constructor tests>>=
test constructor-3.0 {
    define a state model -- defined initial state
} -setup {
    stsatcl STSAClass create test6 {
        statemodel {
            state s1 {} {
                puts "[self] in s1"
            }
            transition s1 - e1 -> s2

            state s2 {a} {
                puts $a
```

[5] or the first state defined if the `initialState` command is not invoked

```
            }
            transition s2 - e2 -> s1

            initialState s2
        }
    }
} -cleanup {
    test6 destroy
} -body {
    test6 create inst6
    set [info object namespace inst6]::__currentstate__
} -result {s2}
```

```
<<constructor tests>>=
test constructor-3.1 {
    define a state model -- default initial state
} -setup {
    stsatcl STSAClass create test6 {
        statemodel {
            state s1 {} {
                puts "[self] in s1"
            }
            transition s1 - e1 -> s2

            state s2 {a} {
                puts $a
            }
            transition s2 - e2 -> s1
        }
    }
} -cleanup {
    test6 destroy
} -body {
    test6 create inst6
    set [info object namespace inst6]::__currentstate__
} -result {s1}
```

```
<<constructor tests>>=
test constructor-3.2 {
    define state model -- check transition
} -setup {
    stsatcl STSAClass create test6 {
        attribute Count {Maximum 20}
        statemodel {
            state s1 {} {
                puts "[self] in s1"
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    test6 destroy
} -body {
    test6 create inst6
    namespace upvar [info object namespace test6] transitions transitions
    dict get $transitions s1 e1
} -result {s1}
```

Constructing in a State

XUML execution semantics support the idea of creating an object in a specified state. The XUML classes here support that notion by augmenting `create` and `new` with the `createin` and `newin` methods. It is worth repeating that creating an object either its default initial state or a specified state does **not** execute any activity associated with the state. The initial state only determines how the object reacts to events in the future.

`STSAClass create` *xumlclass* `{ ...` *configuration script* `...}`

xumlclass `createin` *objname initialstate attr1 value1 attr2 value2 ...*

xumlclass `newin` *initialstate attr1 value1 attr2 value2 ...*

`initialstate`
: the name of the state into which the newly created object is place.

The implementation strategy for these methods is to use the `create` and `new` methods and then, after construction, set the current state to the given *initialstate*.

```
<<in state creation methods>>=
method createin {name initialstate args} {
    my variable states
    if {$initialstate in $states} { # ❶
        if {[string range $name 0 1] ne "::"} {
            set ns [string trimright [uplevel 1 namespace current] :]
            set name ${ns}::${name}
        }
        set inst [my create $name {*}$args]
        set [info object namespace $inst]::__currentstate__ $initialstate
    } else {
        tailcall ::stsatcl::DeclError UNKNOWN_STATE $initialstate $states
    }

    return $inst
}
```

```
<<error code formats>>=
UNKNOWN_STATE   {unknown state, "%s", should be one of, "%s"}
```

❶ We must deal with the complication that *name* may not be fully qualified. If we hand an unqualified name to `create` it will resolve it to the namespace in which we are running rather than the namespace of the caller. We have to undertake the desired namespace resolution here so that `create` will see a fully-qualified name.

```
<<in state creation methods>>=
method newin {initialstate args} {
    my variable states
    if {$initialstate in $states} {
        set inst [my new {*}$args]  ; # ❶
        set [info object namespace $inst]::__currentstate__ $initialstate
    } else {
        tailcall ::stsatcl::DeclError UNKNOWN_STATE $initialstate $states
    }

    return $inst
}
```

❶ Unlike `createin`, here we can depend upon `new` creating an object name automatically.

In State Constructor Tests

```
<<constructor tests>>=
test constructor-4.0 {
    create an instance in a given state
} -setup {
    stsatcl STSAClass create test4 {
        statemodel {
            state sA {} {
                puts "[self] in sA"
            }
            transition sA - e1 -> sB

            state sB {a} {
                puts $a
            }
            transition sB - e2 -> sA

            initialState sB
        }
    }
} -cleanup {
    test4 destroy
} -body {
    test4 createin inst4 sA
    set [info object namespace inst4]::__currentstate__
} -result {sA}
```

```
<<constructor tests>>=
test constructor-4.1 {
    create an instance in a given state using newin
} -setup {
    stsatcl STSAClass create test4 {
        statemodel {
            state sA {} {
                puts "[self] in sA"
            }
            transition sA - e1 -> sB

            state sB {a} {
                puts $a
            }
            transition sB - e2 -> sA

            state sC {b} {
                puts $b
            }
            transition sC - e1 -> sB

            initialState sB
        }
    }
} -cleanup {
    test4 destroy
} -body {
    set inst [test4 newin sC]
    set [info object namespace $inst]::__currentstate__
} -result {sC}
```

Destructor

Destroying an XUML class object involves release resource associated with the event dispatch mechanisms. As with the constructor, we will defer that discussion until later.

The destructor logic involves testing if this class has a state model and if so, then releasing any event dispatch resources. It suffices to test for the existance of the current state to determine if the object has an associated state mode.

```
<<state model methods>>=
destructor {
    my variable __currentstate__
    if {[info exists __currentstate__]} {
        <<unbinding state machine events>>
    }
    classvariable dscript ; # ❶
    my eval $dscript
}
```

❶ Note that any additional destructor script code was defined when the XUML class was created, it is contained in a variable in the class itself.

Access to Attributes

XUML execution semantics provide for reading and updating the attributes of an object. Some readers will observe that there is no notion of private or hidden attributes. Privacy is not a concept that enters into XUML and so is not dealt with here.

Reading Attribute Values

The `readAttributes` method provides access to the values of an object's attributes.

xumlobj `readAttributes` *?attrname1 attrname2 . . . ?*

`attrname ...`
The name of an attribute to be read. If one *attrname* argument is given, then the method return value is the simple scalar for the given attribute. If multiple *attrname* arguments are given then the method return value is a dictionary whose keys are the attribute names given by the *attrname* arguments. If no attribute names are given, then the return value of the method is a dictionary of all the attributes of *xumlobj*.

```
<<attribute access methods>>=
method readAttributes {args} {
    set nargs [llength $args]
    if {$nargs == 1} {
        set attrName [lindex $args 0] ; # ❶
        my variable $attrName
        try {
            return [set $attrName]
        } on error {} {
            tailcall ::stsatcl::DeclError UNKNOWN_ATTRIBUTE $attrName
        }
    }

    set attrNames $args ; # ❷
    if {$nargs == 0} {
        classvariable attrInfo
        set attrNames [dict keys $attrInfo]
    }
    my variable {*}$attrNames ; # ❸
```

```
    set result [dict create]
    foreach attrName $attrNames {
        try {
            dict set result $attrName [set $attrName]
        } on error {} {
            tailcall ::stsatcl::DeclError UNKNOWN_ATTRIBUTE $attrName
        }
    }
    return $result
}
```

1. For a single attribute, just return the value. It's just easier to deal with the most common case in that way.
2. When requesting multiple attributes we will return a dictionary.
3. Bring all the named attributes into scope.

Read Attributes Tests

```
<<exported tests>>=
test readAttributes-1.0 {
    read single attribute value
} -setup {
    stsatcl STSAClass create test-ra1 {
        attribute Count {Maximum 27}
    }
} -cleanup {
    test-ra1 destroy
} -body {
    test-ra1 create inst1-ra1
    inst1-ra1 readAttributes Maximum
} -result {27}
```

```
<<exported tests>>=
test readAttributes-2.0 {
    read multiple attribute values
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst1-ra2
    inst1-ra2 readAttributes Maximum Timeout
} -result {Maximum 27 Timeout 1000}
```

```
<<exported tests>>=
test readAttributes-3.0 {
    read all attribute values
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst1-ra2
    inst1-ra2 readAttributes
} -result {Count {} Maximum 27 Timeout 1000}
```

```
<<exported tests>>=
test readAttributes-4.0 {
    read unknown attribute values
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst1-ra2
    inst1-ra2 readAttributes foo
} -result {unknown attribute, "foo"} -returnCodes error
```

```
<<exported tests>>=
test readAttributes-4.1 {
    read unknown attribute values from among many
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst1-ra2
    inst1-ra2 readAttributes Maximum foo Timeout
} -result {unknown attribute, "foo"} -returnCodes error
```

Updating Attribute Values

The `updateAttributes` method sets one or more attributes to the values given as arguments.

xumlobj `updateAttributes` *?attrname1 value1 attrname2 value 2... ?*

`attrname`
: The name of the attribute to be read.

`value`
: The corresponding value to assign to the attribute.

The return value is the empty string. It is an error to attempt to update an attrbute not defined for the class.

```
<<attribute access methods>>=
method updateAttributes {args} {
    classvariable attrInfo

    dict for {attribute value} $args {
        if {[dict exists $attrInfo $attribute]} {
            my variable $attribute
            set $attribute $value
        } else {
            tailcall ::stsatcl::DeclError UNKNOWN_ATTRIBUTE $attribute
        }
    }
    return
}
```

Update Attributes Tests

```
<<exported tests>>=
test updateAttributes-1.0 {
    update attribute value
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst-ra2
    inst-ra2 updateAttributes Count 33
    inst-ra2 readAttributes Count
} -result {33}
```

```
<<exported tests>>=
test updateAttributes-2.0 {
    update multiple attribute values
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst-ra2
    inst-ra2 updateAttributes Count 33 Maximum 50
    inst-ra2 readAttributes Count Maximum
} -result {Count 33 Maximum 50}
```

```
<<exported tests>>=
test updateAttributes-3.0 {
    unknown attribute
} -setup {
    stsatcl STSAClass create test-ra2 {
        attribute Count {Maximum 27}
    }
} -cleanup {
    test-ra2 destroy
} -body {
    test-ra2 create inst-ra2
    inst-ra2 updateAttributes Foo 42
} -result {unknown attribute, "Foo"} -returnCodes error
```

Linkage Methods

As we discussed above, the model notion of a relationship is implemented by one or more links. There are three fundamental operations on relationships:

1. Creating a relationship.
2. Destroying a relationship.
3. Navigating a relationship.

In this section we discuss the methods provide to XUML classes to operate on relationship linkage. When configuring an XUML class, the `reference` and `partition` commands are used to define object linkages that are the implementation realization of a relationship in the XUML model. Each class participating in the relationship that also uses the relationship in some activity need to have a link defined for it. This implies that creating or destroying the model notion of a relationship may require *linking* or *unlinking* in both objects that participate in the relationship.

Link Method

The `link` method is used to establish a relationship linkage between two XUML objects.

xumlobj `link` *rname target*

`rname`
: the name of the relationship link. This will be one of the link names given as an argument to either the `reference` or `partition` command.

`target`
: the object command name of the target object of the link. The *target* must be an instance of the class (or subclasses in the case of a `partition` link) defined for the link.

returns
: the fully qualified *target* command name.

```
<<linkage access methods>>=
method link {rname target} {
    set target [my ResolveObj $target] ; # ❶
    my CheckLinkName $rname
    my CheckReferenceObj $rname $target

    classvariable linkInfo
    my variable $rname
    set multiple [expr {[dict get $linkInfo $rname type] eq "reference" ?\
            [dict get $linkInfo $rname mult] : 0}] ; # ❷

    if {$multiple} {
        ::struct::set include $rname $target ; # ❸
    } else {
        if {[set $rname] eq {}} {
            set $rname $target
        } else {
            tailcall ::stsatcl::DeclError OVERLINK [self] [set $rname]\
                   $rname $target ; # ❹
        }
    }
    return $target
}
```

❶ We use a number of utility methods described below to handle object command name resolution and checking that the linkage exists and the target object is of the correct class.

❷ All we need to determine here is whether or not the linkage is singular or multiple. For partition links, they are always singular.

❸ For links with multiple cardinality, it is important that the list of target object commands be a set.

❹ For singular links, we insist that the link variable be empty, *i.e.* that it has been properly unlinked before we overwrite the new linkage value.

```
<<error code formats>>=
OVERLINK        {attempt to over-link: %s is currently linked to %s across %s,\
                 requested link to %s}
```

Link Tests

```
<<exported tests>>=
test link-1.0 {
    link across singular reference
} -setup {
    stsatcl STSAClass create test-lt1 {
        reference R1 -> test-lt2
    }
    stsatcl STSAClass create test-lt2 {
        reference R1 -> test-lt1
    }
} -cleanup {
    test-lt1 destroy
    test-lt2 destroy
} -body {
    test-lt1 create lt1
    test-lt2 create lt2
    lt1 link R1 lt2
} -result {::stsatcl::test::lt2}
```

```
<<exported tests>>=
test link-1.1 {
    overlink error
} -setup {
    stsatcl STSAClass create test-lt1 {
        reference R1 -> test-lt2
    }
    stsatcl STSAClass create test-lt2 {
        reference R1 -> test-lt1
    }
} -cleanup {
    test-lt1 destroy
    test-lt2 destroy
} -body {
    test-lt1 create lt1
    test-lt2 create lt2
    lt1 link R1 lt2
    lt1 link R1 lt2
} -result {attempt to over-link: ::stsatcl::test::lt1 is currently linked to  ↩
    ::stsatcl::test::lt2 across R1, requested link to ::stsatcl::test::lt2}\
-returnCodes error
```

```
<<exported tests>>=
test link-2.0 {
    link across multiple reference
} -setup {
    stsatcl STSAClass create test-lt1 {
        reference R2 ->> test-lt2
    }
    stsatcl STSAClass create test-lt2 {
        reference R2 -> test-lt1
    }
} -cleanup {
    test-lt1 destroy
    test-lt2 destroy
} -body {
    test-lt1 create inst1
    test-lt2 create inst2
    test-lt2 create inst3
    inst1 link R2 inst2
```

```
    inst1 link R2 inst3
    llength [set [info object namespace inst1]::R2]
} -result {2}
```

```
<<exported tests>>=
test link-2.1 {
    link across multiple reference, duplicate references
} -setup {
    stsatcl STSAClass create test-lt1 {
        reference R2 ->> test-lt2
    }
    stsatcl STSAClass create test-lt2 {
        reference R2 -> test-lt1
    }
} -cleanup {
    test-lt1 destroy
    test-lt2 destroy
} -body {
    test-lt1 create inst1
    test-lt2 create inst2
    inst1 link R2 inst2
    inst1 link R2 inst2
    llength [set [info object namespace inst1]::R2]
} -result {1}
```

```
<<exported tests>>=
test link-3.0 {
    link across partition
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R3 -> super
    }

    ::stsatcl STSAClass create sub2 {
        reference R3 -> super
    }
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    super create sup1
    sub1 create inst1
    sup1 link R3 inst1
    llength [set [info object namespace sup1]::R3]
} -result {1}
```

Unlink Method

A link between objects is destroyed by invoking the `unlink` method.

xumlobj `unlink` *rname target*

`rname`
: the name of the relationship link. This will be one of the link names given as an argument to either the `reference` or `partition` command.

target
: the object command name of the target object of the link. The *target* must be an instance that is currently linked across *rname*.

```
<<linkage access methods>>=
method unlink {rname target} {
    set target [my ResolveObj $target]
    my CheckLinkName $rname
    my CheckReferenceObj $rname $target

    classvariable linkInfo
    my variable $rname
    switch -exact -- [dict get $linkInfo $rname type] {
        reference {
            set multiple [dict get $linkInfo $rname mult]
            set match [::struct::set contains [set $rname] $target]
        }
        partition {
            set multiple 0
            set match [expr {$target eq [set $rname]}]
        }
        default {
            tailcall ::stsatcl::DeclError UNKNOWN_LINKTYPE\
                    [dict get $linkInfo $rname type]
        }
    }
    if {$match} {
        if {$multiple} {
            ::struct::set exclude $rname $target
        } else {
            set $rname {}
        }
    } else {
        tailcall ::stsatcl::DeclError NOTLINKED $target $rname
    }

    return
}
```

```
<<error code formats>>=
UNKNOWN_LINKTYPE {unknown linkage type, "%s"}
NOTLINKED       {object, "%s", is not linked to "%s"}
```

Unlink Tests

```
<<exported tests>>=
test unlink-1.0 {
    unlink across singular reference
} -setup {
    stsatcl STSAClass create test-ul1 {
        reference R1 ->c test-ul2
    }
    stsatcl STSAClass create test-ul2 {
        reference R1 -> test-ul1
    }
} -cleanup {
    test-ul1 destroy
    test-ul2 destroy
} -body {
```

```
    test-ul1 create inst-ul1
    test-ul2 create inst-ul2
    inst-ul1 link R1 inst-ul2
    inst-ul1 unlink R1 inst-ul2
    set [info object namespace inst-ul2]::R1
} -result {}
```

```
<<exported tests>>=
test unlink-2.0 {
    unlink across multiple reference
} -setup {
    stsatcl STSAClass create test-ul3 {
        reference R1 ->> test-ul4
    }
    stsatcl STSAClass create test-ul4 {
        reference R1 -> test-ul3
    }
} -cleanup {
    test-ul3 destroy
    test-ul4 destroy
} -body {
    test-ul3 create inst1
    test-ul4 create inst2
    test-ul4 create inst3

    inst1 link R1 inst2
    inst1 link R1 inst3
    inst1 unlink R1 inst2
    llength [info object namespace inst1]::R1
} -result {1}

test unlink-3.0 {
    unlink across partition
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R3 -> super
    }

    ::stsatcl STSAClass create sub2 {
        reference R3 -> super
    }
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    super create inst-super
    sub1 create inst-sub1
    inst-super link R3 inst-sub1
    inst-super unlink R3 inst-sub1
    set [info object namespace inst-super]::R3
} -result {}
```

Link Navigation Method

The -> method is used to navigate one or more links.

xumlobj -> rname1 ?rname2 ... ?

rname
: The name of a relationship link. If *rname* refers to a *partition* link and the navigation is from the superclass to the subclass, then *rname* must be followed by the name of the subclass at which the navigation is directed.

There are two particular points in navigating links. First, references and partition have different characteristics in the navigation. A partition, when navigated from superclass to subclass[6] must specify which subclass of the partition is to be navigated and consequently whether the currently linked instance is of the specified subtype. Second, we want to be able to have a chain of links specified and so navigation can be seen as a recursive operation without having to nest command invocations.

