ROSEA

A Relation Oriented Software Execution Architecture

Andrew Mangogna
amangogna@modelrealization.com

22nd Annual Tcl/Tk Conference
October 19-23, 2015

Abstract

Rosea is a Tcl package that implements a data and execution domain that is suitable as the target platform for translating executable software models into programs using Tcl as the implementation language. Translation is accomplished by capturing the characteristics of the software model using a domain specific language from which the data structures used by the run time component are built. This paper presents the design and implementation of rosea showing how Executable UML model semantics are mapped to Tcl language constructs. Model classes are held in relation variables and class relationships are enforced as referential integrity constraints between the relation variables. Dynamics are implemented as Moore type finite state machines. Processing is event driven and transactional in nature with the state of the program data being rolled back if integrity checks fail.

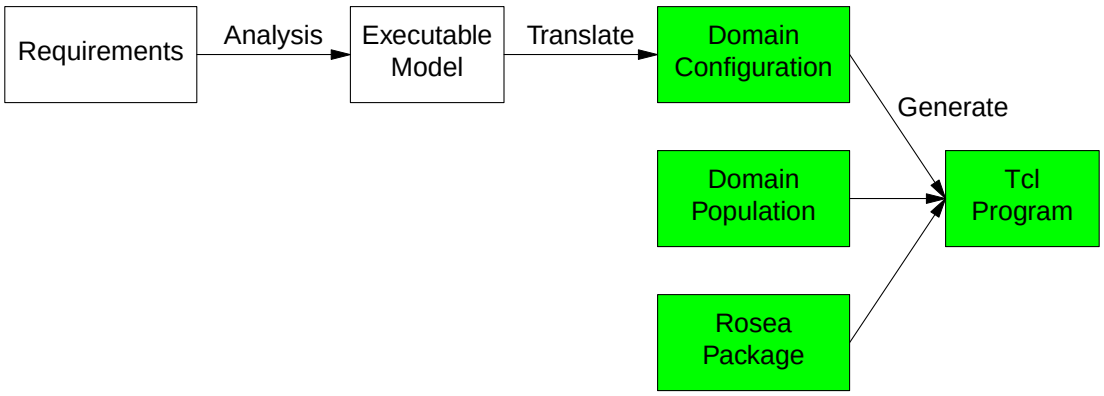
Introduction

Rosea is an acronym for Relation Oriented Software Execution Architecture. Rosea is a Tcl package written entirely in Tcl script but is heavily dependent upon TclRAL a "C" based Tcl extension that implements the relational algebra.

The purpose of rosea is to provide an implementation target for translating executable models of software into running programs using Tcl as the implementation language. In this paper we present two distinct views of the rosea package. First, rosea is discussed from a modeling / translating point of view. From this perspective, rosea provides a means for model execution semantics to be mapped onto Tcl language constructs. Secondly, rosea is discussed from a strictly Tcl language point of view. From that perspective, rosea is an object system based on namespace ensembles with object data stored as relation variables.

The Modeling View of Rosea

The diagram below shows a general outline of a software development workflow that is accomplished by translation.



Requirements are analyzed to give *Executable Models*. A software system is divided into *Domains*. A domain encapsulates a cohesive subject matter. A domain is then translated into the rosea configuration DSL (Domain Specific Language). That configuration used to *generate* data that drives the run time execution of the domain. Each domain is supplied with a initial *population* of class instances. By combining the translated domain with its initial population and the rosea package, a running Tcl program is obtained.

Space considerations do now allow us to discuss in this paper how requirements are gathered and then analyzed to yield executable models. We will only cover those items colored green the in above diagram.

We insist that the models must be executable if we are to translate them and to that end we insist that models conform to the semantics of Executable UML. Executable UML uses a well defined subset of UML that has rigorous execution semantics. There are several good books on Executable UML[1] [2] [3] and readers are encourage to consult at least one of them.

From the modeling perspective, rosea provides an implementation of the Executable UML semantics using Tcl as the implementation language. The rosea package consists of three principle components:

(1) A domain specific language that is used to encode the model graphic. The DSL is consists of a set of Tcl commands and so the DSL would be classified as an *internal* DSL. The DSL provides commands to define classes, attribute and relationships and we shall see examples below.

