Relation Oriented Software Execution Architecture

A Tcl Package for Translating XUML Models

Legal Notices and Information

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	August 3, 2014	Initial coding.	GAM
1.0a1	October 5, 2014	First alpha release.	GAM
1.0a2	October 14, 2014	Second alpha release.	GAM
1.0a3	January 4, 2015	Third alpha release.	GAM
1.0a4	January 11, 2015	Fourth alpha release.	GAM
1.0a6	February 17, 2015	Sixth alpha release.	GAM
1.0b1	March 5, 2015	First beta release.	GAM
1.0b2	March 16, 2015	Second beta release.	GAM
1.0b3	March 25, 2015	Third beta release.	GAM
1.0b4	March 26, 2015	Fourth beta release.	GAM
1.0b5	March 31, 2015	Fifth beta release.	GAM
1.0b6	July 22, 2015	Sixth beta release.	GAM
1.0b7	September 10, 2015	Seventh beta release.	GAM
1.0	October 3, 2015	Release of 1.0. End of beta releases.	GAM
1.1	October 13, 2015	Changed the semantics of creation events to delay instance creation until the event is actually dispatched.	GAM
1.2	October 18, 2015	Corrected the argument interfaces for set operations on instance references. Improved error handling when state activities fail.	GAM

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.3	October 27, 2015	Corrected problem with associative classes that also have a state model. The link and unlink commands were not accounting for the current state instances.	GAM

Contents

List of Figures

List of Tables

Abstract

This document describes a Tcl package named, `rosea`. `Rosea` is an execution architecture intended as the target for translating XUML models using Tcl as the implementation language. The package consists of a run time execution library that implements the XUML semantics and a domain specific configuration language that is used to specify the XUML domain model and generate the required data for the run time execution. `Rosea` uses TclRAL as the basis of its data architecture and that enables referential integrity constraint checking based on the XUML class diagram relationships and supports set-at-a-time operations on domain class instances. Processing is transactional and constraint violations result in rolling back the changes to a known good state.

Chapter 1

Introduction

This document is about a Tcl package named, `rosea`. `Rosea` is a data and execution domain that is intended as the target for translating Executable UML (XUML) models using Tcl as the implementation language.

`Rosea` takes a distinctly different approach to translating models into running programs. Using `rosea` to translate an XUML model is a manual process. The artifacts that capture the model, the class diagram, state model diagrams and action language definitions are transcribed into a domain specific language. This is quite different from the *large tool* approach where the artifacts of the model are directly captured and translated into an implementation with minimal further input. What `rosea` lacks in automation it makes up for in simplicity and the ability to intervene in the details of the implementation that are simply not under your control when the translation is completely automated.

What's Next

Chapter 2 discusses the design of the `rosea` package. Several alternatives are considered and the mapping of XUML model concepts onto Tcl is shown.

Chapter 3 discusses several XUML model concepts and describes how those concepts are realized in `rosea`.

Chapter 4 gives the details of the `rosea` package procedures. These are the Tcl procedures that are invoked from state activities and operations to translate XUML model concepts onto the Tcl implementation.

Chapter 5 explains the domain specific language that is used to define the domain characteristics which provide structure to the translation.

Chapter 6 gives the detailed code and explanation of how the data gathered by the domain configuration language is transformed into the run time data and Tcl constructs that form the execution structure of the domain. This is the process of generating the code for the domain, although in the Tcl environment we don't actually generate source text rather we generate commands that are simply executed in place.

Chapter 7 describes more domain specific language constructs that are used to define the initial instance population for a domain. The initial instance population set the values of objects that are exist when the program starts running.

Chapter 8 discusses a bridging technique directly supported by `rosea`. This technique allows the domain encapsulation to be broken in a controlled manner allowing external actions to be realized as an instance operation.

Chapter 9 discusses how the state of a domain may be serialized to a file.

Chapter 10 describes the very important feature of tracing state machine event dispatch. This is an essential debugging and testing capability that `rosea` supports.

Chapter 11 gives a worked out example of a simple automatic clothes washer. This shows how all the pieces of `rosea` come together to produce a running program.

Chapter 12 shows exactly how the code files are organized and how they can be extracted from the source file.

Finally, there is a bibliography of books and articles that provide some much needed background to understand the overall techniques of XUML and where translation fits into the process.

This document is also a literate program. As such, it contains all the descriptions, source code and test cases for the `rosea` package. An appendix on literate programming describes the conventions used to intersperse code and text that enables the production of documents like this while retaining the ability to produce the source code files needed by the Tcl interpreter.

How To Read This Document

This document contains a lot of material. It contains the complete source code, test cases and a runnable example. Reading the document from beginning to end in order would be an exceptional undertaking. Skipping around is encouraged. For example, you may wish to skip over the test cases although they do provide interesting examples in the small of how `rosea` constructs are used.

For those readers wishing a quick look at what `rosea` provides, the example in Chapter 11 is a good starting point. The example shows a complete, albeit simple, model translated into `rosea`. The resulting translation is a runnable Tcl script and the output of running the example is shown.

For those readers who will attempt a model translation using `rosea`, understanding the domain configuration language in Chapter 5 is an essential starting point. The details of how executable model concepts are realized in a relational algebraic manner are discussed in Chapter 3 and this chapter has important information needed to understand how class instances are referred to. Once the static aspects of the domain, *i.e.* the class definitions and the state model definitions, have been translated, it is then time to understand the package procedures in Chapter 4 as they are used to translate the state activities and operations into Tcl.

The document index also provides a convenient place to look up particular details. When reading an electronic version of the document, the index and the table of contents are hyperlinked for easier navigation.

For those readers interested in an extended example of how relational algebra can be used to query a set of relation variables to produce code structures, Chapter 6 which shows the code generation from data obtained from the configuration DSL to produce the run time data used by the execution mechanisms. The chapter has many examples of advanced relational algebraic expressions.

A Few Words About Tcl

Some readers may find it unusual that Tcl would be chosen as the implementation language for model translation. Tcl is a distinctly different type of programming language. It is command oriented and has very minimal syntax. Tcl is also a relatively old language having been around for at least 20 years. Tcl has many powerful capablities one of which is its ease of extensiblity. That extensibilty enables language extensions such as TclRAL which provides the fundamental relational algebraic operations upon which XUML is based. Relational algebraic operations are not usually available outside of relational database management systems, however, they provide an elegant means of supporting the XUML semantics. Also, Tcl conveniently supports event driven programming which is fundamental to the state machine event dispatch semantics of XUML.

It is also the case that in a translation scheme, the characteristics of the implementation language are simply not as important as when a program is directly coded. When directly coding a program, the implementation language is the primary interface between the programmer and the computer. As such it tends to be very intrusive on the programmer's working habits and individual preferences govern, to a large extent, how convenient it is to code programming constructs. When translating a model, the program logic is contained in the structure of the model. After all, most of the design effort has already been expended in creating the model. Translation does not change the model logic and only brings to bear the appropriate implementation technlogy to obtain a running program. Code segments for the state activities and operations tend to be small and during the translation the goal is to precisely represent the logic already designed in the model as programming language statements. This is a distinctly different mind set since so much of the structural aspects of the program are supplied by the data and execution domain itself.

Tcl is a very misunderstood language. Many programmers simply don't *get* Tcl, especially if their primary experience is with expression-oriented languages that have more elaborate syntax. To those programmers, I would suggest that the primary focus of this document is the concept of using a domain specific language to encode an XUML model and from that generating a running program. The specifics of the implementation language are secondary. In other words, don't get hung up on Tcl as there are deeper concepts in play here.

Chapter 2

Design Strategy

One of main design choices for `rosea` was to base its data storage on TclRAL. TclRAL is a relational algebra library. It is available as a "C" based extension under the package name `ral`.

The choice of TclRAL allows the data architecture to implement referential integrity constraints. Indeed the constraints implemented by TclRAL were designed specifically to support the integrity requirements of XUML. Many XUML execution architectures do not attempt to enforce the referential integrity inherent in the class model relationships in XUML. This is understandable for some platforms, such as embedded micro-controllers, where the cost of enforcement is relatively high and the class of applications fielded on the platforms tend to have static instance and relationships populations. With static populations, the referential integrity can be checked at initialization time and is usually not compromised when the system runs.

However, when running on a more capable compute platform, we want to take advantage of the referential integrity constraints as a convenient, declarative way to better insure the correct operation of the software. By targeting Tcl as the implementation language and using TclRAL as the basis of the data architecture, we achieve an XUML execution architecture that is very robust with respect to insuring that the resulting program runs by transitioning from one correct state to another.

Comparing Approaches

Organizing XUML execution semantics under Tcl can be accomplished in several ways. Tcl has a number of core concepts that can be put to this task.

It is possible to translate directly onto TclRAL and use another package such as the `moore` package to handle state machine event dispatch. Indeed the author has written several such programs. The difficulty arises in relating the class instance data to the state model behavior. Using two packages that are unaware of each other inevitably results in writing some *glue* code to tie the packages together. In particular, some means of association class instances to state machines and *vice versa* is required so that state activities can access class instance data.

Since Tcl version 8.6, the core has acquired object oriented commands that are useful to build object based systems. In TclOO, one can define classes and create objects from the class. This was the approach taken in the `raloo` package. However, the match between TclOO concepts and XUML concepts is not very close. Using TclOO brings in a lot of OO machinery that is simply not used. Having commands represent instances runs up against managing the command lifetimes as instances are created and deleted. Trying to store data in TclRAL relvars contained with TclOO objects is complicated especially when referential constraints are taken into account.

It is possible to build a useful XUML software architecture on top of TclOO. That is the approach taken by the `STSATcl` package. However, this package does not attempt to use TclRAL, stores its instance data in the TclOO objects and does not attempt to enforce referential integrity constraints. It is an XUML architecture that is conceptually similar to `STSA` and its companion `pycca`.

The core Tcl capability that best suits `rosea` is the namespace ensemble. Namespace ensembles allow for creating object-oriented-like commands. It does not support any traditional object oriented concepts of inheritance but then XUML does not use those concepts. The namespace ensemble approach also requires that you manage the instance data directly but then we intend to use TclRAL for that purpose. In the next section we describe how we use the namespace ensemble capabilities of Tcl to support XUML execution semantics.

Namespace Ensemble Design Approach

The `namespace` command allows you to define an ensemble command which has subcommands that are mapped to Tcl command prefixes. Because the mapping is to a command prefix, arbitrary arguments may be added and one use of those arguments is to include an object identifier. In this way the ensemble command can be associated to a particular data element.

Namepaces provide other useful functions for our purposes. In XUML, domains are the unit of encapsulation and mapping domain entities into a namespace provides a convenient way to prevent naming conflicts between the several domains that usually make up an application. Namespaces also have the ability to control the command resolution path and this is a feature we will use extensively. By default, unqualified command names are resolved either in the local namespace or the global namespace. By setting the command resolution path, we can include other namespaces into which we have placed common procedures and be able to invoke those procedures using unqualified names.

The following diagram shows the mapping of XUML concepts to Tcl and TclRAL features.

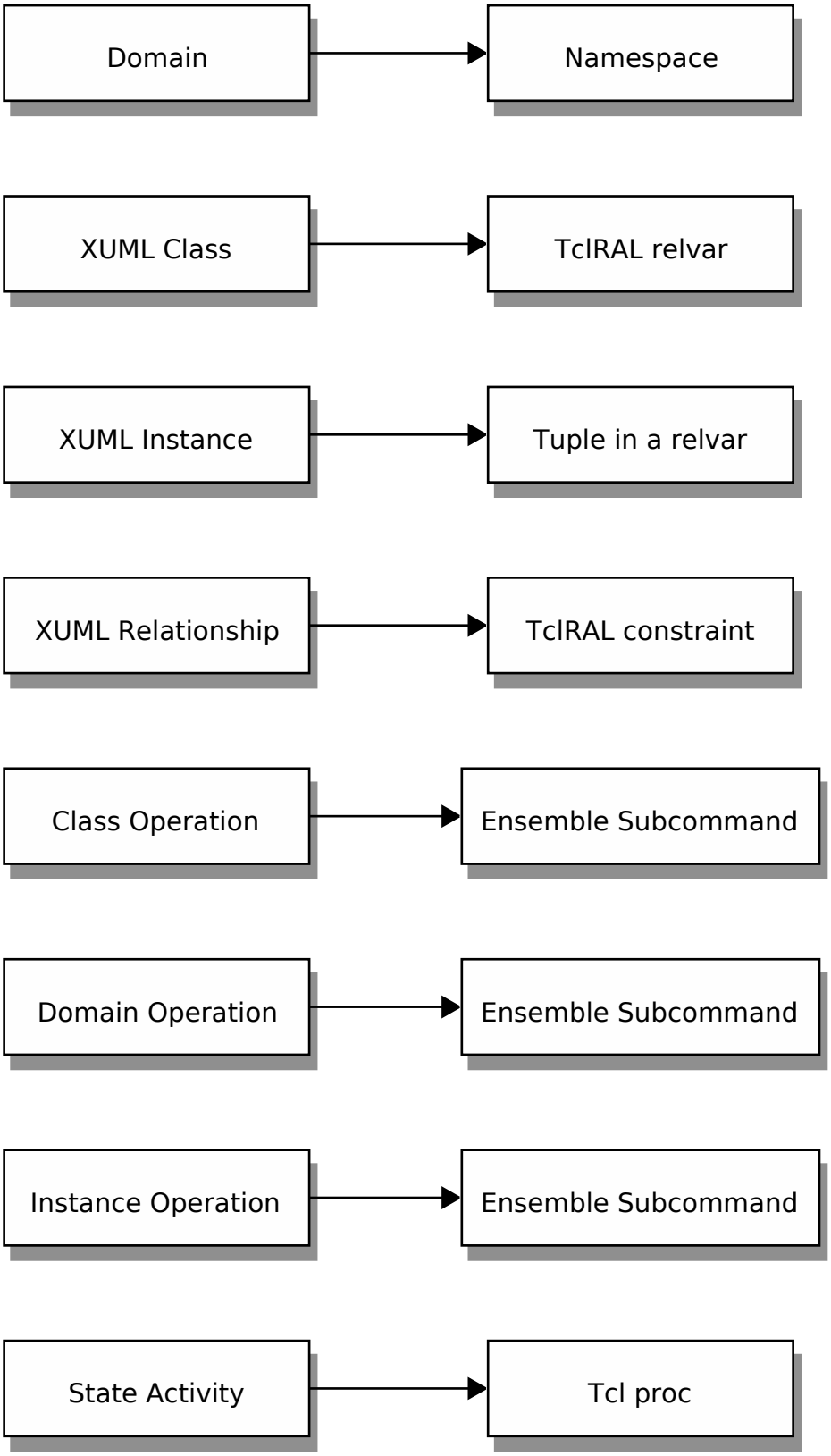

Figure 2.1: Mapping of XUML Concepts to Tcl Entities

Domains will be located in a namespace. Domain operations are constructed as ensemble commands of the domain namespace. Each class is realized as a TclRAL relvar and class instances are then the individual tuples of the relation value held in the class relvar. Relationships between classes are realized as TclRAL relvar constraints. Each class will have an ensemble command by the same name and any class operations as ensemble subcommands. State model activities are converted into ordinary Tcl procedures. This has the distinct advantage of supporting Tcl execution traces and tracing can be used for a number of aspect oriented uses when bridging together domains into a complete application.

Domain Organization

Each domain of an application will be placed in its own namespace. The diagram below show how the components of the domain are organized in that namespace.

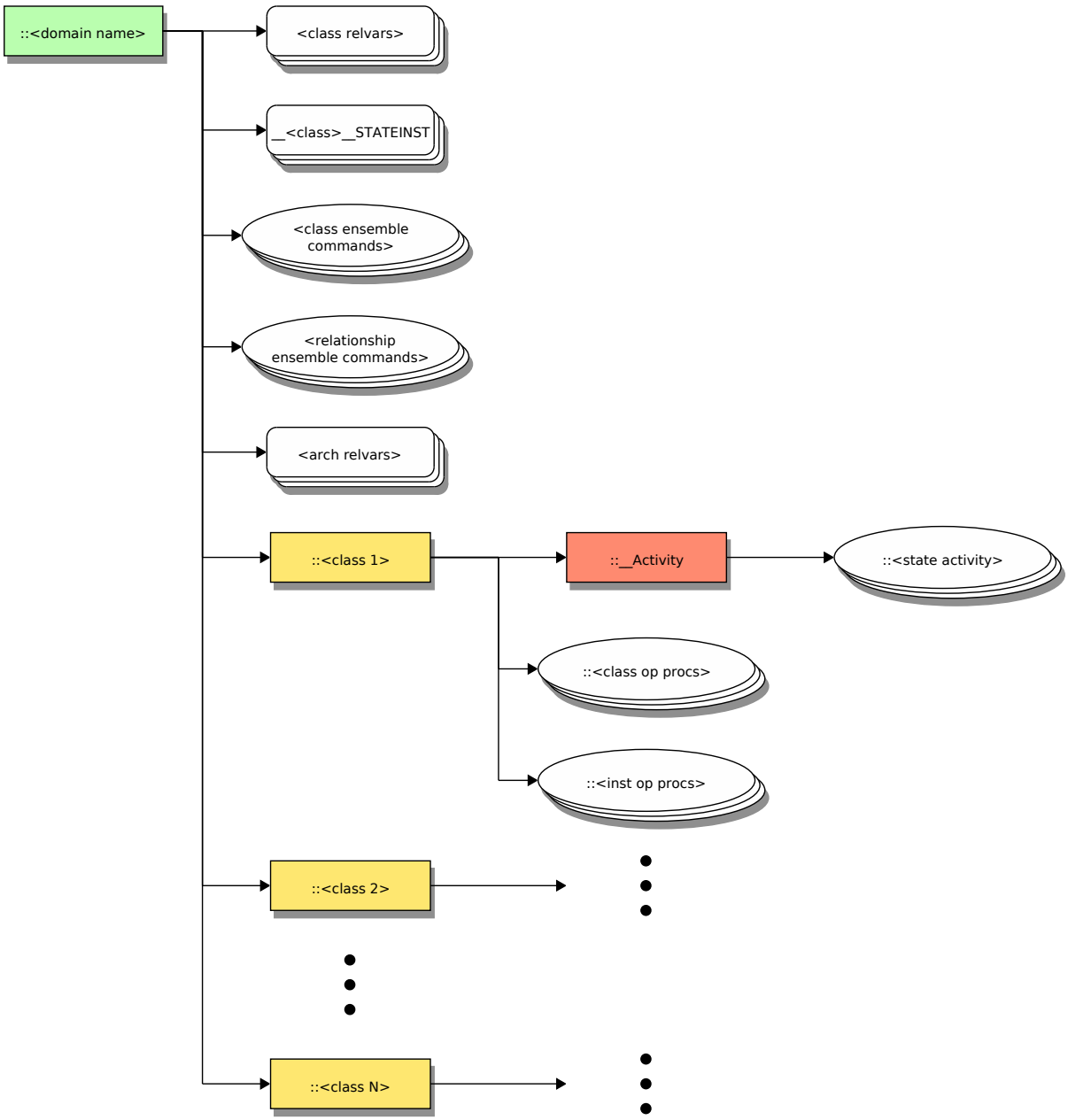

Figure 2.2: Namespace Organization for a Domain

The domain is placed in a namespace that has the same name as the domain. Although the diagram shows the domain placed as a child of the global namespace, in fact it may be located in an arbitrary namespace.

Class Relvars

The TclRAL relvars are placed directly into the domain namespace. Relvars form a shadow variable system to ordinary Tcl variables and use the same name qualification scheme as Tcl variables. Classes with state models also have an additional relvar to hold the current state of the instances and this is also contained in the domain namespace.

Class Ensemble Commands

The ensemble commands for each class also reside in the domain namespace. The class ensemble commands behave as object oriented commands with subcommands to represent the operations on the class. The ensemble subcommands are mapped either to `rosea` package procedures or to procedures in a child namespace that is the same name as the class.

Relationship Ensemble Commands

Each relationship in the domain is also given an ensemble command that resides in the domain namespace. The subcommands of the relationship ensemble provide the operations supported by the relationship and these operations will depend upon the specifics of the relationships.

Class Namespaces

In addition to the class ensemble command, a child namespace of the domain namespace is created to hold the class specific information. This namespace will hold class and instance based operations particular to the class and these serve as the procedures where class ensemble subcommands are mapped.

Class Activity Procedures

For classes that have a state model, the state activities are converted into Tcl procedures and placed in the `__Activity` namespace as a child of the class namespace.

Architecture Relvars

Associated with each domain is a set of data used by the `rosea` run time procedures to drive the model level operations. We will have much more to say about both the data and the procedures. Here we show two things. First, the architecture relvars have names that are prefixed with `__Arch_`. The diagram below shows the names of the relvars that make up this architectural data. The `rosea` run time is strictly data driven and the same code provides the operations for all domains. The data values in the architectural relvars provide the information required by the run time to tailor the operations to the domain specifics.

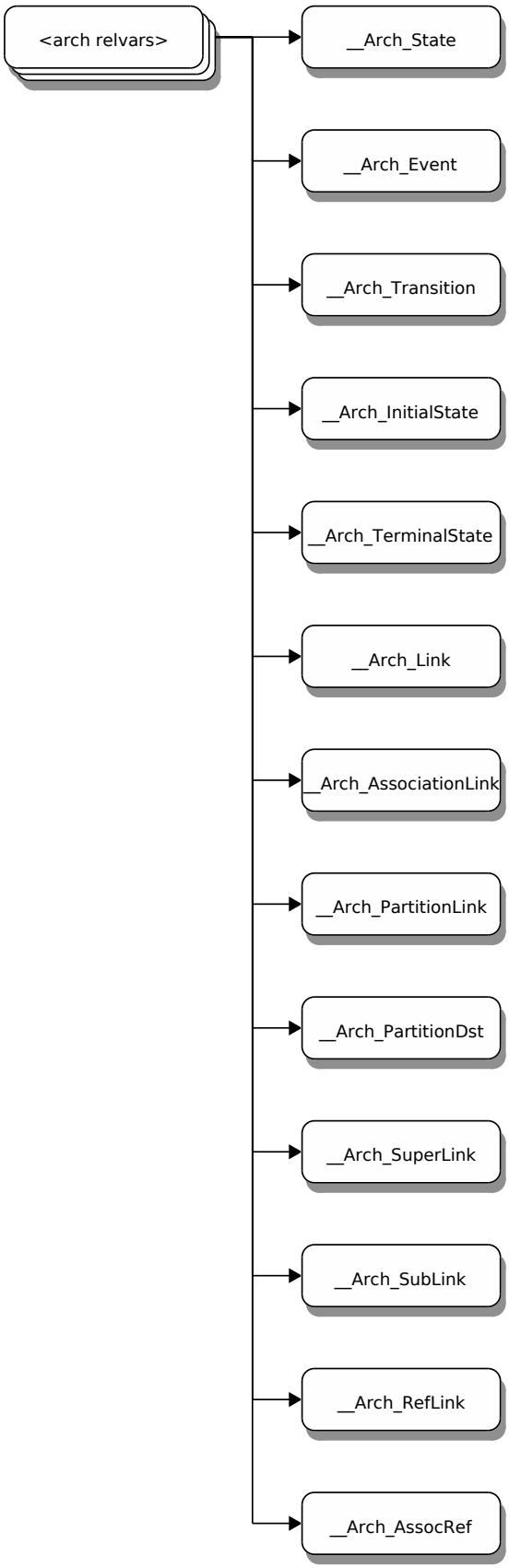

Figure 2.3: Relvars Containing Architecture Information

Package Organization

Up to now we have been discussing the layout of a domain into its namespace. This layout essentially serves as a template for how the elements of a domain are mapped into a Tcl namespace and onto other Tcl language concepts.

In this section, we discuss the layout of the `rosea` package itself. There are three major pieces of the package.

1. Procedures to handle common XUML semantics. These procedures form the interface invoked by domain activities to implement XUML semantics.
2. A configuration language to describe the facets of a domain such as its classes, relationships, state models and processing.
3. A code generator to compute the architectural data structures and layout the domain as we described above.

The package is placed in the `::rosea` namespace and the diagram below show the child namespaces where the components of the package are placed.

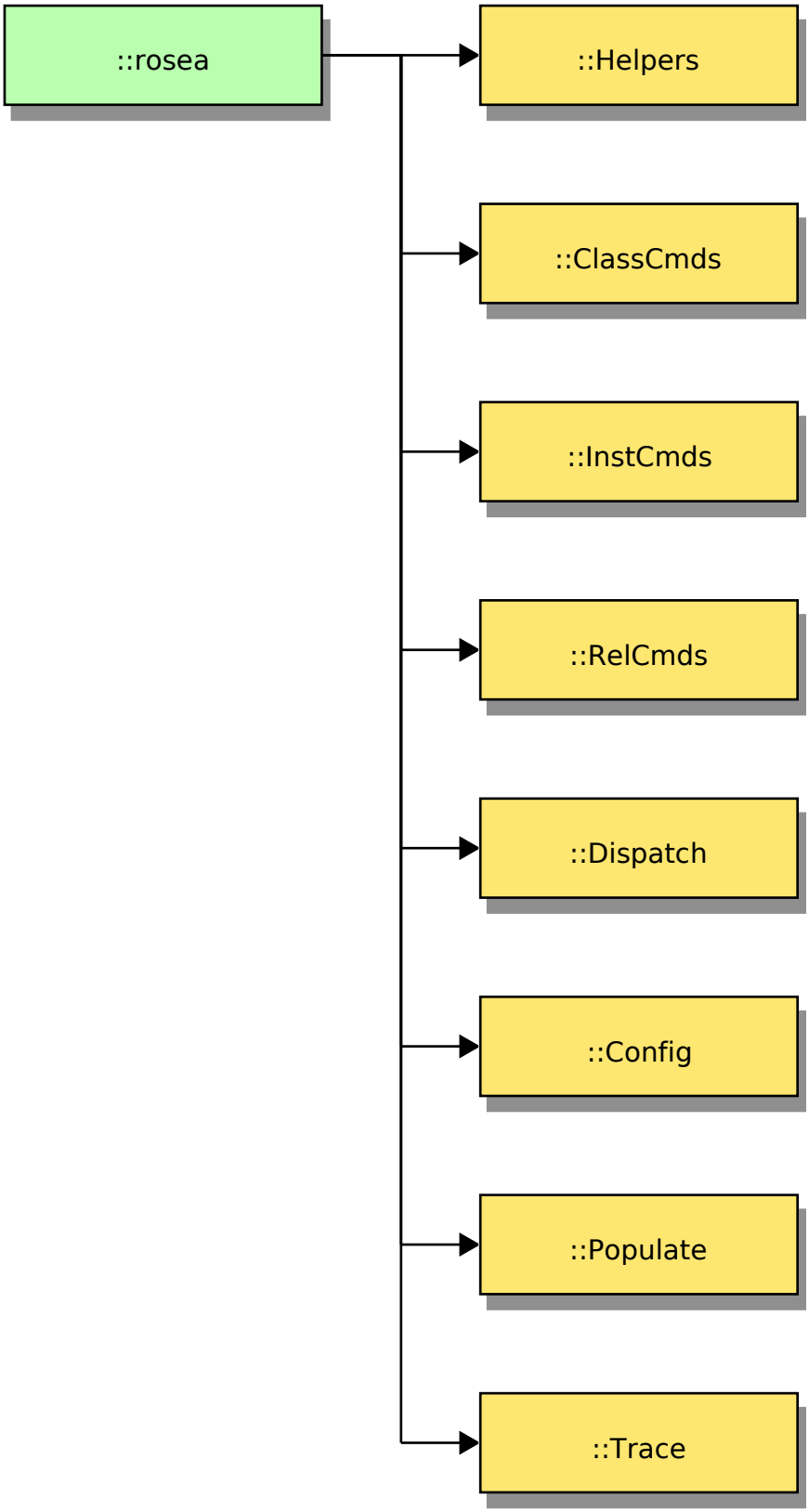

Figure 2.4: Namespace Organization for a rosea Package

We will cover in detail below the commands that reside in each child namespace. Here we summarize the contents of the child namespaces.

Helpers
: Commonly used procedures.

ClassCmds
: Common operations for all classes.

InstCmds
: Common operations for all class instances.

RelCmds
: Operations for relationships.

Dispatch
: Procedures to dispatch state machine events.

Config
: Procedures that implement a domain specific language for specifying the details of the domain.

Populate
: Procedures that implement a domain specific language for specifying initial instance values.

Trace
: Procedures to control, query and format state machine event dispatch traces.