```
<<linkage access methods>>=
method -> {rname args} {
    classvariable linkInfo
    my CheckLinkName $rname

    my variable $rname
    set relobjs [set $rname]

    switch -exact -- [dict get $linkInfo $rname type] {
        reference {
            if {![dict get $linkInfo $rname cond] && [llength $relobjs] == 0} {
                tailcall ::stsatcl::DeclError UNCOND $rname
            }
        }
        partition {
            if {$relobjs eq {}} {
                tailcall ::stsatcl::DeclError UNCOND $rname
            }
            if {[llength $args] < 1} { # ❶
                tailcall ::stsatcl::DeclError NOSUBCLASS $rname
            }

            set subclass [lindex $args 0]
            set args [lrange $args 1 end]

            set subclass [my ResolveObj $subclass]
            if {$subclass ni [dict get $linkInfo $rname subclasses]} {
                tailcall ::stsatcl::DeclError NOTSUBCLASS $subclass $rname
            }
            if {[llength [info class instances $subclass $relobjs]] == 0} {
                set relobjs [list]
            }
        }
        default {
            tailcall ::stsatcl::DeclError UNKNOWN_LINKTYPE\
                    [dict get $linkInfo $rname type]
        }
    }

    if {[llength $args] != 0} {
        set related [list]
        foreach robj $relobjs {
            ::struct::set add related [$robj -> {*}$args] ; # ❷
        }
    } else {
        set related $relobjs
    }

    return [expr {[llength $related] < 2 ? [lindex $related 0] : $related}] ; #❸
```

[6] Navigating from subclass to superclass us just an unconditional singular link with no special characteristics

```
}
export ->
```

```
<<error code formats>>=
UNCOND      {reference, "%s", is unconditional, yet no instances were found}
NOSUBCLASS  {navigation of "%s" from supertype to subtype\
             requires the destination subtype class}
NOTSUBCLASS {subclass, "%s", is not a subclass of partition "%s"}
```

1. When navigating from a superclass to a subclass we must state which subclass of the partition we are interested in and so an extra argument is required.
2. The instances related by a link form a set.
3. We return a single scalar value when possible. argument must be supplied.

Traversal Tests

```
<<exported tests>>=
test traverse-1.0 {
    traverse a singular relationship
} -setup {
    stsatcl STSAClass create rt1 {
        reference R1 -> rt2
    }
    stsatcl STSAClass create rt2 {
        reference R1 -> rt1
    }
} -cleanup {
    rt1 destroy
    rt2 destroy
} -body {
    rt1 create inst1
    rt2 create inst2
    inst1 link R1 inst2
    inst1 -> R1
} -result {::stsatcl::test::inst2}
```

```
<<exported tests>>=
test traverse-1.1 {
    traverse multiple singular relationships
} -setup {
    stsatcl STSAClass create rt1 {
        reference R1 -> rt2
    }
    stsatcl STSAClass create rt2 {
        reference R1 -> rt1
    }
} -cleanup {
    rt1 destroy
    rt2 destroy
} -body {
    rt1 create inst1
    rt2 create inst2
    inst1 link R1 inst2
    inst2 link R1 inst1

    inst1 -> R1 R1
} -result {::stsatcl::test::inst1}
```

```
<<exported tests>>=
test traverse-1.2 {
    traverse many singular relationships -- violate conditionality
} -setup {
    stsatcl STSAClass create rt1 {
        reference R1 -> rt2
    }
    stsatcl STSAClass create rt2 {
        reference R1 -> rt1
    }
} -cleanup {
    rt1 destroy
    rt2 destroy
} -body {
    rt1 create inst1
    rt2 create inst2
    inst1 link R1 inst2
    inst1 -> R1 R1
} -result {reference, "R1", is unconditional, yet no instances were found}\
  -returnCodes error
```

```
<<exported tests>>=
test traverse-2.0 {
    traverse a multiple relationship
} -setup {
    stsatcl STSAClass create rt1 {
        reference R2 ->> rt2
    }
    stsatcl STSAClass create rt2 {
        reference R2 -> rt1
    }
} -cleanup {
    rt1 destroy
    rt2 destroy
} -body {
    rt1 create inst1
    rt2 create inst2
    rt2 create inst3
    inst1 link R2 inst2
    inst1 link R2 inst3
    inst1 -> R2
} -result {::stsatcl::test::inst2 ::stsatcl::test::inst3}
```

```
<<exported tests>>=
test reference-2.1 {
    traverse a multiple relationship, then a singular
} -setup {
    stsatcl STSAClass create rt1 {
        reference R2 ->> rt2
    }
    stsatcl STSAClass create rt2 {
        reference R2 -> rt1
    }
} -cleanup {
    rt1 destroy
    rt2 destroy
} -body {
    rt1 create inst1
    rt2 create inst2
    rt2 create inst3
    inst1 link R2 inst2
```

```
    inst1 link R2 inst3
    inst2 link R2 inst1
    inst3 link R2 inst1

    inst1 -> R2 R2
} -result {::stsatcl::test::inst1}
```

```
<<exported tests>>=
test traverse-3.0 {
    traverse across partition
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R3 -> super
    }

    ::stsatcl STSAClass create sub2 {
        reference R3 -> super
    }
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    super create sup1
    sub1 create inst1

    sup1 link R3 inst1
    sup1 -> R3 sub1
} -result {::stsatcl::test::inst1}
```

```
<<exported tests>>=
test reference-3.1 {
    traverse across partition and single reference
} -setup {
    ::stsatcl STSAClass create super {
        partition R10 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R10 -> super
        reference R11 -> c3
    }

    ::stsatcl STSAClass create sub2 {
        reference R10 -> super
    }
    ::stsatcl STSAClass create c3 {
        reference R11 -> sub1
    }
    super create sup1
    sub1 create s1
    c3 create inst3
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
    c3 destroy
```

```
} -body {
    sup1 link R10 s1
    s1 link R10 sup1
    s1 link R11 inst3
    inst3 link R11 s1
    sup1 -> R10 sub1 R11
} -result {::stsatcl::test::inst3}
```

```
<<exported tests>>=
test reference-3.2 {
    traverse across partition and single reference -- come up empty
} -setup {
    ::stsatcl STSAClass create super {
        partition R10 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R10 -> super
        reference R11 -> c3
    }

    ::stsatcl STSAClass create sub2 {
        reference R10 -> super
    }
    ::stsatcl STSAClass create c3 {
        reference R11 -> sub1
    }
    super create sup1
    sub1 create s1
    c3 create inst3
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
    c3 destroy
} -body {
    sup1 link R10 s1
    s1 link R10 sup1
    s1 link R11 inst3
    inst3 link R11 s1
    sup1 -> R10 sub2 R11
} -result {}
```

```
<<exported tests>>=
test traverse-3.3 {
    traverse across partition -- wrong subclass
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        reference R3 -> super
    }

    ::stsatcl STSAClass create sub2 {
        reference R3 -> super
    }
} -cleanup {
    super destroy
    sub1 destroy
```

```
    sub2 destroy
} -body {
    super create sup1
    sub1 create inst1

    sup1 link R3 inst1
    sup1 -> R3 sub3
} -result {subclass, "::stsatcl::test::sub3",\
        is not a subclass of partition "R3"} -returnCodes error
```

Migrate Method

The `migrate` method provides the common operation of migrating one of the subclass instances in a partition to a different subclass. Conceptually, this is accomplished by:

1. Unlink in the subclass instance.
2. Destroy the unlinked instance.
3. Create a new instance of the required subclass.
4. Link the new subclass instance to the superclass instance.

xumlobj `migrate` *rname subclass ?attr1 value1 attr2 value2 . . . ?*

`rname`
: the name of a partition link. Subclass migration is only defined for partition links.

`subclass`
: the name of the subclass to which the migration will occur. The *subclass* must be one defined for the *rname* partition.

`attr value ...`
: attribute name / attribute value pairs that given as arguments to the new constructed subclass instance.

```
<<linkage access methods>>=
method migrate {rname subclass args} {
    set subclass [my ResolveObj $subclass]
    my CheckLinkName $rname

    classvariable linkInfo
    if {[dict get $linkInfo $rname type] ne "partition"} {
        tailcall ::stsatcl::DeclError NOTPARTITION $rname
    }
    if {$subclass ni [dict get $linkInfo $rname subclasses]} {
        tailcall ::stsatcl::DeclError NOTSUBCLASS $subclass $rname
    }

    my variable $rname
    set subinst [set $rname]
    if {$subinst eq {}} {
        tailcall ::stsatcl::DeclError UNCOND $rname
    }
    $subinst destroy

    set $rname [$subclass new {*}$args]
}
```

```
<<error code formats>>=
NOTPARTITION    {linkage "%s" is not a partition}
```

Migrate Tests

```
<<exported tests>>=
test migrate-1.0 {
    migrate subclasses
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        attribute {A1 20}
    }

    ::stsatcl STSAClass create sub2 {
        attribute {A1 30}
    }
    super create sup1
    sub1 create inst1
    sup1 link R3 inst1
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    sup1 migrate R3 sub2 A1 50
    # Traverse the partition from superclass to subclass
    # to obtain the attribute value. Demonstrates the link was made.
    [sup1 -> R3 sub2] readAttributes A1
} -result {50}
```

```
<<exported tests>>=
test migrate-2.0 {
    migrate subclasses -- not a partition
} -setup {
    ::stsatcl STSAClass create fake {
        reference R32 -> baz
    }
    ::stsatcl STSAClass create baz

    fake create i1
} -cleanup {
    fake destroy
    baz destroy
} -body {
    i1 migrate R32 sub2
} -result {linkage "R32" is not a partition} -returnCodes error
```

```
<<exported tests>>=
test migrate-3.0 {
    migrate subclasses -- unknown subclass
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
    }

    ::stsatcl STSAClass create sub1 {
        attribute {A1 20}
    }

    ::stsatcl STSAClass create sub2 {
```

```
        attribute {A1 30}
    }
    super create sup1
    sub1 create inst1
    sup1 link R3 inst1
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    sup1 migrate R3 sub3
} -result {subclass, "::stsatcl::test::sub3",\
        is not a subclass of partition "R3"} -returnCodes error
```

```
<<exported tests>>=
test migrate-4.0 {
    migrate subclasses -- missing link
} -setup {
    ::stsatcl STSAClass create super {
        partition R5 sub1 sub2
    }
    super create sup1
} -cleanup {
    super destroy
} -body {
    sup1 migrate R5 sub2
} -result {reference, "R5", is unconditional, yet no instances were found}\
    -returnCodes error
```

Example Initial Instance Population

In Tcl, all the XUML class instances are created at “run time” since there is no concept of “compile time” in Tcl. However, most domain models benefit from being able to assume that some population of instances exists when the domain starts running. This collection of instances that is put into place before the system is started is known as the *initial instance population*. For our simple example, the initial instance population is the entire population of instances. Now that we know how to create instances and link objects in relationships we can define that population here.

We start with Washing Cycles. We’ll define a few common washing use cases and set the WashingCycle attributes to match those use cases.

```
<<initial instance population>>=
WashingCycle create normal\
    CycleType Normal\
    WashWaterTemp Cold\
    RinseWaterTemp Cold\
    WashDuration 20\
    RinseDuration 10\
    SpinDuration 10\
    AgitationSpeed Medium\
    SpinSpeed Medium
WashingCycle create whites\
    CycleType Whites\
    WashWaterTemp Hot\
    RinseWaterTemp Cold\
    WashDuration 20\
    RinseDuration 10\
    SpinDuration 20\
    AgitationSpeed High\
    SpinSpeed High
WashingCycle create permpress\
```

```
    CycleType PermPress\
    WashWaterTemp Warm\
    RinseWaterTemp Cold\
    WashDuration 15\
    RinseDuration 10\
    SpinDuration 15\
    AgitationSpeed Medium\
    SpinSpeed Medium
WashingCycle create delicate\
    CycleType Delicate\
    WashWaterTemp Cold\
    RinseWaterTemp Cold\
    WashDuration 15\
    RinseDuration 10\
    SpinDuration 10\
    AgitationSpeed Low\
    SpinSpeed Low
```

We intend to have only a single WashingMachine instance but note that the model will run correctly regardless of how many washing machines we are trying to control. We can also link up our washer with its initial washingCycle.

```
<<initial instance population>>=
WashingMachine create washer MachineID WasherOne
washer link R4 normal
```

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

```
<<initial instance population>>=
ClothesTub create tub
washer link R1 tub      ; # ❶
tub link R1 washer
```

❶ We must link `R1` from both sides. Instances of both classes *navigate* `R1` during their state activities and so the link appears in the definitions of both participating classes.

Each washing machine, by design, has three motors to pump water, agitate the tub and to spin the tub.

```
<<initial instance population>>=
Motor create pump MotorID Pump
Motor create agitator MotorID Agitator
Motor create spin MotorID Spin

tub link R2 pump
tub link R2 agitator
tub link R2 spin
```

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

```
<<initial instance population>>=
WaterValve create hot ValveID Hot
WaterValve create cold ValveID Cold
WaterValve create drain ValveID Drain

tub link R3 hot
tub link R3 cold
tub link R3 drain
```

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

```
<<initial instance population>>=
WaterLevelSensor create sensor

tub link R5 sensor
sensor link R5 tub
```

Instance Selection

In this section we describe the methods provided to search for class instances. There are two types of searches that are quite general and are provided by the package.

1. Selecting from all instances of a class.
2. Selecting from related instances of a class.

To find an instance set from among all the instances of a class is a method on the XUML class itself. Finding related instances is a method on XMUL objects.

Note

The searches here are simple linear searches and are entirely suitable for a small number of instances. More sophisticated searches can be implemented by creating and maintaining indices or other techniques. However, those techniques are beyond that provided by this package. Constructing more sophisticated instance selection can be done by adding class-based operations and carefully building the required auxiliary data structures. It remains for a future enhancement to add the ability to use a more sophisticated search index scheme.

The methods in this section use a `tcllib` package to do the actual work of evaluating the selection critierion and building the resulting list of instances.

```
<<required packages>>=
package require struct::list
```

Select Where Method

xumlclass `selectWhere` *varname expr*

`varname`
: The name of a variable to which each object command from *xumlclass* is assigned during the search.

`expr`
: An expression that is evaluated by the `::expr` command.

The `selectWhere` method is defined on each XUML class. The method iterates over each instance of the *xumlclass* and assigns the object command to the variable given by *varname* and evaluates the expression given by *expr*. The result of evaluating *expr* is interpreted as a boolean and if true, the instance command name is included in the returned list of instance commands.

```
<<select instance methods>>=
method selectWhere {varname expr} {
    tailcall ::struct::list filterfor $varname [info class instances [self]]\
            $expr ; # ❶
}
```

❶ Note the use of the `tailcall` command here. The `filterfor` procedure will evaluate the expression in the stack level of the caller. Using `tailcall` insures that another stack level is *not* introduced so that the expression is evaluated in the stack level of the caller of the `selectWhere` method, which is exactly what we want.

Select Where Tests

```
<<exported tests>>=
test selectWhere-1.0 {
    find instances based on criteria
} -setup {
    ::stsatcl STSAClass create sw1 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    sw1 destroy
} -body {
    sw1 create inst1
    sw1 create inst2 Maximum 30 Timeout 1000
    sw1 create inst3 Timeout 2000
    set min 1000
    sw1 selectWhere inst {
        [$inst readAttributes Timeout] == $min
    }
} -result {::stsatcl::test::inst1 ::stsatcl::test::inst2}
```

Select One Where Method

There are times when it is known, *a priori*, that the return of a selection operation will yield exactly one result or only one is desired in any case. Since `selectWhere` returns a list, we use the `selectOneWhere` method to pick the first result off of the list.

xumlclass `selectOneWhere` *varname expr*

```
<<select instance methods>>=
method selectOneWhere {varname expr} {
    tailcall lindex [uplevel 1 [list ::struct::list filterfor $varname\
        [info class instances [self]] $expr]] 0 ; # ❶
}
```

❶ Okay, this is psychedelic but here's what's going on. We want to run `::struct::list filterfor` to do the heavy lifting for selecting based on some expression. No reason to duplicate all that well tested and often run logic here. We are filtering a list of class instances the XUML class itself. We need to run the selection in the stack frame of the caller (`uplevel 1`) so we get any substitution of variables, *etc.* that might be referenced in the *expr* to happen there. Note we hand `uplevel` a proper list built with the `list` command. Since we are doing variable substitution of the method arguments we don't want to get confused with embedded spaces as might happen if we just let `uplevel` concatenate its arguments into a script. Finally, we just take the first element of the returned list and since this is the end of the method, we can `tailcall` directly into `lindex`. Thus, we get the `selectOne` quality we are after. And yes some local variables might have made it easier to read, but hey! you got the nice comment.