(2) A set run time procedures that activities in the model may invoke to access data, navigate relationships and signal state machine events. The run time code is tailored to match the semantics of the models and includes, among other functions, procedures to access instance data, navigate class relationships and signal events.

(3) A generator that translates the configuration information provided by the DSL into the data structures used by the run time. The run time is completely data driven. The exact behavior, in modeling terms, that a domain exhibits depends upon the data values

---

[1] Stephen J. Mellor and Marc J. Balcer, *Executable UML: a foundation for model-driven architecture*, Addison-Wesley (2002), ISBN 0-201-74804-5.

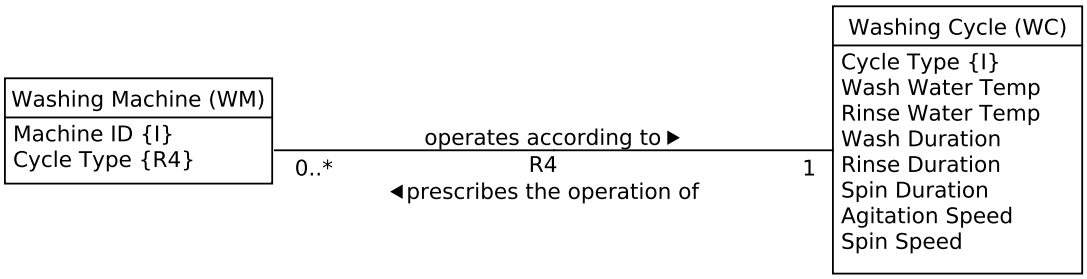
[2] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, *Model Driven Architecture with Executable UML*, Cambridge University Press (2004), ISBN 0-521-53771-1.

[3] Leon Starr, *How to Build Shlaer-Mellor Object Models*, Yourdon Press (1996), ISBN 0-13-207663-2.

supplied to the run time code For example, classes that exhibit state behavior supply state transition information to the run time code. The behavior of the class depends upon the sequence of events received by a class instance as sequenced by the state transitions.

Examples of Translating Data

To get a sense of the rosea domain configuration DSL, we will show an example of how to translate a model using rosea. The example we show here is fully worked out in the literate program document that is part of the rosea package. In this paper, we will examine only fragments of the model. Consider the following fragment of a class diagram for a washing machine control domain. This domain controls a primitive automatic clothes washing machine.



This class model fragment states, roughly, that there exists a **Washing Machine** class that is identified by a **Machine ID** attribute and is characterized by a **Cycle Type** attribute. There is also a **Washing Cycle** class that is identified by a **Cycle Type** attribute and characterized by a number of attributes that hold water temperatures, durations and speeds. There also an association, **R4**, between **Washing Machine** instances and **Washing Cycle** instances. The semantics of the association is stated by the verb phrases on the class model graphic. A **Washing Machine operates according to** *exactly one* **Washing Cycle** and each **Washing Cycle prescribes the operation of** *zero or more* **Washing Machines**

As we set about to translate this fragment, recall that models are organized into domains and the DSL has a `domain` command. The `domain` command supplies a name for the domain and a script which defines the characteristics of the domain.

```
domain wmctrl {
    # Domain definition for the Washing Machine Control Domain
    # .....
}
```

The script for the `domain` command is an ordinary Tcl script, but is evaluated in a context where specific commands may be invoked to configure the domain. In the remaining portion of the example, we will assume that the commands shown are invoked inside the script portion of a `domain` command.