Chapter 3

Realizing Model Concepts in `rosea`

In this section we discuss some XUML concepts and the details of how these concepts are realized in `rosea`. Because the underlying data architecture is based on relational algebra concepts, there are a few issues that need clarification in order to understand the mapping of XUML model concepts into the translation implementation.

Relation Values, Relvars and Instance References — Oh My!

Although the ideas of variables and values are well known, it is worth clarifying the differences between variables and values and how TclRAL constructs are involved. Some of the confusion in this area arises from the conventional use of symbol names in programming languages to refer both to the *storage location* of a variable and the *value* contained therein.

For example, consider the following "C" code.

```
1 int counter ;
2
3 counter = 27 ;
4
5 counter = counter + 3 ;
```

As line 1 shows, `counter` is clearly a variable of integer type. It is assigned the *value* 27 on line 3. Line 5 assigns the sum of the *value* held by the `counter` variable and the *value* `3` back into the memory of the `counter` *variable*. In "C" and many other languages, the interpretation of the symbol `counter` depends upon whether it stands for a value in an expression or as the target of an assignment. In "C" parlance, the meaning is determined by whether `counter` is used as an *lvalue* or an *rvalue*. Normally, we do not speak in such ultra-precise terms since context shows whether we are interested in the *value* contained by the *variable* or whether we are talking about the memory allocated to the *variable*.

We can speak of relation values by analogy to *scalar* values. A *relation value* is defined as a *heading* and a *body*. The heading consists of a set of named attributes and their corresponding data types. The body is a set of named values that correspond to the heading. The tuples that are the body of the relation value form a set, *i.e.* there are no duplicated tuples. In TclRAL, a relation value has a string representation (like all well behaved Tcl values) that is a two element list representing the heading and the body. Relation values in TclRAL are first class Tcl values integrated into the internal Tcl type system and consequently may be assigned to an ordinary Tcl variable and undergo the normal life cycle of Tcl values. This latter point is important as we shall see later.

It is common to display relation values as tables. This is a convenient representation that helps show the two dimensional nature of a relation values as a matrix of columns for the attributes and rows for the tuples. However, it is necessary always to remember that relation values have no inherent order of columns or rows. The implementation will choose to store things in some order but that order is not specified to the external operations of the algebra. Access to the contents of a relation value is always determined by the *values* of attributes. There are no operations that act as an index or pointer and the implementation is allowed to change the order of storing attributes and rows in whatever way it finds convenient. The only guarantees are those provided by the relational algebraic operators.

A *relation variable* (or relvar) is a special type of variable that holds a *relation value*. The concept of a relvar is supplied by TclRAL and a relvar is *not* an ordinary Tcl variable. TclRAL maintains a *shadow* variable system for relvars. The naming conventions follow those of ordinary Tcl variables in that they can be qualified using namespace syntax. The reason that relation variables are distinct from Tcl variables is that they are also subject to *constraints* for identity and referential integrity. This is one of the major distinguishing aspects of relvars. TclRAL has the subcommand, `relvar`, which operates directly on relvars and a distinct subcommand, `relation`, that operates on relation values. The command distinction helps to maintain the value / variable distinction.

We must also consider how we will deal with class instances. We make the correspondence between an XUML class and a relvar and between an XUML class instance and a tuple in the relation value stored in the class relvar. Consider that in XUML, events are signaled to instances of classes. So, we will find it necessary to refer to particular instances when signaling events. We will also find it necessary to refer to sets of instances when, for example, we search for instances that meet some criteria.

With these considerations, we define an *instance reference* concept. In the relational view, the only way that you can refer to tuples in the relation values contained in relvars is by the *values* of identifying attributes. There is no notion of indexing and order within the relation values is not specified. Each relvar must have at least one identifier. An identifier is a set of attributes (often a single attribute but not always), the values of which must be distinct from those of the other tuples in the relvar. To refer to an instance we need to know the relvar in question and the *values* of the attributes that constitute an identifier. We will represent an instance reference as a two element list:

- A fully qualified relvar name.
- A relation value whose heading consists of the set attributes that correspond to an identifier of the relvar.

This representation has several useful characteristics.

- An empty relation value indicates the *nil* reference.
- The cardinality of the relation value can be greater than one and as such represents a reference to a set of instances.
- An instance reference is just an ordinary Tcl value that can be stored in an ordinary Tcl variable whose scope and lifetime is properly managed by Tcl itself.

So we do not need to distinguish between an instance reference and an instance reference set as distinct entities (with potentially different data types) as is often done in architectures that target statically typed languages. Our representation of an instance reference can refer to zero, one or as many instances as needed.

Having defined an instance reference, we will need several operations on them.

Creating References

Internally, `rosea` commands have need to create instance references. The `ToRef` command takes a fully qualified relvar name and a relation value that is a subset of the value held in the relvar and returns an instance reference to the values. This command is typically not used by application code.

Implementation

```
<<helper commands>>=
proc ToRef {relvar relvalue} {
    tailcall list $relvar [relation project $relvalue\
        {*}[lindex [relvar identifiers $relvar] 0]] ; # ❶
}
```

❶ Let's step through this one command at a time, starting at the inner nesting.

- `relvar identifiers` returns a list of the identifiers of the relvar. Since each relvar has at least one identifier, we can always index off the first identifier.

- `relation project` returns a new relation value that contains only the attributes requested. In this case we are asking for those attributes that form one of the identifiers.
- We compose the two element list that defines an instance reference and we can use `tailcall` since this is the last command in the procedure.

Tests

```
<<helper command tests>>=
test ToRef-1.0 {
    Compute a reference to a relvar tuple
} -setup {
    relvar create A {
        Attr1   int
        Attr2   string
    } Attr1
} -cleanup {
    relvar unset A
} -body {
    set value [relation create {Attr1 int Attr2 string} {
        Attr1 20 Attr2 foo
    } {
        Attr1 40 Attr2 bar
    }]
    ::rosea::Helpers::ToRef [namespace current]::A $value
} -result {::rosea::test::A {{Attr1 int} {{Attr1 20} {Attr1 40}}}} -match ref
```

Dereferencing an Instance

We will also need to obtain the tuples referenced by an instance reference. In other words we will need to dereference the instance.

`deRef` *instref ?attr1 attr2 ... ?*

The `deRef` command takes an instance reference and returns a relation value that contains all the tuples from the associated relvar that match the identifying attribute values contained in the reference. Optionally, the `deRef` command takes a set of attribute names. In this case the returned instance will be a *projection* of the full instance relation, containing only the attributes that form an identifier and the attributes given as arguments.

Implementation

```
<<instance commands>>=
proc deRef {instref args} {
    lassign $instref relvar ref
    set inst [relation semijoin $ref [relvar set $relvar]] ; # ❶
    if {[llength $args] != 0} {
        set inst [relation project $inst\
            {*}[relation attributes $ref] {*}$args] ; # ❷
    }
    return $inst
}
```

❶ Dereferencing the instance is just a simple matter of `semijoin`ing the relation value in the instance reference to the value contained in the relvar. Like the core `::set` command, invoking the `relvar set` command without any relation value simply returns the current value held in the *relvar*. The `relation semijoin` command returns a relation value whose heading matches that of its final argument which in our case is that of the referenced relvar value.

❷ If we are requesting specific attributes, then we project those out. Note also that we need to project out the attributes of the `instref` so that we will have an identifier of the relvar in the resulting relation value. This allows us to use the result as an argument to the class `update` command.

Tests

```
<<helper command tests>>=
test deRef-1.0 {
    Dereference to get a relation value
} -setup {
    relvar create A {
        Attr1   int
        Attr2   string
    } Attr1
    relvar insert A {
        Attr1 20 Attr2 foo
    } {
        Attr1 40 Attr2 bar
    }
} -cleanup {
    relvar unset A
} -body {
    set ref [list [namespace current]::A {{Attr1 int} {{Attr1 40}}}]
    rosea::InstCmds::deRef $ref
} -result {{Attr1 int Attr2 string} {{Attr1 40 Attr2 bar}}} -match relation
```

```
<<helper command tests>>=
test deRef-1.1 {
    Dereference to get a projection of the instance
} -setup {
    relvar create A {
        Attr1   int
        Attr2   string
        Attr3   string
    } Attr1
    relvar insert A {
        Attr1 20 Attr2 foo Attr3 buzz
    } {
        Attr1 40 Attr2 bar Attr3 zub
    }
} -cleanup {
    relvar unset A
} -body {
    set ref [list [namespace current]::A {{Attr1 int} {{Attr1 40}}}]
    rosea::InstCmds::deRef $ref Attr3
} -result {{Attr1 int Attr3 string} {{Attr1 40 Attr3 zub}}} -match relation
```

Nil Instance Reference

The nil instance reference is an instance reference that does not refer to anything. The nil reference value has an empty string as the relvar name and DUM[1] as its referring relation value.

`nilInstRef`
The `nilInstRef` command returns an instance reference that does not refer to any instance.

[1] A relation value with an empty heading and an empty list of tuples. The relation value with an empty heading and a single empty tuple is sometimes called DEE.

Implementation

```
<<helper commands>>=
proc nilInstRef {} {
    return {{} {{} {}}}
}
```

Counting References

Since commands that search for instances return instance references we will need some commands that can determine if we are indeed referring to anything.

`isEmptyRef` *instref*

The `isEmptyRef` command returns a boolean indicating if `instref` does not refer to any instances.

Implementation

```
<<instance commands>>=
proc isEmptyRef {instref} {
    tailcall relation isempty [lindex $instref 1]
}
```

Note that in relational algebra the empty relation, *i.e.* what we have called here the *empty reference* is perfectly valid. It is well defined that dereferencing a empty reference simply returns a relation value that contains no tuples. It should *not* be mistaken for the concept of the *nil* or *NULL* pointer available in many languages where it is usually implemented as a special address and dereferencing the `nil` pointer is not allowed. Nor should the empty reference be confused with the idea of a `NULL` attribute value as implemented by Relational Database Management Systems that support SQL as a query language.[2]

`isNotEmptyRef` *instref*

The `isNotEmptyRef` command returns a boolean indicating if `instref` refers to any instances.

Implementation

```
<<instance commands>>=
proc isNotEmptyRef {instref} {
    tailcall relation isnotempty [lindex $instref 1]
}
```

`refMultiplicity` *instref*

The `refMultiplicity` command returns the number of instances referred to by `instref`.

[2] Despite the similarity of the relational concepts between TclRAL and many RDMS, we have no concept of a `NULL` value nor do we find any need for one. The kindest thing that can be said about SQL `NULL` values is that is a serious misfeature, at worst it is a conceptual abomination.

Implementation

```
<<instance commands>>=
proc refMultiplicity {instref} {
    tailcall relation cardinality [lindex $instref 1]
}
```

`isRefSingular` *instref*

The `isRefSingular` command returns true if `instref` refers to only one instance.

Implementation

```
<<instance commands>>=
proc isRefSingular {instref} {
    expr {[refMultiplicity $instref] == 1}
}
```

Instance Reference Equality

The concept of reference equality is well defined. It is the same idea as set equality.

`isRefEqual` *instref1 instref2*

The `isRefEqual` command returns true if `instref1` and `instref2` both refer to the same set of tuples.

Implementation

```
<<instance commands>>=
proc isRefEqual {instref1 instref2} {
    lassign $instref1 relvar1 inst1
    lassign $instref2 relvar2 inst2
    return [expr {$relvar1 eq $relvar2 && [relation is $inst1 equal $inst2]}]
}
```

Instance Reference Set Operations

It is also useful to be able to perform set like operations on instance references. We can factor out some common code as shown below.

Implementation

```
<<instance commands>>=
proc RefSetCommand {op instref1 instref2} {
    lassign $instref1 relvar1 inst1
    lassign $instref2 relvar2 inst2
    if {$relvar1 ne $relvar2} {
        tailcall DeclError SAME_CLASS $relvar1 $relvar2
    }
    return [list $relvar1 [relation $op $inst1 $inst2]]
}
```

```
<<error code formats>>=
SAME_CLASS   {operation is only allowed for references to the same class,\
                got "%s" and "%s"}
```

We can now define operations on instance references by simply supplying the operation subcommand. The command below support the most common set operations on instance references.

Implementation

```
<<instance commands>>=
proc refUnion {instref1 instref2} {
    tailcall RefSetCommand union $instref1 $instref2
}
```

Tests

```
<<helper command tests>>=
test refUnion-1.0 {
    Compute the union of two instance references
} -setup {
    relvar create A {
        A_ID   int
        A_Attr   string
    } A_ID
    set A_value [relation create {A_ID int} {A_ID 20}]
    set aref [::rosea::Helpers::ToRef [namespace current]::A $A_value]
    set B_value [relation create {A_ID int} {A_ID 30}]
    set bref [::rosea::Helpers::ToRef [namespace current]::A $B_value]
} -cleanup {
    relvar unset A
} -body {
    ::rosea::InstCmds::refUnion $aref $bref
} -result {::rosea::test::A {{A_ID int} {{A_ID 20} {A_ID 30}}}} -match ref
```

Implementation

```
<<instance commands>>=
proc refIntersect {instref1 instref2} {
    tailcall RefSetCommand intersect $instref1 $instref2
}
```

Tests

```
<<helper command tests>>=
test refIntersect-1.0 {
    Compute the intersection of two instance references
} -setup {
    relvar create A {
        A_ID   int
        A_Attr   string
    } A_ID
    set A_value [relation create {A_ID int} {A_ID 20} {A_ID 30}]
    set aref [::rosea::Helpers::ToRef [namespace current]::A $A_value]
```

```
    set B_value [relation create {A_ID int} {A_ID 30} {A_ID 40}]
    set bref [::rosea::Helpers::ToRef [namespace current]::A $B_value]
} -cleanup {
    relvar unset A
} -body {
    ::rosea::InstCmds::refIntersect $aref $bref
} -result {::rosea::test::A {{A_ID int} {{A_ID 30}}}} -match ref
```

Implementation

```
<<instance commands>>=
proc refMinus {instref1 instref2} {
    tailcall RefSetCommand minus $instref1 $instref2
}
```

Tests

```
<<helper command tests>>=
test refMinus-1.0 {
    Compute the difference of two instance references
} -setup {
    relvar create A {
        A_ID   int
        A_Attr   string
    } A_ID
    set A_value [relation create {A_ID int} {A_ID 20} {A_ID 30}]
    set aref [::rosea::Helpers::ToRef [namespace current]::A $A_value]
    set B_value [relation create {A_ID int} {A_ID 30} {A_ID 40}]
    set bref [::rosea::Helpers::ToRef [namespace current]::A $B_value]
} -cleanup {
    relvar unset A
} -body {
    ::rosea::InstCmds::refMinus $aref $bref
} -result {::rosea::test::A {{A_ID int} {{A_ID 20}}}} -match ref
```

Iterating on References

We will find it convenient to iterate on the instances referred to by an instance reference. Recall, that an instance reference can refer to many tuples and there are times when we want to visit each instance and run some code. As is typical in Tcl and following the pattern of the `foreach` command, we implement a control structure procedure to perform the iteration.

`forAllRefs` *varname instref body*
The `forAllRefs` command assigns an instance reference from a set to a variable and executes a script for each instance. Each iteration insures that the instance reference assigned to the variable references only a single tuple.

Implementation

```
<<instance commands>>=
proc forAllRefs {varname instref body} {
    lassign $instref relvar refs
    upvar 1 $varname inst
    relation foreach ref $refs {
```

```
        set inst [list $relvar $ref]
        uplevel 1 $body
    } ; # ❶
    return
}
```

❶ The key command here is `relation foreach` which performs an iteration across a relation value, one tuple at a time. We need only construct a new instance reference for each singular relation value that `relation foreach` generates.

Forming and Breaking Relationships

In XUML, the real world associations between classes are modeled as relationships. Classes have their foundation in predicate logic and relationships have their foundation in the referential integrity concepts of relational algebra. In this section, we discuss how relationships are formed and broken and how XUML relationships correspond to TclRAL relvar constraints.

When translating an XUML class model onto a data architecture that is based on relational algebra, we must consider the two fundamental ways classes may be associated with each other.

- Defining a correlation between instances.
- Dividing a set of instances into a disjoint union.

Correlations

XUML models associate the instances of classes together to model the semantics of subject matter rules. From a relational algebra point of view, class instances from two classes may be correlated by representing each class as a relvar and building a third class to contain attributes that refer to the identifiers of the participating classes. We will call this a *class based association* and we will call the class that performs the correlation by holding referential attributes an *associator*.

We need to be clear about the referential use of attributes. We will often refer to attributes as *referential*. By that we mean that attributes in one class have the same *value* as attributes in another class. In relational algebraic terms, we can only refer to particular tuples in a relation by specifying the *values* of the attributes. To realize a relationship using referential attributes, we further insist that the referential attributes have the same values as identifying attributes of the referenced class. Since a class may be identified by more that one attribute and may have more than one identifier, we must be clear about which referential attribute refers to which identifying attribute. In the common and simple case, there is only a single identifier for a class and that identifier consists of a single attribute. Consequently, to create a reference to an instance of such a class requires only a single referential attribute. Although referential attributes may seem like a complicated formalism, in practice is usually amounts to specifying the values involved in a binary relation. We will use the terms referring and referenced to distinguish classes that contain referential attributes and referenced identifiers.

Each instance of the associator class represents an instance of the correlation that is formed between the instances of the participating classes. Forming an instance of a relationship between class instances amounts to creating instances of the associator class with the referential attributes set to the proper values of the identifiers of the participating classes. Correspondingly, destroying the correlation can be accomplished by deleting instances of the associator class.

It is important to say at this point, that whether or not adding or deleting instances of the associator class is a valid operation depends upon the *multiplicity* and *conditionality* of the relationship. It may be the case that adding an instance of an associator will be rejected on the basis that the multiplicity of the relationship (and therefore the relational algebraic referential constraint) does not allow instances of a particular class to be referenced multiple times or demands that an instance be referenced at least once.

Analyzing the properties of the associator classes will show that under the circumstances of one-to-one and one-to-many associations, we can simplify our schema by placing the required referential attributes into one of the participating classes and thereby eliminate the need for a separate associator class. We will call such arrangements *simple associations*. They are quite common and usually form the preponderance of relationships in any XUML model. In the case of many-to-many associations or when the associator class itself has attributes other than the required referential ones, then simplifying the schema by eliminating the associator and moving the referential attributes into one of the participant classes is not available.

For simple associations, instances of the relationship are manipulated by updating the values of the referential attributes that are part of the referring class. Again, whether such updates are valid will depend upon the multiplicity and conditionality specified for the relationship.

Under one particular circumstance we have a small problem in attempting to update a referential attribute. Consider the case of a one-to-many relationship that is conditional on the many side. If we implement that by placing referential attributes in the class on the many side, then we must decide what value to use when updating the referential attributes for the case when we want to delete the association. Traditionally, this is solved in data base management systems using a **NULL** value. We specifically eschew NULL valued logic as an abomination. We must also account for the fact that Tcl has no null value and, consequently, TclRAL has no concept of a null value. In this situation, a null would represent a value that is not a valid value for the data type of the referential attributes. In some languages you may define *algebraic data types* that can implement this concept. That is not available to us in Tcl but, by special dispensation, TclRAL does allow the empty string as a valid value of any data type. Now as it turns out, the empty string is a perfectly valid value and under many circumstances cannot be used as some extraordinary value to indicate the absence of an instance of a correlation. As tempting as it may be, the empty string really isn't the same as a null value or even the same as a pointer value of all zero bits (*i.e.* as is defined in many programming languages).

In this particular circumstance we must make some decision. The options are:

- Translate the XUML relationship to use a class-based association to implement it. In this case the instances of the associator correspond to the instances of the correlation and there is no quandary.
- Use the empty string as a *special* value to indicate the lack of a correlation. This often works because the empty string can be used as a value outside of the range of the data type of the referential attribute.
- Find some other extraordinary value that can represent a missing correlation (*e.g.* -999 for some positive valued integer type).

In practice, the empty string works very well and allows us to keep the simpler formulation of the correlation, but using an associator class *always* works.

Operations on Correlations

XUML action languages support the operations of *link* and *unlink* as a means of creating and deleting associations between class instances. For many XUML translation architectures, particularly those targeting a statically typed programming language, relationships are represented using pointer values. In those cases, the *link* and *unlink* operations are translated into updating pointer values. For example, `STSA` uses this technique. When using relational algebra to implement a correlation type relationship, we need only create or delete instances of the associator or update values of referential attributes to control the association of class instances.

There are a few inferences we can draw from this.

- For class-based associations it is always necessary to unlink instances before re-linking them to other instances. This arises because the instance of the associator class has a direct correspondence to an instance of the relationship.
- For simple associations it is never necessary to unlink an instance if it is to be relinked immediately. This arises because we are only going to update a referential attribute value. It is not incorrect to unlink the instance, it is just superfluous in this particular case.

In `rosea`, we provide `link` and `unlink` commands for associative relationships as part of the namespace ensemble created for the relationship.

Partitions

A partition is the other kind of relationship between classes and is the complement to correlation. This is signified in UML as a generalization relationship. In the XUML usage of a generalization relationship, the instances of the superclass are partitioned into a disjoint union of the subclasses. This partitioning insures that each subclass instance refers to exactly one superclass instance and each superclass instance is referenced by exactly one subclass instance *from among all the subclasses* of the generalization.

A close examination of partitioning shows that there is no real need for operations to link and unlink instances in a generalization relationship. Since the characteristics of a partition insure that there is a one-to-one correspondence between class instances, the partition is created when the superclass and subclass are created with the subclass having the proper values for its referential attributes. Equally, an instance of the partition is deleted when the corresponding superclass and subclass instances are deleted. The only operation that is useful for a generalization relationship is that of *migrating* the generalization from one subclass to a different one and that is the only operation provided by `rosea` for generalization relationships.

Navigating the Model

In the relational model of data, one can specify a set of constraints and have those constraints enforced on the relvar values. For TclRAL, the form of these constraints was specifically designed to match those needed to support XUML model semantics. From a relational algebra point of view, the constraints declare the multiplicity and conditionality of the association between tuples in relvars.

From an XUML point of view, model relationships enforce both referential integrity and provide a way to navigate around the class data. It is common for state activities to read and update data in the model and to use the relationship declarations of the model to find related instances. It is in this way that the semantics of the real-world association represented by the relationship are realized by the code execution. Ultimately for an architecture based on relational algebra, navigating the XUML class model via relationships must be translated into a sequence of `semijoin` operations. In this section we discuss the concepts and syntax for accomplishing that.

Conventionally, XUML action languages specify the relationship navigation by giving the relationship name and the destination class name, *e.g.*

```
self -> R23[MOTOR]
```

would be a statement to find the instances across that are related to those instances of `self` via the `R23` relationship that targets the `MOTOR` class. This technique has an ambiguity for reflexive relationships[3] as it gives no way to determine the direction of the traversal since the starting and ending of the traversal is the same class. This usually has been resolved by using the relationship annotation from the class model.

However, we specifically avoid using the model annotation to disambiguate the reflexive relationship case. We consider the phrases associated with the relationship in a model as vital semantic annotation but that they should have no bearing on how the model executes. Annotation is just that — annotation. Changing model annotation should not invalidate action language statements. The approach we adopt in `rosea` is to give each relationship an explicit direction and to use the destination class only in those cases where the relationship type allows for multiple paths and consequently multiple destination classes. We will specify navigation in the forward direction using the relationship name, *e.g.* `R1`, and in the reverse direction using the relationship name prefixed by a tilde character, *e.g.* `~R1`. Note that the use of the tilde character introduces a syntax convention for the relationship traversal direction.

We will consider each type of relationship and show how the use of a direction and sometimes a destination class allows for the navigation of the class model. Note also that this choice of specifying the relationship navigation has no bearing on the XUML view of relationships in general. We are not implying that the concept of a direction is a model level concept. It is a particular technique that will be used as part of the syntax for specifying the navigation operations for the `rosea` architecture.

Simple Associations

The figure below shows the case of a simple associative relationship. We choose the direction of the relationship to be the same as the direction of reference of the referential attributes. We deem the source of the navigation to be the class that contains the referential attributes and the destination of the navigation to be the referenced class. For the case of singular relationships, the choice of where to place the referential attributes is sometimes arbitrary, but wherever the analyst has placed the referential attribute determines the direction of the relationship.

[3] A reflexive relationship is one formed between a class and itself

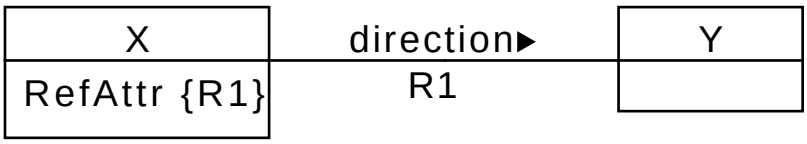

Figure 3.1: Simple Associative Relationships

So, starting with an instance of `X` we can find the related `Y` instance or instances by navigating across `R1`. The reverse navigation from `Y` to `X` is across `~R1`.

Generalization

The figure below shows the case of a generalization relationship. In a generalization, the subclasses always contain the referential attributes and therefore always refer to the superclass and the direction of a generalization is always from the subclasses to the superclass.

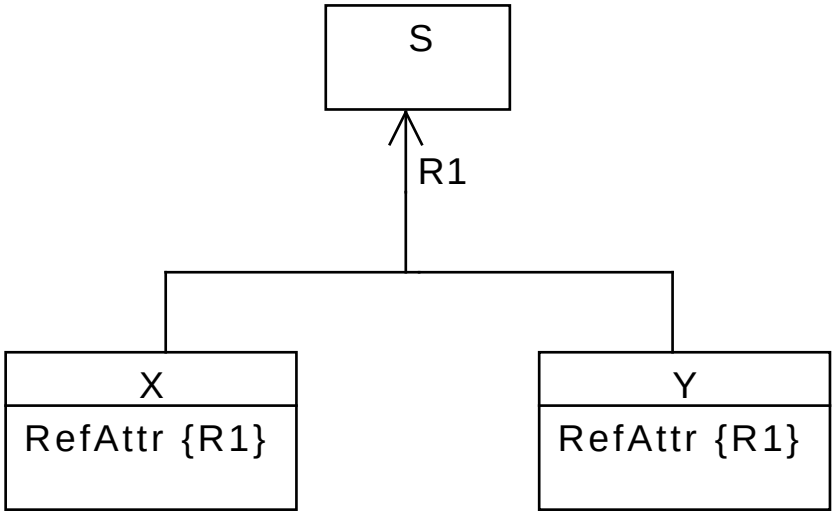

Figure 3.2: Generalization Relationship

So we can say that we navigate from a subclass, say `X`, to the superclass, `S`, by `R1`. In the case of navigating from the superclass to a subclass, we will find it necessary to specify the destination class name. So navigation from `S` to `Y` is specified as `~R1 Y`. Because of the nature of a generalization, navigating from superclass to a specific subclass will result in at most one instance (and possibly zero) being found and navigating from subclass to superclass results in exactly one found instance.

Associative Classes

For class based associations, it is the associator class that holds the referential attributes and it will hold attributes that refer to both participating classes. In this case, the direction of the relationship must be arbitrarily assigned. In the diagram below, we have decided that the forward direction is traversing from `X` to `Y`.

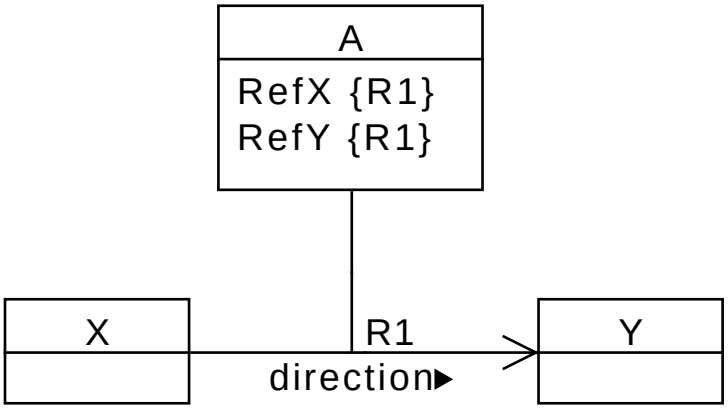

Figure 3.3: Associative Class Relationship

So given an instance of `X`, we can navigate to `Y` via `R1` and, conversely, navigate from `Y` to `X` via `~R1`.

Associator classes may also be the target of relationship navigation. So we may wish to go from `X` to `A`. In this case we represent that as `R1 A`, *i.e.* we go forward along `R1` but stop at `A`. Conversely, we can navigate from `Y` to `A` along `~R1 A`.