```
<<exported tests>>=
test selectOneWhere-1.0 {
    find a single instance based on criteria
} -setup {
    ::stsatcl STSAClass create sw1 {
        attribute Count {Maximum 27} {Timeout 1000}
    }
} -cleanup {
    sw1 destroy
} -body {
    sw1 create inst1
    sw1 create inst2 Maximum 30 Timeout 1000
    sw1 create inst3 Timeout 2000
```

```
    set min 1000
    sw1 selectOneWhere inst {
        [$inst readAttributes Timeout] == $min
    }
} -result {::stsatcl::test::inst1}
```

Select Related Where Method

The `selectRelatedWhere` method performs a selection based on the set of instances found by navigating one more links.

xumlobj `selectRelatedWhere` *links varname expr*

`links`
: a list of links representing a navigation chain as described in the → method.

`varname`
: the name of a variable

`expr`
: a Tcl expression as defined by the `expr` core command.

The `selectRelatedWhere` command iterates over the set of object command instances that are found by navigating the *links* chain of linkages. Each object command is assigned to the variable named, *varname*, and *expr* is evaluated. The result of the evaluation is interpreted as a boolean to indicate whether the object command name is to be included in the returned result set.

```
<<select related methods>>=
method selectRelatedWhere {rlist varname expr} {
    tailcall ::struct::list filterfor $varname [my -> {*}$rlist] $expr
}
```

Select Related Where Tests

```
<<exported tests>>=
test selectRelatedWhere-1.0 {
    select related instances based on criteria
} -setup {
    ::stsatcl STSAClass create frw1 {
        attribute Count {Maximum 27} {Timeout 1000}
        reference R1 ->> frw2
    }
    ::stsatcl STSAClass create frw2 {
        attribute Reason Temp Pressure Current
    }

    frw1 create i1

    frw2 create c1 Temp 20
    frw2 create c2 Temp 30
    frw2 create c3 Temp 40
    frw2 create c4 Temp 50

    i1 link R1 c1
    i1 link R1 c2
    i1 link R1 c3
    i1 link R1 c4
} -cleanup {
    frw1 destroy
    frw2 destroy
```

```
} -body {
    set targetTemp 30
    i1 selectRelatedWhere R1 selected {
        [$selected readAttributes Temp] > $targetTemp
    }
} -result {::stsatcl::test::c3 ::stsatcl::test::c4}
```

```
<<exported tests>>=
test selectRelatedWhere-2.0 {
    select instances related by multiple links
} -setup {
    ::stsatcl STSAClass create frw1 {
        attribute Count {Maximum 27} {Timeout 1000}
        reference R1 ->> frw2
    }
    ::stsatcl STSAClass create frw2 {
        reference R2 ->> frw3
    }
    ::stsatcl STSAClass create frw3 {
        attribute Reason Temp Pressure Current
    }

    frw1 create i1

    frw2 create m1
    frw2 create m2
    i1 link R1 m1
    i1 link R1 m2

    # These two will be on the chain, but not match the criterion
    frw3 create c1 Temp 20
    frw3 create c2 Temp 30
    m1 link R2 c1
    m1 link R2 c2

    # These two will match the criterion
    frw3 create c3 Temp 40
    frw3 create c4 Temp 50
    m2 link R2 c3
    m2 link R2 c4
} -cleanup {
    frw1 destroy
    frw2 destroy
    frw3 destroy
} -body {
    set targetTemp 30
    i1 selectRelatedWhere {R1 R2} selected {
        [$selected readAttributes Temp] > $targetTemp
    }
} -result {::stsatcl::test::c3 ::stsatcl::test::c4}
```

Select One Related Where Method

By analogy to `selectOneWhere` the `selectOneRelatedWhere` method is provided.

xumlobj `selectOneRelatedWhere` *links varname expr*

```
<<select related methods>>=
method selectOneRelatedWhere {rlist varname expr} {
    tailcall lindex [uplevel 1\
```

```
        [list ::struct::list filterfor $varname [my -> {*}$rlist] $expr]] 0 ; # ❶
}
```

❶ See the selectOneWhere code for the details of what is going on here.

Instance Computation Methods

In this section we show methods that, like the instance selection methods above, iterate over instances. For these methods we provide a means to compute on each of the instances and thereby build up higher order summaries of data. The fundamental capablities are provided by the `lmap` command.

Map Instances Method

xumlclass `mapInstances` *varname script*

`varname`
The name of a variable to which each instance of *xumclass* is sequentially assigned.

`script`
A Tcl script that is evaluated once for each instance of *xumlclass*.

The `mapInstances` method iterates across all the instances of *xumlclass*, assigning the instance command name to *varname*. Then *script* is evaluated and the result of the that evaluation is added to an accumlator list. The return value of method is the accumlated list of script results.

```
<<map instance methods>>=
method mapInstances {varname script} {
    uplevel 1 [list ::lmap $varname [info class instances [self]] $script]
}
```

Map Instances Tests

```
<<exported tests>>=
test mapInstances-1.0 {
    iterate on instances to produce an attribute list
} -setup {
    ::stsatcl STSAClass create fw1 {
        attribute Count Maximum Timeout
    }
} -cleanup {
    fw1 destroy
} -body {
    fw1 create inst1 Maximum 20
    fw1 create inst2 Maximum 30
    fw1 create inst3 Maximum 40

    set divisor 10
    fw1 mapInstances inst {
        expr {[$inst readAttributes Maximum] / $divisor}
    }
} -result {2 3 4}
```

Map Related Instances Method

The `mapRelatedInstances` command operates like the `mapInstances` command except the iteration happens only on those instances that are related to a *xumlobj* by some relationship chain.

xumlobj `mapRelatedWhere` *links varname script*

`links`
a list of links representing a navigation chain as described in the → method.

`varname`
The name of a variable to which each instance of *xumclass* is sequentially assigned.

`script`
A Tcl script that is evaluated once for each instance of *xumlclass*.

```
<<map related methods>>=
method mapRelatedInstances {rlist varname script} {
    uplevel 1 [list ::lmap $varname [my -> {*}$rlist] $script]
}
```

Map Related Instances Tests

```
<<exported tests>>=
test mapRelatedInstances-1.0 {
    iterate on related instances to produce a list
} -setup {
    ::stsatcl STSAClass create frw1 {
        attribute Count {Maximum 27} {Timeout 1000}
        reference R1 ->> frw2
    }
    ::stsatcl STSAClass create frw2 {
        attribute Reason Temp Pressure Current
    }

    frw1 create i1

    frw2 create c1 Temp 20
    frw2 create c2 Temp 30
    frw2 create c3 Temp 40
    frw2 create c4 Temp 50

    i1 link R1 c1
    i1 link R1 c2
    i1 link R1 c3
    i1 link R1 c4
} -cleanup {
    frw1 destroy
    frw2 destroy
} -body {
    set divisor 10
    i1 mapRelatedInstances R1 inst {
        expr {[$inst readAttributes Temp] / $divisor}
    }
} -result {2 3 4 5}
```

State Machine Execution

At long last we will now discuss how state machine execution operates.

We must start by clarifying some terms. The term **event** is heavily overloaded in the Tcl world and in most of the computing world. It can be used to describe Tk window events or used in conjunction with event oriented programming concepts. Here we will overload the term one more time and discuss state machine events. We will use the term **state machine event** to denote a particular named occurrence of an event that can drive the transitions of a state machine. The state model execution rules need to be made explicit.

- The state machine formalism is of the Moore type. type. Activities associated with states are executed when the state is entered. There is no support for the Mealy formalism and no support for hierarchical or other composed forms of state models[7].
- There are three types of state machine events, ordinary transitioning events (or just simply events), polymorphic events and creation events.
- Events signalled to an instance are not lost. The event will be delivered at the next available opportunity.
- The order of event dispatch is not specified, however, events signalled by one instance to another are received in the order they were generated.
- Events signalled by an instance to itself are dispatched before events that are signalled from a different instance.
- Events and polymorphic events may be *delayed*, *i.e.* you may request the system to signal events at some time in the future.
- There may be only one outstanding delayed event of a given event name between any sending / receiving pair of instances.
- Polymorphic events can only be signalled to instances that have defined a `partition` link. A polymorphic event is signalled down the generalization hierarchy to the currently related subclass instance. A polymorphic event sent to a superclass instance propagates down **all** the generalization hierarchies for which the instance is a superclass.
- Creation events are signalled to the class and result in an instance of that class being created in its initial pseudo-state and an event signalled to the newly created instance.

One of the important properties of state activities is that they *execute to completion*. During the execution of the activity of a state, signalling a state machine event does *not* cause any side effects on the state of the executing instance and does not immediately cause any transition in the target state machine. Also, this architecture is single threaded, so there is no concurrency implied by the state machine events. Concurrency can be achieved by following the Tcl threading model which supports the notion of multiple concurrent interpreters.

From a design point of view, we must decide how to implement the above rules. The execute-to-completion rule can be accomplished using a queue for the events. In this design, each XUML class instance has its own event queue for non-delayed events. Signalling events to an instance will result in inserting an event on the event queue associated with the target instance.

Another important consideration is how to drive the dispatch of events on the instance queues. Fortunately, the `after` command provides us with all we need. For non-delayed signalling, we will use the `::after 0` command idiom to queue up Tcl event that can be used for state machine event dispatch. For delayed signalling, we use the `::after` *ms* form of the after command.

At instance construction time, we must set up some variables to use to track the event dispatch.

```
<<binding state machine events>>=
my variable __event_queue__
set __event_queue__ [list] ; # ❶
# We also need to keep track of the timer ID's for the non-delayed signals.
my variable __signals__
array set __signals__ {}
# Delayed signals require some additional bookkeeping.
my variable __delayedSignals__
array set __delayedSignals__ {}
```

[7] The Mealy formalism is mathematically equivalent to the Moore formalism. Hierarchical state models are an unnecessary abomination.

❶ We keep a queue of events as a simple list.

At object destruction time, we must clear up any allocated timer resources.

```
<<unbinding state machine events>>=
my variable __signals__ ; # ❶
foreach {sigid timerid} [array get __signals__] {
    after cancel $timerid
}
my variable __delayedSignals__
foreach {sigid sigdetails} [array get __delayedSignals__] {
    after cancel [dict get $sigdetails timerid]
}
```

❶ Both non-delayed and delayed evnts are ultimately mapped to timer events via the `::after` command so we must cancel anything that is currently outstanding.

Signal Method

The primary operation on state machines is to generate events to them and that is accomplished by the `signal` method.

xumlobj `signal` *event ?param1 param2 ... ?*

`event`
The name of the event to be signalled.

`param1 param2 ...`
Optional parameters given to the state activity.

The `signal` method generates *event* to *xumlobj* passing along any parameter arguments. The state activity that is invoked by the transition caused by *event* receives the values of the *paramN* arguments. State activities have a signature that defines the formal arguments that they expect. It is a rule of the Moore formalism that all events that cause a transition *into* a given state must carry parametric data values that match those expected by the state activity signature. In this architecture, a run time error is generated if the arguments don't match.

The implementation of the `signal` method has two major considerations. First, we must determine if the signalled event is polymorphic or not. Polymorphic events are just forwarded on to the related subclass instance. Eventually, the polymorphic event arrives at a subclass instance where it is consumed as an ordinary event. Second for ordinary events, we must place the event into the event queue. It is here we enforce the rule that self-directed events are dispatched before any events generated by other instances.

```
<<state model methods>>=
method signal {event args} {
    set src [my EventSource]
    classvariable polyEvents
    if {$event in $polyEvents} {
        my ForwardPolyEvent signal $src $event $args
    } else {
        my ValidateEvent $event
        set eventInfo [dict create\
            src $src\
            event $event\
            params $args\
        ]
        my variable __event_queue__
        # Self directed events are queued to the front.  However, we handle the
        # case where a state activity may signal multiple self-directed events.
        # In that, admittedly unusual case, we want to preserve the order of
        # the signalled events.
```

```
        if {$src eq [self]} {
            for {set nsindex 0} {$nsindex < [llength $__event_queue__]}\
                    {incr nsindex} { # ❶
                set cmpevent [lindex $__event_queue__ $nsindex]
                if {[dict get $cmpevent src] ne [self]} {
                    break ; # ❷
                }
            }
            set __event_queue__ [linsert $__event_queue__ $nsindex $eventInfo]
        } else {
            lappend __event_queue__ $eventInfo ; # ❸
        }
        my variable __event_counter__
        incr __event_counter__
        set timerid [::after 0 [mymethod Dispatch $__event_counter__]]
        my variable __signals__
        set __signals__($__event_counter__) $timerid
    }
}
```

❶ We iterate through the event queue using an index because we want to use `linsert` to place the event in the correct location.

❷ Stop the iteration at the first event that is non-self directed. The `linsert` command inserts the item into a list *before* its argument index, which is just what we want.

❸ Non-self directed events are simply appended to the event queue.

Signal Method Tests

```
<<exported tests>>=
test signal-1.0 {
    signal an event from outside of a state machine context
} -setup {
    ::stsatcl STSAClass create sg1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    sg1 destroy
} -body {
    sg1 create inst1
    inst1 signal e1 20
    namespace upvar [info object namespace inst1] __event_queue__ evtqueue
    dict get [lindex $evtqueue 0] src
} -result {}
```

```
<<exported tests>>=
test signal-2.0 {
    signal an event from inside of a state machine context
} -setup {
    ::stsatcl STSAClass create sg1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
```

```
            }
            transition s1 - e1 -> s1
        }
    }
    ::stsatcl STSAClass create sg2 {
        reference R1 -> sg1
        instop sig_sg1 {event args} {
            [my -> R1] signal $event {*}$args
        }
    }
} -cleanup {
    sg1 destroy
    sg2 destroy
} -body {
    sg1 create inst1
    sg2 create inst2
    inst2 link R1 inst1
    inst2 sig_sg1 e1 20

    namespace upvar [info object namespace inst1] __event_queue__ evtqueue
    dict get [lindex $evtqueue 0] src
} -result {::stsatcl::test::inst2}
```

```
<<exported tests>>=
test signal-3.0 {
    signal multiple self directed events
} -setup {
    ::stsatcl STSAClass create sg1 {
        attribute {Count 0}
        statemodel {
            state s1 {} {
                my variable Count
                incr Count
                my signal e1
                my signal e2
            }
            transition s1 - e1 -> s1

            state s2 {} {
                my variable Count
                incr Count
            }
            transition s2 - e2 -> s1
        }
    }
    ::stsatcl STSAClass create sg2 {
        reference R1 -> sg1
        instop sig_sg1 {event} {
            [info object namespace [my -> R1]]::my Receive\
                [dict create src {} event e1 params {}]
            [my -> R1] signal e1
        }
    }
} -cleanup {
    sg1 destroy
    sg2 destroy
} -body {
    sg1 create inst1
    sg2 create inst2
    inst2 link R1 inst1
    inst2 sig_sg1 e1
```

```
    namespace upvar [info object namespace inst1] __event_queue__ evtqueue
    set checklen [expr {[llength $evtqueue] == 3}]
    set check1 [expr {[dict get [lindex $evtqueue 0] event] eq "e1"}]
    set check2 [expr {[dict get [lindex $evtqueue 1] event] eq "e2"}]
    set check3 [expr {[dict get [lindex $evtqueue 2] src] eq\
            "::stsatcl::test::inst2"}]
    expr {$checklen && $check1 && $check2 && $check3}
} -result {1}
```

```
<<exported tests>>=
test signal-4.0 {
    signal a polymorphic event
} -setup {
    ::stsatcl STSAClass create super {
        partition R3 sub1 sub2
        polymorphic *e1
    }

    ::stsatcl STSAClass create sub1 {
        reference R3 -> super
        attribute {Count 0}
        statemodel {
            state sb1 {} {
                my variable Count
                incr Count
            }
            transition sb1 - *e1 -> sb1
        }
    }

    ::stsatcl STSAClass create sub2 {
        reference R3 -> super
        attribute {Count 0}
        statemodel {
            state sb1 {} {
                my variable Count
                incr Count
            }
            transition sb1 - *e1 -> sb1
        }
    }
} -cleanup {
    super destroy
    sub1 destroy
    sub2 destroy
} -body {
    super create sup1
    sub1 create inst1
    sup1 link R3 inst1
    inst1 link R3 sup1

    sup1 signal *e1
    namespace upvar [info object namespace inst1] __event_queue__ evtqueue
    dict get [lindex $evtqueue 0] src
} -result {::stsatcl::test::sup1}
```

Delayed Signal Method

The `delayedSignal` method requests an event to be signalled at some time in the future.

xumlobj `delayedSignal` *time event ?param1 param2 . . . ?*

`time`
The minimum number of milliseconds in the future when the event will be signalled. This value must be non-negative.

`event`
The name of the event to be signalled.

`param1 param2 ...`
Optional parameters given to the state activity.

There are several design considerations here. Clearly, we have to have access to some timing resource. The `after` command provides basic time driven events. However, there are two other operations on delayed signals that must be considered. It is possible to cancel a delayed signal. This is conveniently mapped to the `after cancel` command. It is also necessary to be able to determine the amount of time remaining before a delayed signal is to be generated. For this function, we have to provide some additional machinery.

We must also enforce the rule that there can only be one outstanding delayed signal for a given event name between any sending / receiving pair of instances. The first complication arises in determining what to do if a duplicate request is made. In this architecture we interpret an attempt to have a duplicated delayed event as a request to cancel the first signal and reinstate the signal at the newly requested time. This turns out to be the most common intent in practice and it prevents have to deal with a potential error situation.