Starting with the **Washing Cycle** class, it is defined as follows.

```
class WashingCycle {
    attribute CycleType string -id 1
    attribute WashWaterTemp string
    attribute RinseWaterTemp string
    attribute WashDuration int
    attribute RinseDuration int
    attribute SpinDuration int
    attribute AgitationSpeed string
    attribute SpinSpeed string
}
```

The `class` command gives a name to the class and takes a script that defines the properties of the class. Class attributes are defined by invoking the `attribute` command, which takes a name for the attribute and a Tcl data type. It may seem unusual to specify a data type for a language like Tcl, where *everthing is a string* is such a firmly held principle. This is an integrity check on the values that may be assigned to an attribute. For example, the **WashDuration** attribute is declared as an `int`. This shows the intent to store numbers in the attribute and potentially to perform arithmetic operations on the attribute values. Consequently, the **WashDuration** attribute is not allowed to be set to a value that cannot reasonable be converted into an integer (by the usual Tcl rules). This prevents attributes being set to values that have no possibility of being correct. More detailed control of subsets of integer values is also available. Of course, `string` is the universal Tcl type and any attribute whose type is declared as `string` can be set to anything.

All classes must have one or more identifiers. An identifier consists of one or more attributes whose values must be unique. For the case of **Washing Cycle**, there is one identifier and that identifier consists on only one attribute, as indicated by the `-id 1` option on the **CycleType** attribute[4]. This option corresponds to the **{I}** notation on the model graphic.

The two classes in the model also participate in an *association* named, **R4**. Associations define the way that class instances are related to each other. The model graphic indicates the multiplicity and conditionality of the relationship using standard UML annotations and indicates the semantics of the relationship using verb phrases. The `association` command captures the name of association, the participating classes and the multiplicity and conditionality.

```
association R4 WashingMachine 0..*--1 WashingCycle
```

There is a direct correspondence between the `association` command arguments and the graphical notion for the association.

Associations are realized by having attributes of one class refer to identifying attributes of another class. By *refer to*, we mean that the value of the referring attribute is equal to the value of the referenced identifying attribute. Readers familiar with relational database management systems will recognize this concept as similar to a *foreign key constraint* found in SQL. The model graphic shows how the relationship is realized by the **{R4}** annotation. The **Cycle Type** attribute in the **Washing Machine** class is annotated to indicate its role in realizing the **R4** relationship. This shows up in the definition of the **WashingMachine** class shown below.

[4] If there were multiple attributes in the first identifier, then the `-id 1` option would be given for each attribute in the identifier.

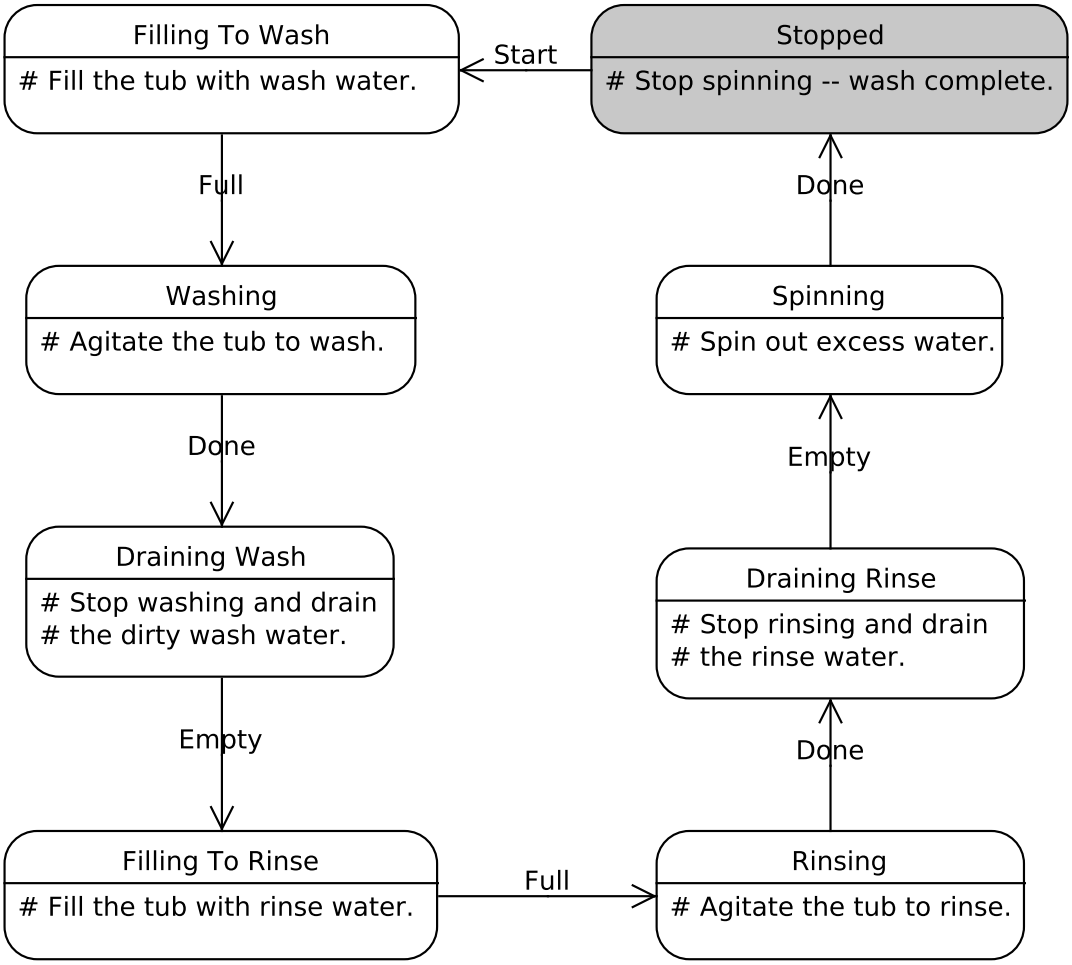
```
class WashingMachine {
    attribute MachineID string -id 1
    attribute CycleType string
    reference R4 WashingCycle -link CycleType

    # Other aspects of the Washing Machine class
    # ...
}
```