Chapter 4

Package Procedures

In this section we begin showing all the procedures associated with the `rosea` run time code. These will be divided into sections that correspond to the figure above that showed how the namespace for the package would be organized.

We start with some preliminaries. Since we are using TclRAL extensively, we need to pull in the package and its utility package.

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

We will find it convenient to import some of the commands from TclRAL since we will be using them repeatedly.

```
<<tclral imports>>=
namespace import\
    ::ral::relation\
    ::ral::tuple\
    ::ral::relformat\
    ::ralutil::pipe
```

The `::ralutil::pipe` command is used extensively in the code for `rosea`. This command has nothing to do with relational algebra but rather is a control structure. It allows a sequence of operations to be written as a linear "pipe" where the result of one command is used as an argument to the next. It turns out that such sequencing is common in relational algebraic processing. What the command does is rewrite the sequence into a set of nested procedure invocations. It is a classic Tcl control structure type of procedure and makes it much easier to understand a sequence of operations without having to tease apart the procedure nesting or introduce superfluous variables.

The namespace is organized as shown above.

```
<<rosea namespace layout>>=
<<helper commands namespace>>
<<class commands namespace>>
<<instance commands namespace>>
<<relationship commands namespace>>
<<dispatch commands namespace>>
<<configuration commands namespace>>
<<population commands namespace>>
<<trace commands namespace>>
```

A Diversion for Testing

We will be intermixing the code and test cases in this document and will be using the `tcltest` package to execute the tests. We will attempt to keep test cases and the code they test lexically near each other. Since we are using TclRAL, we will often in our test cases want to compare the test result with a relation value. We define to `tcltest` a couple of custom matching functions that we will be using in the test result matching.

```
<<test utility procs>>=
proc refMatch {expected actual} {
    lassign $expected erelvar einst
    lassign $actual arelvar ainst
    return [expr {$erelvar eq $arelvar && [ral relation is $einst == $ainst]}]
}
customMatch ref [namespace current]::refMatch

proc relationMatch {expected actual} {
    return [::ral::relation is $expected == $actual]
}
customMatch relation [namespace current]::relationMatch
```

Since state machine dispatch happens in the Tcl event loop, we will need to be able to synchronize a test case execution with the event loop. This means that we need a way to enter the event loop and then leave the event loop when conditions are met. The procedures below are useful for using the event loop in tests.

```
<<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] ; # ❶
    vwait $varname
    after cancel $tid
    set $varname
}
```

❶ We need some kind of timeout to make sure that we don't hang forever in a test if the condition of the test is not met.

Another useful thing to be able to do is catch background errors. Since state activities are run as `after` script callbacks, any errors are reported via the interpreter `bgerror` mechanism. The procedure below help sync to these background errors.

```
<<test utility procs>>=
proc waitForBGError {{timeout 1000}} {
    set bg [interp bgerror {}] ; # ❶
    interp bgerror {} [lambda@ [namespace current] {result opts} {
        syncToTest [dict get $opts -errorcode]
    }]

    set err [waitForSync $timeout]
    interp bgerror {} $bg
    return $err
}
```

❶ We want to save the existing `bgerror` handler so that it can be restored below.

Class Commands

In this section we show the class oriented commands of `rosea`. These commands are the architecture supplied class based operations. They are the target of the ensemble mappings for each of the classes in a domain. Every class in the domain will have an ensemble command that is the same name as the class. All classes will have the commands discussed in this section plus any class based operations that are defined for the class. The invocation synopses below show how to invoke the class named

ensemble command. The procedure implementations have a slightly different interface since the command ensemble mechanism of Tcl is used to supply an additional argument of the name of the relvar upon which the operation will apply.

Class oriented commands fall into these categories:

- Commands that search for instances.
- Commands that create and destroy instances.
- Commands that update the value of the underlying relvar.

First we create the namespace where the commands will reside.

```
<<class commands namespace>>=
namespace eval ClassCmds {
    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path [namespace parent]::Helpers ; # ❶
    <<class commands>>
}
```

❶ The helper commands are used everywhere so it is convenient to include them in the command resolution path.

Find All Instances

The simplest search for instances is just to return them all. The `findAll` command returns an instance reference that refers to all the instances of a class.

class `findAll`

The `findAll` command returns an instance reference that refers to all the instances of *class*.

Implementation

```
<<class commands>>=
proc findAll {relvar} {
    tailcall ToRef $relvar [relvar set $relvar]
}
```

Find an Instance by its Identifier

Sometimes the values of the identifying attributes are known and we can directly locate the instance.

class `findById` *idattr1 idvalue1* ...

`idattN`
The name of an identifying attribute.

`idvalueN`
The value of an identifying attribute.

The `findById` subcommand returns a reference to at most one instance of *class*. The arguments are a set of name / value pairs for the attributes that form an identifier for *class*. The returned reference will be the empty reference if no instance could be found that matches the values of the identifying attributes.

Implementation

```
<<class commands>>=
proc findById {relvar args} {
    tailcall ToRef $relvar [relvar restrictone $relvar {*}$args]
}
```

Tests

```
<<class command tests>>=
test findById-1.0 {
    Find tuple by value of the identifier
} -setup {
    relvar create A {
        Attr1   int
        Attr2   string
    } Attr1
    relvar insert A {
        Attr1 20 Attr2 foo
    } {
        Attr1 40 Attr2 bar
    }
} -cleanup {
    relvar unset A
} -body {
    rosea::ClassCmds::findById [namespace current]::A Attr1 20
} -result {::rosea::test::A {{Attr1 int} {{Attr1 20}}}} -match ref
```

Find Instances Meeting a Criteria

The general case is to find instances based on the result of evaluating an expression.

class `findWhere` *expression*

`expression`
A Tcl expression which selects which instances are returned in the reference.

The `findWhere` command takes an expression and returns an instance reference for those instances of *class* where the expression evaluates to true. During execution, the values of the attributes will be placed in Tcl variable that have the same name as the attribute. The *expression* argument may refer to the attribute-named variables so that the selection can be based on the values in the instance.

Implementation

```
<<class commands>>=
proc findWhere {relvar expr} {
    tailcall ToRef $relvar [uplevel 1 [list ::ral relation restrictwith\
        [relvar set $relvar] $expr]]
}
```

Tests

```
<<class command tests>>=
test findWhere-1.0 {
    Find tuples based on an expression
} -setup {
    relvar create A {
        Attr1   int
        Attr2   string
    } Attr1
    relvar insert A {
        Attr1 20 Attr2 foo
    } {
        Attr1 40 Attr2 bar
    } {
        Attr1 60 Attr2 baz
    }
} -cleanup {
    relvar unset A
} -body {
    # Make sure the expression is evaluated in the proper context
    # and can access local variables
    set lower 20
    set upper 60
    rosea::ClassCmds::findWhere [namespace current]::A {
        $Attr1 > $lower && $Attr1 < $upper
    }
} -result {::rosea::test::A {{Attr1 int} {{Attr1 40}}}} -match ref
```

More Testing Infrastructure

As we proceed, in order to perform some of the tests we will need to set up the data and namespace layout that the procedures expect. Here we present some test support procedures that are used as part of the test case setup and take down.

```
<<test utility procs>>=
proc setupDomain {name} {
    set domns [namespace current]::$name
    namespace eval ${domns} {
        <<arch relvar definitions>>
    }
}

proc cleanupDomain {name} {
    set domns [namespace current]::$name
    relvar constraint delete {*}[relvar constrain names ${domns}::*]
    relvar unset {*}[relvar names ${domns}::*]
    namespace delete $domns
}

proc addClass {domain class heading id} {
    set domns [namespace current]::$domain
    set idheading [list]
    foreach attr $id {
        lappend idheading $attr [dict get $heading $attr]
    }
    relvar create ${domns}::__${class}__STATEINST\
        [concat $idheading [list __State string]]\
        $id
    relvar create ${domns}::$class $heading $id
    namespace eval ${domns}::${class}::__Activity {}
```

```
}

proc addState {domain args} {
    relvar insert [namespace current]::${domain}::__Arch_State $args
}

proc addEvent {domain args} {
    relvar insert [namespace current]::${domain}::__Arch_Event $args
}

proc addTransition {domain args} {
    relvar insert [namespace current]::${domain}::__Arch_Transition $args
}

proc addInitialState {domain args} {
    relvar insert [namespace current]::${domain}::__Arch_InitialState $args
}

proc addAssocLink {domain linkattrs assocattrs} {
    relvar eval {
        relvar insert [namespace current]::${domain}::__Arch_Link $linkattrs
        relvar insert [namespace current]::${domain}::__Arch_AssociationLink\
                $assocattrs
    }
}

proc addParitionLink {domain linkattrs} {
    relvar insert [namespace current]::${domain}::__Arch_Link $linkattrs
    relvar insert [namespace current]::${domain}::__Arch_PartitionLink\
            $linkattrs
}

proc addPartitionDst {domain dstattrs} {
    relvar insert [namespace current]::${domain}::__Arch_PartitionDst\
        $dstattrs
}
```

Create an Instance

The `create` command creates instances. If the class has a state model, then the instance is placed in its default initial state.

class `create` *attr1 value1 attr2 value2 ...*

`attrN`
The name of an attribute of *class*.

`valueN`
The corresponding value to be given to the attribute.

When creating an instance, it is necessary to give each attribute a value, unless a default value was defined for the attribute. The command returns an instance reference to the newly created instance.

Implementation

```
<<class commands>>=
proc create {relvar args} {
```

```
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }
    set ref [ToRef $relvar [relvar insert $relvar $args]] ; # ❶
    SplitRelvarName $relvar domain class
    CreateInInitialStateFromRef $domain $class $ref
    return $ref
}
```

❶ Here we see the correspondence between an XUML class instance and a tuple in a relvar. The instance is create by inserting a tuple into the relvar.

Tests

```
<<class command tests>>=
test create-1.0 {
    Create an instance -- no state model
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
} -cleanup {
    cleanupDomain foo
} -body {
    rosea::ClassCmds::create [namespace current]::foo::c1 A1 20 A2 bar
} -result {::rosea::test::foo::c1 {{A1 int} {{A1 20}}}} -match ref
```

Create an Instance in a Given State

It is sometimes desirable to create an instance in a state other than its default initial state.

class `createin` *state attr1 value1 attr2 value2 ...*

`state`
The name of the state in which the new instance is placed. The action of that state is **not** executed.

`attrN`
The name of an attribute of *class*.

`valueN`
The corresponding value to be given to the attribute.

When creating an instance, it is necessary to give each attribute a value, unless a default value was defined for the attribute. The command returns an instance reference to the newly created instance.

Implementation

```
<<class commands>>=
proc createin {relvar state args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }
    SplitRelvarName $relvar domain class
    set initstate [relvar restrictone ${domain}::__Arch_State\
        Class $class State $state]
```

```
    if {[relation isempty $initstate]} { # ❶
        tailcall DeclError UNKNOWN_STATE $state $relvar
    }
    set ref [ToRef $relvar [relvar insert $relvar $args]]
    CreateStateInstanceFromRef $domain $class $state $ref
    return $ref
}
```

❶ First, we must make sure that requested state is indeed a state of the class.

```
<<error code formats>>=
UNKNOWN_STATE   {unknown state, "%s", for class, "%s"}
```

Tests

```
<<class command tests>>=
test createin-1.0 {
    Create an instance in a given state
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State Idle
        addEvent foo Class c1 Event e1
        addTransition foo Class c1 State Idle Event e1 NewState Idle
        addInitialState foo Class c1 State Idle
    }
} -cleanup {
    cleanupDomain foo
} -body {
    set ref [::rosea::ClassCmds::createin [namespace current]::foo::c1\
        Idle A1 40 A2 bar]
    set statetest [pipe {
        relvar restrictone ::rosea::test::foo::__c1__STATEINST A1 40 |
        relation extract ~ __State
    }]
    expr {$statetest eq "Idle" &&\
        [refMatch {::rosea::test::foo::c1 {{A1 int} {{A1 40}}}} $ref]}
} -result {1}
```

We factor out the code to create the instance tuple that tracks the state of the instance.

```
<<helper commands>>=
proc CreateStateInstance {domns class state idattrs} {
    tailcall relvar insert ${domns}::__${class}__STATEINST\
            [concat $idattrs [list __State $state]]
}
```

In `CreateStateInstance`, the `idattrs` parameter is a list of attribute name / value pairs for the identifier for `class`. The current state of a class instance that has a state machine defined for it is kept in a separate relvar. That relvar has a heading of the identifying attributes plus the `__State` attribute that holds the current state of the instance.

We also have a procedure that creates an state instance tuple for the default initial state.

```
<<helper commands>>=
proc CreateInInitialState {domns class idattrs} {
    set initstate [relvar restrictone ${domns}::__Arch_InitialState\
        Class $class]
```

```
    if {[relation isnotempty $initstate]} {
        CreateStateInstance $domns $class [relation extract $initstate State]\
            $idattrs
    }
    return
}
```

It is also useful to create the state instance tuples from a corresponding instance reference.

```
<<helper commands>>=
proc CreateStateInstanceFromRef {domns class state ref} {
    relation foreach inst [lindex $ref 1] { # ❶
        CreateStateInstance $domns $class $state\
            [tuple get [relation tuple $inst]]
    }
    return
}
```

❶ We iterate across the reference because we may be creating multiple state instance tuples.

It is also convenient to have a procedure to look up the default initial state when dealing with instance references.

```
<<helper commands>>=
proc CreateInInitialStateFromRef {domns class ref} {
    set initstate [relvar restrictone ${domns}::__Arch_InitialState\
        Class $class]
    if {[relation isnotempty $initstate]} {
        CreateStateInstanceFromRef $domns $class\
            [relation extract $initstate State] $ref
    }
    return
}
```

Asynchronous Instance Creation

The above creation commands are used to create class instances synchronously. For those cases, the instance is placed in an initial state (if it has an associated state model), but no state activity is executed as part of the creation.

By contrast, asynchronous creation happens when an event is dispatched. A new class instance is created and placed in a pseudo-initial state. The event is dispatched immediately to the newly created instance to cause a transition and, necessarily, have a state activity executed. These events are sometimes referred to as *creation* events.

class `createasync` *event eventparams attr1 value1 attr2 value2 ...*

`event`
The name of the event to signal to the newly created instance.

`eventparams`
A list of event parameters that are to be sent along with the event. If the event does not carry any parameters, then this argument must be specified as the empty list.

`attrN`
The name of an attribute of *class*.

`valueN`
The corresponding value to be given to the attribute.

The command returns the empty string. Note that the instance is not created nor does any transition occur until the creation event is dispatched.

Implementation

The implementation of `createasync` resembles that of signaling and ordinary event.

```
<<class commands>>=
proc createasync {relvar event eventparams args} {
    SplitRelvarName $relvar domain class

    <<createasync: validate event>>
    <<createasync: validate pseudo initial state>>
    <<createasync: construct event information>>
    <<createasync: queue event>>
    return
}
```

We want to do some validation before puting an event on the event queue in an effort to detect errors as early as we can. So, we insist that the creation event be one of the events defined for the class state model.

```
<<createasync: validate event>>=
if {[relation isempty [relvar restrictone ${domain}::__Arch_Event\
        Class $class Event $event]]} {
    tailcall DeclError UNKNOWN_EVENT $event $class
}
```

```
<<error code formats>>=
UNKNOWN_EVENT   {unknown event, "%s", for class, "%s"}
```

It is also necessary that the state model be defined as having as transition from the pseudo initial state. You can't send a creation event to just any state model. The pseudo-initial state is named `@` and so that must be one of the defined states for the class.

```
<<createasync: validate pseudo initial state>>=
if {[relation isempty [relvar restrictone ${domain}::__Arch_State\
        Class $class State @]]} {
    tailcall DeclError NO_CREATION_EVENTS $class
}
```

```
<<error code formats>>=
NO_CREATION_EVENTS   {class, "%s", has no defined creation events}
```

Now we just construct the event information. There is a `type` member of the event dictionary to indicate that this is a creation event and has to be processed slightly differently.

```
<<createasync: construct event information>>=
set srcref [SelfInstRef]
set eventInfo [dict create\
    type creation\
    src $srcref\
    dstClass $relvar\
    dstAttrs $args\
    event $event\
    params $eventparams\
]
```

For a creation event, we need to know the class where the new instance will be created and the attribute values for the class.

Finally, we determine if the creation event will start a thread of control or not. This depends upon whether or not we are sending the event from a state activity.

```
<<createasync: queue event>>=
if {[::rosea::InstCmds::isRefEqual $srcref [nilInstRef]]} {
    lappend ::rosea::Dispatch::toc_queue $eventInfo
```

```
} else {
    lappend ::rosea::Dispatch::event_queue $eventInfo
}
::after 0 ::rosea::Dispatch::DispatchEvent
```

Tests

```
<<class command tests>>=
test createasync-1.0 {
    Signal a creation event
} -setup {
    rosea configure {
        domain foo {
            class A {
                attribute A_Id string -id 1
                attribute A_Attr string

                statemodel {
                    transition @ - New -> Initialize

                    state Initialize {bid battr} {
                        B create B_Id $bid B_Attr $battr\
                            A_Id [readAttribute $self A_Id]
                        ::rosea::test::syncToTest $bid
                    }
                }
            }
            class B {
                attribute B_Id string -id 1
                attribute B_Attr string
                attribute A_Id string
                reference R1 A -link A_Id
            }

            association R1 B 1--1 A

            operation newA {} {
                A createasync New {5 foo} A_Id 17 A_Attr bar
            }
        }
    }

    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    ::foo newA
    waitForSync
} -result {5}
```

```
<<class command tests>>=
test createasync-1.1 {
    Signal a creation event to class without pseudo initial state
} -setup {
    rosea configure {
        domain foo {
            class A {
                attribute A_Id string -id 1
```

```
                attribute A_Attr string

                statemodel {
                    transition Initialize - New -> Initialize

                    state Initialize {bid battr} {
                        B create B_Id $bid B_Attr $battr\
                            A_Id [readAttribute $self A_Id]
                    }
                }
            }
            class B {
                attribute B_Id string -id 1
                attribute B_Attr string
                attribute A_Id string
                reference R1 A -link A_Id
            }

            association R1 B 1--1 A

            operation newA {} {
                A createasync New {5 foo} A_Id 17 A_Attr bar
            }
        }
    }

    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    ::foo newA
} -result {class, "A", has no defined creation events} -returnCodes error
```

```
<<class command tests>>=
test createasync-1.2 {
    Signal a creation event, instance creation fails.
} -setup {
    rosea configure {
        domain foo {
            class A {
                attribute A_Id string -id 1
                attribute A_Attr string

                statemodel {
                    transition @ - New -> Initialize

                    state Initialize {bid battr} {
                        B create B_Id $bid B_Attr $battr\
                            A_Id [readAttribute $self A_Id]
                        ::rosea::test::syncToTest $bid
                    }
                }
            }
            class B {
                attribute B_Id string -id 1
                attribute B_Attr string
                attribute A_Id string
                reference R1 A -link A_Id
            }

            association R1 B 1--1 A
```

```
            operation newA {} {
                A createasync New {5 foo} A_Id 17
            }
        }
    }

    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    ::foo newA
    lindex [waitForBGError] 1
} -result {ASYNC_CREATION_FAILED}
```

Updating Class Values

It is often more convenient to perform operations on a set of instances and then have the results of the operations updated. This command will update an entire set of non-identifying attribute values.

class `update` *relationvalue*
Update the non-identifying attributes of *class* based on the values in the heading of *relationvalue*. The head of *relationvalue* must include the attributes of at least one identifier. The tuples in *class* that match the identifying attribute values will have the remaining non-identifying attributes changed to match the values found in *relationvalue*

Implementation

```
<<class commands>>=
proc update {relvar relvalue} {
    tailcall ToRef $relvar [relvar updateper $relvar $relvalue]
}
```

Tests

```
<<class command tests>>=
test update-1.0 {
    Update a set of attribute values
} -setup {
    setupDomain foo
    addClass foo c1 {A1 string Temp int Volume double} A1
    rosea::ClassCmds::create [namespace current]::foo::c1\
        A1 tank1 Temp 100 Volume 3.3
    rosea::ClassCmds::create [namespace current]::foo::c1\
        A1 tank2 Temp 200 Volume 6.3
    rosea::ClassCmds::create [namespace current]::foo::c1\
        A1 tank3 Temp 300 Volume 10.3
} -cleanup {
    cleanupDomain foo
} -body {
    set ref [rosea::ClassCmds::findAll [namespace current]::foo::c1]
    # This adds 10 to the Temp and multiplies the Volume by 2 for
    # all the instances. We do it by extending relation value with
    # new attributes and then eliminating the old attributes and renaming
```

```
    # the new attributes to have the old attribute names.
    set newvalues [relation update [rosea::InstCmds::deRef $ref Temp Volume]\
            newtuple {1} {
        tuple update $newtuple\
            Temp [expr {[tuple extract $newtuple Temp] + 10}]\
            Volume [expr {[tuple extract $newtuple Volume] * 2.0}]
    }]
    log::debug [relformat $newvalues newvalues]
    set ref [rosea::ClassCmds::update [namespace current]::foo::c1 $newvalues]
    # Extract one of the instances and make sure the attribute values
    # were properly updated.
    pipe {
        rosea::InstCmds::deRef $ref |
        relation restrictwith ~ {$Temp < 200} |
        relation extract ~ Temp Volume
    }
} -result {110 6.6}
```

Relationship Commands

In this section we show the relationship oriented commands of `rosea`. These commands are supplied by the architecture and are the the commands to which the relationship ensemble subcommands are mapped.

We place the relationship commands into a separate namespace.

```
<<relationship commands namespace>>=
namespace eval RelCmds {
    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path [list\
        [namespace parent]::Helpers\
        [namespace parent]::InstCmds\
    ]
    <<relationship commands>>
}
```

Linking Simple Associations

We first consider linking across simple relationships. Later we discuss linking for class based associations. They are very similar, but class based associations may require additional arguments and so the procedure interface is different.

relationship `link` *instref1 instref2*

`instref1`
: An instance reference to a class that participates in *relationship*.

`instref2`
: An instance reference to the other class that participates in *relationship*.

The `link` command creates a relationship linkage between the class instances referenced by *instref1* and *instref2* for the simple association, *relationship*.

Implementation

The figure below shows the architectural relvars required to link instances.

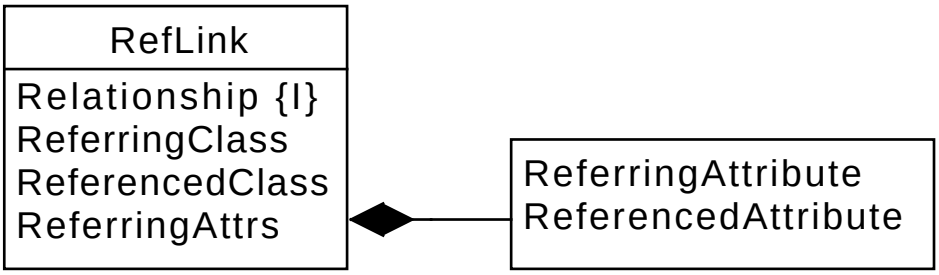

Figure 4.1: Data Required for Linking Simple Associations

The `RefLink` relvar contains the information required to support linking instances at run time. The `ReferringAttrs` attribute is a relation valued attribute and we use the UML composition notation to show the nested attributes of the `ReferringAttrs` attribute. For the case of a simple association, we have one class that plays the role of the `ReferringClass` that contains attribute references to the `ReferencedClass`. Since referential attributes refer to an identifier and since identifiers may consist of multiple attributes, the `ReferringAttrs` attribute is a relation value that gives the mapping with the `ReferringAttribute` / `ReferencedAttribute` tuples specifying the attribute names holding the referential correspondence. This relvar is populated by the domain generation process. The relationship command that links instances in a simple association then uses this data at run time to perform the required operations.

As we discussed above, linking across simple relationships is accomplished by updating referential attributes. We want to be able to give instance references in any order so we will need to be able to determine which class is the referring class and which is the referenced class. The linking is accomplished by storing the values of the referenced attributes into the referential attributes That correspondence is given in the appropriate tuple of the `RefLink` relvar which tells us which values to query and which values will be updated. We can outline the `linkSimple` procedure as shown below.

```
<<relationship commands>>=
proc linkSimple {rname instref1 instref2} {
    <<linkSimple: validate arguments>>
    <<linkSimple: look up link info>>
    <<linkSimple: determine reference direction>>
    <<linkSimple: obtain referenced values>>
    <<linkSimple: update referring values>>

    return
}
```

We need to break apart the arguments into their component parts and insure that we aren't attempting something absurd such as linking across domains.

```
<<linkSimple: validate arguments>>=
lassign $instref1 relvar1 inst1
SplitRelvarName $relvar1 domain1 class1

lassign $instref2 relvar2 inst2
SplitRelvarName $relvar2 domain2 class2

if {$domain1 ne $domain2} {
    tailcall DeclError NO_CROSS_DOMAIN $domain1 $domain2
}
```

```
<<error code formats>>=
NO_CROSS_DOMAIN         {cannot link instances across domains,\
                             got "%s" and "%s"}
```

We next find the instance of `RefLink` that matches the relation name. The graphical representation of the linkage data is translated into TclRAL relvars. This shows the structure of the relation values with which we will be dealing.

```
<<arch relvar definitions>>=
ral relvar create __Arch_RefLink {
    Relationship    string
    ReferringClass  string
    ReferencedClass string
    ReferringAttrs  {Relation
            {ReferringAttribute string ReferencedAttribute string}}
} Relationship
```

Using the relationship name as the identifier, we can lookup the link information and assign the attributes into Tcl variables. This will allow us to detect any unknown relationships. Note that even though `ReferringAttrs` is a relation valued attribute, its value can be stored in an ordinary Tcl variable.

```
<<linkSimple: look up link info>>=
set reflink [relvar restrictone ${domain1}::__Arch_RefLink Relationship $rname]
if {[relation isempty $reflink]} {
    tailcall DeclError UNKNOWN_RELATIONSHIP $rname
}
relation assign $reflink\
    {ReferringClass referringClass}\
    {ReferencedClass referencedClass}\
    {ReferringAttrs referringAttrs} ; # ❶
```

❶ The `relation assign` command allows us to specify both the attributes to place into variables as well as the name those variables should have.

```
<<error code formats>>=
UNKNOWN_RELATIONSHIP    {unknown relationship, "%s"}
```

At this point we have the linkage information in a convenient form with which to work.

As we stated, we don't want to impose an order on the instance references, *i.e.* we don't want to have to specify the referring class and the referenced class in any particular order of the arguments. The system already *knows* this information and can take on the burden of determining the class roles in the association. The role determination is made by comparing the class names of the instance references to those found in the `RefLink` tuple. This also allows us to check that the classes actually participate in the given relationship.

```
<<linkSimple: determine reference direction>>=
if {$class1 eq $referringClass && $class2 eq $referencedClass} {
    set fromrelvar $relvar1
    set frominsts $inst1
    set torelvar $relvar2
    set toinst $inst2
} elseif {$class2 eq $referringClass && $class1 eq $referencedClass} {
    set fromrelvar $relvar2
    set frominsts $inst2
    set torelvar $relvar1
    set toinst $inst1
} else {
    tailcall DeclError NON_PARTICIPANTS $relvar1 $relvar2 $rname\
        $referringClass $referencedClass
}
```

```
<<error code formats>>=
NON_PARTICIPANTS        {"%s" and "%s" don't participate in %s,\
                            expected "%s" and "%s"}
```

You may link many referring instances to the same referenced instance, but there can only be one referenced instance.

```
<<linkSimple: determine reference direction>>=
if {[relation cardinality $toinst] != 1} {
    tailcall MUST_BE_SINGULAR $torelvar [relation cardinality $toinst]
}
```

```
<<error code formats>>=
MUST_BE_SINGULAR    {number of referred to instances for "%s" must be one,\
                        got %d}
```

Notice that this code does not consider the multiplicity of the underlying relationship. Even if the relationship is defined to be singular, this code will link multiple referring instances. The error will be detected when the TclRAL transaction ends since such actions will violate referential integrity. This greatly simplifies this code since the TclRAL enforces all the integrity constraints.