```
<<state model methods>>=
method delayedSignal {time event args} {
    if {!([string is integer -strict $time] && $time >= 0)} {
        tailcall ::stsatcl::DeclError INVALIDTIME $time
    }
    set src [my EventSource]
    classvariable polyEvents
    if {$event in $polyEvents} {
        my ForwardPolyEvent [list delayedSignal $time] $src $event $args
    } else {
        # Cancel any existing delayed signal that might have been in place.
        my CancelDelayedSignal $src $event

        my variable __delayedSignals__
        set __delayedSignals__($src,$event) [dict create\
            timerid\
                [::after $time [mymethod DelayedDispatch $src $event]]\
            expire [expr {[clock milliseconds] + $time}]\
            info [dict create\
                src $src\
                event $event\
                params $args\
            ]
        ]
    }
    return
}
```

```
<<error code formats>>=
INVALIDTIME    {invalid signal delay time, "%ld"}
```

Delayed Signal Method Tests

```
<<exported tests>>=
test delayedSignal-1.0 {
    signal a delayed event from outside of a state machine context
```

```
} -setup {
    ::stsatcl STSAClass create ds1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    ds1 destroy
} -body {
    ds1 create inst1
    set begin [clock milliseconds]
    inst1 delayedSignal 100 e1 20
    namespace upvar [info object namespace inst1] __delayedSignals__ dsigs
    set expire [dict get $dsigs(,e1) expire]

    expr {$expire - $begin - 100 <= 2}
} -result {1}
```

Cancel Method

A delayed event may be cancelled using the `cancel` method. It is not an error to cancel an event that was never scheduled or might have alread been delivered.

xumlobj `cancel` *event*

`event`
: The name of the event to be cancelled.

The method returns a boolean indicating whether *event* was actually cancelled.

```
<<state model methods>>=
method cancel {event} {
    return [my CancelDelayedSignal [my EventSource] $event]
}
```

Cancel Method Tests

```
<<exported tests>>=
test cancel-1.0 {
    cancel a delayed event
} -setup {
    ::stsatcl STSAClass create ds1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    ds1 destroy
} -body {
    ds1 create inst1
    set begin [clock milliseconds]
    inst1 delayedSignal 100 e1 20
```

```
    inst1 cancel e1

    namespace upvar [info object namespace inst1] __delayedSignals__ dsigs
    parray dsigs
    info exists dsigs(,e1)
} -result {0}
```

Remaining Method

The final operation on delayed events is to obtain the amount of time `remaining` before the event is dispatched.

xumlobj `remaining` *event*

`event`
The name of the event for which the remaining time is to be returned.

The method returns the number of milliseconds before the event is to be signalled. A return value of 0 indicates that the event has expired or was not found as a delayed event.

```
<<state model methods>>=
method remaining {event} {
    set src [my EventSource]
    my variable __delayedSignals__
    return [expr {[info exists __delayedSignals__($src,$event)] ?\
            max([dict get $__delayedSignals__($src,$event) expire]\
                - [clock milliseconds], 0) : 0}] ; # ❶
}
```

❶ The remaining time is the difference between the expiration time recorded when the event was signalled and the current time. We return 0, if the delayed signal doesn't exists or if by some chance the current time is past the expiration time. The current time may be past the expiration time if the we have been off executing outside of the event loop. The logic is nested together in one expression under some, probably misguided as the units of time here are milliseconds, impression that calculations on time should be "efficient".

Remaining Method Tests

```
<<exported tests>>=
test remaining-1.0 {
    obtain the remaining time for a delayed event
} -setup {
    ::stsatcl STSAClass create ds1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    ds1 destroy
} -body {
    ds1 create inst1
    inst1 delayedSignal 100 e1 20
    set remain [inst1 remaining e1]
    expr {$remain - 100 <= 2}
} -result {1}
```

```
<<exported tests>>=
test remaining-2.0 {
    remaining time for non-existent event
} -setup {
    ::stsatcl STSAClass create ds1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    ds1 destroy
} -body {
    ds1 create inst1
    inst1 remaining e1
} -result {0}
```

Force Method

The `force` method causes an unconditional transition into the given *state* and executes the state activity there. Athough this method is subject to abuse, it can be very useful under certain error and / or recovery conditions.

xumlobj `force` *state*

`state`
The name of the state into which an unconditional transition is to be made.

```
<<state model methods>>=
method force {state args} {
    classvariable states
    if {$state ni $states} {
        tailcall ::stsatcl::DeclError UNKNOWN_STATE $state $states
    }
    my variable __currentstate__
    set __currentstate__ $state
    my ${__currentstate__}__STATE_ {*}$args
    return
}
```

Current State Method

The `currentstate` method returns the name of the current state of the object. This method is also subject to abuse and is provided for diagnostic and introspection purposes. Please do not make the logic of your application dependent upon the return of the `currentstate`. It is not necessary and there is always a better way to partition the control logic. However, debugging insight can depend upon knowledge of where you are and the current state of an instance.

xumlobj `currentstate`

```
<<state model methods>>=
method currentstate {} {
    my variable __currentstate__
    return $__currentstate__
}
```

Receive Method

This section begins that discussion of several unexported methods that are used to implement the internals of the state machine event dispatch code. These methods form the heart of the design of the execution mechanisms and warrant close study if you wish to understand how the architecture actually works.

The `Receive` method is responsible for performing a transition on the state machine and executing the activity associated with the destination state. This happens synchronously as this method is what is invoked along the callback path that event signalling starts.

The design handles a few key considerations:

- Computing the transition is accomplished by accessing a dictionary. Recall that the transition matrix is fully populated and the new state is a function of the current state and the event. So a two level dictionary serves as the new state function.
- The “IG” and “CH” pseudo-transition must be accounted for. “IG” causes nothing to happen, “CH” is a serious error.
- Assuming an actionable transition, the state activity is invoked. Recall that all state activities were turned into methods, albeit with a convention enforced on the method naming.
- Finally, we have to deal with asynchronous deletion by examining the set of terminal states for the state model. At the end of the execution of the activity for a terminal state, the instance is deleted.

The code below is a direct implementation of these considerations.

```
<<state model methods>>=
method Receive {eventInfo} {
    my variable __currentstate__
    set event [dict get $eventInfo event]
    set params [dict get $eventInfo params]

    classvariable transitions
    set newState [dict get $transitions $__currentstate__ $event] ; # ❶

    ::stsatcl::TraceTransition [dict get $eventInfo src] $event [self]\
        $__currentstate__ $newState $params

    if {$newState eq "CH"} { # ❷
        tailcall ::stsatcl::DeclError CH_TRANSITION $__currentstate__ $event
    } elseif {$newState ne "IG"} {
        set __currentstate__ $newState ; # ❸
        try {
            my ${__currentstate__}__STATE__ {*}$params ; # ❹
        } finally {
            classvariable terminals ; # ❺
            if {$__currentstate__ in $terminals} {
                my destroy
            }
        }
    }
    return
}
```

❶ In the Moore formalism, the new state is a function of the current state and the event.

❷ In this architecture, we deem the state machine to be “in” the new state before the activity is executed. Some XUML software architectures do the opposite, *i.e.* changing state only after the activity is executed. The difference is subtle and applications should *not* depend upon such nuances, but one rational for this order is to allow a state activity to read the current state and get a value that matches its own state name.

❸ Check for pseudo-transitions.

❹ Despite the “line noise” appearance of this statement, we are simply invoking the method that is the state activity, constructing its name according to some established conventions and giving it the arguments specified in the event.

❺ If this is a terminal state, then we destroy ourselves. This is the asynchronous deletion that corresponds to asychronous creation.

```
<<error code formats>>=
CH_TRANSITION   {can't happen transition: %s - %s -> CH}
```

Receive Method Tests

```
<<exported tests>>=
test Receive-1.0 {
    synchronously receive a state machine event
} -setup {
    ::stsatcl STSAClass create rc1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    [info object namespace inst1]::my Receive\
            [dict create src {} event e1 params 20]
} -result {} -output {20}
```

```
<<exported tests>>=
test Receive-2.0 {
    cause can't happen transition
} -setup {
    ::stsatcl STSAClass create rc1 {
        statemodel {
            state s1 {a} {
                puts -nonewline $a
            }
            transition s1 - e1 -> s1
            transition s1 - e2 -> CH
        }
    }
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    [info object namespace inst1]::my Receive\
            [dict create src {} event e2 params {}]
} -result {can't happen transition: s1 - e2 -> CH} -returnCodes error
```

```
<<exported tests>>=
test Receive-3.0 {
    enter a terminal state
} -setup {
    ::stsatcl STSAClass create rc3 {
        statemodel {
```

```
            terminal s2

            state s1 {} {
            }
            transition s1 - e1 -> s2

            state s2 {} {
            }
        }
    }
} -cleanup {
    rc3 destroy
} -body {
    rc3 create inst1
    [info object namespace inst1]::my Receive\
            [dict create src {} event e1 params {}]
    llength [info commands ::stsatcl::test::inst1]
} -result {0}
```

```
<<exported tests>>=
test Receive-4.0 {
    ignore an event
} -setup {
    ::stsatcl STSAClass create rc4 {
        statemodel {
            state s1 {} {
            }
            transition s1 - e1 -> s1
            transition s1 - e2 -> IG
        }
    }
} -cleanup {
    rc4 destroy
} -body {
    rc4 create inst1
    [info object namespace inst1]::my Receive\
            [dict create src {} event e2 params {}]
    inst1 currentstate
} -result {s1}
```

Dispatch Method

The dispatch of non-delayed events is accomplished by the `Dispatch` method. Each non-delayed state machine event that is signalled causes a Tcl event to which this method is attached as the Tcl event handler.

The purpose of this method is to turn a Tcl event into a state machine event. The logic of dispatch is simple. We pull the first event off of the event queue and, using the `Receive` method, sychronously cause the transition to happen and, consequently, the state activity to be executed. Note that we always pull the dispatched event off of the front of the event queue. The event queue is always kept in dispatch order and recall that inserting an event it the event queue accounts for the rules about self-directed events being dispatched before non-self-directed events.

The only minor complication in the design is that we have to attend to the bookkeeping of the timer ID's assocated with the Tcl `_after` command. Recall that the Tcl events that are handled here are created using the `after 0` command sequence. When the Tcl events are created, we save the timer ID returned from `after`. Here we have to remove our memory of that timer ID. Any timer ID's remaining at `destroy` time are then cancelled insuring basic resource cleanup occurs properly.

```
<<state model methods>>=
method Dispatch {eventid} {
    my variable __signals__
    array unset __signals__ $eventid
```

```
    my variable __event_queue__
    if {[llength $__event_queue__] != 0} { # ❶
        # Pull the event from the front of the queue.
        set eventInfo [lindex $__event_queue__ 0]
        set __event_queue__ [lrange $__event_queue__ 1 end]
        tailcall my Receive $eventInfo
    }
}
```

❶ The event queue should contain something here, but we test anyway.

Dispatch Method Tests

To test asynchronous dispatch of events into state machines, we will use some utility procedures to help synchronize things for back to the test script. The design idea is to have one procedure that can be called by state activities to write to a well known variable and another one that will perform the necessary `vwait` command to enter the event loop.

```
<<test utility procs>>=
proc syncToTest {{value {}}} {
    set [namespace current]::testDone $value
}

proc waitForSync {{timeout 1000}} {
    set varname [namespace current]::testDone
    set tid [::after $timeout set $varname TIMEOUT]
    vwait $varname
    after cancel $tid
    set $varname
}
```

```
<<exported tests>>=
test Dispatch-1.0 {
    dispatch an event to a state machine.
} -setup {
    ::stsatcl STSAClass create rc1 {
        statemodel {
            state s1 {a} {
                ::stsatcl::test::syncToTest $a
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    inst1 signal e1 20
    waitForSync
} -result {20}
```

```
<<exported tests>>=
test Dispatch-2.0 {
    dispatch multiple events to a state machine.
} -setup {
    ::stsatcl STSAClass create rc1 {
        attribute {Count 0}
        attribute {Max 2}
        statemodel {
            state s1 {} {
```

```
                my variable Count Max
                incr Count
                if {$Count >= $Max} {
                    ::stsatcl::test::syncToTest $Count
                }
            }
            transition s1 - e1 -> s1
        }
    }
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    inst1 signal e1
    inst1 signal e1
    waitForSync
} -result {2}
```

Delayed Dispatch Method

Dispatch of delayed events is accomplished by the `DelayedDispatch` method. One difference between delayed dispatch and non-delayed dispatch is we don't have to consult the event queue. Recall that we use the `after` *ms* form of the `after` command to generate Tcl events at the appropriate delay time. The `DelayedDispatch` command is tied to the Tcl event and, like the `Dispatch` command is responsible for translating the Tcl event into a state machine event. We have arranged for all the event information, in the form of a dictionary value, to be stored along with the other information required to support delayed events.

Also, like the `Dispatch` method we must keep track of the timer ID's and other information associated with the delayed event.

```
<<state model methods>>=
method DelayedDispatch {src event} {
    my variable __delayedSignals__
    if {[info exists __delayedSignals__($src,$event)]} { # ❶
        set eventInfo [dict get $__delayedSignals__($src,$event) info]
        array unset __delayedSignals__ $src,$event
        tailcall my Receive $eventInfo ; # ❷
    }
}
```

❶ Now that the delayed event is delivered, we can obtain the event information and do the bookkeeping on the set of outstanding delayed events.

❷ The `Receive` method does the heavy lifting of actually causing the state machine transition.

Delayed Dispatch Method Tests

```
<<exported tests>>=
test DelayedDispatch-1.0 {
    dispatch a delayed event to a state machine.
} -setup {
    ::stsatcl STSAClass create rc1 {
        statemodel {
            state s1 {} {
                ::stsatcl::test::syncToTest [clock milliseconds]
            }
            transition s1 - e1 -> s1
        }
    }
```

```
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    set delay 100

    set start [clock milliseconds]
    inst1 delayedSignal $delay e1
    set end [waitForSync]

    expr {$end - $start - $delay <= 2}
} -result {1}
```

```
<<exported tests>>=
test DelayedDispatch-2.0 {
    dispatch multiple delayed events to a state machine.
} -setup {
    ::stsatcl STSAClass create rc1 {
        attribute {Count 0}
        attribute {Max 2}
        statemodel {
            state s1 {} {
                my variable Count Max
                incr Count
                if {$Count >= $Max} {
                    ::stsatcl::test::syncToTest $Count
                }
            }
            transition s1 - e1 -> s2

            state s2 {} {
                my variable Count
                incr Count
            }
            transition s2 - e2 -> s1
        }
    }
} -cleanup {
    rc1 destroy
} -body {
    rc1 create inst1
    set delay 100

    set start [clock milliseconds]
    inst1 delayedSignal $delay e1
    inst1 delayedSignal $delay e2
    set end [waitForSync]

    expr {$end - $start - 2 * $delay <= 2}
} -result {1}
```

Cancel Delayed Signal

The `CancelDelayedSignal` method factors common code for tending to the bookkeeping of delayed signals into one place.

```
<<state model methods>>=
method CancelDelayedSignal {src event} {
    my ValidateEvent $event
    # Check if we have the delayed signal. It might not exist
    # or have already been dispatched. We return an indication
```

```
    # of whether the signal was indeed canceled.
    my variable __delayedSignals__
    if {[info exists __delayedSignals__($src,$event)]} {
        after cancel [dict get $__delayedSignals__($src,$event) timerid]
        array unset __delayedSignals__ "$src,$event"
        set result true
    } else {
        set result false
    }
    return $result
}
```

Example State Activities

At long last we now have enough information to code the state model activities and the remaining processing for our example washing machine. Let's recap where we are with the example.

First, we saw a class diagram and used that diagram to define the attributes and linkages for the classes that perform washing machine control. Next we saw a state model and showed how that model could be encoded in the `STSAClass` DSL. All along in the presentation of the example we have deferred showing any state activities code. Now that we have the XUML class methods available, we are prepared to show the third and final facet of how a model is made into a running program.

To make the correspondence clear, we will visit each state activity, show its action language pseudo-code and then present the Tcl code that implements the activity. It is in that Tcl code we shall see the invocation of the XUML class methods discussed above. A few points are worth noting.

- There is substantial repetition in the action language. No attempt has been made to factor common sequences into instance operations as would be the case for more complicated activities. This was deliberate to avoid having to look in multiple places to be able to determine exactly what is happening.
- The transliteration of action language statements to Tcl commands is quite literal. In many places, a more natural Tcl coding style would have "nested" several commands together. Again, this was deliberate to be explicit about the correspondence between the Tcl commands and the action language statements.

Washing Machine State Activities

We start with the Washing Machine class. In reading this section, you may find it helpful to refer to the state model and class diagram.

Stopped Activity

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

Stopped Implementation

```
<<WM stopped activity>>=
set ct [my -> R1]
$ct signal Stop
```

Filling To Wash Activity

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

Filling To Wash Implementation

```
<<WM filling to wash activity>>=
set wc [my -> R4]
set ct [my -> R1]
$ct signal Fill [$wc readAttributes WashWaterTemp]
```

Washing Activity

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

Washing Implementation

```
<<WM washing activity>>=
set ct [my -> R1]
$ct signal Agitate
set wc [my -> R4]
my delayedSignal [expr {[$wc readAttributes WashDuration] * 1000}] Done ; # ❶
```

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

Draining Wash Activity

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

Draining Wash Implementation

```
<<WM draining wash activity>>=
set ct [my -> R1]
$ct signal Drain
```

Filling To Rinse Activity

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

Filling To Rinse Implementation

```
<<WM filling to rinse activity>>=
set wc [my -> R4]
set ct [my -> R1]
$ct signal Fill [$wc readAttributes RinseWaterTemp]
```

Rinsing Activity

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

Rinsing Implementation