The `reference` command states that the **WashingMachine** class realizes the association, **R4**, to the **WashingCycle** class by linking the value of the **CycleType** attribute in **WashingMachine** to be the same value as the **CycleType** attribute for some instance in the **WashingCycle** class. The rosea run time code insures that this invariant is not violated and any attempt to do so results in the state of the program being rolled back to a valid set of values.

Examples of Translating State Models

Executable models sequence and coordinate processing by using state models. Below is a state model diagram for the **Washing Machine** class.



This diagram shows how the washing machine washes clothes by progressing through a series of states filling the clothes tub with water, washing, draining, rinsing and spinning the tub to obtain clean clothes. The transitions from one state to another are driven by the events as shown on the labels of the arrows directed from one state to another.

This is a Moore type state model, *i.e.* each state is associated with an activity that is executed upon entry into the state. The initial state, **Stopped**, is shown with the grayed out background. The rosea configuration DSL supports directly declaring a state model. This is done with the `statemodel` command that is part of a class definition. Below the **WashingMachine** class is shown with its state model definition.

```
class WashingMachine {
    attribute MachineID string -id 1
    attribute CycleType string
    reference R4 WashingCycle -link CycleType

    statemodel {
        initialstate Stopped

        state Stopped {} {
            # Stop spinning -- wash complete.
        }
        transition Stopped - Start -> FillingToWash
        state FillingToWash {} {
            # Fill the tub with wash water.
        }
        transition FillingToWash - Full -> Washing
        state Washing {} {
            # Agitate the tub to wash.
        }
        transition Washing - Done -> DrainingWash
        state DrainingWash {} {
            # Stop washing and drain
            # the dirty wash water.
        }
        transition DrainingWash - Empty -> FillingToRinse
        state FillingToRinse {} {
            # Fill the tub with rinse water.
        }
        transition FillingToRinse - Full -> Rinsing
        state Rinsing {} {
            # Agitate the tub to rinse.
        }
        transition Rinsing - Done -> DrainingRinse
        state DrainingRinse {} {
            # Stop rinsing and drain
            # the rinse water.
        }
        transition DrainingRinse - Empty -> Spinning
        state Spinning {} {
            # Spin out excess water.
        }
        transition Spinning - Done -> Stopped
    }
}
```