The update we want to perform is to modify the values of the referential attributes to match the values of the referenced attributes. First, we obtain the class instance by following the instance reference. We perform the `semijoin` in place since we have already split apart the instance reference into its components.

```
<<linkSimple: obtain referenced values>>=
set refedvalue [relation semijoin $toinst [relvar set $torelvar]]
```

Our strategy for updating the referential attributes is to create a dictionary whose keys are the referential attribute names and whose values are those of the referenced attributes. It is important to maintain the correspondence between the referential and referenced attributes. The `RefLink` relvar does this by having the `ReferringAttrs` attribute as a relation valued attribute. By extending that relation to include the value of the referenced attribute, we can then convert the result into a dictionary containing the referring attribute name along with the referenced value.

```
<<linkSimple: update referring values>>=
set tovalues [pipe {
    relation extend $referringAttrs rfa RefValue string {
        [relation extract $refedvalue\
            [tuple extract $rfa ReferencedAttribute]]} |
    relation dict ~ ReferringAttribute RefValue
}]
```

Now we iterate across the referring instances updating the referential attributes to be the values of the dictionary we just computed.

```
<<linkSimple: update referring values>>=
relation foreach frominst $frominsts {
    relvar updateone $fromrelvar fromtup\
            [tuple get [relation tuple $frominst]] { # ❶
        tuple update $fromtup {*}$tovalues ; # ❷
    }
}
```

❶ The `relvar updateone` command needs a list of attribute name/value pairs that constitute an identifier. The instance reference accomplishes that and we need only get the attributes and values of the instance reference into the right form. The `relation tuple` command takes a singleton relation value and turns it into a tuple value. The `tuple get` command takes a tuple value and returns a list of alternating attribute names and values which is exactly the form we need to supply to `relvar updateone`.

❷ Here we treat the dictionary, which has attribute names as keys, as a list and obtain the attribute name/value pairs that the `tuple update` command needs.

Linking Class Based Associations

When a class based association is linked, we must create an instance of the associator class. In general, the associator may have additional attributes other than the referential attributes involved in making the association. If so, then values must be supplied

for those attributes and this requirement dictates a different interface for the `link` command when it is applied to class based associations. So for class based associations, the relationship ensemble `link` command will be mapped to the `linkAssoc` procedure given below.

relationship `link` *instref1 instref2 ?attr1 value1 attr2 value2 . . . ?*

`instref1`
An instance reference to a class that participates in *relationship*.

`instref1`
An instance reference to the other class that participates in *relationship*.

`attrN valueN`
The values for attributes of the associator class, other than the referential attributes, are given as attribute name / value argument pairs.

Implementation

The run time data required to create class based associations must describe both the classes participating in the association as well as the associator class itself.

The diagram below shows the structure of the architectural relvars that support linking class based associations.

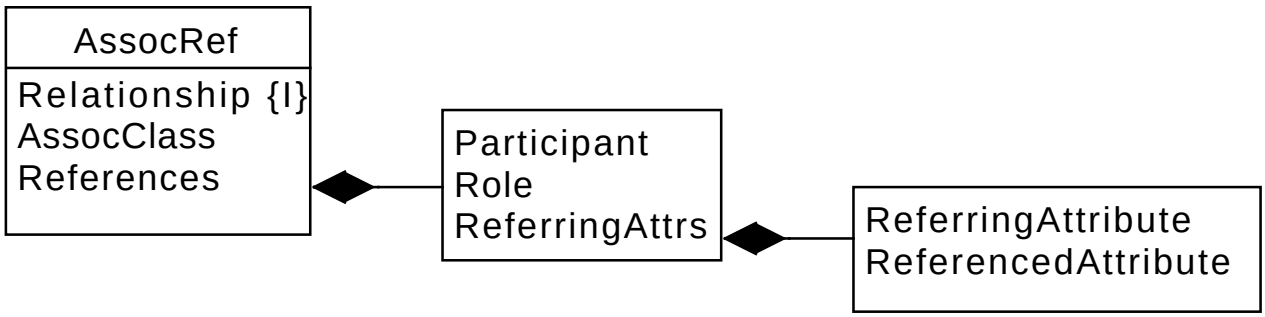

Figure 4.2: Data Required for Linking Class Based Associations

The `References` attribute is relation valued (as shown by the UML composition construct). The relation gives the roles of the participating classes and the attributes that make up the references. The `ReferringAttrs` attribute is also relation valued and has the same heading as the `ReferringAttrs` attribute for simple associations.

As expected, the implementation of `linkAssoc` follows the outline as that for `linkSimple`.

```
<<relationship commands>>=
proc linkAssoc {rname instref1 instref2 args} {
    <<linkAssoc: validate arguments>>
    <<linkAssoc: look up link info>>
    <<linkAssoc: determine reference direction>>
    <<linkAssoc: obtain referenced values>>
    <<linkAssoc: update referring values>>
    <<linkAssoc: create assoc class instances>>
}
```

We need to access the components of the instance references and insure that there is no cross domain link attempt.

```
<<linkAssoc: validate arguments>>=
lassign $instref1 relvar1 inst1
SplitRelvarName $relvar1 domain1 class1

lassign $instref2 relvar2 inst2
SplitRelvarName $relvar2 domain2 class2

if {$domain1 ne $domain2} {
    tailcall DeclError NO_CROSS_DOMAIN $domain1 $domain2
}
```

The graphic of the class based association link data is again a straight forward transliteration into TclRAL. The fact that we have a relation valued attribute that, in turn, has an attribute that is relation valued makes for some deeper nesting of type information.

```
<<arch relvar definitions>>=
ral relvar create __Arch_AssocRef {
    Relationship    string
    AssocClass      string
    References      {Relation\
        {Participant string Role string ReferringAttrs {Relation\
            {ReferringAttribute string ReferencedAttribute string}}}}
} Relationship
```

Now we look up the instance of `AssocRef` that corresponds to the relationship and assign the attributes into Tcl variables.

```
<<linkAssoc: look up link info>>=
set assocref [relvar restrictone ${domain1}::__Arch_AssocRef\
        Relationship $rname]
if {[relation isempty $assocref]} {
    tailcall DeclError UNKNOWN_RELATIONSHIP $rname
}

relation assign $assocref\
    {AssocClass assocClass}\
    {References references}
```

The `references` variable contains a relation value so the code below will use `relation` commands to access the value. For class based associations, we need to identify the participants and check that the instance references refer to relvars that indeed participate in the relationship.

```
<<linkAssoc: determine reference direction>>=
set ref1 [relation restrictwith $references {$Participant eq $class1}]
if {[relation isempty $ref1]} {
    tailcall DeclError NOT_IN_ASSOCIATION $class1 $rname
}
```

```
<<error code formats>>=
NOT_IN_ASSOCIATION          {"%s" $relvar1 not a participant in "%s"}
```

Reflexive associations have to be considered. If the association is reflexive then the above query will yield two tuples since the values of the `Participant` attribute are the same name. In this case, we query for the "target" as a means of disambiguating the reflexive relationship.

For the non-reflexive case, we can just find the other participant, checking that it is indeed there.

```
<<linkAssoc: determine reference direction>>=
if {[relation cardinality $ref1] > 1} {
    set ref2 [relation restrictwith $ref1 {$Role eq "target"}]
    set ref1 [relation minus $ref1 $ref2] ; # ❶
} else {
    set ref2 [relation restrictwith $references {$Participant eq $class2}]
```

```
    if {[relation isempty $ref2]} {
        tailcall DeclError NOT_IN_ASSOCIATION $class2 $rname
    }
}

set rattr1 [relation extract $ref1 ReferringAttrs]
set rattr2 [relation extract $ref2 ReferringAttrs]
```

❶ We are justified in this code sequence because we know that there are never more than two participants in the association and that the generation of the architectural values will always create a *target* role for one of the participants.

At this point, `rattr1` and `rattr2` contain the attribute reference information corresponding to `instref1` and `instref2`. Note that they are relation valued as we have a situation where one relation valued attribute contains an attribute that is itself relation valued.

We use a similar updating strategy here as with `linkSimple`. First we de-reference the instance references against the relvar. Then we want to create dictionaries with the referring attributes as keys and the referenced values as the dictionary values.

```
<<linkAssoc: obtain referenced values>>=
set values1 [relation semijoin $inst1 [relvar set $relvar1]]
set values2 [relation semijoin $inst2 [relvar set $relvar2]]
```

Since linking a class based association amounts to inserting tuples in the associator relvar, we will accumulate the set of tuples and perform a single `relvar insert` of all the tuples. Note that we use the same extend and convert to a dictionary as we did in `linkSimple`.

```
<<linkAssoc: update referring values>>=
set assoctuples [list]
relation foreach v1 $values1 { # ❶
    set refto1 [pipe {
        relation extend $rattr1 rfa RefValue string {
            [relation extract $v1 [tuple extract $rfa ReferencedAttribute]]
        } |
        relation dict ~ ReferringAttribute RefValue
    }]
    relation foreach v2 $values2 {
        set refto2 [pipe {
            relation extend $rattr2 rfa RefValue string {
                [relation extract $v2\
                    [tuple extract $rfa ReferencedAttribute]]
            } |
            relation dict ~ ReferringAttribute RefValue
        }]

        lappend assoctuples [dict merge $args $refto1 $refto2] ; # ❷
    }
}
```

❶ The instance references may refer to multiple instances so we need to compute the Cartesian product of the instance references. The nested iteration will insure that we end up with an associator instance for each reference in each of the instance references of the participating classes.

❷ It may be the case that the associator class has other attributes. Those are passed as attribute/value pairs in the `args`. The merge order is important here just in case the caller tried to set one of the referential attributes in the invocation. By merging the referential attributes last, they will take precedence over any caller arguments.

Finally, we insert the tuples that we have created into the association class relvar and return an instance reference to the new created instances.

```
<<linkAssoc: create assoc class instances>>=
set ref [ToRef ${domain1}::$assocClass\
        [relvar insert ${domain1}::$assocClass {*}$assoctuples]]
    CreateInInitialStateFromRef $domain1 $assocClass $ref ; # ❶
return $ref
```

❶ Note that there is no way to create an associative class that has a state model in anything but the default initial state when using the `link` command. Although technically acceptable to create an associative class instance in an arbitrary state, it is a very rare need. If this situation arises, the programmer will have to do the creation using `createin` and arrange the referential attribute values properly to implement the link.

Tests

```
<<class command tests>>=
test linkAssoc-1.0 {
    Create associative links.
} -setup {
    rosea configure {
        domain assoc {
            class Dog {
                attribute Name string -id 1
                attribute Breed string
            }
            class Owner {
                attribute Name string -id 1
                attribute Age int
            }
            class Ownership {
                attribute DogName string -id 1
                attribute OwnerName string -id 1
                reference R1 Owner -link {OwnerName Name}
                reference R1 Dog -link {DogName Name}

                statemodel {
                    state Owning {} {
                        puts "Own dog [readAttribute $self DogName]"
                    }
                    transition Owning - Sold -> Disposed

                    state Disposed {} {
                        puts "No longer own dog [readAttribute $self DogName]"
                    }
                }
            }

            association R1 Owner 0..*--0..* Dog -associator Ownership
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::assoc
} -body {
    set d1 [assoc::Dog create Name Fluffy Breed Poodle]
    set d2 [assoc::Dog create Name Tiger Breed Mut]
    set dogs [rosea::InstCmds::refUnion $d1 $d2]
    set owners [assoc::Owner create Name Fred Age 32]
    set a [assoc::R1 link $dogs $owners]
```

```
    testConditions\
        {[rosea::InstCmds::refMultiplicity $a] == 2}\
        {[relation cardinality [relvar set ::assoc::__Ownership__STATEINST]]\
            == 2}
} -result {1}
```

Unlinking Simple Associations

The inverse of linking instances in a relationship is to unlink them. In this case, the interface is much simpler. Since the system knows the details of the relationship, it is only necessary to specify an instance reference to one of the participants that is to be unlinked.

relationship `unlink` *instref*

instref
An instance reference to a class that participates in *relationship*.

Implementation

The implementation of unlink must determine the role of the class instances in the relationship and if the relationship is simple or class based. We perform that work here to ease the burden on the translation since we have all the information required to make the necessary determination of role. For simple associations, we will set the referring attributes to the empty string. This code follows the same pattern as for `linkSimple` and so we present it without any additional commentary.

```
<<relationship commands>>=
proc unlinkSimple {rname instref} {
    lassign $instref relvar inst
    SplitRelvarName $relvar domain class

    set reflink [relvar restrictone ${domain}::__Arch_RefLink\
            Relationship $rname]
    if {[relation isempty $reflink]} {
        tailcall DeclError UNKNOWN_RELATIONSHIP $rname
    }
    relation assign $reflink\
        {ReferringClass referringClass}\
        {ReferencedClass referencedClass}\
        {ReferringAttrs referringAttrs}

    if {$class eq $referringClass} {
        set fromrelvar $relvar
        set frominsts $inst
    } elseif {$class eq $referencedClass} {
        lassign [::rosea::InstCmds::findRelated $instref ~$rname]\
            fromrelvar frominsts
    } else {
        tailcall DeclError NOT_IN_ASSOCIATION $relvar $rname
    }

    set tovalues [pipe {
        relation extend $referringAttrs rfa RefValue string {{}} |
        relation dict ~ ReferringAttribute RefValue
    }]

    relation foreach frominst $frominsts {
        relvar updateone $fromrelvar fromtup\
                [tuple get [relation tuple $frominst]] {
```

```
            tuple update $fromtup {*}$tovalues
        }
    }

    return
}
```

Unlinking Class Based Associations

For relationships that are classed based, the unlink command will be mapped to `unlinkAssoc` procedure as given below. We maintain the same interface but the end result will be deleting instances of the associator class.

Implementation

The `unlinkAssoc` follows the same pattern as `linkAssoc`. The goal here is to delete instances of the associative class.

```
<<relationship commands>>=
proc unlinkAssoc {rname instref} {
    lassign $instref relvar insts
    SplitRelvarName $relvar domain class

    set assocref [relvar restrictone ${domain}::__Arch_AssocRef\
            Relationship $rname]
    if {[relation isempty $assocref]} {
        tailcall DeclError UNKNOWN_RELATIONSHIP $rname
    }
    relation assign $assocref\
        {Relationship relationship}\
        {AssocClass associator}\
        {References references}

    # If we are given instances to the associator class, then there is no
    # more work to do.
    if {$class ne $associator} {
        # Otherwise, we have to find the associator class instances ourselves.
        set part [relation restrictwith $references {$Participant eq $class}]
        set partcard [relation cardinality $part]
        if {$partcard == 0} {
            tailcall DeclError NOT_IN_ASSOCIATION $class $rname
        } elseif {$partcard > 1} {
            tailcall DeclError AMBIGUOUS_UNLINK $rname $relvar ; # ❶
        } else {
            # Find associative class instances.  We have to determine the
            # direction of the relationship traversal.
            set navdir [expr {[relation extract $part Role] eq "source" ?\
                "$relationship" : "~$relationship"}]
            set instref [::rosea::InstCmds::findRelated $instref\
                [list $navdir [namespace tail $associator]]]
            lassign $instref relvar insts
            SplitRelvarName $relvar domain class
        }
    }

    # We are going to delete tuples from the relvar that holds the class
    # instances.
    set delrelvars [list ${domain}::$class]
    # If the associative class also has a state model, then we need to also
    # delete the state instance tuples.
    if {[relvar exists ${domain}::__${class}__STATEINST]} {
```

```
        lappend delrelvars ${domain}::__${class}__STATEINST
    }

    # Now, we iterate across the associative class instances and delete all the
    # tuples in class relvar and, if necessary, the state instances.
    relation foreach inst $insts {
        set idattrs [tuple get [relation tuple $inst]]
        foreach rv $delrelvars {
            relvar deleteone $rv {*}$idattrs
        }
    }
    return $instref
}
```

❶ Because reflexive relationships have an ambiguity, they can only be unlinked if the caller supplies a reference to associator class instances. So, for associative classes, the caller must navigate to the associator class before invoking `unlink`.

```
<<error code formats>>=
AMBIGUOUS_UNLINK         {"%s" is reflexive and linking via "%s" is ambiguous}
```

Tests

```
<<class command tests>>=
test unlinkAssoc-1.0 {
    Destroy associative links.
} -setup {
    rosea configure {
        domain assoc {
            class Dog {
                attribute Name string -id 1
                attribute Breed string
            }
            class Owner {
                attribute Name string -id 1
                attribute Age int
            }
            class Ownership {
                attribute DogName string -id 1
                attribute OwnerName string -id 1
                reference R1 Owner -link {OwnerName Name}
                reference R1 Dog -link {DogName Name}

                statemodel {
                    state Owning {} {
                        puts "Own dog [readAttribute $self DogName]"
                    }
                    transition Owning - Sold -> Disposed

                    state Disposed {} {
                        puts "No longer own dog [readAttribute $self DogName]"
                    }
                }
            }

            association R1 Owner 0..*--0..* Dog -associator Ownership
        }
    }
    rosea generate
```

```
    set owner [assoc::Owner create Name Fred Age 32]
    assoc::R1 link $owner [assoc::Dog create Name Fluffy Breed Poodle]
    assoc::R1 link $owner [assoc::Dog create Name Tiger Breed Mut]
} -cleanup {
    cleanupConfigData
    removeDomain ::assoc
} -body {
    set dogs [assoc::Dog findAll]
    assoc::R1 unlink $dogs
    testConditions\
        {[relation cardinality [relvar set ::assoc::Ownership]] == 0}\
        {[relation cardinality [relvar set ::assoc::__Ownership__STATEINST]]\
            == 0}
} -result {1}
```

Migrating Subclasss In Generalizations

For the case of generalization relationships, the notion of linking and unlinking across the relationship doesn't make much sense. To understand why, recall that in the XUML usage a generalization relationship partitions the superclass instances into a disjoint union of the subclasses. This means that each subclass instance is unconditionally related to exactly one superclass instance and each superclass instance is related to exactly one subclass instance from among all the subclasses of the generalization. If you were to unlink a subclass from its superclass the only operations that would not violate referential integrity is to link a new subclass instance or delete both the superclass and subclass instances. So the only meaningful operation that does not violate referential integrity is to *migrate* a subclass instance. Conceptually, migration consists of:

- Unlink the superclass and subclass.
- Delete the subclass instance.
- Create a new instance of a subclass of the generalization.
- Link the new instance to the superclass.

One additional complication is that the newly created subclass instance may have attributes that need to be set. These are included in the invocation of `migrate` as attribute name / value pairs.

relationship `migrate` *instref subclass ?attr1 value1 attr2 value2 ... ?*

`instref`
: An instance reference to a subclass that participates in *relationship*.

`subclass`
: The name of a subclass in the generalization, *relationship*, to which the instances referred to by *instref* will be migrated.

`attrN valueN`
: Arguments that are taken as attribute name / value pairs representing attribute values that will be given to the newly created instance of *subclass*. Any attributes of *subclass* other than those of the identifying referential ones that realize the generalization must be specified here or have declared default values.

This command returns an instance reference to the newly migrated instance.

Implementation

The diagram below shows the structure of the architectural relvars required to support subtype migration.

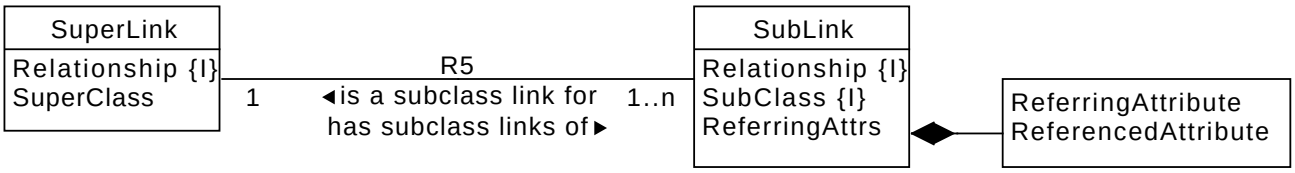

Figure 4.3: Data Required for Migrating Generalization Relationship

The graphic becomes the TclRAL commands as shown below.

```
<<arch relvar definitions>>=
ral relvar create __Arch_SuperLink {
    Relationship    string
    SuperClass      string
} Relationship

ral relvar create __Arch_SubLink {
    Relationship    string
    SubClass        string
    ReferringAttrs  {Relation\
        {ReferringAttribute string ReferencedAttribute string}}
} {Relationship SubClass}

ral relvar association __Arch_R5\
    __Arch_SubLink Relationship +\
    __Arch_SuperLink Relationship 1
```

```
<<relationship commands>>=
proc migrate {rname instref subclass args} {
    if {![isRefSingular $instref]} {
        tailcall DeclError SINGLE_REF_REQUIRED [refMultiplicity $instref]
    }
    lassign $instref relvar inst
    SplitRelvarName $relvar domns class

    set link [relvar restrictone ${domns}::__Arch_AssociationLink Name $rname\
            SrcClass $class]
    if {[relation isempty $link]} {
        tailcall DeclError NO_SUBCLASS $rname $relvar
    }

    # Traverse the relationship to the superclass. We need to get the values of
    # the referenced attributes to use in creating the new subclass instance.
    set superinst [pipe {
        relvar set ${domns}::[relation extract $link DstClass] |
        relation semijoin $inst ~ -using [relation extract $link Attrs]
    }]

    set sublink [relvar restrictone ${domns}::__Arch_SubLink\
            Relationship $rname SubClass $subclass]
    if {[relation isempty $sublink]} {
        tailcall DeclError NO_SUBCLASS $rname $subclass
    }

    # Using the same relation extend strategy, we compute a dictionary of the
    # attribute names / values as they must appear in the subclass we are about
```

```
    # to create.
    set refedvalues [pipe {
        relation extract $sublink ReferringAttrs |
        relation extend ~ rval Value string {
            [relation extract $superinst\
                [tuple extract $rval ReferencedAttribute]]
        } |
        relation dict ~ ReferringAttribute Value
    }]

    # We use the class commands to delete the old subclass instance and create
    # the new one. This will make sure that if the subclass has a state model
    # that the initial state is set correctly.
    relvar eval {
        ::rosea::InstCmds::delete $instref
        set ref [::rosea::ClassCmds::create ${domns}::$subclass\
                {*}[dict merge $args $refedvalues]]
    }
    return $ref
}
```

```
<<error code formats>>=
NO_SUBCLASS         {relationship "%s" does not have a "%s"}
```

Signaling an Assigner

An association that is competitive in nature may have an assigner defined for it. This procedure is used to signal events to an assigner. More specifically, associations with defined assigners that are singular will have the `signalAssigner` proc mapped to the ensemble `signal` subcommand.

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

`event`
The name of an event to be signaled to the assigner associated with *relationship*.

`paramN`
Optional event parameters are passed as additional arguments.

Implementation

```
<<relationship commands>>=
proc signalAssigner {rname event args} {
    SplitRelvarName $rname domain relationship
    set assignrelvar ${domain}::__${relationship}__STATEINST
    if {![relvar exists $assignrelvar]} {
        tailcall DeclError NO_ASSIGNER $rname
    }

    set dstref [pipe {
        relvar set $assignrelvar |
        relation project ~ {*}[lindex [relvar identifiers $assignrelvar] 0] |
        list $rname ~
    }]
    ::rosea::Dispatch::SignalEvent [SelfInstRef] $dstref $event $args

    return
}
```

```
<<error code formats>>=
NO_ASSIGNER         {relationship, "%s", does not have an assigner}
```

Signaling a Multi-Assigner

For associations that have a multi-assigner defined on them, the `signal` command interface must include the set of attributes names and values that identify the instance of the multi-assigner that is to be signaled. So for multi-assigners, the `signal` subcommand is mapped to the `signalMultiAssigner` procedure described here.

relationship `signal` *idvalues event ?param1 param2 . . . ?*

`event`
: The name of an event to be signaled to the assigner associated with *relationship*.

`idvalues`
: A dictionary keyed by attribute names whose values identify an instance of the multiple assigner associated with *relationship*. The attribute names must be the same as those given when the multiple assigner was defined.

`paramN`
: Optional event parameters are passed as additional arguments.

Implementation

We will find it convenient deal with attribute names as sets and will rely on the `struct::set` package from `tcllib` to operate on attribute name sets.

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

```
<<relationship commands>>=
proc signalMultiAssigner {rname idvalues event args} {
    SplitRelvarName $rname domain relationship
    set assignrelvar ${domain}::__${relationship}__STATEINST
    if {![relvar exists $assignrelvar]} {
        tailcall DeclError NO_ASSIGNER $rname
    }
    set idattrs [lindex [relvar identifiers $assignrelvar] 0]
    # We will insist that the idvalues contain the necessary identifying
    # attributes.
    if {![struct::set equal [dict keys $idvalues] $idattrs]} {
        tailcall DeclError NO_IDENTIFIER [dict keys $idvalues] $idattrs
    }
    set assigninsts [relvar set $assignrelvar]
    set assignrefs [relation project $assigninsts {*}$idattrs]

    # We need to find the tuple in "assignrefs" that matches the attributes and
    # values in "idvalues". We will do this by creating a relation value from
    # the idvalues list and semijoining that to "assignrefs". If we come up
    # non-empty, then we have found our multi-assigner instance.
    set heading [relation heading $assignrefs]
    set dstinst [pipe {
        tuple create $heading $idvalues |
        tuple relation ~ |
        relation semijoin ~ $assignrefs
    }]
    if {[relation isempty $dstinst]} {
```

```
        tailcall DeclError UNKNOWN_ASSIGNER $idvalues $rname
    }

    set dstref [list $rname $dstinst]
    ::rosea::Dispatch::SignalEvent [SelfInstRef] $dstref $event $args ; # ❶

    return
}
```

❶ We use the same event dispatch mechanisms as for class instances.

```
<<error code formats>>=
NO_IDENTIFIER       {bad identifying attributes, expected "%s", got "%s"}
UNKNOWN_ASSIGNER    {unknown assigner instance, "%s", for relationship, "%s"}
```

Creating a Multi-Assigner

Since associations can have multi-assigners, we must provide a means to create instances of the assigners.

relationship `create` *?attr1 value1 attr2 value2 . . . ?*

`attrN valueN`
Attribute name / value pairs that form the identifier for an instance of the multi-assigner associated with *relationship*.

Implementation

```
<<relationship commands>>=
proc createMultiAssigner {rname args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }
    SplitRelvarName $rname domain relationship
    set assignrelvar ${domain}::__${relationship}__STATEINST
    if {![relvar exists $assignrelvar]} {
        tailcall DeclError NO_ASSIGNER $rname
    }
    set idattrs [lindex [relvar identifiers $assignrelvar] 0]
    if {![struct::set equal [dict keys $args] $idattrs]} {
        tailcall DeclError NO_IDENTIFIER [dict keys $args] $idattrs
    }
    CreateInInitialState $domain $relationship $args
}
```

Instance Commands

In this section we discuss the `rosea` commands that operate on instance references. These commands form the foundation of operations that state activities use to access the underlying execution architecture.

As shown above, the instance commands are placed in their own namespace. This make it easier to provide unqualified access to these commands via the `namespace path` settings.

```
<<instance commands namespace>>=
namespace eval InstCmds {
    <<tclral imports>>
    namespace path [namespace parent]::Helpers
    <<instance data>>
    <<instance commands>>
}
```

Finding Related Instances

Navigating the class diagram to find related instances is a fundamental operation in most state activities. Relationship navigation is mapped to a series of relational `semijoin` operations. To support the run-time processing, we require a set of data structures that facilitate invoking a series of relational `semijoin` operations. The class diagram for these data structures is shown below.