```
<<WM rinsing activity>>=
set ct [my -> R1]
$ct signal Agitate
set wc [my -> R4]
my delayedSignal [expr {[$wc readAttributes RinseDuration] * 1000}] Done
```

Draining Rinse Activity

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

Draining Rinse Implementation

```
<<WM draining rinse activity>>=
set ct [my -> R1]
$ct signal Drain
```

Spinning Activity

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

Spinning Implementation

```
<<WM spinning activity>>=
set ct [my -> R1]
$ct signal Spin
set wc [my -> R4]
my delayedSignal [expr {[$wc readAttributes SpinDuration] * 1000}] Done
```

Clothes Tub State Activities

The other state model in our example is for the Clothes Tub class Again, you may find it helpful to refer to the state model and class diagram.

Empty Activity

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

Empty Implementation

```
<<CT empty activity>>=
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Pump"
}]
$mtr Stop
set wv [my selectOneRelatedWhere R3 selected {
    [$selected readAttributes ValveID] eq "Drain"
}]
$wv Close
set wls [my -> R5]
$wls Disable
set wm [my -> R1]
$wm signal Empty
```

Filling Activity

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

```
        wv.Open()
    endfor
end if
```

Filling Implementation

```
<<CT filling activity>>=
set wls [my -> R5]
$wls Enable
if {$temp eq "Hot"} {
    set wv [my selectOneRelatedWhere R3 selected {
        [$selected readAttributes ValveID] eq "Hot"
    }]
    $wv Open
} elseif {$temp eq "Cold"} {
    set wv [my selectOneRelatedWhere R3 selected {
        [$selected readAttributes ValveID] eq "Cold"
    }]
    $wv Open
} elseif {$temp eq "Warm"} {
    set wvs [my selectRelatedWhere R3 selected {
        [$selected readAttributes ValveID] eq "Hot" ||\
        [$selected readAttributes ValveID] eq "Cold"
    }]
    foreach wv $wvs {
        $wv Open
    }
}
```

Full Activity

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

Full Implementation

```
<<CT full activity>>=
set wls [my -> R5]
$wls Disable
set wvs [my selectRelatedWhere R3 selected {
    [$selected readAttributes ValveID] eq "Hot" ||\
    [$selected readAttributes ValveID] eq "Cold"
}]
foreach wv $wvs {
    $wv Close
}
set wm [my -> R1]
$wm signal Full
```

Agitating Activity

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

Agitating Implementation

```
<<CT agitating activity>>=
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Agitator"
}]
$mtr Start
```

Emptying Activity

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

Emptying Implementation

```
<<CT emptying activity>>=
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Agitator"
}]
$mtr Stop
set wv [my selectOneRelatedWhere R3 selected {
    [$selected readAttributes ValveID] eq "Drain"
}]
$wv Open
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Pump"
}]
$mtr Start
set wls [my -> R5]
$wls Enable
```

Spinning Activity

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

Spinning Implementation

```
<<CT spinning activity>>=
set wv [my selectOneRelatedWhere R3 selected {
    [$selected readAttributes ValveID] eq "Drain"
}]
$wv Open
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Pump"
}]
$mtr Start
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Spin"
}]
$mtr Start
```

Stopping Spin Activity

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

Stopping Spin Implementation

```
<<CT stopping spin activity>>=
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Spin"
}]
$mtr Stop
set mtr [my selectOneRelatedWhere R2 selected {
    [$selected readAttributes MotorID] eq "Pump"
}]
$mtr Stop
set wv [my selectOneRelatedWhere R3 selected {
    [$selected readAttributes ValveID] eq "Drain"
}]
$wv Close
```

State Machine Trace

In this section we discuss the capabilites and design of tracing state machine event dispatch. Again, we are confronted with a heavily overloaded term, *trace*. There are many kinds of traces in the Tcl world and we do not wish to confuse Tcl variable and command tracing with `stsatcl` package state machine tracing. Here we are discussing the ability of the package to produce a chronologically ordered sequence of the results of dispatching state machine events.

It's hard to overemphasize the importance of the event dispatch trace for a set of state machines. Since the majority of the processing in state machine based application is in the form of callbacks for dispatched events, it is difficult to simply read the code base sequentially and have a good sense of what will happen during execution. Of course, a different sequence of events will order the code execution differently. That is, after all, what we are trying to achieve with a state model.

The fact that the path of code execution does not easily correspond to the sequence of the code statements is objectionable enough to some that they avoid a state model based approach. Others try to cast state behavior into more sequential appearing code by using other techniques such as coroutines. All of these considerations lead to the conclusion that capturing a chronological trace of the event dispatch of a state machine based applications is indispensable to understanding and testing.

In this section we discuss the design and implementation of the state machine tracing implemented by this package. First we lay out some basic rules.

- Tracing can be controlled. It is necessary to be able to start and stop the trace capture and to clear out any accumulated traces.
- Tracing must capture all the semantics of event dispatch, including polymorphic and creation events.
- It is only necessary to trace event dispatch. Signalling, *i.e.* when events are generated, is not captured.
- Common operations on the collected trace data must be supported including the ability to save the trace data into some persistent form, (*e.g.* a file).

To meet these requirements, trace data is captured by the event dispatch methods and stored in appropriate data structures. The following sections discuss the manner in which this is done. We divide the discussion into the these parts:

- The structure of the trace data.
- Procedures to gather the trace data.
- Procedures to query and format the trace data.

Trace Data

The figure below shows a class diagram in UML notation of the state machine trace data.

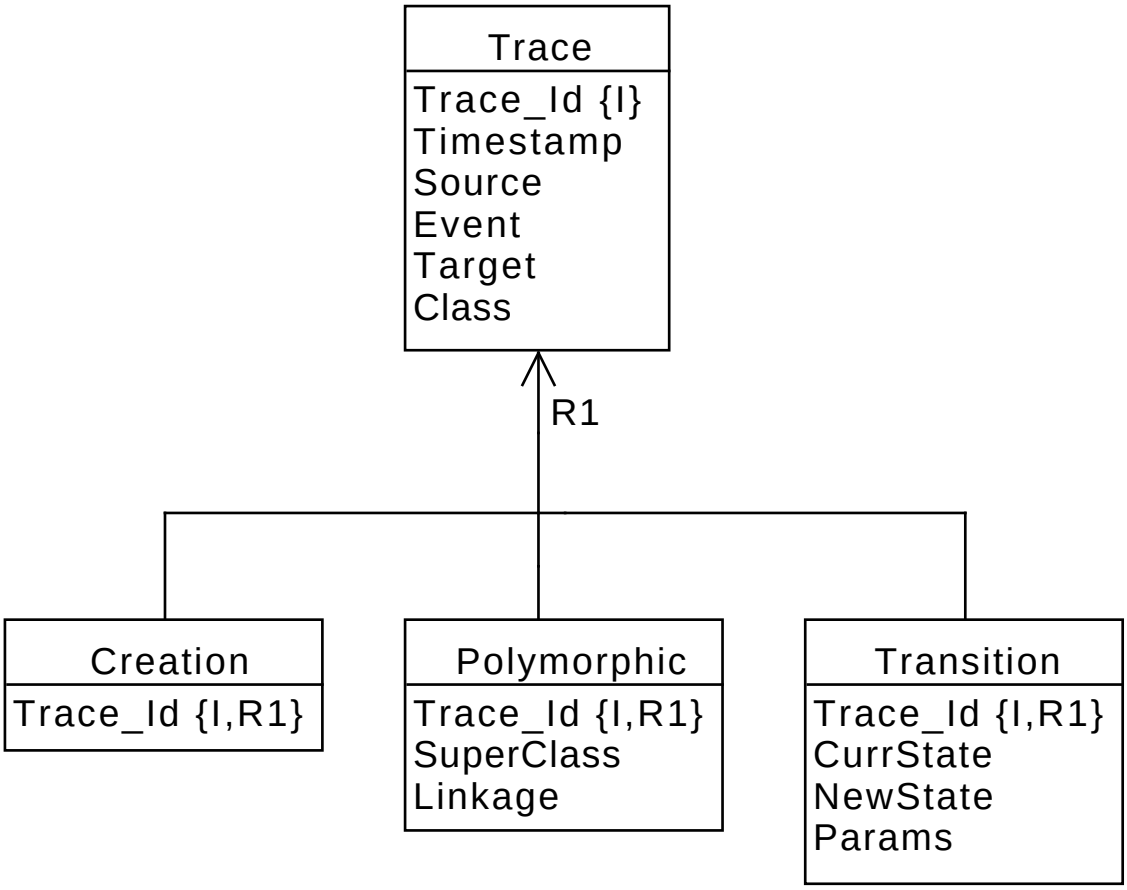

Figure 6: Trace Data Class Model

A trace is identified by an arbitary identifier. We will use a sequential integer. Each trace has a `Timestamp`. This needs to be of relatively high resolution so we will use the return from the `clock microseconds` command. The `Target` of a trace is the object command name of the instance to which the event is directed. We also store the `Class` command name that corresponds to the `Target` so we can perform queries on a class rather than instance basis (*e.g.* it is interesting to determine how all instances of some set of classes pass events without any reference to particular instances).

The three types of event dispatch types are captured as by the `R1` generalization. **Creation** events result in an instance being created followed by an ordinary **Transition** event. So any creation event will have two trace entries. **Polymorphic** events map, at runtime, an event directed at a superclass instance to a corresponding event in the related subclass instance. So each polymorphic event will have at least one ordinary transition associated with it as the polymorphic event is finally mapped down to a leaf subclass and consumed there. Ordinary **Transition** events are most common and we record the object that is the source of the event, parameters associated with the event and the outcome of the transition.

The most direct implementation of this data schema is to use the `TclRAL` package. `TclRAL` has direct support for defining the relation variables and partition constraints implied by the class diagram and all the required operations to query and serialize the captured trace data.

```
<<sm trace packages>>=
package require ral
package require ralutil
```

The correspondence between the class diagram and the `TclRAL` commands required to implement it is direct and amounts to little more than casting the class diagram into Tcl command syntax.

```
<<sm trace data>>=
namespace import ::ral::*
namespace import ::ralutil::*

relvar create Trace {
    Trace_Id    int
    Timestamp   bignum
    Source      string
    Event       string
    Target      string
    Class       string
} Trace_Id

relvar create Creation {
    Trace_Id    int
} Trace_Id

relvar create Polymorphic {
    Trace_Id    int
    SuperClass  string
    Linkage     string
} Trace_Id

relvar create Transition {
    Trace_Id    int
    CurrState   string
    NewState    string
    Params      list
} Trace_Id

relvar partition R1 Trace Trace_Id\
    Creation Trace_Id\
    Polymorphic Trace_Id\
    Transition Trace_Id
```

Despite the value of tracing in testing and debugging, it is a rather heavy weight operation and this design requires as “C” based extension to implement it. So we will make it optional at to whether or not to pull in all the tracing machinery. To do that we will use some variable to hold the state of the tracing operations.

```
<<stsatcl data>>=
variable traceInitialized false
variable traceState off
```

We also need some ordinary variables to keep track of the state of trace capture, a counter for generating the `Trace_Id` attribute values and details of trace logging.

```
<<sm trace data>>=
variable traceNumber 0
variable traceLogState off
variable traceLogLevel info
```

In addition to simply accumulating trace data, it is sometimes convenient to produce the trace as a log in real time. We support logging via the `logger` package available in `tcllib`.

```
<<sm trace packages>>=
package require logger
```

We log to a service that is the same name as the package.

```
<<sm trace packages>>=
variable traceLogCmd [::logger::init stsatcl]
```

Trace Control

The `traceControl` procedure provides the interface necessary to control the aspects of state machine tracing.

```
<<stsatcl commands>>=
proc ::stsatcl::traceControl {op args} {
    variable traceInitialized       ; # ❶
    if {!$traceInitialized} {
        set traceInitialized true
        <<sm trace packages>>
        <<sm trace data>>
    }
    switch -exact -- $op {
        on {
            variable traceState on
        }
        off {
            variable traceState off
            set traceState off
        }
        status {
            variable traceState
            return $traceState
        }
        clear {
            relvar eval {
                foreach rvar {Trace Creation Polymorphic Transition} {
                    relvar set $rvar [relation emptyof [relvar set $rvar]]
                }
            }
            variable traceNumber 0
        }
        logon {
            variable traceLogState on
        }
        logoff {
            variable traceLogState off
        }
        loglevel {
            variable traceLogLevel
            if {[llength $args] != 0} {
                set traceLogLevel [lindex $args 0]
            }
            return $traceLogLevel
        }
        save {
            if {[llength $args] == 0} {
                DeclError NO_SAVEFILE
            }
            storeToSQLite [lindex $args 0] [relvar names [namespace current]::*]
        }
        default {
            DeclError BAD_TRACEOP $op
        }
    }
}

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

❶ So we only pull in all the required tracing packages and define the tracing data structures if some form of control operation on the tracing is invoked.

```
<<error code formats>>=
BAD_TRACEOP     {unknown trace operation, "%s"}
NO_SAVEFILE     {no save file name provided}
```

Trace Population

Each of the XUML class methods that is involved with dispatching events into a state machine invokes a procedure to capture the dispatch data. There are three such procedures corresponding to the three types of event dispatch. Each as the same basic structure, namely determining if tracing is enabled, inserting the trace data to the appropriate relvars and logging the trace instance.

```
<<stsatcl commands>>=
proc ::stsatcl::TraceCreation {source event target} {
    variable traceState
    if {$traceState} {
        relvar eval {
            set trace [NewTrace $source $event $target]
            relvar insert Creation [list\
                Trace_Id    [relation extract $trace Trace_Id]\
            ]
        }
        LogTrace $trace
    }
}
```

```
<<stsatcl commands>>=
proc ::stsatcl::TracePolymorphic {source event target super link} {
    variable traceState
    if {$traceState} {
        relvar eval {
            set trace [NewTrace $source $event $target]
            relvar insert Polymorphic [list\
                Trace_Id    [relation extract $trace Trace_Id]\
                SuperClass  $super\
                Linkage     $link\
            ]
        }
        LogTrace $trace
    }
}
```

```
<<stsatcl commands>>=
proc ::stsatcl::TraceTransition {source event target curr new params} {
    variable traceState
    if {$traceState} {
        relvar eval {
            set trace [NewTrace $source $event $target]
            relvar insert Transition [list\
                Trace_Id    [relation extract $trace Trace_Id]\
                CurrState   $curr\
                NewState    $new\
                Params      $params\
            ]
        }
        LogTrace $trace
    }
}
```

Code to number the trace and insert it into the `Trace` relvar is factored to a separate procedure.

```
<<stsatcl commands>>=
proc ::stsatcl::NewTrace {src event target} {
    variable traceNumber
    return [relvar insert Trace [list\
        Trace_Id    [incr traceNumber]\
        Timestamp   [clock microseconds]\
        Source      $src\
        Event       $event\
        Target      $target\
        Class       [info object class $target]\
    ]]
}
```

Each of the above three procedures that inserts a particular type of trace data into the data store also logs the trace. Trace logging first checks the state of the logging and then formats an appropriate string for the log.

```
<<stsatcl commands>>=
proc ::stsatcl::LogTrace {trace} {
    variable traceLogState
    if {$traceLogState} {
        set rec [FormatTraceRec [lindex [TracesToRecords $trace] 0]] ; # ❶

        variable traceLogCmd
        variable traceLogLevel
        ${traceLogCmd}::${traceLogLevel} $rec
    }
}
```

❶ The `TracesToRecords` procedure returns a list and knowing that the list contains only one element, we extract that element to format.

```
<<exported tests>>=
test LogTrace-1.0 {
    log trace data
} -setup {
    ::stsatcl STSAClass create tr1 {
        attribute {Count 0} {Max 2}
        statemodel {
            state s1 {} {
                my variable Count Max
                incr Count
                if {$Count >= $Max} {
                    ::stsatcl::test::syncToTest $Count
                }
            }
            transition s1 - e1 -> s2

            state s2 {} {
                my variable Count
                incr Count
                my signal e2
            }
            transition s2 - e2 -> s1
        }
    }

    turnOnTraceLog
} -cleanup {
    turnOffTraceLog
```

```
    tr1 destroy
} -body {
    tr1 create inst1
    inst1 signal e1
    waitForSync
} -result {2}
```

```
<<exported tests>>=
test LogTrace-2.0 {
    log trace data for a polymorphic event
} -setup {
    ::stsatcl STSAClass create lt2 {
        partition R1 lt2-sub1 foo
        polymorphic *e1 *e2
    }

    ::stsatcl STSAClass create lt2-sub1 {
        reference R1 -> lt2
        statemodel {
            state sub1-s1 {} {
                ::stsatcl::test::syncToTest true
            }
            transition sub1-s1 - *e1 -> sub1-s2

            state sub1-s2 {} {
                [my -> R1] delayedSignal 50 *e2
            }
            transition sub1-s2 - *e2 -> sub1-s1
        }
    }

    lt2 create super
    lt2-sub1 create sub1
    super link R1 sub1
    sub1 link R1 super

    turnOnTraceLog
} -cleanup {
    turnOffTraceLog

    lt2 destroy
    lt2-sub1 destroy
} -body {
    super signal *e1
    waitForSync
} -result {true}
```

```
<<exported tests>>=
test LogTrace-3.0 {
    log trace data for a creation event
} -setup {
    ::stsatcl STSAClass create lt3 {
        statemodel {
            transition @ - e1 -> s1

            state s1 {value} {
                ::stsatcl::test::syncToTest $value
            }
        }
    }
```

```
    turnOnTraceLog
} -cleanup {
    turnOffTraceLog

    lt3 destroy
} -body {
    set inst [lt3 signal e1 10]
    waitForSync
} -result {10}
```

```
<<test utility procs>>=
proc turnOnTraceLog {} {
    variable prevloglevel

    ::stsatcl traceControl clear
    ::stsatcl traceControl on
    set logcmd [::logger::servicecmd stsatcl]
    set prevloglevel [${logcmd}::currentloglevel]
    ${logcmd}::setlevel [::stsatcl traceControl loglevel]
    ::stsatcl traceControl logon
}

proc turnOffTraceLog {} {
    variable prevloglevel

    set logcmd [::logger::servicecmd stsatcl]
    ${logcmd}::setlevel $prevloglevel
    ::stsatcl traceControl logoff
    ::stsatcl traceControl off
    ::stsatcl traceControl clear
}
```

Trace Operations

It is difficult to provide all the conceivable operations that one may wish to perform on the captured state machine trace data. In this package we will provide some common operations, but special cases can be handled by constructing appropriate queries on the `relvar` data of the package and passing the result to be formatted. The required primative operations are here to support a wider range of custom trace queries.