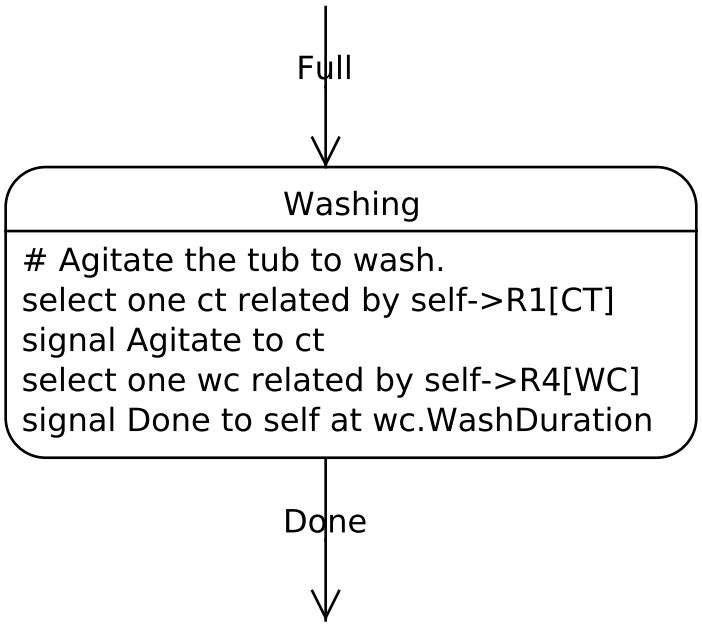
The `state` command specifies the name of the state, any arguments to the state and the Tcl script to be executed when the state is entered. The interface to the `state` command is the same as the `proc` command[5]. We do not show any of the processing above, only the structural aspects of the state model. We will show the processing for one state below.

[5] Indeed state activities are realized as Tcl proc's.

The `transition` command defines which new state is entered when the state machine is some starting state and an event is received. Transitions and states may be defined in any order. The order we have chosen above emphasizes the state with its outgoing transition and the simple circular nature of this particular state model.

Examples of Translating State Activities

Each state has an activity associated with it that is a Tcl script which is run when the state is entered. Rosea provides a number of commands that state activities may invoke to accomplish any model level actions that it may need to do. Since the state activity is also a Tcl script, it has access to all the functionality provided by the Tcl language itself. We will not detail all the model level actions a state activity may do, but two common ones are to navigate a relationship and to signal an event. The graphic below shows a portion of the state model for the **Washing Machine** class, complete with the action specification of what happens when the state is entered. The **Washing** state of the **Washing Machine** class is entered by receiving the **Full** event and remains in that state until the **Done** event is received. The processing is expressed in an action language that describes what the state activity must accomplish.



This action language description of the processing is translated into Tcl as follows.

```
state Washing {} {
    # Agitate the tub to wash.
    set ct [findRelated $self ~R1]
    signal $ct Agitate
    set wc [findRelated $self R4]
    delaysignal [readAttribute $wc WashDuration] $self Done
}
```

Each state activity has an implicitly defined variable named `self`. This variable holds a reference to the particular class instance for which the state activity is executing. The `findRelated` command traverses a relationship and finds all the class instances that are related. The `signal` command signals an event. The `delaysignal` command signals an event at a given time in the future. The `readAttribute` command obtains the values of one or more attributes given a reference to a class instance.

For the **Washing** state, the related **Clothes Tub** instance is signaled with the **Agitate** event to start the tub agitating the clothes. The agitation will continue for the length of time specified in the related **Washing Cycle** instance as given by the **Wash Duration** attribute. From the fragment of the model graphic we saw above, we know that traversing relationship, **R4**, will yield exactly one related instance of **Washing Cycle** because of the multiplicity and conditionality of the **R4** relationship on the **Washing Cycle** side (*i.e.* that side is marked as **1**). The **Done** event is signaled to ourselves when the wash duration amount of time has elapsed. When the **Done** event is received, the instance transitions to another state and performs the processing required to stop washing and continue the other steps of obtaining clean clothes.

Run Time Processing

Interacting state machines play an important role in sequencing the processing of a domain. Events are signaled to class instances and state machine receive the events and transition to a new state. Rosea uses the Tcl event loop to implement state machine events and state activities are executed as an `after` command callback.