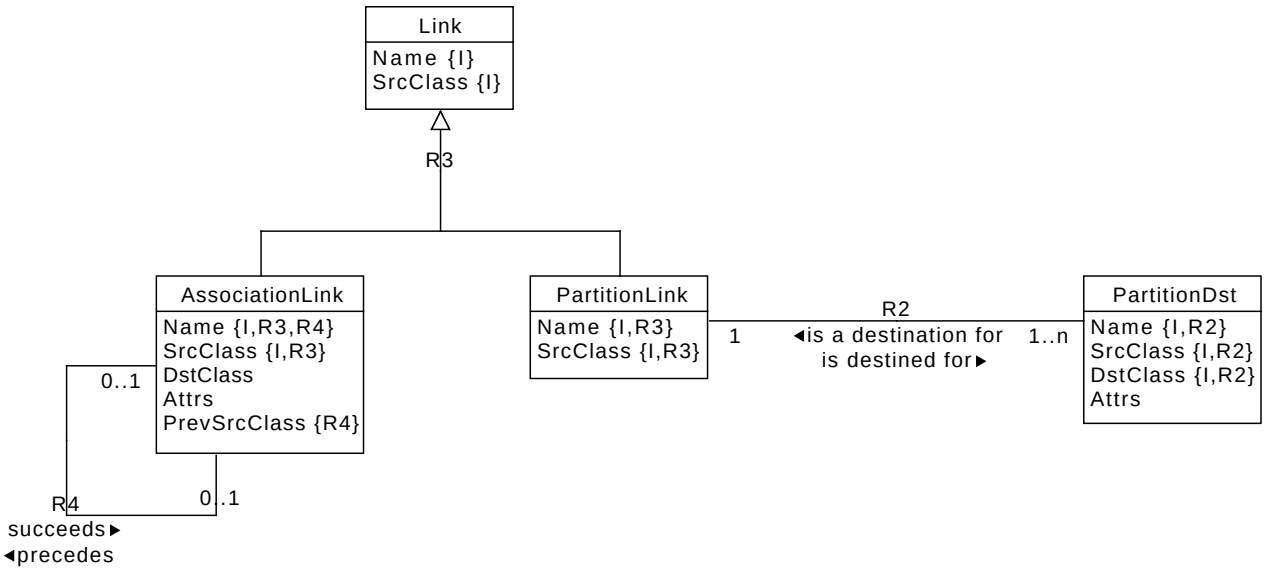

Figure 4.4: Link Information Class Diagram

A **Link** can be of two types (**R3**). An **AssociationLink** corresponds to the linkage of an association type relationship and a **Partition** link corresponds to a generalization relationship. For class based associations, it is necessary to traverse to both the associator class and then to the other participating class (**R4**) To traverse from a superclass to a subclass we must know what are the possible destination subclasses and the attributes needed to locate a subclass instance (**R2**).

The transliteration of the graphic into TclRAL relvars and constraints follows the usual pattern.

```
<<arch relvar definitions>>=
ral relvar create __Arch_Link {
    Name            string
    SrcClass        string
} {Name SrcClass}
ral relvar create __Arch_AssociationLink {
    Name            string
    SrcClass        string
    DstClass        string
    Attrs           list
    PrevSrcClass    string
```

```
} {Name SrcClass}
ral relvar association __Arch_R4\
    __Arch_AssociationLink {Name PrevSrcClass} ?\
    __Arch_AssociationLink {Name SrcClass} ?
ral relvar create __Arch_PartitionLink {
    Name            string
    SrcClass        string
} {Name SrcClass}
ral relvar partition __Arch_R3 __Arch_Link {Name SrcClass}\
    __Arch_AssociationLink {Name SrcClass}\
    __Arch_PartitionLink {Name SrcClass}
ral relvar create __Arch_PartitionDst {
    Name            string
    SrcClass        string
    DstClass        string
    Attrs           list
} {Name SrcClass DstClass}
ral relvar association __Arch_R2\
    __Arch_PartitionDst {Name SrcClass} +\
    __Arch_PartitionLink {Name SrcClass} 1
```

There are two features of navigating that we provide to make translation easier.

- Often several relationships are traversed to get to the desired destination. We provide the ability to specify a relationship chain to traverse multiple relationships in a single invocation. This interface saves the nesting that would be otherwise required if only a single relationship could be traversed in one invocation.
- Although much less common, it is sometimes useful to find the set of *unrelated* instances. We provide that functionality since obtaining the unrelated instance set is otherwise tedious and TclRAL provides the necessary mechanism in the `semiminus` command.

We factor all the common processing for relationship navigation into a procedure below and then provide specific procedures intended to be invoked by the model processing code.

The strategy for navigating relationships is to iterate across the relationship navigation chain querying the linkage relvars to determine the type of linkage and formulating TclRAL `relation semijoin` (or `relation semiminus` if we are seeking unrelated instances).

```
<<instance commands>>=
proc FindRelatedInsts {instref op args} {
    <<FindRelatedInst: set up data access>>
    # We start by dereferencing the instance reference.
    set related [deRef $instref]
    foreach linkage $args {
        <<FindRelatedInst: traverse linkage>>
    }
    # N.B. that we are not returning an instance reference but rather the
    # complete relation value of the related instances. This allows us
    # to further filter the result in the "findRelatedWhere" command.
    return [list $srcrelvar $related]
}
```

```
<<FindRelatedInst: set up data access>>=
set srcrelvar [lindex $instref 0]
SplitRelvarName $srcrelvar domain class

# This procedure queries the architectural data about relationship linkage,
# so we bring the relvar variables into scope.
namespace upvar ${domain}\
        __Arch_Link Link\
        __Arch_AssociationLink AssociationLink\
```

```
        __Arch_PartitionLink PartitionLink\
        __Arch_PartitionDst PartitionDst
variable relatedQuery ; # ❶
```

❶ The query to obtain the related instances is used many places and so worth factoring out.

The query below shows the relational algebraic equivalent to traversing a relationship. The key here is that we want to specify the attributes across which the `semijoin` (or `semiminus`) operation occurs. In TclRAL, you may specify the attribute names for the join directly rather than having to go through an additional renaming operation. Since the referential attributes and the corresponding identifying attributes may not be named the same, the run-time linkage information stores them in a directly usable form.

```
<<instance data>>=
pipe {
    relvar set ${domain}::$DstClass |
    relation $op $related ~ -using $Attrs
} relatedQuery
```

By placing the result of the `pipe` command into a namespace variable (`relatedQuery` in this case), we save the work of rewriting the pipe syntax into a Tcl nested command each time it is invoked. We will use this technique in many places. Also note that it is in this query where the `op` argument is substituted and this allows us to select `semijoin` to query for the related instances or `semiminus` to query for the unrelated ones.

The linkage specification may contain a destination class for those types of linkages where there may be multiple paths along the relationship (i.e. navigating to associative classes or subclasses in a generalization).

```
<<FindRelatedInst: traverse linkage>>=
lassign $linkage lname dst ; # split out the linkage spec

set link [relvar restrictone ${domain}::__Arch_Link\
        Name $lname SrcClass $class]
if {[ral relation isempty $link]} {
    tailcall DeclError UNKNOWN_LINKAGE $lname $class
}
# First we determine if we are dealing with an associative link or a
# partition link.
set assoc [relation join $link $AssociationLink]
if {[relation isnotempty $assoc]} {
    <<FindRelatedInst: traverse association>>
} else {
    <<FindRelatedInst: traverse to subclass>>
}
# Continue the iteration by setting the next source to the current
# destination class.
set class $DstClass
set srcrelvar ${domain}::$class
```

If this is a navigation across an associative relationship, then the following cases arise.

- A simple association where we need only join across one set of referential attributes.
- A class based association where we will need to perform two joins, first to the associator class and then to the other participant.
- A class based association where we want to stop the traversal at the associator class.

The first two cases are distinguished by whether the instances of **AssociationLink** are related across **R4** . The third case is determined if the linkages spec included a path to a destination class.

```
<<FindRelatedInst: traverse association>>=
relation assign $assoc DstClass Attrs
# Check if additional path information was provided.
if {$dst eq {}} {
    # Find the set of instances related to the current set.
    set related [eval $relatedQuery]
    # We must see if this is a class-based association.  If so, then the first
    # traversal was to the associator and we must traverse again to the other
    # class.
    set assoc [relation semijoin $assoc $AssociationLink\
        -using {Name Name SrcClass PrevSrcClass}]
    if {[relation isnotempty $assoc]} {
        relation assign $assoc DstClass Attrs
        set related [eval $relatedQuery]
    }
} else {
    # If a destination was specified in the linkage, then we need to verify
    # that it is actually part of the relationship.
    if {$dst ne $DstClass} {
        tailcall DeclError PATH_ERROR $lname $class $DstClass\
            $dst
    }
    set related [eval $relatedQuery]
}
```

If the linkage is not associative, then it must be a superclass to subclass traversal. In this case, we simply find the partition linkage information. A subclass to superclass traversal is treated as just an ordinary associative linkage and would be handled by the associative traversal code.

```
<<FindRelatedInst: traverse to subclass>>=
set partdst [relvar restrictone ${domain}::__Arch_PartitionDst\
    Name $lname SrcClass $class DstClass $dst]
if {[relation isempty $partdst]} {
    tailcall DeclError PATH_ERROR $lname $class $DstClass $dst
}
relation assign $partdst DstClass Attrs
set related [eval $relatedQuery]
```

```
<<error code formats>>=
UNKNOWN_LINKAGE {unknown relationship, "%s", for class, "%s"}
PATH_ERROR      {relationship, "%s", from "%s" to "%s", does not end at "%s"}
```

The set of commands intended for state activities are just variations on the invocation of `FindRelatedInsts`.

`findRelated` *instref relationship1 relationship2 ...*

instref
: An instance reference that is the start of the relationship chain navigation.

relationshipN
: A specifier indicating the relationship path to traverse. Each relationship specifier is a one or two element list. If a single element, then it is the name of a relationship (*e.g.* R27) that is to be traversed in the forward direction or the name of a relationship prefixed by a tilde (*e.g.* ~R27) giving the name of a relationship to be traversed in its reverse direction. If the relationship specifier is a two element list, then the traversal has alternate paths and the second element names the class that is to be the destination of the traversal. For class based associations this would be the name of the associator class and for generalizations this would be the name of one of the subclasses.

The `findRelated` command returns an instance reference to the set of instances that are found along the relationship navigation chain.

Implementation

```
<<instance commands>>=
proc findRelated {instref args} {
    tailcall ToRef {*}[FindRelatedInsts $instref semijoin {*}$args] ; # ❶
}
```

❶ All the heavy lifting is done by `FindRelatedInsts`. Here we need only specify that we want to use `semijoin` which finds the related instances and then create an instance reference from the returned result.

The procedure for finding the unrelated instances has the same interface.

`findUnrelated` *instref relationship1 relationship2 ...*

`instref`
An instance reference that is the start of the relationship chain navigation.

`relationshipN`
A specifier indicating the relationship path to traverse. Each relationship specifier is a one or two element list. If a single element, then it is the name of a relationship (*e.g.* R27) that is to be traversed in the forward direction or the name of a relationship prefixed by a tilde (*e.g.* ~R27) giving the name of a relationship to be traversed in its reverse direction. If the relationship specifier is a two element list, then the traversal has alternate paths and the second element names the class that is to be the destination of the traversal. For class based associations this would be the name of the associator class and for generalizations this would be the name of one of the subclasses.

The `findUnrelated` command returns an instance reference to the set of instances that are **not** related to those instances referenced by `instref` along the relationship navigation chain.

Implementation

```
<<instance commands>>=
proc findUnrelated {instref args} {
    tailcall ToRef {*}[FindRelatedInsts $instref semiminus {*}$args] ; # ❶
}
```

❶ This is just like `findRelated` except that we now use the `semiminus` operation to find those instances that are *not* related.

It is also useful to filter further the set of related instances by selecting those that satisfy some expression. These variations are described below.

`findRelatedWhere` *instref rchain expression*

instref
An instance reference that is the start of the relationship chain navigation.

rchain
A list of relationship specifiers that form a navigation chain. Each element of the list is specified as for the `findRelated` command.

expression
A expression suitable for the `::expr` command. Each instance found at the end of the navigation chain is evaluated against `expression` and if the expression evaluates to true, then the instance is included in the result. During the evaluation of the `expression` the values of each attribute are assigned to Tcl variables that are the same name as the attribute. These values may be referenced in the `expression` and provide a means of selecting instances based on their attribute values.

The `findRelatedWhere` command returns an instance reference to the set of instances that are found along the relationship navigation chain and for which *expression* evaluates to true.

Implementation

```
<<instance commands>>=
proc findRelatedWhere {instref rchain expr} {
    lassign [FindRelatedInsts $instref semijoin {*}$rchain] relvar insts
    tailcall ToRef $relvar [uplevel 1\
        [list ::ral relation restrictwith $insts $expr]] ; # ❶
}
```

❶ The only new wrinkle here is that we want to further restrict the returned relation value by evaluating `expr`. Note that we perform the `relation restrictwith` in the stack frame of the caller to insure proper resolution of any variables in `expr`.

`findUnrelatedWhere` *instref rchain expression*

instref
An instance reference that is the start of the relationship chain navigation.

rchain
A list of relationship specifiers that form a navigation chain. Each element of the list is specified as for the `findRelated` command.

expression
A expression suitable for the `::expr` command. Each instance found at the end of the navigation chain is evaluated against `expression` and if the expression evaluates to true, then the instance is included in the result. During the evaluation of the `expression` the values of each attribute are assigned to Tcl variables that are the same name as the attribute. These values may be referenced in the `expression` and provide a means of selecting instances based on their attribute values.

The `findUnrelatedWhere` command returns an instance reference to the set of instances that are *not* related to those referenced by `instref` along the relationship navigation chain and for which *expression* evaluates to true.

Implementation

```
<<instance commands>>=
proc findUnrelatedWhere {instref rchain expr} {
    lassign [FindRelatedInsts $instref semiminus {*}$rchain] relvar insts
    tailcall ToRef $relvar [uplevel 1\
        [list ::ral relation restrictwith $insts $expr]]
}
```

Tests

```
<<instance command tests>>=
test findRelated-1.0 {
    Navigate a relationship -- simple association
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    addClass foo c2 {B1 int B2 string} B1
    addAssocLink foo {Name R1 SrcClass c1}\
        {Name R1 SrcClass c1 DstClass c2 Attrs {A1 B1} PrevSrcClass {}}
    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::ClassCmds::create [namespace current]::foo::c2 B1 5 B2 bar
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref R1
} -result {::rosea::test::foo::c2 {{B1 int} {{B1 5}}}} -match ref
```

```
<<instance command tests>>=
test findRelated-1.1 {
    Navigate a relationship -- multiple simple association
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    addClass foo c2 {B1 int B2 string} B1
    addClass foo c3 {C1 int C2 string} C1

    addAssocLink foo {Name R1 SrcClass c1}\
        {Name R1 SrcClass c1 DstClass c2 Attrs {A1 B1} PrevSrcClass {}}
    addAssocLink foo {Name R2 SrcClass c2}\
        {Name R2 SrcClass c2 DstClass c3 Attrs {B1 C1} PrevSrcClass {}}

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::ClassCmds::create [namespace current]::foo::c2 B1 5 B2 bar
    rosea::ClassCmds::create [namespace current]::foo::c3 C1 5 C2 bar
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref R1 R2
} -result {::rosea::test::foo::c3 {{C1 int} {{C1 5}}}} -match ref
```

```
<<instance command tests>>=
test findRelated-1.2 {
    Navigate a relationship -- unknown link
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
} -cleanup {
```

```
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref R1
} -result {unknown relationship, "R1", for class, "c1"}\
-returnCodes error
```

```
<<instance command tests>>=
test findRelated-1.3 {
    Navigate a relationship -- bad path
} -setup {
    setupDomain foo
    addClass foo X {A1 int A2 string} A1
    addClass foo Y {B1 int B2 string} B1
    addClass foo A {A1 int B1 int} {A1 B1}
    addClass foo c1 {A1 int A2 string} A1

    # Forward direction is from X -> Y via A
    addAssocLink foo {Name R1 SrcClass X}\
        {Name R1 SrcClass X DstClass A Attrs {A1 A1} PrevSrcClass {}}
    addAssocLink foo {Name R1 SrcClass A}\
        {Name R1 SrcClass A DstClass Y Attrs {B1 B1} PrevSrcClass X}
    addAssocLink foo {Name ~R1 SrcClass Y}\
        {Name ~R1 SrcClass Y DstClass A Attrs {B1 B1} PrevSrcClass {}}
    addAssocLink foo {Name ~R1 SrcClass A}\
        {Name ~R1 SrcClass A DstClass X Attrs {A1 A1} PrevSrcClass Y}
    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref {R1 A}
} -result {unknown relationship, "R1", for class, "c1"}\
    -returnCodes error
```

```
<<instance command tests>>=
test findRelated-2.0 {
    Navigate a relationship -- associative class
} -setup {
    setupDomain foo
    addClass foo X {A1 int A2 string} A1
    addClass foo Y {B1 int B2 string} B1
    addClass foo A {A1 int B1 int} {A1 B1}
    # Forward direction is from X -> Y via A
    addAssocLink foo {Name R1 SrcClass X}\
        {Name R1 SrcClass X DstClass A Attrs {A1 A1} PrevSrcClass {}}
    addAssocLink foo {Name R1 SrcClass A}\
        {Name R1 SrcClass A DstClass Y Attrs {B1 B1} PrevSrcClass X}
    addAssocLink foo {Name ~R1 SrcClass Y}\
        {Name ~R1 SrcClass Y DstClass A Attrs {B1 B1} PrevSrcClass {}}
    addAssocLink foo {Name ~R1 SrcClass A}\
        {Name ~R1 SrcClass A DstClass X Attrs {A1 A1} PrevSrcClass Y}

    set ref [rosea::ClassCmds::create [namespace current]::foo::X A1 5 A2 baz]
    rosea::ClassCmds::create [namespace current]::foo::Y B1 10 B2 bar
    rosea::ClassCmds::create [namespace current]::foo::A A1 5 B1 10
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref R1
} -result {::rosea::test::foo::Y {{B1 int} {{B1 10}}}} -match ref
```

```
<<instance command tests>>=
```

```
test findRelated-2.1 {
    Navigate a relationship -- path to associative class
} -setup {
    setupDomain foo
    addClass foo X {A1 int A2 string} A1
    addClass foo Y {B1 int B2 string} B1
    addClass foo A {A1 int B1 int} {A1 B1}
    # Forward direction is from X -> Y via A
    addAssocLink foo {Name R1 SrcClass X}\
        {Name R1 SrcClass X DstClass A Attrs {A1 A1} PrevSrcClass {}}
    addAssocLink foo {Name R1 SrcClass A}\
        {Name R1 SrcClass A DstClass Y Attrs {B1 B1} PrevSrcClass X}
    addAssocLink foo {Name ~R1 SrcClass Y}\
        {Name ~R1 SrcClass Y DstClass A Attrs {B1 B1} PrevSrcClass {}}
    addAssocLink foo {Name ~R1 SrcClass A}\
        {Name ~R1 SrcClass A DstClass X Attrs {A1 A1} PrevSrcClass Y}

    set ref [rosea::ClassCmds::create [namespace current]::foo::X A1 5 A2 baz]
    rosea::ClassCmds::create [namespace current]::foo::Y B1 10 B2 bar
    rosea::ClassCmds::create [namespace current]::foo::A A1 5 B1 10
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref {R1 A}
} -result {::rosea::test::foo::A {{A1 int B1 int} {{A1 5 B1 10}}}} -match ref
```

```
<<instance command tests>>=
test findRelated-3.0 {
    Navigate a generalization
} -setup {
    setupDomain foo
    addClass foo S {S1 int S2 string} S1
    addClass foo X {X1 int X2 string} X1
    addClass foo Y {Y1 int Y2 string} Y1

    addAssocLink foo {Name R1 SrcClass X}\
        {Name R1 SrcClass X DstClass S Attrs {X1 S1} PrevSrcClass {}}
    addAssocLink foo {Name R1 SrcClass Y}\
        {Name R1 SrcClass Y DstClass S Attrs {Y1 S1} PrevSrcClass {}}
    relvar eval {
        addParitionLink foo {Name ~R1 SrcClass S}
        addPartitionDst foo {Name ~R1 SrcClass S DstClass X Attrs {S1 X1}}
        addPartitionDst foo {Name ~R1 SrcClass S DstClass Y Attrs {S1 Y1}}
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::S S1 5 S2 baz]
    rosea::ClassCmds::create [namespace current]::foo::X X1 5 X2 bar
} -cleanup {
    cleanupDomain foo
} -body {
    ::rosea::InstCmds::findRelated $ref {~R1 X}
} -result {::rosea::test::foo::X {{X1 int} {{X1 5}}}} -match ref
```

Updating Attributes

Another fundamental data operation is to update the values of attributes. The `updateAttribute` command is used to modify the value of one or more attributes.

`updateAttribute` *instref ?attr1 value1 attr2 value2?*

instref
An instance reference to those instances that are to be updated.

attrN valueN
The *attrN valueN* arguments are a set of attribute name / value pairs. The attributes given will be updated to the corresponding value. If `instref` refers to multiple instances, then the attributes of all referenced instances are updated.

Implementation

```
<<instance commands>>=
proc updateAttribute {instref args} {
    <<updateAttribute: parse arguments>>
    <<updateAttribute: check for identifiers>>
    <<updateAttribute: extend instance reference>>
    <<updateAttribute: update class relvar>>
    return
}
```

We insist that the arguments be given as attribute name / value pairs and the instance reference be singular.

```
<<updateAttribute: parse arguments>>=
if {[llength $args] % 2 != 0} {
    tailcall DeclError ARG_ERROR $args
}

lassign $instref relvar insts
if {[relation cardinality $insts] != 1} {
    tailcall MUST_BE_SINGULAR $relvar [relation cardinality $insts]
}
```

One complication of `updateAttribute` is to disallow updates to identifying attributes. Changes to identifiers must be accomplished by creating and deleting instances. So we want to make sure that none of the attributes listed in the arguments are actually identifying attributes.

We obtain the set of identifying attributes by performing the union of the attributes of all the identifiers of the relvar. Then if the intersection of the identifying attributes with the attributes to be update is non-empty, we disallow the update. In theory, one could continue on and simply exclude the identifying attributes from the update set, but this is deemed an error especially since there is no convenient way to return to the caller that all the attributes might not have been updated. Analysis models should not be trying to update identifying attributes.

```
<<updateAttribute: check for identifiers>>=
set idattrs [list]
foreach identifier [relvar identifiers $relvar] {
    ::struct::set add idattrs $identifier
}
set idupdates [::struct::set intersect $idattrs $args]
if {![::struct::set empty $idupdates]} {
    tailcall DeclError ID_UPDATE [join $idupdates {, }]
}
```

The update strategy is to use the `relvar updateper` command. This command was tailored for this particular purpose. It takes a relation value, uses the identifiers in that value and will update the corresponding non-identifying attributes. So we can create a relation value to contain the updated attribute values by extending the instance reference value with the attributes to be updated. Conveniently, the instance reference already contains the identifying attributes. To perform the extend operation we

will need the type of the attribute which we can get from the heading of the relvar. Since we can have an arbitrary number of attributes to update and since the `relation extend` command will add an arbitrary number of attributes, we iterate over the attributes to build up a `relation extend` command.

```
<<updateAttribute: extend instance reference>>=
set extcmd [list relation extend $insts exttuple]
set heading [relation heading [relvar set $relvar]]
foreach {attr value} $args {
    if {![dict exists $heading $attr]} {
        tailcall DeclError UNKNOWN_ATTRIBUTE $attr
    }
    lappend extcmd $attr [dict get $heading $attr] \"$value\"
}
```

With the `relvar extend` command synthesized, it is just a matter of evaluating it and then using `relvar updateper` to do the work of modifying the relvar in place.

```
<<updateAttribute: update class relvar>>=
relvar updateper $relvar [eval $extcmd]
```

```
<<error code formats>>=
ARG_ERROR           {attribute updates must be name / value pairs, got "%s"}
ID_UPDATE           {cannot update identifying attributes, "%s"}
UNKNOWN_ATTRIBUTE   {unknown attribute, "%s"}
```

Tests

```
<<instance command tests>>=
test updateAttribute-1.0 {
    Update single attribute value
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::updateAttribute $ref A2 foo
    relation extract [::rosea::InstCmds::deRef $ref] A2
} -result {foo}
```

```
<<instance command tests>>=
test updateAttribute-1.1 {
    Update multiple attribute values
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string A3 int} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1\
        A1 5 A2 baz A3 27]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::updateAttribute $ref A3 42 A2 foo
    relation extract [::rosea::InstCmds::deRef $ref] A2 A3
} -result {foo 42}
```

```
<<instance command tests>>=
test updateAttribute-1.2 {
    Update identifying attributes
```

```
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::updateAttribute $ref A1 27
} -result {cannot update identifying attributes, "A1"} -returnCodes error
```

```
<<instance command tests>>=
test updateAttribute-1.3 {
    Update non-existent attributes
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::updateAttribute $ref C3 27
} -result {unknown attribute, "C3"} -returnCodes error
```

Modifying Attributes

It is frequently the case that an update to an attribute depends upon the current value of the attribute. The simple example of incrementing an integer values attribute by one involves reading the current value, performing the addition and updating the new value. To support this operation conveniently, we will supply a control structure type of command patterned off of the `dict with` command.

`withAttribute` *instref attrvarpair1 ?attrvarpair2 . . . ? body*

`instref`
: An instance reference to the instances whose attributes are to be updated. This instance reference must be a singular reference.

`attrvarpairN`
: Each `attrvarpairN` argument is interpreted as a two element list. The first element is the name of an attribute and the second element is the name of a variable into which the attribute value is assigned. If the second element is missing, then the attribute value is assigned into a Tcl variable that is named the same as the attribute. Attributes may not be identifying attributes.

`body`
: A Tcl script that is executed.

The `withAttribute` command assigns the values of the attributes given by the `attrvarpairN` arguments into Tcl variables and evaluates `body`. After the evaluation of `body` the values of the attributes are updated. Updates to identifying attributes are not allowed. Any variable holding an attribute that is unset by `body` is silently ignored and not updated.

Implementation

The implementation of `withAttribute` follows the same pattern as that for `updateAttribute`.

```
<<instance commands>>=
proc withAttribute {instref args} {
    <<withAttribute: parse arguments>>
```

```
    <<withAttribute: check for identifiers>>
    <<withAttribute: execute body>>
    <<withAttribute: extend instance reference>>
    <<withAttribute: update class relvar>>

    return
}
```

The interface has the complication of a variable number of attribute name / variable name pairs since we want to be able to specify the evaluated body as the last argument (as is typical in these control oriented procedures).

```
<<withAttribute: parse arguments>>=
if {[llength $args] < 2} {
    tailcall DeclError WITH_ATTR_USAGE
}

lassign $instref relvar insts
if {[relation cardinality $insts] != 1} {
    tailcall MUST_BE_SINGULAR $relvar [relation cardinality $insts]
}

set body [lindex $args end]
set attrspecs [lrange $args 0 end-1]
set attrnames [list]
set varnames [list]
foreach attrspec $attrspecs {
    set speclen [llength $attrspec]
    if {$speclen == 1} {
        lappend attrnames [lindex $attrspec 0]
        lappend varnames [lindex $attrspec 0]
    } elseif {$speclen == 2} {
        lappend attrnames [lindex $attrspec 0]
        lappend varnames [lindex $attrspec 1]
    } else {
        tailcall DeclError ATTR_VAR_SPEC $attrspec
    }
}
```

```
<<error code formats>>=
WITH_ATTR_USAGE {wrong arguments, should be,\
        "withAttribute attrvarpair1 ?attrvarpair2 ...? body"}
ATTR_VAR_SPEC   {attribute / variable argument must be a one or two element\
        list, got "%s"}
```

As with `updateAttribute`, identifiers may not be modified.

```
<<withAttribute: check for identifiers>>=
set idattrs [list]
foreach identifier [relvar identifiers $relvar] {
    ::struct::set add idattrs $identifier
}
set idupdates [::struct::set intersect $idattrs $attrnames]
if {![::struct::set empty $idupdates]} {
    tailcall DeclError ID_UPDATE [join $idupdates {, }]
}
```

Assign the attributes to variables and evaluate *body* in the context of the caller.

```
<<withAttribute: execute body>>=
uplevel 1 [list ral relation assign [deRef $instref] {*}$attrspecs]
uplevel 1 $body
```

As with `updateAttribute`, we will use the `relvar updateper` command. The only complication here is to make sure that the variables to which the attributes were assigned still exist.

```
<<withAttribute: extend instance reference>>=
set extcmd [list relation extend $insts exttuple]
set heading [relation heading [relvar set $relvar]]
foreach attr $attrnames var $varnames {
    if {[uplevel 1 [list info exists $var]]} {
        upvar 1 $var varvalue
        lappend extcmd $attr [dict get $heading $attr] \"$varvalue\"
    }
}
```

Finally, we perform the update.

```
<<withAttribute: update class relvar>>=
relvar updateper $relvar [eval $extcmd]
```

Tests

```
<<instance command tests>>=
test withAttribute-1.0 {
    Update single attribute value
} -setup {
    setupDomain bar
    addClass bar c1 {A1 string A2 int} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 baz A2 5]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::withAttribute $ref A2 {
        incr A2
    }
    relation extract [::rosea::InstCmds::deRef $ref] A2
} -result {6}
```

```
<<instance command tests>>=
test withAttribute-1.1 {
    Update multiple attribute values
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string A3 int} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1\
            A1 5 A2 baz A3 27]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::withAttribute $ref {A3 x} {A2 y} {
        set x [expr {$x + 10}]
        set y foo
    }
    relation extract [::rosea::InstCmds::deRef $ref] A2 A3
} -result {foo 37}
```

```
<<instance command tests>>=
test withAttribute-1.2 {
    Unset update variable
} -setup {
    setupDomain bar
```

```
    addClass bar c1 {A1 int A2 string A3 int} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1\
            A1 5 A2 baz A3 27]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::withAttribute $ref {A3 x} {A2 y} {
        set x [expr {$x + 10}]
        unset y
    }
    relation extract [::rosea::InstCmds::deRef $ref] A2 A3
} -result {baz 37}
```

Reading Attributes

The complement to updating attributes is, of course, to read them. Reading attributes is just a matter of dereferencing the instance reference and extracting the requested attributes. We deal with the boundary case of reading no attributes by returning the empty string. If only one attribute is read, then a simple scalar value is returned. If multiple attributes are read, then a list of attribute values is returned.