We divide the decoding of the trace data into two parts:

- Casting the `relvar` data into an ordered list of dictionaries.
- Formatting trace dictionary data into human readable form.

This separation makes handling data programmatically via the dictionaries much easier yet still allows the production of human readable output. There are many ways that a program can use the state machine trace data. For example, the trace data can be used to compute the coverage of states and transitions for testing purposes. This is much easier to accomplish if the data is *not* in string form but rather with known named fields.

In this section we will cover the layout of the trace record dictionaries and then a set of query functions to produce sets of records. Afterward, we will discuss procedures to format trace dictionaries into human readable strings.

Trace Dictionary Structure

Although the `relvars` defined above contain all the trace data in a form that is easy to write queries on, transforming relational data into a dictionary provides a convenient interface for handling sets of trace data. Here we dicuss the keys that the trace data dictionary have and the procedures which produce them.

The trace data dictionary has to account for the three different types of dispatched events. We will have common keys that apply to all types of events and a `type` key that can be used to determine the event type specific keys. The common keys are:

`id`
An integer number that identifies the sequence of the trace entry.

`time`
The time since the epoch, in microseconds, that the trace was captured.

`source`
The object command name of the instance that signalled the event. If the source of the event is outside of an object, then this will be the empty string.

`event`
The name of the event.

`target`
The object command name of the target of the event.

`class`
The command name of the class that corresponds to `target`.

`type`
The type of the dispatched event: one of `creation`, `polymorphic`, or `transition`.

For `creation` type events, there are no additional keys. The `target` value is interpreted as the object command name of the newly created instance and the `class` value is the command name of the instance creator (which is, necessarily, also the class of the `target`).

For `polymorphic` type events, the following additional keys are available:

`super`
The object command name of the superclass instance mapping the event.

`link`
The name of the parition linkage across which the event is mapped.

For polymorphic type events, the `target` value is the command name of the subclass instance onto which the polymorphic event is mapped.

For `transition` type events, the following additional keys are available:

`current`
The name of the state of `target` when the event was received.

`new`
The name of the state of `target` after the transition happened.

`params`
A list of values giving the parametric data passed with the event to the state activity.

The procedure, `TracesToRecords`, takes a relation value that is a subset (proper or improper) of the value contained in the `Trace` relvar and converts it into a list of trace records. Each trace record is a dictionary of the form described above.

As we will see below, this procedure is used by all the query oriented procedures to convert the trace data into lists of dictionaries. One difficulty of this transformation is to account for the three different types of event traces. This is done by joining the `Trace` relation values against each of the three `relvars` that participate in `R1`. This join is done in such a way that three new relation valued attributes are created and, given the disjoint union implied by the partition constraint, only one of the new attributes will contain any tuples.

```
<<stsatcl commands>>=
proc ::stsatcl::TracesToRecords {traces} {
    foreach rvname {Creation Polymorphic Transition} { # ❶
        set traces [rvajoin $traces [relvar set $rvname] $rvname]
    }

    set result [list]
    relation foreach trace $traces -ascending Trace_Id { # ❷
        relation assign $trace
        set labeled [dict create\
            id      $Trace_Id\
            time    $Timestamp\
            source  $Source\
            event   $Event\
            target  $Target\
            class   $Class\
        ] ; # ❸
        if {[relation isnotempty $Transition]} { # ❹
            relation assign $Transition
            dict set labeled type transition
            dict set labeled current $CurrState
            dict set labeled new $NewState
            dict set labeled params $Params
        } elseif {[relation isnotempty $Polymorphic]} {
            relation assign $Polymorphic
            dict set labeled type polymorphic
            dict set labeled super $SuperClass
            dict set labeled link $Linkage
        } elseif {[relation isnotempty $Creation]} {
            dict set labeled type creation
        }

        lappend result $labeled
    }
    return $result
}
```

❶ Perform a **relation valued join** of the traces against the three different types of trace data. Each `rvajoin` produces a new attribute whose name we choose to be the same as the `relvar` from which it came and whose value is a relation value containing those tuples whose value of `Trace_Id` match. Because the `R1` partition represents a disjoint union and `Trace_Id` is an identifier, only one of the three new attributes will have any tuples and the non-empty one will contain exactly one tuple.

❷ We can now iterate across the joined trace data in the order that it was generated (*i.e.* by `-ascending Trace_Id`) so that the resulting list of dictionaries is in the same order. This was the primary reason for performing all the `rvajoin` operations, *i.e.* to make it more convenient to preserve the event dispatch order into the resulting list.

❸ Add all the common keys, except `type`.

❹ Add the keys that are specific to the event type.

Decode All Traces

The first of the query functions simple decodes all available trace data.

```
<<stsatcl commands>>=
proc ::stsatcl::decodeAllTraces {} {
    tailcall TracesToRecords [relvar set Trace]
}
```

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

Decode All Traces Tests

```
<<exported tests>>=
test decodeAllTraces-1.0 {
    display trace data
} -setup {
    ::stsatcl STSAClass create tr1 {
        attribute {Count 0} {Max 2}
        statemodel {
            state s1 {} {
                my variable Count Max
                incr Count
                if {$Count >= $Max} {
                    ::stsatcl::test::syncToTest $Count
                }
            }
            transition s1 - e1 -> s2

            state s2 {} {
                my variable Count
                incr Count
            }
            transition s2 - e2 -> s1
        }
    }
} -cleanup {
    ::stsatcl traceControl off
    ::stsatcl traceControl clear
    tr1 destroy
} -body {
    ::stsatcl traceControl on
    tr1 create inst1
    inst1 signal e1
    inst1 signal e2
    waitForSync
    set traces [::stsatcl decodeAllTraces]
    dict get [lindex $traces 0] event
} -result {e1}
```

Decode Class Traces

```
<<stsatcl commands>>=
proc ::stsatcl::decodeClassTraces {args} {
    tailcall TracesToRecords [pipe {
        relvar set Trace |
        relation restrictwith ~ {$Class in $args}
    }]
}

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

Decode Class Traces Tests

```
<<exported tests>>=
test decodeClassTraces-1.0 {
    display trace data for specific class
} -setup {
    ::stsatcl STSAClass create dct1 {
        reference R1 -> dct2
        attribute {Power 20}
        statemodel {
            state Off {} {
                [my -> R1] signal Lower
            }
            transition Off - TurnOn -> On

            state On {power} {
                my variable Power
                set Power $power
                [my -> R1] signal Raise
            }
            transition On - TurnOff -> Off
        }
    }

    ::stsatcl STSAClass create dct2 {
        reference R1 -> dct1
        statemodel {
            state Down {} {
                ::stsatcl::test::syncToTest [[my -> R1] readAttributes Power]
            }
            transition Down - Raise -> Up

            state Up {} {
                [my -> R1] delayedSignal 50 TurnOff
            }
            transition Up - Lower -> Down
        }
    }

    ::stsatcl traceControl clear
    ::stsatcl traceControl on
} -cleanup {
    ::stsatcl traceControl off
    ::stsatcl traceControl clear
    dct1 destroy
    dct2 destroy
} -body {
    dct1 create inst1
    dct2 create inst2
    inst1 link R1 inst2
    inst2 link R1 inst1
    inst1 signal TurnOn 30
    waitForSync
    # puts [::stsatcl formatTraces [::stsatcl decodeAllTraces]]
    set traces [::stsatcl decodeClassTraces [namespace current]::dct2]
    expr {[llength $traces] == 2 &&\
            [dict get [lindex $traces 1] event] eq "Lower"}
} -result {1}
```

Decode Target Traces

```
<<stsatcl commands>>=
proc ::stsatcl::decodeTargetTraces {args} {
    tailcall TracesToRecords [pipe {
        relvar set Trace |
        relation restrictwith ~ {$Target in $args}
    }]
}

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

Format Traces

Human readable output for state machine traces can be obtain via the `formatTraces` procedure. This procedure takes a list of trace records, as defined above and as returned from the various trace decode procedures, and returns a string that has a human readable representation of the trace records. Each trace record is separated by a line terminator character. The returned string may be written on any channel the caller wishes.

```
<<stsatcl commands>>=
proc ::stsatcl::formatTraces {tracerecs} {
    if {[llength $tracerecs] == 0} {
        return {}
    }
    set prevtime [dict get [lindex $tracerecs 0] time]
    foreach rec $tracerecs {
        dict with rec {
            append result "[FormatTimestamp $time]: "
            append result "[FormatTimeAsSec [expr {$time - $prevtime}]]: "
            set prevtime $time
            append result [FormatTraceRec $rec] \n
        }
    }
    return [string trimright $result]
}

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

Format Traces Tests

```
<<exported tests>>=
test formatTraces-1.0 {
    display human readable trace data
} -setup {
    ::stsatcl STSAClass create tr1 {
        attribute {Count 0} {Max 2}
        statemodel {
            state s1 {} {
                my variable Count Max
                incr Count
                if {$Count >= $Max} {
                    ::stsatcl::test::syncToTest $Count
                }
            }
            transition s1 - e1 -> s2
```

```
            state s2 {} {
                my variable Count
                incr Count
            }
            transition s2 - e2 -> s1
        }
    }
} -cleanup {
    ::stsatcl traceControl off
    ::stsatcl traceControl clear
    tr1 destroy
} -body {
    ::stsatcl traceControl on
    tr1 create inst1
    inst1 signal e1
    inst1 signal e2
    waitForSync
    set traces [::stsatcl formatTraces [::stsatcl::decodeAllTraces]]
    llength [split $traces \n]
} -result {2}
```

Format Trace Record

An individual trace record may be formatted with the `FormatTraceRec` procedure.

```
<<stsatcl commands>>=
proc ::stsatcl::FormatTraceRec {rec} {
    dict with rec {
        switch -exact -- $type {
            transition {
                append result\
                    "Transition: "\
                    "[list $source] - $event \{$params\} -> $target ==> "\
                    "$current -> $new"
            }
            polymorphic {
                append result\
                    "Polymorphic : "\
                    "[list $source] - $event -> $super ==> "\
                    "$link -> $target"
            }
            creation {
                append result\
                    "Creation: "\
                    "[list $source] - $event -> $class ==> $target"
            }
            default {
                DeclError BAD_TRACETYPE $type
            }
        }
    }
    return $result
}
```

```
<<error code formats>>=
BAD_TRACETYPE   {unknown trace type, "%s"}
```

Format Time Stamp

```
<<stsatcl commands>>=
proc ::stsatcl::FormatTimestamp {time} {
    set sec [clock format [expr {$time / 1000000}] -format %T] ; # ❶
    set time [expr {$time % 1000000}]
    set msec [expr {$time / 1000}]
    set usec [expr {$time % 1000}]

    return [format %s.%03ld.%03ld $sec $msec $usec]
}
```

❶ We assume time in units of microseconds.

Format Time As Seconds

```
<<stsatcl commands>>=
proc ::stsatcl::FormatTimeAsSec {time} {
    set sec [expr {$time / 1000000}]
    set time [expr {$time % 1000000}]
    set msec [expr {$time / 1000}]
    set usec [expr {$time % 1000}]

    return [format %3ld.%03ld.%03ld $sec $msec $usec]
}
```

Sequence Diagrams

Another way to view trace information is in the form of a sequence diagram. The seqdiag program can layout and render UML sequence diagrams from a simple textual specification of the diagram's components. The syntax of the sequence diagram is similar to that of **dot**. Using the captured state machine trace information, we can view the interactions of state models. We present several procedures in this group.

Diagram Traces

```
<<stsatcl commands>>=
proc ::stsatcl::diagtraces {traces args} {
    set result {}
    append result "seqdiag \{\n"
    append result "    activation = none;\n"
    foreach {option value} $args {
        append result "    $option = $value;\n"
    }

    relation foreach trace $traces -ascending Trace_Id {
        relation assign $trace
        if {$Source eq {}} {
            set Source EXTERNAL
        } else {
            set Source [namespace tail $Source]
        }
        set Target [namespace tail $Target]
        append result "    $Source ->> $Target \[label=\"$Event\"\];\n"
    }

    append result "\}\n"
```

```
    return $result
}
```

Diagram All Traces

```
<<stsatcl commands>>=
proc ::stsatcl::diagAllTraces {filename args} {
    set chan [open $filename w]
    try {
        chan puts $chan [diagtraces [relvar set Trace] {*}$args]
    } finally {
        chan close $chan
    }
    return
}

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

Diagram Class Traces

```
<<stsatcl commands>>=
proc ::stsatcl::diagClassTraces {filename classes args} {
    set chan [open $filename w]
    try {
        set traces [pipe {
            relvar set Trace >
            relation restrictwith ~\
                {$Class in $classes || [namespace tail $Class] in $classes}
        }] {} >~
        chan puts $chan [diagtraces $traces {*}$args]
    } finally {
        chan close $chan
    }
    return
}

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

Finishing the Example

Up to this point in discussing our running example, we have ignored the all too apparent fact that we don't have any washing machine hardware. No valves, no pumps, no tubs, nothing. Worse yet, we have no way to start the washing cycle. No buttons knobs or user interface of any kind. And yet we intend to run the example. Clearly, we have some loose ends to tie up.

Our example model describes a single XUML *domain*. Real applications are typically build from multiple domains, each domain confining itself to a single coherent subject matter. This is very much related to the concept of *separation of concerns*. So in our case, we have solved the problem of timing and sequencing activities to wash clothes but have assumed that some other domain will handle the the interaction with the sensors and valves and motors as well as with the our intrepid user who's just trying to wash some clothes.

To resolve this, we must provide two things:

1. A set of operations that other domains may invoke when washing machine control is needed. We call these operations *domain operations*. These operations form a set of services provided by our domain.

2. A set of procedure interfaces that our domain invokes when it needs service from another domain to which it has allocated some responsibility. We call these operations *external operations*.

For our example, we are assuming there exists some user interface that a user may interact with to get his clothes washed. We intend to provide only two domain operations:

1. Select a wash cycle.
2. Start a washing machine.

This will be a distinctly unfriendly washer as once it is started there will be no way to stop it before it completes the entire cycle.

For external operations, our domain assumes that valves can be opened and closed, motors can be started and stopped, and sensors can be enabled and disabled. Additionally, we assume that when the sensor detects the clothes tub as full or empty, it can manage to generate a signal into our domain to indicate the sensor state. We will see how that happens below.

Domain Operations

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

Start Washer

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

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

```
<<domain operations>>=
proc ::wmctrl::startWasher {washer} {
    set wm [WashingMachine selectOneWhere selected {
        [$selected readAttributes MachineID] eq $washer
    }]
    if {$wm eq {}} {
        error "unknown washer, \"$washer\""
    }
    $wm signal Start

    return
}
```

Select Cycle

On the class diagram, relationship `R1` determines which Washing Cycle will be used to control the operations. Selecting a wash cycle mean we must reform the `R1` relationship, *i.e.* unlink the existing cycle and link in a different one. The implementation of the `selectCycle` domain operation does just that.

```
<<domain operations>>=
proc ::wmctrl::selectCycle {washer cycle} {
    set wm [WashingMachine selectOneWhere selected {
        [$selected readAttributes MachineID] eq $washer
    }]
    if {$wm eq {}} {
        error "unknown washer, \"$washer\""
    }
```

```
    set wc [WashingCycle selectOneWhere selected {
        [$selected readAttributes CycleType] eq $cycle
    }]
    if {$wc eq {}} {
        error "unknown cycle, \"$cycle\""
    }
    $wm unlink R4 [$wm -> R4] ; # ❶
    $wm link R4 $wc

    return
}
```

❶ You must provide the instance currently participating in the link in order to unlink it. It's a basic sanity check.

External Operations

For external operations, we don't code the operation itself, it is after all *external*. But we must define its interface and invoke the operation at the appropriate time in our own processing.

For our example, there are six external operations that are assumed, two each for the motor, valve and water level sensor. Each of these operations passes an identifying value to the operation. This allows the external operations to determine exactly what is is to operate upon. We fill in the external operation invocations below.