Typically, an event is delivered from outside of a domain and a cascade of activities is executed as a result of the transition. State machines signal other state machines which causes the *thread of control* to continue as the signaled events are received by the class instances. Eventually, the result of the original event is fully realized, state machines stop signaling events and the domain waits for the next external stimulus. Rosea executes the thread of control as a *transaction* on the underlying data model. At the end of the transaction, all the integrity constraints represented in the class model as relationships are checked. If the data model is not consistent, an error is thrown and the data is rolled back to its previous set of values. In this way, the domain sequences from one consistent data state to another as a result of executing threads of control started by events arriving from the environment of the domain.

The Tcl View of Rosea

The purpose of the rosea package is to provide a convenient means to implement executable model semantics in the Tcl language. We have seen how an executable model of a domain may be expressed in the domain specific language provided by rosea. In this section, we discuss some of the details of how model semantics are mapped onto Tcl language constructs. In the end, we are building Tcl programs regardless of the view of those programs provided by the DSL.

From the Tcl perspective, rosea is an extension that provides an object system. However, the object system provided by rosea is based on the relational model of data rather than the more conventional concepts of type hierarchy and run time polymorphic function dispatch. Since version 8.6, Tcl has had core support for object orientation. The core has basic object commands intended to provide the foundation upon which full featured object systems may be built, *e.g.* IncrTcl. In previous releases, Tcl supported the concept of *namespace ensembles*. A

namespace ensemble provides the means to map subcommand names onto command prefixes. Using this, one may construct object oriented commands in Tcl via namespace ensembles. Rosea uses namespace ensembles extensively. Each domain is realized as namespace ensemble command for its domain operations and has a namespace defined whose name matches that of the domain. In the domain namespace, each class of the domain will have an ensemble command with subcommands being defined for all the class based operations.

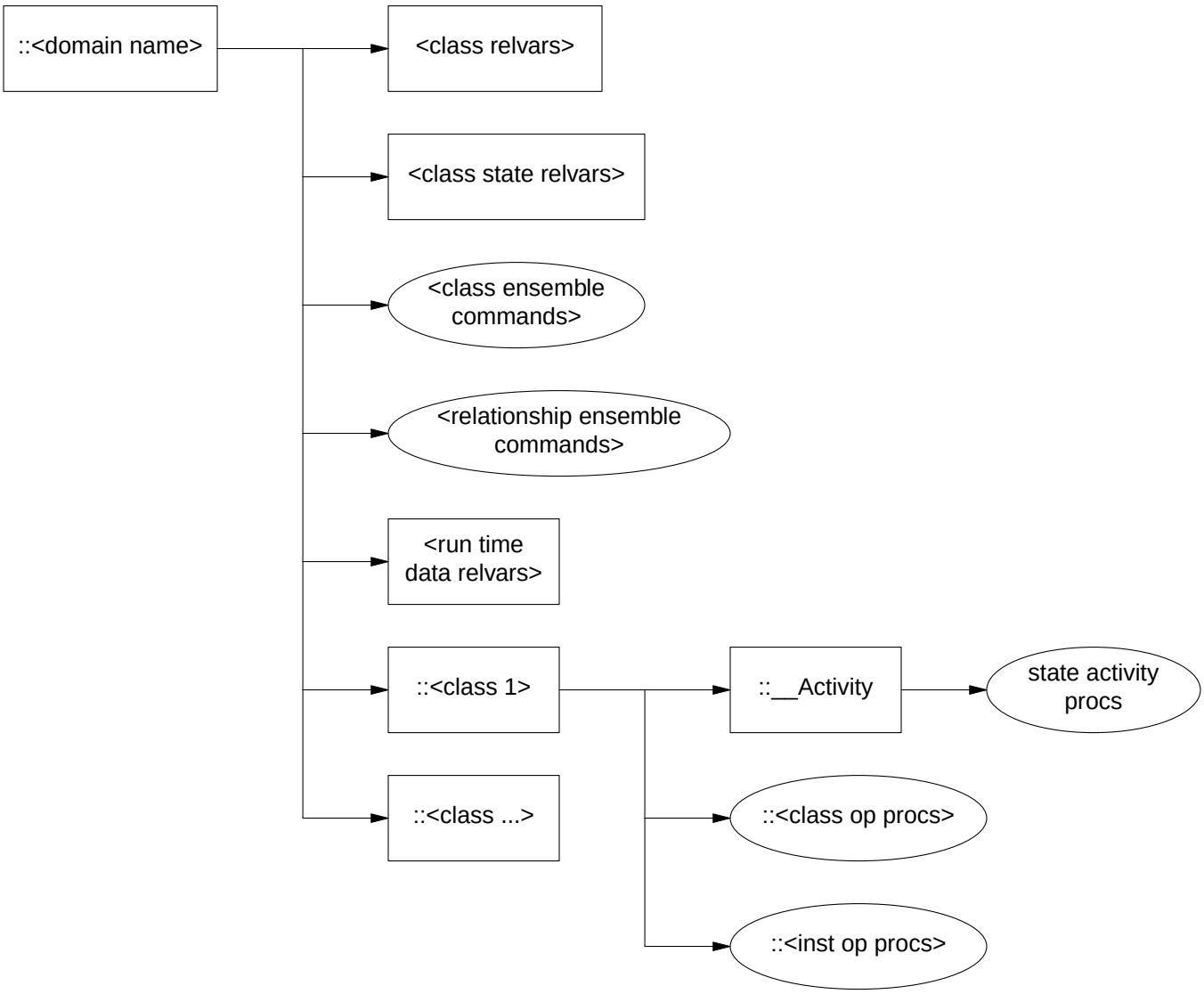
For example, the `wmctrl` domain will be realized as an ensemble command named `::wmctrl`[6] and there will be a `::wmctrl` namespace created. In the `::wmctrl` namespace, a `::wmctrl::WashingMachine` ensemble command is created along with ensemble commands for all the classes in the domain.