This command is a convenience procedure for the common case where we want to deal with a single class instance. The `instref` argument to the command must be singular. If you wish to deal with the attributes sets of class instances, the best way is to dereference with the `deRef` command to get a relation value that can then be used with `ral relation` commands. This is the preferred way to perform *set-at-a-time* computations. However, most actions operate on a single instance only making procedures such as this most convenient.

`readAttribute` *instref ?attr1 attr2 ... ?*

`instref`
: An instance reference to the instances whose attribute is to be read. This instance reference must be a singular reference.

`attrN`
: The `attrN` arguments are a set of attribute names. The values of the attributes will be returned. If no `attrN` arguments are given, the return value is the empty string. If only one attribute argument is given, then a simple scalar value is returned. If multiple attribute arguments are given, then a list of values is returned. The order of the returned list corresponds to the order of the *attrN* arguments.

Implementation

```
<<instance commands>>=
proc readAttribute {ref args} {
    # We insist upon a singular reference for reading attributes.  Multiple
    # references can be handled by dereferences and using TclRAL "relation"
    # commands.
    if {![isRefSingular $ref]} {
        tailcall DeclError SINGLE_REF_REQUIRED [refMultiplicity $ref]
    }

    return [expr {[llength $args] == 0 ? {} :\
        [relation extract [deRef $ref {*}$args] {*}$args]}] ; # ❶
}
```

❶ The `relation extract` command is smart enough to deal with returning a scalar value if only one attribute is extracted and a list of values otherwise.

```
<<instance command tests>>=
test readAttribute-1.0 {
    Read single attribute value
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::readAttribute $ref A2
} -result {baz}
```

```
<<instance command tests>>=
test readAttribute-1.1 {
    Read multiple attribute values
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::readAttribute $ref A1 A2
} -result {5 baz}
```

```
<<instance command tests>>=
test readAttribute-1.2 {
    Attempt to read multiple reference
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz
    rosea::ClassCmds::create [namespace current]::bar::c1 A1 7 A2 foo
    set ref [rosea::ClassCmds::findAll [namespace current]::bar::c1]
} -cleanup {
    cleanupDomain bar
} -body {
    rosea::InstCmds::readAttribute $ref A1
} -result {single valued reference required, 2 found} -returnCodes error
```

It is often convenient to “unwrap” the attributes of an instance into Tcl variables. This is usually done to interface into other procedures or Tcl commands that expect scalar arguments. The `assignAttribute` command accomplishes this.

`assignAttribute` *instref ?attrvarpair1 attrvarpair2 . . . ?*

`instref`
: An instance reference to the instances whose attributes are to be assigned. This instance reference must be a singular reference.

`attrvarpairN`
: The remaining arguments are each interpreted as attribute name / variable name pairs. If no additional arguments are given, then all the attributes referenced by `instref` are assigned to Tcl variables whose name is the same as the attribute name. If additional arguments are present, each argument is interpreted as a two element list. The first element is the name of an attribute and the second element is the name of a variable into which the attribute value is assigned. If the second element is missing, then the attribute value is assigned into a Tcl variable that is named the same as the attribute.

Implementation

```
<<instance commands>>=
proc assignAttribute {ref args} {
    # We must insist upon a singular references, since assigning to scalar
    # variables from a relation value doesn't make a lot of sense.
    if {![isRefSingular $ref]} {
        tailcall DeclError SINGLE_REF_REQUIRED [refMultiplicity $ref]
    }

    uplevel 1 [list ral relation assign [deRef $ref] {*}$args] ; # ❶
    return
}
```

❶ The `relation assign` command does all the heavy lifting with respect to assigning attributes into the requested Tcl variables.

```
<<instance command tests>>=
test assignAttribute-1.0 {
    Assign single attribute value into a given Tcl variable
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::assignAttribute $ref {A2 myvar}
    set myvar
} -result {baz}
```

```
<<instance command tests>>=
test assignAttribute-1.1 {
    Assign multiple attribute values
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string} A1
    set ref [rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain bar
} -body {
    ::rosea::InstCmds::assignAttribute $ref A1 {A2 a2var}
    list $A1 $a2var
} -result {5 baz}
```

Deleting Instances

As we say above, creating instances is a class based operation. Deleting them is a simple matter of invoking the `delete` instance operation. However, deleting class instances that are involved in relationships can become quite complicated. It is the responsibility of the model activities to leave the data model coherent by unlinking relationships as necessary before deleting instance. If this is not done, then you will most assuredly encounter a constraint violation error.

`delete ?instref1 instref2 ...?`

`instrefN`
An instance reference to those instances that are to be deleted.

Implementation

```
<<instance commands>>=
proc delete {args} {
    foreach instref $args {
        lassign $instref relvar insts
        relvar minus $relvar [deRef $instref]

        SplitRelvarName $relvar domain class
        set instrelvar ${domain}::__${class}__STATEINST
        if {[relvar exists $instrelvar]} {
            relvar minus $instrelvar [relation semijoin $insts\
                [relvar set $instrelvar]]
        }
    }

    return
}
```

Tests

```
<<instance command tests>>=
test delete-1.1 {
    delete class instances
} -setup {
    setupDomain bar
    addClass bar c1 {A1 int A2 string A3 int} A1
    rosea::ClassCmds::create [namespace current]::bar::c1 A1 5 A2 baz A3 27
    rosea::ClassCmds::create [namespace current]::bar::c1 A1 6 A2 foo A3 37
    rosea::ClassCmds::create [namespace current]::bar::c1 A1 7 A2 bar A3 47
} -cleanup {
    cleanupDomain bar
} -body {
    set before [relation cardinality [relvar set [namespace current]::bar::c1]]
    rosea::InstCmds::delete [rosea::Helpers::ToRef\
        [namespace current]::bar::c1\
        [relvar set [namespace current]::bar::c1]]
    set after [relation cardinality [relvar set [namespace current]::bar::c1]]
    testConditions\
        {$before == 3}\
        {$after == 0}
} -result {1}
```

Signaling Events

Most state activities (maybe all) end up updating attribute values or signaling events or both. Here we discuss the implementation of signaling an event.

For events, we track both the source and the target of the event. However, we don't want to force the caller to provide the source as an argument. What we are truly seeking to know is whether the signal originates from within a state activity and if so from which instance. We can make this determination because state activities are located in a child namespace called, `__Activity`. We can climb up the call stack until we find a procedure executing in the `__Activity` namespace.

Implementation

If we find that we are in executing in the `__Activity` namespace before we reach the global namespace level, we want to deem the first `self` variable value encountered as the source instance. Each state activity has a variable named `self` generated

for it. However, as we climb the call stack we will test to see if any procedures we visit also have a defined `self` variable. The first value of `self` that we find in the climb up the call stack is the one we will take as the source instance. This handles the case where a state activity finds some related instance, invokes an instance operation (which also has a `self` variable) that signals the event. In this case, we want the source of the event to be the related instance not the instance executing the state activity. So the logic is a little more complicated that just taking the value of `self` when we find a procedure executing in the `__Activity` namespace and we will need some state information to keep track of our progress in finding a value for `self`.

```
<<helper commands>>=
proc SelfInstRef {{level 2}} {
    set foundRef false
    set ref [nilInstRef]

    for {set ns [uplevel $level namespace current]} {$ns ne "::"}\
            {set ns [uplevel [incr level] namespace current]} {
        if {!$foundRef} {
            upvar $level self srcself
            if {[info exists srcself]} {
                set ref $srcself
                set foundRef true
            }
        }
        if {[string match {*__Activity*} $ns]} {
            return $ref ; # ❶
        }
    }
    return [nilInstRef]
}
```

❶ One might conclude that it is not possible to reach this statement without `ref` being set. Under most circumstances that is true. Since only state activities reside in the `__Activity` namespace and all state activities have a `self` variable, we would expect `ref` to be set in the first `if` statement. But this is Tcl, and it is possible to *inject* code into the namespace to be executed. Consider, for example, a Tk button command that is created in a state activity and given the command `[namespace code [list instop $self signal MyEvent]]`. This will cause execution in the `__Activity` namespace but no `self` variable will be defined when that execution takes place. So the initialization of `ref` outside of the `for` loop is necessary to protect against this situation. **N.B.** that the correct way to signal events or perform other instance based operations from outside of the domain (*e.g.* in Tk callbacks) is to use the `rosea tunnel` command.

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

`instref`
An instance reference whose instances are to be signaled.

`event`
The name of the event to signal.

`paramN`
The `paramN` arguments are a set of optional event parameter values. These values are delivered to the state activity when it is run. It is a corollary of the rules of Moore type state machines that any event that causes a transition into a state must supply the parametric event data defined by the signature of the state activity. Failing to do so will result in a runtime error when the event is dispatched.

One complication with signaling events is dealing with polymorphic events. Polymorphic events are sent to a superclass instance and, at run time, are mapped to an event in the subclass instance to which the superclass instance is currently related. In general the rules of mapping polymorphic events can be complex, so we factor that into a procedure. Otherwise, the `signal` procedure is simply a means of invoking a dispatch procedure that queues an event and hooks that action into the Tcl event loop.

Implementation

```
<<instance commands>>=
proc signal {dstset event args} {
    set srcref [SelfInstRef]
    if {![::rosea::Dispatch::MapPolymorphicEvent ::rosea::Dispatch::SignalEvent\
            $srcref $dstset $event $args]} {
        ::rosea::Dispatch::SignalEvent $srcref $dstset $event $args
    }
    return
}
```

Signaling Delayed Events

The XUML modeling semantics supply the concept of a delayed event as a means of dealing with time. Delayed events are requests to deliver an event at some time in the future. Time is specified in units of milliseconds.

There is a rule in the XUML semantics that there may be no more than one outstanding delayed event between any sending / receiving pair of instances. We will see below what happens when duplicate delayed events are signaled. For now we will simply state the rule and state that it is enforced.

`delaysignal` *time instref event ?param1 param2 ... ?*

`time`
The number of milliseconds to delay in delivering the event. The time value must be a non-negative integer. Delay times of zero are acceptable and result in the event being signaled immediately.

`instref`
An instance reference whose instances that are to be signaled.

`event`
The name of the event to signal.

`paramN`
The `paramN` arguments are a set of optional event parameter values. These values are delivered to the state activity when it is run.

Implementation

```
<<instance commands>>=
proc delaysignal {time dstset event args} {
    if {![string is integer -strict $time] && $time >= 0)} {
        tailcall DeclError INVALID_TIME $time
    }
    set srcref [SelfInstRef]
    if {![::rosea::Dispatch::MapPolymorphicEvent\
            [list ::rosea::Dispatch::SignalDelayedEvent $time]\
            $srcref $dstset $event $args]} {
        ::rosea::Dispatch::SignalDelayedEvent $time $srcref $dstset $event $args
    }
    return
}
```

```
<<error code formats>>=
INVALID_TIME    {invalid time value, "%s"}
```

Canceling Delayed Events

You may cancel a delayed event that is outstanding. There are three situations that might arise when a delayed event is to be canceled.

- The delay time has not expired and the signal is still pending.
- The delay time has expired but the event has not yet been delivered.
- The event has already been delivered.

In the third case, canceling has no effect as it is not possible to reach backward in time. It is not an error to cancel a delayed event that is not in flight. For the first two cases, canceling the delayed event will insure that it is *not* delivered. That is to say, that even if the event is queued for delivery, it will be found and discarded.

`canceldelayed` *srcref dstref event*

`srcref`
The instance reference of the sender of the delayed event. The `srcref` reference may not be multiple.

`dstref`
The instance reference of the receiver of the delayed event. The `dstref` reference may be multiple.

`event`
The name of the event to cancel.

Implementation

```
<<instance commands>>=
proc canceldelayed {srcref dstset event} {
    set srcmult [refMultiplicity $srcref]
    if {$srcmult > 1} {
        tailcall DeclError SINGLE_OR_EMPTY_REF_REQUIRED $srcmult
    }
    forAllRefs dstref $dstset {
        ::rosea::Dispatch::CancelDelayedSignal $srcref $event $dstref
    }
    return
}
```

```
<<error code formats>>=
SINGLE_OR_EMPTY_REF_REQUIRED {single valued or nil reference required, %d found}
```

Remaining Time for a Delayed Event

The other operation for delayed is events is to inquire as to the amount of time remaining before the event is signaled.

`delayremaining` *srcref dstref event*

srcref
The instance reference of the sender of the delayed event. The `srcref` reference may not be multiple.

dstref
The instance reference of the receiver of the delayed event. The `dstref` reference must be singular.

event
The name of the event to inquire of the remaining time.

The return value of the command is the number of milliseconds remaining before `event` is signaled. The returned value will be zero if the `event` does not exist or has already been signaled.

```
<<instance commands>>=
proc delayremaining {srcref dstref event} {
    set srcmult [refMultiplicity $srcref]
    if {$srcmult > 1} {
        tailcall DeclError SINGLE_OR_EMPTY_REF_REQUIRED $srcmult
    }
    if {![isRefSingular $dstref]} {
        tailcall DeclError SINGLE_REF_REQUIRED [refMultiplicity $dstref]
    }
    tailcall ::rosea::Dispatch::SignalTimeRemaining $srcref $event $dstref
}
```

```
<<error code formats>>=
SINGLE_REF_REQUIRED     {single valued reference required, %d found}
```

Invoking Instance Operations

`instop` *instref operation ?arg1 arg2 . . . ?*

instref
The instance reference of the class instance on which the operation is to be performed.

operation
The name of the operation.

argN
The arguments to the operation.

Implementation

The implementation of `instop` uses the relvar name that is part of the instance reference to invoke the ensemble command for the appropriate class. This amounts to a convenient shortcut to avoid typing the class name when it can be deduced from the instance reference.

```
<<instance commands>>=
proc instop {instref op args} {
    tailcall [lindex $instref 0]::Instance $instref $op {*}$args
}
```

Dispatch Commands

Let's recap where we are with respect to the `rosea` package commands. So far we have seen commands associated with classes, relationships and instances. These commands were intended to be invoked by the domain processing to translate XUML semantics onto the underlying Tcl implementation. These commands primarily deal with access to data or requesting events to be signaled. In this section we will see how execution is sequenced.

Dispatch commands are those commands in the `rosea` execution architecture that queue and dispatch state machine events and provide the mapping between the XUML semantics of state machine execution and the Tcl event loop. Tcl has support in its core commands for event based programming and in this section we show how those facilities are used to implement the Moore type state machines that are part of the XUML execution model. We will use the `::after` command to provide access to the event loop.

The dispatch commands are not intended to be directly invoked by state activities or other domain processing code. These commands are invoked as either part of requesting an event be signaled or as the means of delivering an event to a class instance.

Like the other aspects that support XUML semantics, there are a set of architectural relvars that contain the data needed to support the run-time dispatch of events. The structure of this data is shown below.

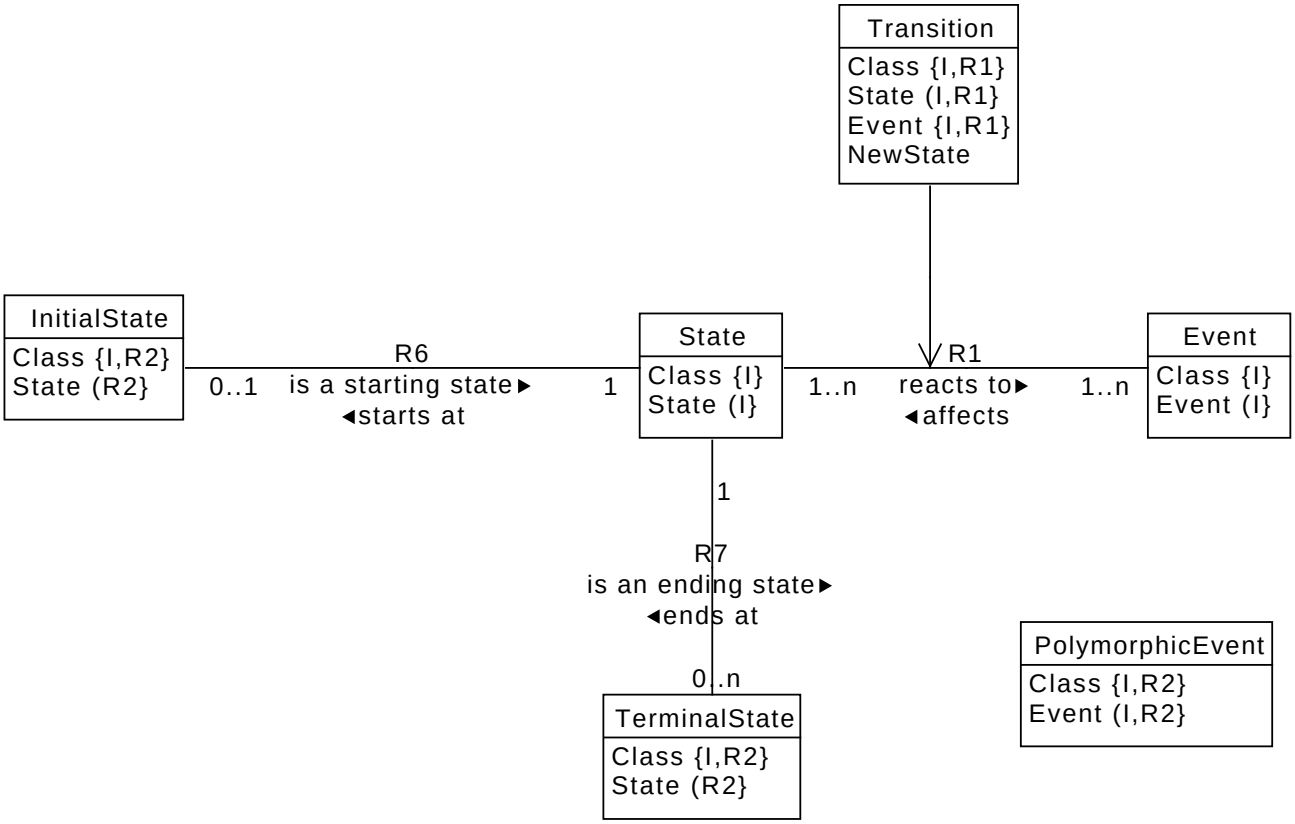

Figure 4.5: Class State Model Architecture Data

There are two distinct concerns regarding event dispatch.

- Ordinary events are dispatched via a **Transition** table.
- Polymorphic events must be mapped at run-time to ordinary events in the currently related subclass.

The **Transition** table here is the Cartesian product of the **State** and **Event** instances (**R1**). Every class has a well defined **InitialState** (**R6**). Some states also serve as a **TerminalState** (**R7**) and the system will delete instances that transition into those states. To map a **PolymorphicEvent**, we need only know its event name.

The class diagram is directly transliterated into TclRAL commands.

```
<<arch relvar definitions>>=
ral relvar create __Arch_State {
    Class string
    State string
} {Class State}
ral relvar create __Arch_Event {
    Class string
    Event string
} {Class Event}
ral relvar create __Arch_Transition {
    Class string
    State string
    Event string
    NewState string
} {Class State Event}
ral relvar correlation __Arch_R1 __Arch_Transition\
    {Class State} + __Arch_State {Class State}\
    {Class Event} + __Arch_Event {Class Event}
ral relvar create __Arch_InitialState {
    Class string
    State string
} Class
ral relvar association __Arch_R6\
    __Arch_InitialState {Class State} ?\
    __Arch_State {Class State} 1
ral relvar create __Arch_TerminalState {
    Class string
    State string
} {Class State}
ral relvar association __Arch_R7\
    __Arch_TerminalState {Class State} *\
    __Arch_State {Class State} 1
```

We place the dispatch commands in their own namespace to avoid any accidental invocation of these commands from state activity code.

```
<<dispatch commands namespace>>=
namespace eval Dispatch {
    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path [list\
        [namespace parent]::Helpers\
        [namespace parent]::InstCmds\
    ]
    <<dispatch data>>
    <<dispatch commands>>
}
```

Dispatching Events

The core part of XUML execution sequencing is rooted in the dispatch of events to state machines and the subsequent execution of the state activities. There are several issues which the design must handle.

- Since we are storing data using TclRAL and we wish to use TclRAL's ability to enforce referential integrity, we must define what constitutes a *transaction* on the domain data model. Transactions are necessary since there will be times when a set of

operations must be executed to have a consistent data model. TclRAL supports the required transaction functionality, it is only necessary to define what constitutes the processing in a transaction.

- Support for delayed events requires support for canceling and querying the remaining time. When mapping XUML event concepts onto Tcl event concepts we will find it necessary to store some additional data to support fully delayed event operations.
- Self directed events, *i.e.* those events that an instance signals to itself must be delivered before non-self directed events directed to the same instance. This is a rule of the XUML execution model.[1]

The XUML execution rules require that a state activity either leave the domain data in a consistent state or generate events that will cause further execution that eventually makes the data model consistent. In `rosea`, we interpret this rule to mean that the data model must be consistent at the end of each *thread of control*. We define a thread of control as beginning with an event that is signaled from *outside* of a state machine activity. We define the end of the thread of control to occur when the state activities initiated by the thread of control starting event have signaled no more events.

Consider an event arising from the invocation of a domain operation. When that event is dispatched, it starts a thread of control. The state activity executed when the event is dispatched will, in general, cause other events to be signaled. These other events are then dispatched and the execution of the state activities may yet signal more events. Eventually, signaling new events stops and all the events generated by state activities is consumed. At that time, the thread of control is over and the data model must be consistent with respect to its relationships. So, the implementation will start a `relvar transaction` at the beginning of the thread of control and end that transaction when the thread of control ends. When the transaction ends, TclRAL will evaluate the `relvar` constraints that are associated with the class relationships. If one or more constraints fail, then the data is rolled back to its state before the transaction began.

There is one other subtle aspect of event dispatch and transactions. We consider dispatching a delayed event as arising from *outside* of a state activity. Although the delayed event may be signaled from inside a state activity, that signaling operation is interpreted as a request to the system to signal the event at some future time. The event ultimately is delivered by the system and so is considered to have arisen from outside of any state activity that made the delayed signaling request. We consider time to be somewhat magical, pervasive and as existing outside of any domain. This interpretation is in effect even if the delay time is zero. A state activity signaling a delayed event with a zero delay time will result, when the event is dispatched, in the start of a new thread of control. This boundary condition case can actually be used to cause a state machine to *yield* the thread of control allowing other processing to occur. Models that rely on such behavior should be considered suspect but the effect can be used as a means of yielding execution from a long running computation to allow other threads of control to be run. Given the single threaded nature of this implementation, that situation is sometimes needed.

To implement these ideas we will use two queues, one to hold events that arise outside of a state machine context and another one to hold those that are signaled by state machine activities.

```
<<dispatch data>>=
variable toc_queue [list]
variable event_queue [list]
```

We implement the queues as simple Tcl lists. The elements of the list are dictionaries holding the event information.

For the `toc_queue` new events that originate outside of a state activity are simply appended to the list and the inherent ordering of the list determines the sequence of execution for the threads of control. For the `event_queue` the inherent order of the events placed in the list must account for self directed events. We want the `event_queue` list order to be the order of event dispatch and this implies that when a self directed event is signaled it will be inserted into the `event_queue` list at the beginning.

SignalEvent

The `SignalEvent` procedure is invoked by the various instance commands that need to cause event generation.

Implementation

We first decide whether the signaled event will start a new thread of control.

[1] The reason for this rule is beyond our scope here but, despite the special case nature of the rule, in the final analysis the rule greatly simplifies reasoning about the analytical model.

- If the event is signaled from outside of a state activity it starts a new thread of control. This is determined by testing the source of the event against the instance reference that represents no instance reference at all.
- Self directed events have to be placed in the event queue at the proper location.
- Non-self directed events are queued at the end of the queue.

```
<<dispatch commands>>=
proc SignalEvent {srcref dstset event arglist} {
    <<SignalEvent: validate event>>

    set callback [namespace code DispatchEvent]
    set eventInfo [dict create\
        type ordinary\
        src $srcref\
        event $event\
        params $arglist\
    ]
    if {[isRefEqual $srcref [nilInstRef]]} {
        variable toc_queue
        forAllRefs dstref $dstset {
            # Queue the thread of control events.
            dict set eventInfo dst $dstref
            lappend toc_queue $eventInfo
            ::after 0 $callback
        }
    } else {
        variable event_queue
        forAllRefs dstref $dstset {
            dict set eventInfo dst $dstref
            if {[isRefEqual $srcref $dstref]} { # ❶
                <<SignalEvent: queue self directed event>>
            } else {
                # Non-self directed events are always simply appended to the
                # event queue.
                lappend event_queue $eventInfo
            }
            ::after 0 $callback
        }
    }
}
```

❶ We deem the event to be self directed if the `srcref` and `dstref` are equal. There is a bit of a strange case here. We would expect that when signaling a self directed event that `dstset` would be a singular reference, probably just the value of `self` in the state activity. It's not clear that signaling the same event to `self` multiple times is a particularly meaningful operation. However, since instance references can be computed in many ways and since there is nothing blatantly wrong with signaling yourself with the same event multiple times, we iterate through the target instance references for self directed events even though we would expect under the vast majority of cases that there would be only one target instance for a self directed event.

Finally, we use the `::after` command to interface to the Tcl event loop which ultimately causes the event to be delivered via the associated callback. The above code insures that each event placed in queue is also accompanied by creating a Tcl event using `::after`. In this way each state machine event is mapped to a Tcl event.

We validate the event to insure sure that the class instances accept the event.

```
<<SignalEvent: validate event>>=
lassign $dstset relvar insts
SplitRelvarName $relvar domain class
if {[relation isempty [relvar restrictone ${domain}::__Arch_Event\
```

```
        Class $class Event $event]]} {
    tailcall DeclError UNKNOWN_EVENT $event $class
}
```

Finding the proper place for a self directed event requires inserting the event at the front of the event queue. However, there is a minor complication. If there is already a self directed event on the event queue, we don't want to place the new one directly on the front. That would invert the order of the self directed event generation. So we search the event queue to find the boundary between self directed and non-self directed events and insert the new event there.

Note there is another design alternative here. We could simply keep the self directed events on their own queue and then the problem of ordering of the self directed events is solved by appending new self directed events to the queue. It is, however, a trade off between another queue and the logic surrounding both selecting the next event to deliver and determining when the thread of control has finished. In this case we have decided upon a single queue, expecting that we will make only a single comparison in the loop below. In the majority of cases, a self directed event will be placed directly on the front of the event queue.

```
<<SignalEvent: queue self directed event>>=
for {set eqindex 0} {$eqindex < [llength $event_queue]} {incr eqindex} {
    set cmpevent [lindex $event_queue $eqindex]
    if {![isRefEqual [dict get $cmpevent src] [dict get $cmpevent dst]]} {
        break
    }
}
set event_queue [linsert [K $event_queue [set event_queue {}]]\
        $eqindex $eventInfo] ; # ❶
```

❶ This code uses the K combinator trick. This is an optimization that exploits a particular design characteristic of Tcl. The net effect is to prevent the sharing of the `event_queue` value and save the copy that would otherwise have to be made when the list value is modified by the list insertion command. See the Tcl wiki for a complete discussion. We will use this optimization in several places where we are dealing with the event queues as they can, in principle, be quite large and dealing with the event queues is a frequent operation.