```
<<MTR start operation>>=
my variable MotorID
::wmctrl::MOTOR::start $MotorID

<<MTR stop operation>>=
my variable MotorID
::wmctrl::MOTOR::stop $MotorID

<<WV open operation>>=
my variable ValveID
::wmctrl::VALVE::open $ValveID

<<WV close operation>>=
my variable ValveID
::wmctrl::VALVE::close $ValveID

<<WLS enable operation>>=
set machine [my -> R5 R1] ; # ❶
::wmctrl::SENSOR::enable [$machine readAttributes MachineID]

<<WLS disable operation>>=
set machine [my -> R5 R1]
::wmctrl::SENSOR::disable [$machine readAttributes MachineID]
```

❶ Sensors are identified in the same way as washing machines. So, to get a value for that identifier, we navigate the linkage back to where it is stored. This is valid since the class diagram constrains each Washing Machine to have exactly on Clothes Tub and each Clothes Tub to have exactly one Water Level Sensor.

At this point the example domain is fully coded. We still have a bit more work to actualy get the domain to run and we will finally do that below.

Introspection

It is very common in the Tcl world to provide the means to query the internals of things (*e.g.* the `info` command). Access to this kind of data can provide powerful opportunities for meta-programming in addition to aiding testing and debugging.

```
<<stsaclass constructor>>=
::oo::objdefine [self] {
    <<class introspection methods>>
}
```

Info Method

The `info` method provides a simple interface properties of the class as defined through the configuration script given used to construct the class.

xumlclass `info` *topic*

The method returns data associated with the *topic* argument which may be one of the following:

`attributes`
: Returns a list of the attribute names.

`links`
: Returns a list of the relationship links.

`states`
: Returns a list of state names.

`events`
: Returns a list of events.

`transitions`
: Returns a list of transitions. Each transition is also a list of three elements giving the source state, event and target state of the transition. Note that the entire <states> by <events> transition matrix is returned, *i.e.* the length of the returned list is the product of the number of states and the number of events. The order of the returned list is arbitrary.

`intialstate`
: Returns the initial state.

`defaulttrans`
: Returns the default transition, either “IG” or “CH”.

`terminals`
: Returns a list of terminal states.

`polyevents`
: Returns a list of polymorphic events.

```
<<class introspection methods>>=
method info {topic} {
    switch -exact -- $topic {
        attributes {
            my variable attrInfo
            return [dict keys $attrInfo]
        }
        links {
            my variable linkInfo
            return [dict keys $linkInfo]
        }
        states {
```

```
            my variable states
            return $states
        }
        events {
            my variable events
            return $events
        }
        transitions {
            my variable transitions
            set result [list]
            dict for {src trans} $transitions {
                dict for {evt dst} $trans {
                    lappend result [list $src $evt $dst]
                }
            }
            return $result
        }
        initialstate {
            my variable initialstate
            return $initialstate
        }
        defaulttrans {
            my variable defaulttrans
            return $defaulttrans
        }
        terminals {
            my variable terminals
            return $terminals
        }
        polyevents {
            my variable polyEvents
            return $polyEvents
        }
        default {
            tailcall ::stsatcl::DeclError UNKNOWN_INFO $topic
        }
    }
}
```

```
<<error code formats>>=
UNKNOWN_INFO        {unknown information topic, "%s"}
```

Info Method Tests

```
<<exported tests>>=
test info-1.0 {
    introspection on a class
} -setup {
    ::stsatcl STSAClass create in1 {
        attribute {Count 0} {Max 2}
        reference R23 -> foo
        statemodel {
            state s1 {} {
                puts "in s1"
            }
            transition s1 - e1 -> s2

            state s2 {} {
                puts "in s2"
            }
```

```
            transition s2 - e2 -> s1
        }
    }
} -cleanup {
    in1 destroy
} -body {
    set pass 1

    set attrs [in1 info attributes]
    set pass [expr {$pass && "Count" in $attrs && "Max" in $attrs}]
    set links [in1 info links]
    set pass [expr {$pass && "R23" in $links}]
    set states [in1 info states]
    set pass [expr {$pass && "s1" in $states && "s2" in $states}]
    set events [in1 info events]
    set pass [expr {$pass && "e1" in $events && "e2" in $events}]
    set trans [in1 info transitions]
    set pass [expr {$pass && [llength $trans] == 4}]
    set istate [in1 info initialstate]
    set pass [expr {$pass && $istate eq "s1"}]
    set deftrans [in1 info defaulttrans]
    set pass [expr {$pass && $deftrans eq "CH"}]
    set terms [in1 info terminals]
    set pass [expr {$pass && [llength $terms] == 0}]
    set poly [in1 info polyevents]
    set pass [expr {$pass && [llength $poly] == 0}]

    set pass
} -result {1}
```

Dot Method

The dot program is magical software that can layout and render directed graphs from a simple textual specification of the graph's nodes and arcs. It is often useful to view graphically the implementation aspects of a state model. The dot method produces a string that describes the state model in the syntax compatible with the dot program.

```
<<class introspection methods>>=
method dot {} {
    set result {}
    append result "digraph [namespace tail [self]] \{" \n
    append result "    node\[shape=\"box\"]" \n

    my variable states
    foreach state $states {
        if {$state eq "@"} { # ❶
            append result "    \"@\"\["\
                "shape=\"point\""\
                ",label=\"\""\
                "\]\n" ; # ❷
        } else {
            lassign [info class definition [self] ${state}__STATE__]\
                    arguments body ; # ❸
            set code "$state \{$arguments\} \{"
            append code\
                    [textutil::adjust::indent [textutil::adjust::undent $body]\
                    {    }] ; # ❹
            append code "\n\}"
            set labelCode {}
            set escapemap [list \\ \\\\ \" \\\"] ; # ❺
            foreach line [split $code \n] {
```

```
                append labelCode\
                    [string map $escapemap $line]\
                    "\\l"
            }
            set stProps "label=\"$labelCode\""

            my variable initialstate
            if {$state eq $initialstate} {
                append stProps ",style=\"bold\""
            }
            append result "    \"$state\"\[$stProps]" \n
        }
    }

    my variable transitions
    dict for {currstate trans} $transitions {
        dict for {event dststate} $trans {
            if {!($dststate eq "IG" || $dststate eq "CH")} {
                append result "    \"$currstate\" -> \"$dststate\"\
                        \[label=\"$event\"]" \n
            }
        }
    }

    append result "\}"

    return $result
}
```

1. The initial pseudo state has no activity associated with it.
2. Careful escaping is necessary since “dot” uses many characters in its syntax that have special meaning in Tcl.
3. we obtain the details of the state activity to insert the action as the “label” for the node.
4. This just makes the indentation consistent by removing all the indentation and then re-indenting the code with a fixed spacing.
5. We need a round of escape character substitution to insure “dot” does not interpret Tcl syntax incorrectly.

To perform some cleanup on the state activities, we use the `textutil::adjust` package from `tcllib`.

```
<<required packages>>=
package require textutil::adjust
```

Dot File Method

The `dotfile` method provides a convenient interface to write the `dot` description of the state model to a given *filename*.

xumlclass `dotfile` *filename*

`filename`
The name of the file to which the `dot` description of the state model is written.

```
<<class introspection methods>>=
method dotfile {filename} {
    set chan [open $filename w]
    try {
        chan puts $chan [my dot]
```

```
    } finally {
        chan close $chan
    }
    return
}
```

Draw Method

The `draw` method provides a convenient interface to render the state model graph using the `dot` program. It is necessary that `dot` be installed and be found along the execution path.

xumlclass `draw` *options*

`options`
: A set of `dot(1)` command line options that are given to `dot` when it is run. The string "%F" is substituted with the base name of the class, if present. Default value for *options* is "-Gcenter=1 -Gratio=auto -Gsize=7.5,10 -Tps -o%F.ps".

```
<<class introspection methods>>=
method draw {{dotopts {-Gcenter=1 -Gratio=auto -Gsize=7.5,10 -Tps -o%F.ps}}} {
    set dotexec [auto_execok dot]
    if {$dotexec eq {}} {
        tailcall ::stsatcl::DeclError NODOT
    }
    set dotopts [string map [list %F [namespace tail [self]]] $dotopts] ; #❶
    set chan [open "| $dotexec $dotopts" w]
    try {
        chan puts $chan [my dot]
    } finally {
        chan close $chan
    }
    return
}
```

❶ Substitute the file name token. The default file name is just the name of the class minus any namespace qualifiers.

```
<<error code formats>>=
NODOT        {cannot find \"dot\" executable}
```

Dot Method Tests

```
<<exported tests>>=
test dot-1.0 {
    draw state model with dot
} -setup {
    ::stsatcl STSAClass create dot1 {
        statemodel {
            state s1 {} {
                puts "in s1"
            }
            transition s1 - e2 -> s2

            state s2 {} {
                puts "in s2"
            }

            transition @ - e1 -> s1
```

```
        }
    }

    # We use "dotfile" to make an expected result file
    # which is closely examined manually to serve as
    # the comparison file.
    # dot1 dotfile dot_1_results
} -cleanup {
    dot1 destroy
} -body {
    set diagram [dot1 dot]
    string equal $diagram [viewFile dot_1_results]
} -result {1}
```

```
<<exported tests>>=
test draw-1.0 {
    draw and render state model with dot
} -setup {
    ::stsatcl STSAClass create dot1 {
        statemodel {
            state s1 {} {
                puts "in s1"
            }
            transition s1 - e2 -> s2

            state s2 {} {
                puts "in s2"
            }

            transition @ - e1 -> s1
        }
    }
} -cleanup {
    dot1 destroy
} -body {
    dot1 draw
} -result {}
```

The figure below shows our example Washing Machine state model as rendered by **dot**. These types of drawing are not particularly useful during the design of a state model but does represent the "as coded" state model and can be used to insure that the transfer of the analysis state model to code was accurate. Here we can compare the morphology of the state model graph to its design as well as example the translation of the state activity into its implementation.

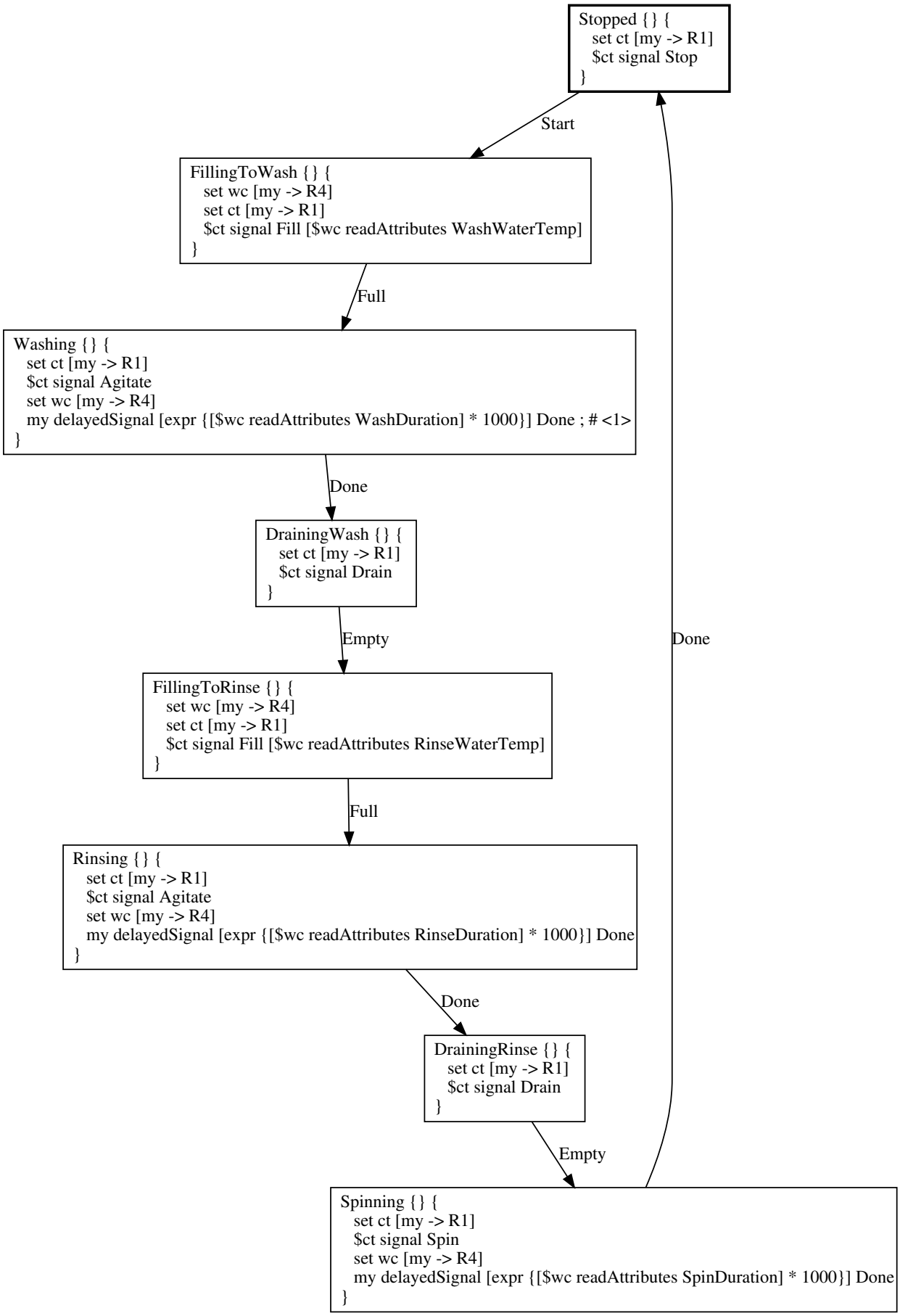

Figure 7: Dot Drawing of Washing Machine State Model

Utility Methods

A number of unexported utility methods are presented here without much discussion. This is simply common code factored into methods. Most of these methods provide argument validation for the DSL commands.

Check Link Name

```
<<utility methods>>=
method CheckLinkName {rname} {
    classvariable linkInfo
    if {![dict exists $linkInfo $rname]} {
        tailcall ::stsatcl::DeclError UNKNOWN_LINKAGE $rname
    }
}
```

```
<<error code formats>>=
UNKNOWN_LINKAGE     {unknown linkage, "%s"}
```

Check Reference Object

```
<<utility methods>>=
method CheckReferenceObj {rname obj} {
    classvariable linkInfo

    set objclass [info object class $obj]
    switch -exact -- [dict get $linkInfo $rname type] {
        reference {
            set destclass [dict get $linkInfo $rname dest]
            if {!($objclass eq $destclass ||\
                    $destclass in [info class superclasses $objclass])} {
                tailcall ::stsatcl::DeclError NOT_AN_INSTANCE\
                    $obj [dict get $linkInfo $rname dest]
            }
        }
        partition {
            set subclasses [dict get $linkInfo $rname subclasses]
            foreach subclass $subclasses {
                if {$objclass eq $subclass ||\
                    $subclass in [info class superclasses $objclass]} {
                    return
                }
            }
            tailcall ::stsatcl::DeclError NOT_AN_SUBCLASS\
                $obj [join $subclasses {, }]
        }
        default {
            tailcall ::stsatcl::DeclError UNKNOWN_LINKTYPE\
                    [dict get $linkInfo $rname type]
        }
    }
}
```

```
<<error code formats>>=
NOT_AN_INSTANCE {instance, "%s", is not an instance of class, "%s"}
NOT_AN_SUBCLASS {instance, "%s", is not an instance of any subclasses, "%s"}
```

Resolve Object

```
<<utility methods>>=
method ResolveObj {obj} {
    if {[string range $obj 0 1] ne "::"} {
        set obj\
            [string trimright [uplevel 2 namespace current] :]::$obj
    }
    return $obj
}
```

Validate Event

```
<<utility methods>>=
method ValidateEvent {event} {
    classvariable events
    if {$event ni $events} {
        tailcall ::stsatcl::DeclError UNKNOWN_EVENT $event
    }
}
```

```
<<error code formats>>=
UNKNOWN_EVENT {unknown event, "%s"}
```

Event Source

```
<<utility methods>>=
method EventSource {} {
    try {
        set src [lindex [uplevel 1 {self caller}] 1] ; # ❶
    } on error {} {
        set src {}
    }
    return $src
}
```

❶ `self caller` throws an error if the caller is not a method.

Forward Polymorphic Event

```
<<utility methods>>=
method ForwardPolyEvent {type src event arglist} {
    classvariable partitions
    foreach partition $partitions { # ❶
        my variable $partition
        set target [set $partition] ; # ❷
        if {$target eq {}} {
            tailcall ::stsatcl::DeclError UNCOND $partition
        }
        ::stsatcl::TracePolymorphic $src $event $target [self] $partition
        $target {*}$type $event {*}$arglist ; # ❸
    }
}
```

1. Polymorphic events are propagated down all the partition linkages that are specified for the class.
2. Polymorphic event dispatch implies a runtime traversal of the partition linkage from superclass to subclass. However unlike the case of a conventional partition traversal, we don't care about the type of the subclass instance. All subclasses must be prepared to handle the polymorphic event.
3. Despite all the variable substitution, this command signals the event to the subclass instance. Here *type* controls whether the signal is delayed and, if delayed, the delay time. Admittedly, the construct is rather strained but was done this way so that there is only a single invocation of `TracePolymorphic` in the package and much of the code for a method specific to delayed event forwarding would be duplication.

Running the Example

We are finally ready to run our example. To do so we have to figure out two last things:

1. How to get the `startWasher` domain operation invoked.
2. What to do about the six external operations.

Put another way, we need to *drive* the domain operations and *stub* the external operations in order to resolve fully all the domain dependencies. One convenient aspect of domain based systems is that the domain will run in isolation as long as you can drive the operations and stub the dependencies. The domain code itself is really none wiser.

Stubbing the External Operations

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

```
<<external operation stubs>>=
namespace eval ::wmctrl::MOTOR {
    ::logger::initNamespace [namespace current] info
    proc start {motor} {
        log::info "starting motor, \"$motor\""
    }
    proc stop {motor} {
        log::info "stopping motor, \"$motor\""
    }
}

namespace eval ::wmctrl::VALVE {
    ::logger::initNamespace [namespace current] info
    proc open {valve} {
        log::info "opening valve, \"$valve\""
    }
    proc close {valve} {
        log::info "closing valve, \"$valve\""
    }
}
```

The SENSOR operations present a bit more difficulty to stub. In this case, there *is* feedback from the interaction. We must signal back the state of water level as being full or empty. So in some sense we must *simulate* the action of the sensor. If you carefully examine the example, you find that we can assume the tub starts empty and each time the sensor is enabled we are trying to detect the opposite state of the tub. So enabling the sensor on an empty tub means we are looking for when the tub goes full and *vice versa*. So it sufficient to simulate the sensor to simply toggle the tub state and then announce the new state at some time when we think the new state will have been reached. We will delay the announcement of the new state for some time to simulate the water filling or draining. To make the example run in reasonable times, we assume the tub will fill or empty in 3 seconds. That's quick!