The namespace ensemble approach can provide commands that follow the typical Tcl object oriented approach, but we still must deal with how to store instance data. Often, simple object system using namespace ensembles store the instance data in a dictionary held in a namespace variable. In this strategy, the dictionary key is some identifier of the instance and the dictionary value holds the instance data. That instance data can then be yet another dictionary so that instance attribute names are the keys of the nested dictionary.

In rosea, TclRAL is used as the data storage mechanism. TclRAL is a “C” based extension that provides native Tcl data types for relation and tuple values. In addition, TclRAL provides a relation variable concept along with relation variable referential constraints. Fundamentally, executable models are based on relational data model concepts and TclRAL was designed to match the semantics of executable models. Thus much of the underlying ability of rosea to navigate relationships and insure data integrity is derived directly from the use of TclRAL to store class data. This includes the transactional nature of the thread of control discussed above.

The diagram below shows the Tcl components and how they are organized for a domain.

[6] By default domains are placed in the global namespace, but may be placed in any desired namespace. We’ll use the global namespace here for simplicity.



It should be noted and readers may have recognized that there is no concept of inheritance in the object system presented by rosea. There is no substantial notion of inheritance in the relational model of data and the emphasis here is on the deep ties that the relational view has to predicate logic and set theory. Executable class models do have the notion of a generalization relationship, but this actually represents a disjoint union[7] of the class instances rather than an inheritance of data type structure. For some, the fact that the object orientation provided by rosea is not based on type hierarchy will be disconcerting. However, object systems based on other concepts can be useful in the right context, *e.g.* the `snit` object system is based heavily on delegation concepts.

[7] The disjoint union being complementary to the Cartesian product which is the conceptual basis for class associations.

Resources

The rosea package is freely available and licensed under the same license as Tcl itself. The source code is available in a fossil repository located at:

http://repos.modelrealization.com/cgi-bin/fossil/mrtools

This repository is mirrored to:

http://chiselapp.com/user/mangoa01/repository/mrtools

The rosea package was written in the literate programming style. The literate program document contains all of the design, code, test cases and the fragmentary example used here is fully worked out. There is also reference manual documentation available.