Testing

```
<<dispatch command tests>>=
test SignalEvent-1.0 {
    Signal an event
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::InstCmds::signal $ref e1
} -cleanup {
    after cancel [namespace eval ::rosea::Dispatch namespace code DispatchEvent]
    cleanupDomain foo
    set ::rosea::Dispatch::toc_queue [list]
} -body {
    set event [lindex $::rosea::Dispatch::toc_queue 0]
    dict get $event dst
} -result {::rosea::test::foo::c1 {{A1 int} {{A1 5}}}} -match ref
```

DispatchEvent

For each invocation of `SignalEvent` we use the `::after` command to have `DispatchEvent` invoked as the callback. The events have been properly ordered on the two queues by `SignalEvent`, so it is the job of `DispatchEvent` to select an event and effect the state machine transition. Layered on the event dispatch is determining the boundaries of the thread of control.

Implementation

Dispatching an event first involves selecting the event. Here we need to make decisions about whether to start a new thread of control. In a Moore type machine, the new state is a function of the current state and the event. Once we have the new state, there is a state activity associated with the new state or the new state is one of the non-transitioning states that causes the event to be ignored or treated as an error.

```
<<dispatch commands>>=
proc DispatchEvent {} {
    <<DispatchEvent: select event>>
    <<DispatchEvent: handle creation events>>
    <<DispatchEvent: get current state>>
    <<DispatchEvent: determine new state>>
    <<DispatchEvent: execute state action>>
    return
}
```

Events are removed from the `event_queue` list first. This queue holds the events generated on the currently running thread of control and it is always our goal to run the current thread of control to completion. Otherwise, we start a new thread of control by removing an event from the `toc_queue` list. If we find both lists empty, then there is nothing to do and we simply return.

```
<<DispatchEvent: select event>>=
variable event_queue
variable toc_queue

if {[llength $event_queue] != 0} {
    set eventInfo [lindex $event_queue 0]
    set event_queue [lrange [K $event_queue [set event_queue {}]] 1 end] ; # ❶
} elseif {[llength $toc_queue] != 0} {
    set eventInfo [lindex $toc_queue 0]
    set toc_queue [lrange [K $toc_queue [set toc_queue {}]] 1 end]
    relvar transaction begin ; # ❷
} else {
    return
}
```

❶ Note the use of the `K` trick here and below.

❷ Each time we start a new thread of control, we begin a transaction.

The event dictionary has a `type` key that determines if the event is just an ordinary event or is a creation event. For a creation event we must actually create an class instance before we dispatch the event to it.

```
<<DispatchEvent: handle creation events>>=
if {[dict get $eventInfo type] eq "creation"} {
    try {
        set dstref [::rosea::ClassCmds::createin\
                [dict get $eventInfo dstClass] @\
                {*}[dict get $eventInfo dstAttrs]] ; # ❶
    } on error {result} {
        catch {
            if {[llength $event_queue] == 0} {
                relvar transaction end
```

```
            }
        }
        tailcall DeclError ASYNC_CREATION_FAILED [dict get $eventInfo dstClass]\
            [dict get $eventInfo dstAttrs] $result
    }
    dict set eventInfo dst $dstref ; # ❷
    ::rosea::Trace::TraceCreation [dict get $eventInfo src]\
        [dict get $eventInfo event] $dstref
}
```

❶ Here is where the new instance is created in the pseudo-initial, `@`, state.

❷ Note that we are just setting the `dst` dictionary key so that now all the members are in place to look just like an ordinary event. The rest of the code simply ignores the dictionary members that it is not concerned with.

```
<<error code formats>>=
ASYNC_CREATION_FAILED     {asynchronous creation of an instance of class, "%s",\
    with attributes, "%s", failed: %s}
```

Each class that has a state model also has an additional relvar that is used to hold the value of the current state for the instances of the class. That relvar is named, `__<class>__STATEINST`, where `<class>` is replaced by the class name. The heading of the current state relvar consists of the attributes of the first identifier of the class plus an attribute named, `__State`, which holds the current state. So obtaining the current state means that we `semijoin` the instance reference to the value of the `__<class>__STATEINST` relvar.

```
<<DispatchEvent: get current state>>=
set dstref [dict get $eventInfo dst]
lassign $dstref relvar ref
SplitRelvarName $relvar domain class

set instrelvar ${domain}::__${class}__STATEINST
set state [relation semijoin $ref [relvar set $instrelvar]]
if {[relation isempty $state]} {
    tailcall DeclError EVENT_IN_FLIGHT [dict get $eventInfo event] $ref
}
set currstate [relation extract $state __State]
```

```
<<error code formats>>=
EVENT_IN_FLIGHT     {event, "%s", sent to "%s", which does not exist}
```

It is possible for an event to be signaled to an instance that has been deleted before the event is actually delivered. This is known as the *event in flight* error and is one of the few run-time errors that is diagnosed by `rosea`. This is considered an analysis error detected at run time. Correct analysis models must insure that no events are in flight to an instance before deleting it. In general, deletion protocols within a domain model can be quite complicated as the model is responsible for all the compensatory operations needed to make the instance population consistent with the relationship constraints.

To obtain the new state, we query the **Transition** relvar that was shown above. When the **Transition** relvar value is set up, it is the Cartesian product of the states and events for the class. So it can be considered the complete transition matrix and the query below will always find a tuple with the new state.

```
<<dispatch data>>=
pipe {
    relvar restrictone ${domain}::__Arch_Transition\
        Class $class State $currstate Event $event |
    relation extract ~ NewState
} transitionQuery
```

Combining the event name and the current state we perform the query to find the new state.

```
<<DispatchEvent: determine new state>>=
set event [dict get $eventInfo event]
variable transitionQuery
set newState [eval $transitionQuery]

::rosea::Trace::TraceTransition [dict get $eventInfo src]\
    $event $dstref $currstate $newState [dict get $eventInfo params] ; # ❶
```

❶ Note that we trace the transition *before* we take any action on it. It is a simpler control path. We will have much more to say about state machine tracing.

Finally, we get to the heart of the event dispatch code. There are three cases to consider:

- The event causes a CH (Can't Happen) transition and that is considered a serious error.
- The event causes an IG (Ignore) transition and we do nothing.
- We transition to the new state and execute the state activity.

In the end, we must decide if the thread of control has ended and if so then the transaction is also ended.

```
<<DispatchEvent: execute state action>>=
try {
    if {$newState eq "CH"} {
        <<DispatchEvent: handle transition error>>
    } elseif {$newState ne "IG"} {
        <<DispatchEvent: transition to new state>>
    }
} on error {result opts} {
    catch {
        if {[llength $event_queue] == 0} {
            relvar transaction end
        }
    } ; # ❶
    return -options $opts $result
}

if {[llength $event_queue] == 0} {
    relvar transaction end
}
```

❶ If there is an error executing a state activity, we may still need to end the thread of control (and hence the `relvar transaction`). In general, the transaction will fail since the state activity did not complete. We don't want the transaction failure to throw another error which would mask the reason the state activity failed in the first place.

The **CH** (can't happen) transition deserves special consideration. You can think of a **CH** transition as an analogy to the `assert` macro in ordinary "C" code. Triggering a **CH** transition is a serious error that should never happen in production code. If it does, usually there is no clear way to proceed and how to handle such situations is very application dependent. Often the only recourse is to exit or reset to some saved checkpoint. However during development and testing, the **CH** transition can uncover a wealth of analysis errors and misunderstandings. Consequently, it is very useful to have some control over what happens should a **CH** transition occur. We delegate the error handling for **CH** transitions to the `TransitionError` [2] procedure.

```
<<DispatchEvent: handle transition error>>=
TransitionError [dict get $eventInfo src] $event $dstref $currstate
```

[2] The fully qualified name is `::rosea::Dispatch::TransitionError`

We supply a default implementation of `TransitionError` which simply throws an error. Application can redefine this procedure to gain control over the error handling.

```
<<dispatch commands>>=
proc TransitionError {src event dst currstate} {
    tailcall DeclError CANT_HAPPEN_EVENT $src $event $dst $currstate
}
```

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

When transitioning to the new state, we update the value of the current state of the instance to be the newly determined state value. The identifiers of the `__<class>__STATEINST` relvar are just the attributes of the instance reference for the instance. Note that the current state is updated to the new state *before* the state activity is executed. Some architectures wait until after the state activity is executed. Which is better or correct is hard to determine and in practice does not matter since state activities are strongly discouraged from having knowledge of their own state.

```
<<DispatchEvent: transition to new state>>=
set idattrs [tuple get [relation tuple $ref]]
relvar updateone $instrelvar is $idattrs {
    tuple update $is __State $newState
}
```

At last we execute the state activity. State activities are ordinary Tcl procedures named the same as the state and placed in the `__Activity` namespace that is a child of the namespace associated to the class. The instance reference of the event target is passed as the value of the `self` parameter and any other parameters carried along with the event are also given to the activity procedure. Here is the explicit reason that the parameters carried with the event must match the arguments that the state activity expects or Tcl will throw an error.

```
<<DispatchEvent: transition to new state>>=
try {
    ${relvar}::__Activity::$newState $dstref\
            {*}[dict get $eventInfo params] ; # ❶
} on error {result opts} {
    return -options $opts $result
} finally {
    set term [relvar restrictone ${domain}::__Arch_TerminalState\
            Class $class State $newState]
    if {[relation isnotempty $term]} {
        relvar deleteone $relvar {*}$idattrs
        relvar deleteone $instrelvar {*}$idattrs
    }
}
```

❶ This command invocation has a lot of variable substitution. Each state activity is actually a procedure. Those procedures are placed in the `__Activity` child namespace of the class so that there is no naming conflict. The value of `dstref` becomes `self` in the state activity. Finally, we expand out the list of event parameters to be any additional arguments to the state activity procedure.

After executing the state activity, we look to see if new state is a terminal state. If the state was marked as terminal when the state model was defined, the the system automatically deletes the class instance. Of course, we must also delete the tuple that records the current state.

Testing

```
<<dispatch command tests>>=
test DispatchEvent-1.0 {
    Dispatch an event and execute a state action
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }
    proc [namespace current]::foo::c1::__Activity::s2 {self} {
        set ::done $self
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::InstCmds::signal $ref e1
} -cleanup {
    cleanupDomain foo
} -body {
    vwait ::done
    set ::done
} -result {::rosea::test::foo::c1 {{A1 int} {A1 5}}} -match ref
```

```
<<dispatch command tests>>=
test DispatchEvent-1.1 {
    Dispatch an event with the wrong arguments
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }
    proc [namespace current]::foo::c1::__Activity::s2 {self} {
    }
} -cleanup {
    cleanupDomain foo
} -body {
    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::InstCmds::signal $ref e1 baz
    lindex [waitForBGError] 1
} -result {WRONGARGS}
```

```
<<dispatch command tests>>=
test DispatchEvent-1.2 {
    Dispatch an event with error in the activity
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
```

```
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }
    proc [namespace current]::foo::c1::__Activity::s2 {self} {
        throw BAD {bad activity}
    }
} -cleanup {
    cleanupDomain foo
} -body {
    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::InstCmds::signal $ref e1
    waitForBGError
} -result {BAD}
```

Dispatching Delayed Events

The mapping of delayed events onto the Tcl event loop again uses the `::after` command. However, there is not enough information available from `::after` to support some of the features of delayed events. Further, callers of delayed event procedures use the source, target and event as identifiers of a delayed event and `::after` uses its own generated identifiers. Consequently, we will maintain a relvar that contains the mapping between delayed event identifying attributes and the `::after` timer id's along with additional data we need to support all the delayed event operations.

```
<<dispatch data>>=
relvar create DelayedSignal {
    RefId           int
    SrcInstRef      list
    Event           string
    DstInstRef      list
    TimerId         string
    Expire          bignum
} RefId {SrcInstRef Event DstInstRef} TimerId
```

The **RefId** attribute is an arbitrary integer identifier that provides an easy handle on to access the delayed event information. We will supply the value from an ordinary variable.

```
<<dispatch data>>=
variable DelayedSignalId 0
```

The **Expire** attributes allows us to keep track of the remaining delay time for an event. We keep that time in units of milliseconds.

We will need some procedures to operate on the **DelayedSignal** relvar. In particular we need to find and delete instances of **DelayedSignal**. There is a complication. The **SrcInstRef** and **DstInstRef** attributes are instance references which are Tcl list types. If TclRAL is left to its own devices, it would compare the string representations of the instance references when performing operations on the relvar that require a comparison for equality. Since part of the instance reference is a relation value we don't want to use string comparison to test for equality. Relation values do not have a canonical string representation. We have defined the necessary procedures to properly compute whether two instance references are equal. So here we define procedures that explicitly use the proper comparison to locate and delete DelayedSignal instances.

```
<<dispatch commands>>=
proc FindDelayedSignal {srcref event dstref} {
    variable DelayedSignal
    return [relation restrictwith $DelayedSignal {
            [isRefEqual $SrcInstRef $srcref] && $Event eq $event &&\
            [isRefEqual $DstInstRef $dstref]}]
}
```

```
<<dispatch commands>>=
proc DeleteDelayedSignal {srcref event dstref} {
```

```
    relvar delete DelayedSignal dsig {
        [isRefEqual [tuple extract $dsig SrcInstRef] $srcref] &&\
        [tuple extract $dsig Event] eq $event &&\
        [isRefEqual [tuple extract $dsig DstInstRef] $dstref]
    }
}
```

SignalDelayedEvent

To signal a delayed event is much the same logic as signaling one that is not delayed. The addition work involves:

- Creating an instance of **DelayedSignal** to record the additional information we need.
- Having `::after` invoke a different callback to handle the event when the delay time has expired.

Here we must also deal with the issue referred to earlier regarding the rules for delayed events. There can be only one outstanding delayed event for any sending / receiving pair of class instances. In this architecture, requests to create duplicate delayed events are interpreted to mean that the outstanding one should be canceled and a new delayed event at the new delay time should be established. This is generally what is convenient.

Implementation

The implementation does validate that the event is known for the class. Like `SignalEvent` we treat the target instance reference as potentially referring to multiple instances and each must have the event signaled to it.

```
<<dispatch commands>>=
proc SignalDelayedEvent {time srcref dstset event arglist} {
    lassign $dstset relvar insts
    SplitRelvarName $relvar domain class
    if {[relation isempty [relvar restrictone ${domain}::__Arch_Event\
            Class $class Event $event]]} {
        tailcall DeclError UNKNOWN_EVENT $event $class
    }

    variable DelayedSignalId
    set eventInfo [dict create\
        id [incr DelayedSignalId]\
        type ordinary\
        src $srcref\
        event $event\
        params $arglist\
    ]
    forAllRefs dstref $dstset {
        dict set eventInfo dst $dstref
        DeleteDelayedSignal $srcref $event $dstref ; # ❶
        relvar insert DelayedSignal [list\
            RefId       $DelayedSignalId\
            SrcInstRef  $srcref\
            Event       $event\
            DstInstRef  $dstref\
            TimerId     [::after $time [list\
                        ::rosea::Dispatch::DispatchDelayedEvent $eventInfo]]\
            Expire      [expr {entier([clock milliseconds]) + entier($time)}]\
        ]
    }
    return
}
```

❶ Note that we only delete the delayed event information here. If the event has already been dispatched to the `toc_queue`, then we will still signal another event here.

DispatchDelayedEvent

Dispatching a delayed event is a simple operation since all delayed events are placed on the `toc_queue`. The only minor complication is that we must delete the instance of **DelayedSignal** that corresponds to the dispatched event. Otherwise, `DispatchEvent` does all the real work.

```
<<dispatch commands>>=
proc DispatchDelayedEvent {eventInfo} {
    relvar deleteone DelayedSignal RefId [dict get $eventInfo id]
    variable toc_queue
    lappend toc_queue $eventInfo
    tailcall DispatchEvent
}
```

Testing

```
<<dispatch command tests>>=
test DispatchDelayedEvent-1.0 {
    Dispatch a delayed event and execute a state action
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }
    proc [namespace current]::foo::c1::__Activity::s2 {self} {
        set ::done $self
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
    rosea::InstCmds::delaysignal 100 $ref e1
} -cleanup {
    cleanupDomain foo
} -body {
    vwait ::done
    set ::done
} -result {::rosea::test::foo::c1 {{A1 int} {{A1 5}}}} -match ref
```

CancelDelayedEvent

To cancel a delayed event, we handle both cases where the delay time has not expired and where it has expired and the event may be queued for dispatch.

Implementation

```
<<dispatch commands>>=
proc CancelDelayedSignal {srcref event dstref} {
    variable DelayedSignal
    set sig [FindDelayedSignal $srcref $event $dstref]
    if {[relation isnotempty $sig]} {
        ::after cancel [relation extract $sig TimerId]
        DeleteDelayedSignal $srcref $event $dstref
    } else {
        # Search the toc_queue to see if the signal delay has already expired
        # and the event is awaiting dispatch.
        variable toc_queue
        for {set i 0} {$i < [llength $toc_queue]} {incr i} {
            set eventInfo [lindex $toc_queue $i]
            dict update eventInfo src evtsrc dst evtdst event evtevent {
                if {[isRefEqual $evtsrc $srcref] &&\
                        [isRefEqual $evtdst $dstref] &&\
                        $evtevent eq $event} {
                    set toc_queue [lreplace\
                            [K $toc_queue [set toc_queue {}]] $i $i]
                    break
                }
            }
        }
    }
}
```

Testing

```
<<dispatch command tests>>=
test CancelDelayedSignal-1.0 {
    Cancel a delayed event
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain foo
} -body {
    rosea::InstCmds::delaysignal 100 $ref e1
    set src [rosea::Helpers::nilInstRef]
    rosea::InstCmds::canceldelayed $src $ref e1
    relation cardinality [rosea::Dispatch::FindDelayedSignal $src e1 $ref]
} -result {0}
```

SignalTimeRemaining

To support determining the remaining delay time for an event, we stored the time at which the event was due to expire into its **DelayedSignal** tuple at the time we created the delayed event. The remaining time computation then just computes the difference

between the expiration time and the current time. There are a few boundary conditions to consider and zero is returned in those cases.

Implementation

```
<<dispatch commands>>=
proc SignalTimeRemaining {srcref event dstref} {
    set sig [FindDelayedSignal $srcref $event $dstref]
    return [expr {[relation isempty $sig] ? 0 :\
        max([relation extract $sig Expire] - [clock milliseconds], 0)}] ; # ❶
}
```

❶ Okay, this is rather dense. If we don't find a matching delayed event, then just return zero under the assumption that it has already been dispatched or never existed in the first place. Otherwise, we want to return the difference between the expiration time and the current time unless that difference is negative. A negative difference means that we were executing when expiration time occurred and we just haven't gotten back to the event loop to dispatch the event. We will soon enough so, just return zero to indicate that the ship has sailed.

Testing

```
<<dispatch command tests>>=
test SignalTimeRemaining-1.0 {
    Obtain remaining time on a delayed signal
} -setup {
    setupDomain foo
    addClass foo c1 {A1 int A2 string} A1
    relvar eval {
        addState foo Class c1 State s1
        addState foo Class c1 State s2
        addEvent foo Class c1 Event e1
        addInitialState foo Class c1 State s1
        addTransition foo Class c1 State s1 Event e1 NewState s2
        addTransition foo Class c1 State s2 Event e1 NewState s1
    }

    set ref [rosea::ClassCmds::create [namespace current]::foo::c1 A1 5 A2 baz]
} -cleanup {
    cleanupDomain foo
} -body {
    rosea::InstCmds::delaysignal 100 $ref e1
    set src [rosea::Helpers::nilInstRef]
    set remain [rosea::InstCmds::delayremaining $src $ref e1]
    rosea::InstCmds::canceldelayed $src $ref e1
    expr {$remain <= 100 && $remain >= 95}
} -result {1}
```

Dispatching Polymorphic Events

We now consider the job of dispatching polymorphic events. Polymorphic events are events directed at the superclass of a generalization that are mapped at run time to an event in the subclass to which the superclass instance is currently related. This is the only type of polymorphism that XUML execution rules support[3].

The rules for polymorphic events can, in general, be rather intricate and complicated. These rules are also the least well specified of the XUML rules, so it is important to state here the rules that `rosea` uses.

[3] In particular note there are no polymorphic operations supported.

- Polymorphic events are directed at superclass instances and are mapped to events in one of the subclasses of the generalization.
- If the superclass of a generalization defines a polymorphic event, then the ultimate leaf subclasses of the generalization hierarchy must have defined state models to consume the polymorphic events. Although, the leaf subclasses may choose to ignore the ordinary events mapped to it, such constructs should be considered as suspect analysis albeit the execution rules operate properly in that case.
- The name of the ordinary event consumed in the leaf subclass instances is the same as the name of the polymorphic event. Using a naming convention to distinguish ordinary events from polymorphic event may be useful but such conventions do not affect the event dispatch.
- Polymorphic events have no effect on the superclass instance. A superclass may have its own state model and react to ordinary events directed at the superclass state model independent of any polymorphic events defined for the state model. If a superclass does have a state model, the names of the events to which it responds must be disjoint from any polymorphic events.
- A class may serve as the superclass for multiple generalization hierarchies, a so called *compound generalization*. For a compound generalization, polymorphic events directed at the superclass will propagate down all the hierarchies for which the class serves as a superclass. This implies that signaling a single polymorphic event may cause multiple events to ultimately be signaled.
- A class may serve as both the subclass of one generalization and the superclass of a different generalization, a so called *repeated specialization*. In this case, polymorphic events defined for the generalization where the class serves as a subclass are inherited by the generalization for which the class serves as a superclass. A class that is subject to repeated specialization may also define it own polymorphic events and the generalization for which it is the superclass will inherit the union of the events that derive from its subclass role and those defined for its superclass role.
- A class may be the subclass of *multiple generalization* hierarchies[4]. In this case, the state model must respond to the union of the events from all the generalization for which the class is a subclass. Note that some care might have to be taken to insure that the polymorphic event names of the multiple hierarchies are distinct.

MapPolymorphicEvent

The `MapPolymorphicEvent` procedure is called by the instance commands that deal with signaling events to determine if the event is indeed polymorphic. It is, then the the event is recursively signaled to the currently related subclass instance. Otherwise, `MapPolymorphicEvent` returns an indication that the event is not polymorphic and the instance signaling commands can then perform the operation of signaling an ordinary event. Ultimately, execution sequencing is only accomplished by delivering ordinary events. Event polymorphism is a convenience and optimization that prevents state activities from performing event mapping themselves.

Implementation

One consideration for the implementation of the `MapPolymorphicEvent` procedure is that we want to use this procedure to handle both immediate and delayed events. The function of mapping the event is the same. The only difference is how the mapped event is handled. So we will pass in a forwarding command prefix as an argument. Once the work is accomplished to determine which subclass instance is to be signaled, the forwarding command will determine if it is a delayed or immediate event.

```
<<dispatch commands>>=
proc MapPolymorphicEvent {frwdcmd srcref dstrefs event arglist} {
    <<MapPolymorphicEvent: check if event is polymorphic>>
    <<MapPolymorphicEvent: find subclasses>>
    <<MapPolymorphicEvent: map event>>

    return true
}
```

[4] XUML rules prohibit multiple generalizations the have a common root class since such arrangements violate the requirement that the subclass instances form a disjoint union.

We use the **PolymorphicEvent** relvar to determine if is the event is polymorphic. If the event is not found here, then we just return and the event will be signaled as an ordinary event.

```
<<arch relvar definitions>>=
ral relvar create __Arch_PolymorphicEvent {
    Class           string
    Event           string
} {Class Event}
```

```
<<MapPolymorphicEvent: check if event is polymorphic>>=
lassign $dstrefs dstrelvar dstinsts
SplitRelvarName $dstrelvar domain dstclass
set polyevent [relvar restrictone ${domain}::__Arch_PolymorphicEvent\
    Class $dstclass Event $event]
if {[relation isempty $polyevent]} {
    return false
}
```

If we indeed have a polymorphic event, we need to determine the generalization hierarchies that are affected. We do that with a query.

```
<<MapPolymorphicEvent: find subclasses>>=
variable polymapQuery
set partitions [eval $polymapQuery]
```

The essential query for dispatching a polymorphic event is to find the set of hierarchies originating at the source relvar.

```
<<dispatch data>>=
pipe {
    relvar set ${domain}::__Arch_PartitionDst |
    relation restrictwith ~ {$SrcClass eq $dstclass} |
    relation group ~ DstLinks DstClass Attrs
} polymapQuery
```

The `polymapQuery` is on the `PartitionDst` relvar which is the same data we used to navigate relationships. This makes sense because mapping a polymorphic event implies that we must navigate the generalization relationships from the superclass to the subclass. The `PartitionDst` relvar contains the information needed to accomplish the generalization traversal.

This query is accomplished by finding all the partition link tuples that originate at the relvar which is the target of the event signaling. The result is grouped so we may consider all the superclass to subclass links for a given hierarchy. We want to do this because we know that a superclass instance will reference exactly one subclass from among all the subclasses in the generalization and it is to that subclass instance that the event will be forwarded.

The heading of the result of this query will be:

Table 4.1: Heading of Polymorphic Event Mapping Query Result

Name	**SrcClass**	**DstLinks**	
string	string	Relation	
		DstClass	**Attrs**
		string	list

Note that the **DstLinks** attribute is relation valued.[5]

The polymorphic event mapping operation consists of three nested iterations. The outside loop iterates over the target superclass instances since we may be signaling several instances in the same call. The next iteration is over all the hierarchies that originate

[5] That is what the `group` operation does.

from the superclass. Typically, this is only one, but it can be several for a compound generalization. Finally, we iterate over all the subclass links.

```
<<MapPolymorphicEvent: map event>>=
relation foreach inst $dstinsts {
    relation foreach partition $partitions {
        relation assign $partition
        relation foreach dstlink $DstLinks {
            relation assign $dstlink
            <<MapPolymorphicEvent: signal subclass>>
        }
    }
}
```

We must traverse the generalization relationship to find the one subclass that is actually linked and forward the event on to that one. Note that we are semijoining to each subclass in the generalization until one is found. This is different from navigating the relationship in the context of the `findRelated` command. There we determined whether the superclass instance is related to a specific subclass instance. Here we are searching for the subclass instance to which the superclass instance is related. Once we know which is the related subclass instance we can formulate an instance reference to it and trace the polymorphic event dispatch. We *forward* the event to the subclass instance by recursively signaling the event. Once forwarding has happened, we can stop looking any further since we *know* that there is exactly one subclass instance linked to any given superclass instance because the partition constraint guarantees that.

```
<<MapPolymorphicEvent: signal subclass>>=
set related [relation semijoin $inst\
    [relvar set ${domain}::$DstClass] -using $Attrs]
if {[relation isnotempty $related]} {
    set dstreference [ToRef ${domain}::$DstClass $related]
    ::rosea::Trace::TracePolymorphic $srcref\
        $event $dstreference ${domain}::SrcClass $Name
    {*}$frwdcmd $srcref $dstreference $event $arglist ; # ❶
    break
}
```

❶ The forward command prefix will either be `SignalEvent` or `SignalDelayedEvent <time>` depending whether we arrived here via the `signal` instance command or the `delaysignal` instance command.

Testing

```
<<test utility procs>>=
proc addPolymorphic {domain args} {
    relvar insert [namespace current]::${domain}::__Arch_PolymorphicEvent\
            $args
}

proc polysignal {srcref dstref event paramlist} {
    puts -nonewline [info level 0]
}
```

```
<<dispatch command tests>>=
test MapPolymorphicEvent-1.0 {
    Map a polymorphic event
} -setup {
    setupDomain bar
    addClass bar S {S1 int S2 string} S1
    addClass bar X {X1 int X2 string} X1
    addClass bar Y {Y1 int Y2 string} Y1
```

```
    addAssocLink bar {Name R1 SrcClass X}\
        {Name R1 SrcClass X DstClass S Attrs {X1 S1} PrevSrcClass {}}
    addAssocLink bar {Name R1 SrcClass Y}\
        {Name R1 SrcClass Y DstClass S Attrs {Y1 S1} PrevSrcClass {}}
    relvar eval {
        addParitionLink bar {Name ~R1 SrcClass S}
        addPartitionDst bar {Name ~R1 SrcClass S DstClass X Attrs {S1 X1}}
        addPartitionDst bar {Name ~R1 SrcClass S DstClass Y Attrs {S1 Y1}}
    }

    addPolymorphic bar Class S Event e1
    set ref [rosea::ClassCmds::create [namespace current]::bar::S S1 5 S2 baz]
    rosea::ClassCmds::create [namespace current]::bar::X X1 5 X2 bar
} -cleanup {
    cleanupDomain bar
} -body {
    rosea::Dispatch::MapPolymorphicEvent [namespace current]::polysignal\
        {} $ref e1 {}
} -result {true} -output {::rosea::test::polysignal {} {::rosea::test::bar::X {{X1 int} {{ ↩
    X1 5}}}} e1 {}}
```

Helper Commands

In this section we present a set of commands that factor common processing used in a number of areas. These commands are placed in a separate namespace which is imported into most of the other package namespaces.

```
<<helper commands namespace>>=
namespace eval Helpers {
    <<tclral imports>>
    namespace import ::ral::relvar
    <<helper data>>
    <<helper commands>>
}
```

DeclError

All error notification in the package is consolidated in the `DeclError` procedure.