```
<<external operation stubs>>=
namespace eval ::wmctrl::SENSOR {
    ::logger::initNamespace [namespace current] info
    variable sensorState TubEmpty
    variable sensorEvent

    proc enable {machine} {
        log::info "enable sensor on machine, \"$machine\""

        variable sensorState
        variable sensorEvent

        set newState\
            [expr {$sensorState eq "TubEmpty" ? "TubFull" : "TubEmpty"}] ; # ❶
        set sensorEvent [after 3000 [namespace code\
            [list trigger $machine $newState]]] ; # ❷
    }

    proc disable {machine} {
        log::info "disable sensor on machine, \"$machine\""
        variable sensorEvent
        after cancel $sensorEvent
    }
```

❶ Toggle the tub state.

❷ Use the `after` *ms* form of the command to delay announcing the new state.

Delivering the indication that the Water Level Sensor has detected a change in the water level really means we want to send the appropriate ClothesTub instance either the **TubFull** or **TubEmpty** event. Conveniently, we have been keeping track of the sensor state using string named the same as the event we intend to deliver. So all we have to do is search for the correct Washing Machine that matches the sensor, navigate `R1` to the ClothesTub and signal an event.

```
<<external operation stubs>>=
    proc trigger {machine value} {
        variable sensorState
        set sensorState $value

        set wm [::wmctrl::WashingMachine selectOneWhere selected {
            [$selected readAttributes MachineID] eq $machine
        }]
        set ct [$wm -> R1]
        $ct signal $sensorState
    }
}
```

Before we can start the example running, we have to figure out how we are going to stop it. Recall that to dispatch state machine events we must enter the Tcl event loop. We will use the `vwait` command to do that. But we need some way to break out of the event loop so that we can look at the results of the run. To do that we will add setting a global variable to the state activity of the Stopping Spin state in the ClothesTub class. This state is entered when the cycle is done and it is our intent to regain control of the execution flow after each washing cycle. Note that we are adding this synchronization solely to be able to run one washing cycle in our example. An actual application would most likely run forever.

```
<<CT stopping spin activity>>=
set ::done 1
```

An finally, yes truly finally, we are in a position to drive the domain opertions to select a cycle and run the washer through the cycle. We will turn on tracing so we can see what happened.

```
<<running the example>>=
::stsatcl traceControl on

::wmctrl::selectCycle WasherOne PermPress
puts "**** Start Run"
::wmctrl::startWasher WasherOne

vwait ::done ; # ❶

puts "**** Finish Run"
::stsatcl traceControl off

puts "**** Trace Begin"
puts [::stsatcl::formatTraces [::stsatcl::decodeAllTraces]]
puts "**** Trace End"

::wmctrl::WashingMachine draw\
    {-Gcenter=1 -Gratio=auto -Gsize=7.5,10 -Tsvg -oimages/WashingMachine.svg} ; # ❷

::stsatcl diagAllTraces images/wmctrl-seqdiag.diag span_height 7 ; # ❸
```

❶ Enter the Tcl event loop. State machine events will then be dispatched and the washer will operate.

❷ While we are at it, we need an example to show how **dot** will draw a state model.

❸ Draw a sequence diagram of the example run. This will allow us to compare the textual log with a graphic containing the same information.

Example Run Results

After running the example we obtain the following output.

Output From Running the Example

```
**** Start Run
[Mon Aug 11 16:45:08 PDT 2014] [wmctrl::SENSOR] [info] 'enable sensor on machine, ↩
   "WasherOne"'
[Mon Aug 11 16:45:08 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Hot"'
[Mon Aug 11 16:45:08 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Cold"'
[Mon Aug 11 16:45:11 PDT 2014] [wmctrl::SENSOR] [info] 'disable sensor on machine, ↩
    "WasherOne"'
[Mon Aug 11 16:45:11 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Hot"'
[Mon Aug 11 16:45:11 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Cold"'
[Mon Aug 11 16:45:11 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Agitator"'
[Mon Aug 11 16:45:26 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Agitator"'
[Mon Aug 11 16:45:26 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Drain"'
[Mon Aug 11 16:45:26 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Pump"'
[Mon Aug 11 16:45:26 PDT 2014] [wmctrl::SENSOR] [info] 'enable sensor on machine, ↩
   "WasherOne"'
[Mon Aug 11 16:45:29 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Pump"'
[Mon Aug 11 16:45:29 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Drain"'
[Mon Aug 11 16:45:29 PDT 2014] [wmctrl::SENSOR] [info] 'disable sensor on machine, ↩
    "WasherOne"'
[Mon Aug 11 16:45:29 PDT 2014] [wmctrl::SENSOR] [info] 'enable sensor on machine, ↩
   "WasherOne"'
[Mon Aug 11 16:45:29 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Cold"'
[Mon Aug 11 16:45:32 PDT 2014] [wmctrl::SENSOR] [info] 'disable sensor on machine, ↩
    "WasherOne"'
```

```
[Mon Aug 11 16:45:32 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Hot"'
[Mon Aug 11 16:45:32 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Cold"'
[Mon Aug 11 16:45:32 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Agitator"'
[Mon Aug 11 16:45:42 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Agitator"'
[Mon Aug 11 16:45:42 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Drain"'
[Mon Aug 11 16:45:42 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Pump"'
[Mon Aug 11 16:45:42 PDT 2014] [wmctrl::SENSOR] [info] 'enable sensor on machine,  ↩
   "WasherOne"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Pump"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Drain"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::SENSOR] [info] 'disable sensor on machine, ↩
    "WasherOne"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::VALVE] [info] 'opening valve, "Drain"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Pump"'
[Mon Aug 11 16:45:45 PDT 2014] [wmctrl::MOTOR] [info] 'starting motor, "Spin"'
[Mon Aug 11 16:46:00 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Spin"'
[Mon Aug 11 16:46:00 PDT 2014] [wmctrl::MOTOR] [info] 'stopping motor, "Pump"'
[Mon Aug 11 16:46:00 PDT 2014] [wmctrl::VALVE] [info] 'closing valve, "Drain"'
**** Finish Run
**** Trace Begin
16:45:08.492.559:   0.000.000: Transition: {} - Start {} -> ::wmctrl::washer ==>  ↩
   Stopped -> FillingToWash
16:45:08.493.200:   0.000.641: Transition: ::wmctrl::washer - Fill {Warm} -> :: ↩
   wmctrl::tub ==> Empty -> Filling
16:45:11.519.556:   3.026.356: Transition: {} - TubFull {} -> ::wmctrl::tub ==>  ↩
   Filling -> Full
16:45:11.520.873:   0.001.317: Transition: ::wmctrl::tub - Full {} -> ::wmctrl:: ↩
   washer ==> FillingToWash -> Washing
16:45:11.521.779:   0.000.906: Transition: ::wmctrl::washer - Agitate {} -> :: ↩
   wmctrl::tub ==> Full -> Agitating
16:45:26.521.940:  15.000.161: Transition: ::wmctrl::washer - Done {} -> ::wmctrl ↩
   ::washer ==> Washing -> DrainingWash
16:45:26.522.373:   0.000.433: Transition: ::wmctrl::washer - Drain {} -> ::wmctrl ↩
   ::tub ==> Agitating -> Emptying
16:45:29.524.205:   3.001.832: Transition: {} - TubEmpty {} -> ::wmctrl::tub ==>  ↩
   Emptying -> Empty
16:45:29.525.591:   0.001.386: Transition: ::wmctrl::tub - Empty {} -> ::wmctrl:: ↩
   washer ==> DrainingWash -> FillingToRinse
16:45:29.526.062:   0.000.471: Transition: ::wmctrl::washer - Fill {Cold} -> :: ↩
   wmctrl::tub ==> Empty -> Filling
16:45:32.526.973:   3.000.911: Transition: {} - TubFull {} -> ::wmctrl::tub ==>  ↩
   Filling -> Full
16:45:32.527.912:   0.000.939: Transition: ::wmctrl::tub - Full {} -> ::wmctrl:: ↩
   washer ==> FillingToRinse -> Rinsing
16:45:32.528.373:   0.000.461: Transition: ::wmctrl::washer - Agitate {} -> :: ↩
   wmctrl::tub ==> Full -> Agitating
16:45:42.528.474:  10.000.101: Transition: ::wmctrl::washer - Done {} -> ::wmctrl ↩
   ::washer ==> Rinsing -> DrainingRinse
16:45:42.528.943:   0.000.469: Transition: ::wmctrl::washer - Drain {} -> ::wmctrl ↩
   ::tub ==> Agitating -> Emptying
16:45:45.530.715:   3.001.772: Transition: {} - TubEmpty {} -> ::wmctrl::tub ==>  ↩
   Emptying -> Empty
16:45:45.531.999:   0.001.284: Transition: ::wmctrl::tub - Empty {} -> ::wmctrl:: ↩
   washer ==> DrainingRinse -> Spinning
16:45:45.532.606:   0.000.607: Transition: ::wmctrl::washer - Spin {} -> ::wmctrl ↩
   ::tub ==> Empty -> Spinning
16:46:00.532.684:  15.000.078: Transition: ::wmctrl::washer - Done {} -> ::wmctrl ↩
```

```
    ::washer ==> Spinning -> Stopped
16:46:00.533.169:   0.000.485: Transition: ::wmctrl::washer - Stop {} -> ::wmctrl ↩
    ::tub ==> Spinning -> StoppingSpin
**** Trace End
```

The first section is the set of log message that show the interaction with the motors, values and sensors. This amounts to a trace of the external side effects that the domain asserts on the washing machine hardware. The second portion show the chronological trace of the state machine event dispatch. The first column of the trace is the time of day and the second column is the time difference between the last trace entry (in the form of *s.ms.us*). The remainder of the trace show the details of the event dispatch. The first portion of the *Transition* trace shows the event being dispatched from a source instance to a target instance. The state machine transition, from current state to new state, is then shown after the "$\Longrightarrow$" symbol.

The figure below shows the trace information in sequence diagram form.

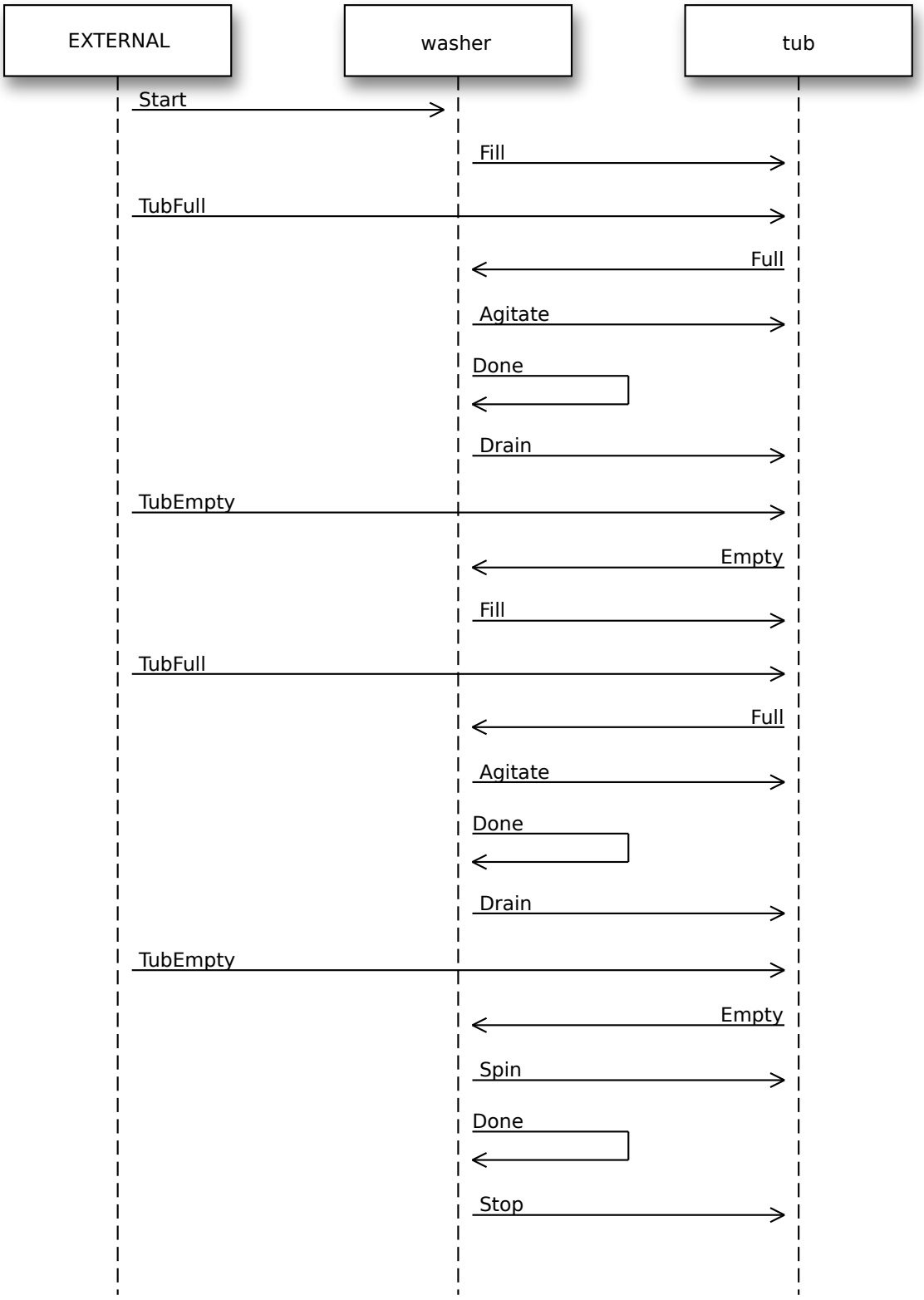

Figure 8: Example Sequence Diagram

The track labeled **EXTERNAL** shows events that originate outside of the domain. The example begins with a **Start** event being sent to the **washer** instance. The other external events arise from the Water Level Sensor detecting the status of the Clothes Tub. The clothes are cleaned by a sequence of events exchanged between the **washer** and **tub** to coordinate the steps of the washing

cycle to clean clothes. Although the sequence diagram does not tell the whole story of the program execution (*e.g.* the side effects caused the state activities is often very important), it does go a long way to showing the chronological unfolding of the example's execution.

Code Organization

In this section we show the organization of the files that can be tangled from the literate source. Each *root chunk* is described in a section below.

Source Code

```
<<stsatcl.tcl>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2014 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.

package require Tcl 8.6

<<required packages>>
```

```
<<helper commands>>

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

    variable version 1.0a2

    <<stsatcl data>>
}

<<stsatcl commands>>

package provide stsatcl $::stsatcl::version
```

Unit Tests

```
<<stsatcl.test>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2014 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.
```

```
package require Tcl 8.6
package require cmdline
package require logger

source ../src/stsatcl.tcl
chan puts "testing stsatcl version: [package require stsatcl]"

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

package require tcltest
eval tcltest::configure $argv

namespace eval ::stsatcl::test {
    ::logger::initNamespace [namespace current] $::options(level)

    namespace import ::tcltest::*

    <<test utility procs>>
    <<meta constructor tests>>
    <<constructor tests>>
    <<exported tests>>

    cleanupTests
}
```

Package Index

We also provide a root chunk to extract a package index file.

```
<<pkgIndex.tcl>>=
package ifneeded stsatcl 1.0a2 [list source [file join $dir stsatcl.tcl]]
```

Example Code

```
<<wmctrl.tcl>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2014 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.

package require Tcl 8.6
package require logger

source stsatcl.tcl

namespace eval ::wmctrl {

    variable version 1.0a2

    ::logger::initNamespace [namespace current] info

    namespace import ::stsatcl::STSAClass

    <<WM class>>
    <<WC class>>
    <<CT class>>
    <<WV class>>
    <<MTR class>>
    <<WLS class>>
    <<initial instance population>>
}
<<domain operations>>
<<external operation stubs>>
<<running the example>>
```

Bibliography

Books

[1] [mb-xuml] Stephen J. Mellor and Marc J. Balcer, Executable UML: a foundation for model-driven architecture, Addison-Wesley (2002), ISBN 0-201-74804-5.

[2] [rs-xuml] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, Model Driven Architecture with Executable UML, Cambridge University Press (2004), ISBN 0-521-53771-1.

[3] [ls-build], Leon Starr, How to Build Shlaer-Mellor Object Models, Yourdon Press (1996), ISBN 0-13-207663-2.

[4] [sm-data] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in Data, Prentice-Hall (1988), ISBN 0-13-629023-X.

[5] [sm-states] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in States, Prentice-Hall (1992), ISBN 0-13-629940-7.

Index