Implementation

The `DeclError` procedure locates a format string based on the error code and applies its arguments to it. We use the `::throw` command to raise the error to insure that we have consistent error code information for the package.

```
<<helper commands>>=
namespace export DeclError

proc DeclError {errcode args} {
    variable errFormats
    set errmsg [format [dict get $errFormats $errcode] {*}$args]
    tailcall throw [list ROSEA $errcode {*}$args $errmsg] $errmsg
}
```

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

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

SplitRelvarName

The `rosea` package uses a naming convention whereby the names of the domain classes are encoded in the name of the relvar that holds the class instances. The class name is the last namespace component of the relvar name and the domain name, including it location relative the global namespace, is the set of leading namespace components. This function splits apart the class name from its domain and returns the two elements via variable references.

Implementation

```
<<helper commands>>=
proc SplitRelvarName {relvar {dvarname domain} {cvarname class}} {
    upvar 1 $dvarname domain $cvarname class
    set domain [namespace qualifiers $relvar]
    set class [namespace tail $relvar]
}
```

Chapter 5

Configuration Language

In this section we consider the domain specific language (DSL) that is used to configure a domain under `rosea`. Let's use this opportunity to recap our intent.

`Rosea` is divided into two main parts:

- A set of procedures that implement the run-time execution model. These procedures are data-driven and assume the existence of particular data structures and namespace organizations. These are the procedures that have been described in the previous section.
- A configuration language to populate the data structures and organize the namespaces as needed by the run-time mechanisms.

Strictly speaking, the configuration language will not do anything that cannot be done manually. Indeed, as we have been testing the run-time mechanisms above, we have done so using *ad hoc* constructed data and namespaces. But such an undertaking is too error prone for productive uses and some type of configuration scheme is needed. Configuring the run-time mechanisms via a DSL has the added benefit of creating a well defined interface that will insulate the package user from minor changes in the underlying data structures and procedural mechanisms.

The configuration language itself will be a Tcl script[1]. Commands in that script will define the aspects of a domain such as its classes, attributes and relationships. The DSL script will be evaluated in a context where the command names will conveniently resolve without any qualification.

There are two basic approaches to the DSL that were considered.

- As each DSL command is executed, create the corresponding run-time structure. One can view that as a type of on-the-fly interpretation.
- Consume all the DSL commands, storing away any data that the commands contain and then generate the run-time structure at the end.

The first approach is certainly simpler to code but has some decided disadvantages. To create the run-time structure as each command is executed will impose a rather strict ordering of the commands. For example, TclRAL insists that the participating relvars exist before a constraint may be defined upon them. This would imply that the DSL would have to be ordered in that way. Also, defining state models needs some semantic checking to insure a reasonable and consistent model is defined. That checking is hard to do in an on-the-fly interpretation scheme.

These disadvantages lead us to choose the second alternative, namely, build the configuration DSL commands to store away their data and then generate the run-time structures from the stored data. This approach requires significantly more programming but leads to considerable flexibility. It has the added benefit of allowing much easier and extensive introspection of the domain than could be possible if we were just examining the run-time structures themselves. As we shall see below, the data structures required to accumulate the domain definition are quite large and we will spend considerable time discussing them. There are a relatively large number of rules about how an XUML domain is specified and we will need data structures to hold the specification data and enforce the rules.

[1] This places the language into a category sometimes known as an *internal DSL*.

Config Namespace Layout

The `::rosea::Config` namespace has a relatively more complex layout than we have seen before. The reason for this is that we want to create a set of child namespaces that mirror the nesting of the configuration language statements. At the top level, the `domain` command is used to define each domain. The `domain` command accepts a script body that then defines the components of the domain. That script will be evaluated in a namespace where the commands that define the domain components, *e.g.* `class` and relationship commands, resolve appropriately. This lets us put commands for the body specifying a particular component into a namespace which prevents any problems of accidentally invoking commands that are inappropriate for that context.

We layout the `::rosea::Config` namespace as shown below.

```
<<configuration commands namespace>>=
namespace eval Config {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path [namespace parent]::Helpers
    <<config data>>
    <<config commands>>
    <<domain config namespace layout>>
}
```

Evaluating Configuration Scripts

You can evaluate the configuration script by simply gathering it together and passing it to the core `namespace eval` command. Unfortunately, the first error that is encountered will terminate the evaluation. This is decidedly inconvenient as you would prefer to continue on and come up with a list of errors for the evaluation much like a conventional language compiler would do when compiling a source file. Discovering errors one at a time is tediously unproductive.

Of course, it is possible to continue evaluating after an error but that requires a bit more code. We will also want to be able to evaluate scripts in different namespace contexts as we described above. We will accomplish all this by taking advantage of the core `::apply` command and its ability to execute a lambda function in a given namespace. So given a script body and an namespace we can evaluate it as shown below.

```
<<config commands>>=
namespace export ConfigEvaluate

proc ConfigEvaluate {ns body} {
    variable evalLambda
    tailcall ::apply [concat $evalLambda [list $ns]] $body ; # ❶
}
```

❶ The `concat` command treats its arguments as a list. On the off chance that the `ns` argument contains embedded whitespace, we need to insure that it is treated as a single element list. Hence the need to include the invocation of the `list` command.

The lambda function that performs the command evaluation is given below. We hold the evaluation lambda as a piece of data.

```
<<config data>>=
variable evalLambda {{body} {
    upvar #0 ::rosea::Config::errcount errcount ; # ❶
    upvar #0 ::rosea::Config::configlineno configlineno
    set lineno $configlineno
    set command {}
    foreach line [split $body \n] { # ❷
        append command $line \n
```

```
        incr lineno ; # ❸
        if {[info complete $command]} {
            try {
                eval $command
            } on error {result} {
                log::error "line $configlineno: \"$result\""
                incr errcount
            }
            set command {} ; # ❹
            set configlineno $lineno
        }
    }
    return $errcount
}}
```

❶ We need to keep track of all the errors encountered and what line of the script we are currently dealing with.

❷ We split the body along lines and then reassemble the lines into a complete command. The `info complete` command tells us when we have something that has some chance of being a real command.

❸ We need to keep track of where we are in the `body`.

❹ After evaluating a command we begin to assemble another one and set our location counter to where that command starts.

Configure

One of our top level `rosea` commands is to configure a domain from a script.

`::rosea configure` *script*

`script`
A Tcl script that is executed in an environment that will resolve *domain configuration* commands that may be used to define the characteristics of a domain.

Implementation

```
<<rosea exports>>=
namespace export configure

<<rosea commands>>=
proc configure {script} {
    namespace upvar Config errcount errcount configlineno configlineno
    set errcount 0
    set configlineno 1

    ConfigEvaluate ::rosea::Config $script

    if {$errcount > 0} {
        tailcall DeclError CONFIG_ERRORS $errcount
    }
    return $errcount
}
```

```
<<error code formats>>=
CONFIG_ERRORS     {encountered %d configuration script errors}
```

ConfigureFromChan

It is often convenient to obtain the `configure` script from an I/O channel. The `rosea configureFromChan` command supports reading the configure script directly from a channel.

`::rosea configureFromChan` *channel*

`channel`
A Tcl channel handle that is readable.

The `configureFromChan` command invokes `configure` on the script obtained by reading *channel* until end of file is reached.

Implementation

```
<<rosea exports>>=
namespace export configureFromChan

<<rosea commands>>=
proc configureFromChan {chan} {
    tailcall configure [::chan read -nonewline $chan]
}
```

ConfigureFromFile

It is often convenient to hold the `configure` script in a file. The `rosea configureFromFile` command supports reading the configure script directly from a file.

`::rosea configureFromFile` *filename*

`filename`
The name of a file containing a domain configuration script.

The `configureFromFile` command invokes `configure` on the script obtained by reading the contents of the file named, *filename*.

Implementation

```
<<rosea exports>>=
namespace export configureFromFile

<<rosea commands>>=
proc configureFromFile {filename} {
    set f [::open $filename r]
    try {
        configureFromChan $f
    } finally {
        ::chan close $f
    }
}
```

Defining a Domain

The *script* handed to the `configure` command should contain invocations of the `domain` command.

As we described above, the domain configuration commands will take the data in their arguments and store it away. After we have finished all the configuration scripts, then the `generate` command is used to layout the data structures and namespace as needed.

The structure of the data that is populated by the configuration scripts is rather complicated itself. There are many rules as to what is allowed or not allowed when configuring the characteristics of a domain. It is important that these rules be enforced to insure that the generated domain is correct. To accomplish that we will formulate a model of the domain rules. Such models-of-models are usually called *meta-models*. An unfortunate consequence of using a meta-model is that they tend to be rather abstract and it is easy to confuse the rules associated with building models with the rules associated with some particular application domain. We will strive to make the distinction clear. To further confound the situation, we will implement the meta-models as a set of TclRAL relvars and relvar constraints. This means we are putting TclRAL commands to two distinct uses: first as a basis for the data architecture of the generated domains and second as the basis for accumulating the domain definition during configuration. Again, we will strive to make the distinction clear.

In this section we start the process of configuring a domain by showing some of the data structures we will use to store the configuration data as it is being collected. The design of this meta-model was first defined in the `raloo` package. We will present these meta-model diagrams in small groups that are associated with the configuration commands being discussed. We will establish a pattern of presenting the meta-model class diagram followed by a discussion of the rules it enforces and finally show the configuration code and how it populates the meta-model relvars.

Below is a UML class diagram for a domain and some of its component parts.

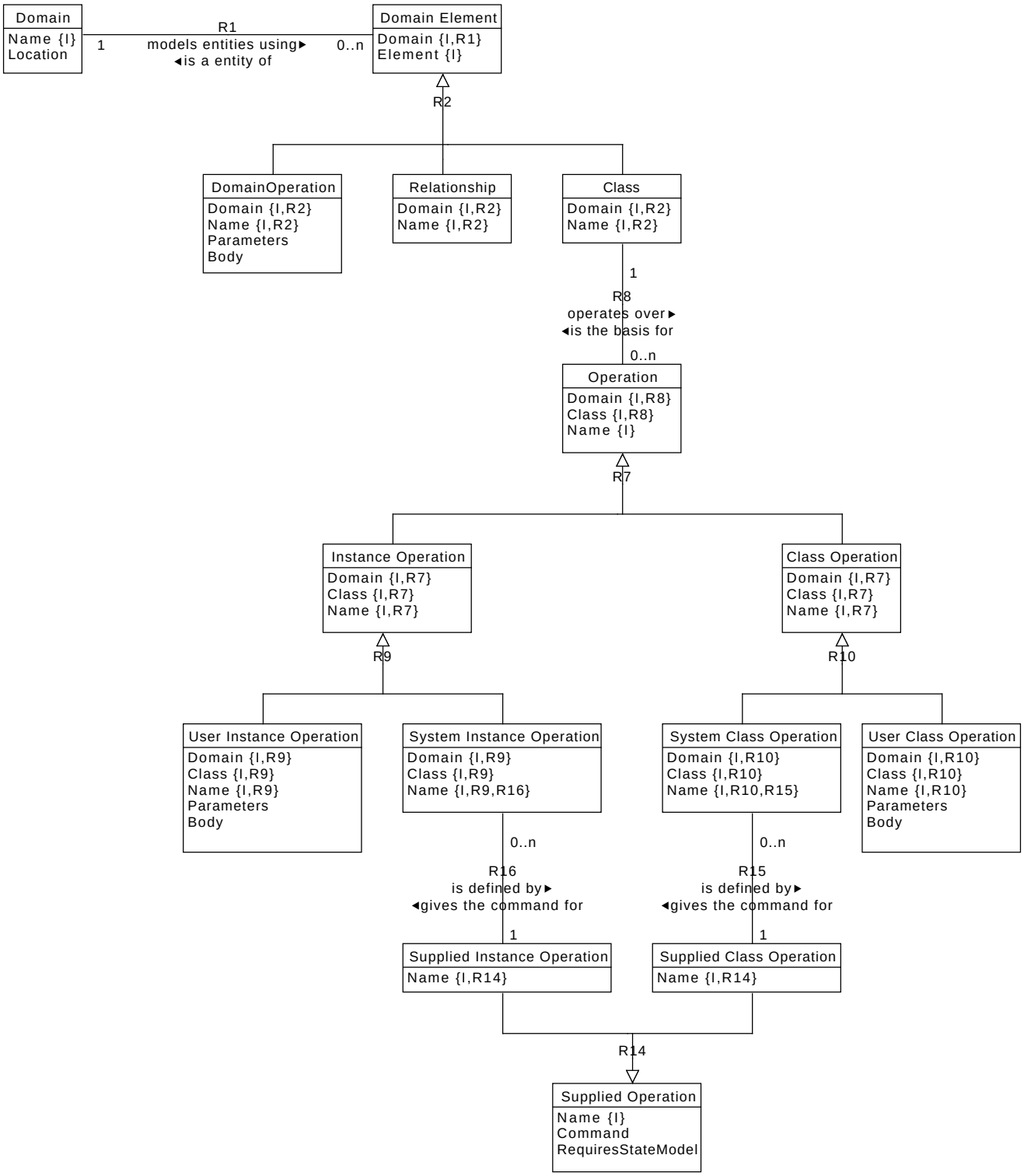

Figure 5.1: Domain Configuration Class Diagram

A **Domain** is characterized by a **Name** and **Location** and consists of a set of **Domain Elements** (**R1**). **Domain Elements** are of three types (**R2**), **Domain Operation**, **Relationship** and **Class**. **R2** insures that no two elements have the same name and this makes sure that there are no naming conflicts with the namespace ensemble commands that are generated for the domain. A

Class may also have a set of operations (**R8**). These operations are of two types (**R7**), **Instance Operation** and **Class Operation**. The difference between the two is that **Instance Operation** will have a generated `self` argument that refers to the instance on which the operation is performed. A **Class Operation** has no such argument. In turn, operations may be either user defined or system defined (**R9** and **R10**). System operations are one of the **Supplied Operation** (**R16** and **R15**) A **Supplied Operation** is mapped to a Tcl **Command** and may be supplied to all classes and instances or to those where **RequiresStateModel** is true.

As we will see repeated many times below, the above UML class diagram has a direct textual correspondence to TclRAL `relvar` commands.

```
<<config data>>=
relvar create Domain {
    Name        string
    Location    string
} Name
relvar create DomainElement {
    Domain  string
    Element string
} {Domain Element}
relvar association R1\
    DomainElement Domain *\
    Domain Name 1
relvar create Class {
    Domain  string
    Name    string
} {Domain Name}
relvar create Relationship {
    Domain  string
    Name    string
} {Domain Name}
relvar create DomainOperation {
    Domain      string
    Name        string
    Parameters  list
    Body        string
} {Domain Name}
relvar partition R2 DomainElement {Domain Element}\
    Class {Domain Name}\
    Relationship {Domain Name}\
    DomainOperation {Domain Name}
```

We postpone the definitions of the relvars for the operations until later.

When configuring a domain, we will evaluate the configuration script in the `DomainDef` child namespace. That namespace will define commands for all the components of the domain. We will use a similar arrangement for other nested aspects of domain definitions.

Domain

`domain` *name body*

`name`
The name of the domain. A domain name must be a non-empty string.

`body`
A Tcl script containing invocation of the domain definition commands to specify the details of the domain configuration.

Implementation

```
<<config commands>>=
proc domain {name body} {
    namespace upvar DomainDef DomainName DomainName ; # ❶
    set DomainName $name

    relvar uinsert ::rosea::Config::Domain [list Name $name Location {}] ; # ❷
    try {
        if {$name eq {}} {
            tailcall DeclError EMPTY_NAME domain
        }

        relvar eval { # ❸
            ConfigEvaluate [namespace current]::DomainDef $body

            # At this point we have the definition of the domain and enough
            # information to tie together the referential attributes and
            # the association definitions.
            <<domain: bind association references>>

            # We must also compute how polymorphic events are inherited
            # down generalization hierarchies.
            <<domain: propagate polymorphic events>>
        }
    } on error {result} {
        ::rosea::Config::HandleConfigError $result
    }
}
```

❶ We place the domain name into the child namespace where the context implies that all the components defined are to be part of this domain.

❷ We allow the `domain` command to be invoked more than once. Domains are open ended definitions and the `uinsert` command will not throw an error upon a duplicate.

❸ Evaluating the script must be done as a `relvar` transaction as we will need to defer the constraint checking until all the domain components have been defined.

```
<<error code formats>>=
EMPTY_NAME      {the empty string is not a valid name for a %s}
```

After evaluating the domain configuration script, it is necessary to patch up the association data. We will defer the discussion about binding the associations and their attribute referenced until later when we have covered those commands. For now, we will say that there are semantic evaluations and checks that can only be done after the entire domain configuration script has been processed. The reason for this lies mainly with the way the DSL script commands were defined to make them more convenient to use when translating an XUML model. Polymorphic events also must be dealt with after the domain configuration is in place. Again we will discuss this further below.

Although we have said that domain configurations are cumulative, clearly the need to resolve association bindings and polymorphic event inheritance means that domain configurations cannot be split arbitrarily. You will need to be aware that certain divisions of domain configuration will end up separating information that is needed at the end of the configuration process.

Tests

```
<<config command tests>>=
test configure-1.0 {
    Define domain
} -setup {
```

```
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            # A domain may be empty of components.
        }
    }
    relation extract [relvar set ::rosea::Config::Domain] Name
} -result {foo}
```

```
<<config command tests>>=
test configure-1.1 {
    Define domain -- bad name
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain {} {
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error
```

We need some more testing support. After running configuration commands, we will need to clean up the configuration data population to ready it for the next test.

```
<<test utility procs>>=
proc cleanupConfigData {} {
    set preserved {
        ::rosea::Config::TransitionRule
        ::rosea::Config::SuppliedOperation
        ::rosea::Config::SuppliedInstanceOperation
        ::rosea::Config::SuppliedClassOperation
        ::rosea::Config::Config_AssocSpec
        ::rosea::Config::Config_DataError
    } ; # ❶
    relvar eval {
        foreach rname [relvar names ::rosea::Config::*] {
            if {$rname ni $preserved} {
                relvar set $rname [relation emptyof [relvar set $rname]]
            }
        }
    }
}
```

❶ Note that some of the configuration data is specification information and needs to be preserved. We keep a list of the preserved relvars so we can skip over them.

Defining Domain Components

In this section we discuss the commands that are used to define the components of a domain. We define child namespaces for those domain component commands that have a nested structure. In this case, it is the `class` command that requires additional configuration commands.

We start with the namespace layout for the `DomainDef` namespace.

```
<<domain config namespace layout>>=
```

```
namespace eval DomainDef {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace import ::rosea::Config::ConfigEvaluate
    namespace import ::rosea::Helpers::DeclError
    <<domain config commands>>

    <<class config namespace layout>>
    <<assigner config namespace layout>>
}
```

Defining Classes

An XUML class represents a real world entity. It is defined by giving it a name and providing a script to further define its characteristics.

Below is a UML class diagram of the configuration data for defining classes.

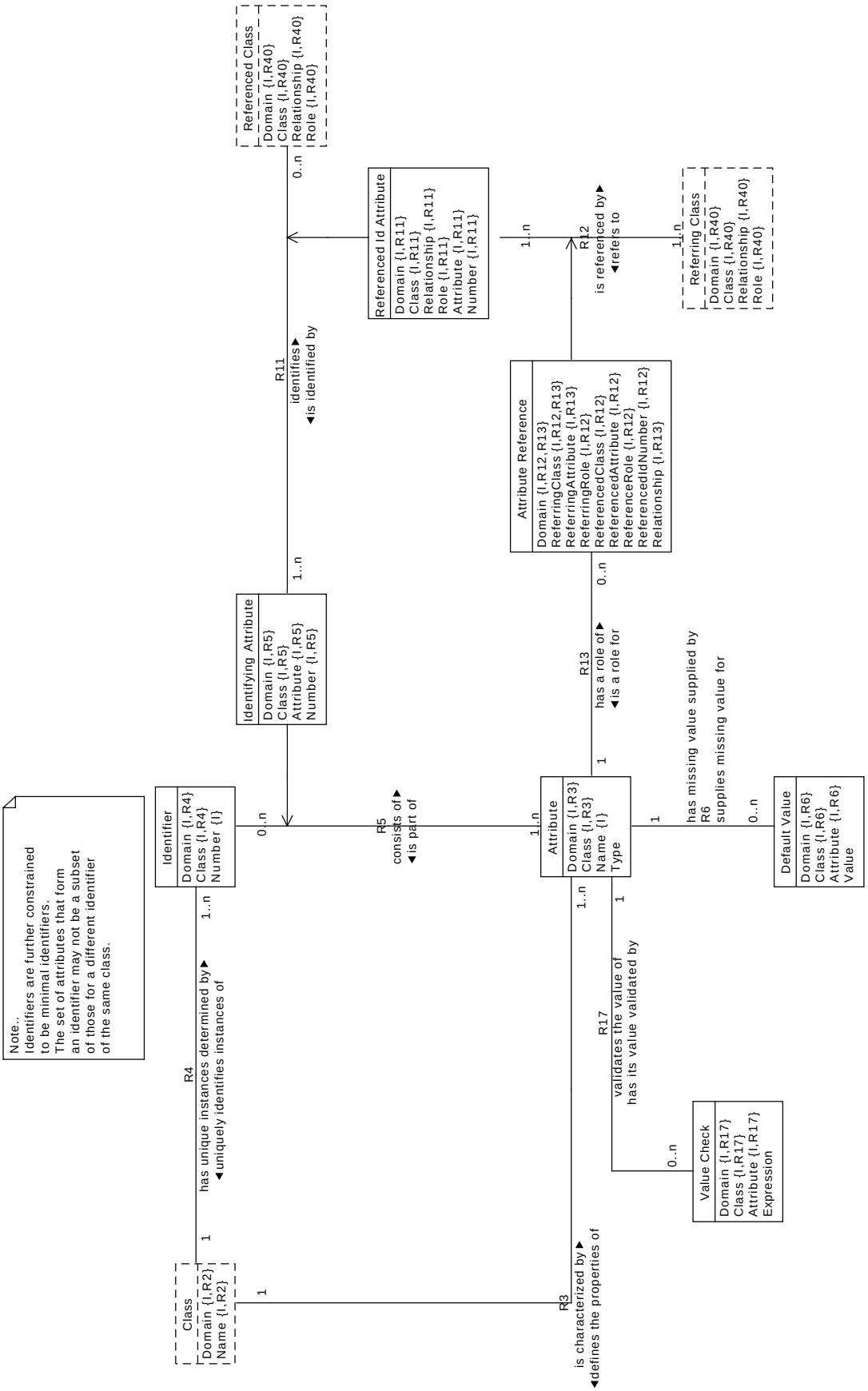

Figure 5.2: Classes Configuration Class Diagram

There are a substantial number of rules about what constitutes a proper XUML class. A **Class** has one or more **Attribute**s (**R3**). It also has one or more **Identfier**s (**R4**). An **Attribute** that serves as part of an **Identifier** is known as an **Identifying Attribute** (**R5**). An **Attribute** may have a **Default Value** (**R6**). An **Identifiying Attribute** is sometimes referenced as part of realizing a relationship and in that role is treated as a **Referenced Id Attribute** (**R11**). Also, classes may contain attributes that are referential and in that role the attribute is treated as an **Attribute Reference** (**R12**). Of course, any **Attribute Reference** must ultimately refer to some **Attribute** that is in a class (**R13**).

As before, the TclRAL relvar definitions corresponding to the class model are are direct text transliteration of the model graphic.

```
<<config data>>=
relvar create Attribute {
    Domain  string
    Class   string
    Name    string
    Type    string
} {Domain Class Name}
relvar association R3\
    Attribute {Domain Class} +\
    Class {Domain Name} 1
relvar create Identifier {
    Domain  string
    Class   string
    Number  int
} {Domain Class Number}
relvar association R4\
    Identifier {Domain Class} +\
    Class {Domain Name} 1
relvar create IdentifyingAttribute {
    Domain      string
    Class       string
    Attribute   string
    Number      int
} {Domain Class Attribute Number}
relvar correlation R5 IdentifyingAttribute\
    {Domain Class Number} + Identifier {Domain Class Number}\
    {Domain Class Attribute} * Attribute {Domain Class Name}
relvar create DefaultValue {
    Domain      string
    Class       string
    Attribute   string
    Value       string
} {Domain Class Attribute}
relvar association R6\
    DefaultValue {Domain Class Attribute} ?\
    Attribute {Domain Class Name} 1
relvar create ValueCheck {
    Domain      string
    Class       string
    Attribute   string
    Expression  string
} {Domain Class Attribute}
relvar association R17\
    ValueCheck {Domain Class Attribute} ?\
    Attribute {Domain Class Name} 1
relvar create AttributeReference {
    Domain              string
    ReferringClass      string
    ReferringAttribute  string
    ReferringRole       string
    ReferencedClass     string
    ReferencedAttribute string
    ReferencedRole      string
```

```
    ReferencedIdNumber  int
    Relationship        string
} {Domain ReferringClass ReferringAttribute ReferringRole\
    ReferencedClass ReferencedAttribute ReferencedRole ReferencedIdNumber\
    Relationship}
relvar association R13\
    AttributeReference {Domain ReferringClass ReferringAttribute} *\
    Attribute {Domain Class Name} 1
relvar create ReferencedIdAttribute {
    Domain          string
    Class           string
    Relationship    string
    Role            string
    Attribute       string
    Number          int
} {Domain Class Relationship Role Attribute Number}
```

Class

The class command is used to define classes and specify their characteristics.

`class` *name script*

name
The name of the class. A class name must be a non-empty string.

script
A Tcl script containing invocation of the class definition commands to specify the details of the class configuration.

Implementation

```
<<domain config commands>>=
proc class {name body} {
    namespace upvar [namespace current] DomainName DomainName ; # ❶
    namespace upvar ClassDef ClassName ClassName
    set ClassName $name

    if {$name eq {}} {
        tailcall DclError EMPTY_NAME class
    }
    try {
        relvar insert ::rosea::Config::DomainElement [list\
            Domain  $DomainName\
            Element $name\
        ]
    } trap {RAL relvar insert DUPLICATE_TUPLE} {} {
        tailcall DclError DUP_ELEMENT_NAME $name
    } ; # ❷
    relvar insert ::rosea::Config::Class [list\
        Domain  $DomainName\
        Name    $name\
    ]
    set instops [pipe {
        relvar set ::rosea::Config::SuppliedInstanceOperation |
        relation extend ~ ioptuple\
            Domain string {[set DomainName]}\
            Class string {[set ClassName]}
```

```
    }]
    relvar union ::rosea::Config::SystemInstanceOperation $instops
    relvar union ::rosea::Config::InstanceOperation $instops
    relvar union ::rosea::Config::Operation $instops
    set classops [pipe {
        relvar set ::rosea::Config::SuppliedClassOperation |
        relation extend ~ coptuple\
            Domain string {[set DomainName]}\
            Class string {[set ClassName]}
    }]
    relvar union ::rosea::Config::SystemClassOperation $classops
    relvar union ::rosea::Config::ClassOperation $classops
    relvar union ::rosea::Config::Operation $classops

    ConfigEvaluate [namespace current]::ClassDef $body
}
```

1. Again we arrange for namespace variables to hold the current domain and class names to provide the context of the definition.
2. We trap this `relvar insert` specifically so we can give a more meaningful error message.

```
<<error code formats>>=
DUP_ELEMENT_NAME    {a class, relationship or domain operation named, "%s",\
                    already exists}
```

Defining Classes Components

Since there are several aspects of classes, the `class` command takes a *script* argument which should invoke the commands we discuss in this section. Following our pattern, we define a namespace where the class body script is evaluated.

```
<<class config namespace layout>>=
namespace eval ClassDef {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace import ::rosea::Config::ConfigEvaluate
    namespace import ::rosea::Helpers::DeclError

    <<class config commands>>

    <<state model config namespace layout>>
}
```

Attribute

Within a class definitions, the `attribute` command specifies the attributes of the class.

`attribute` *name type ?option value ...?*

name
The name of the attribute. Attributes names may not be the empty string.

type
The type of the attribute. The *type* may be any type that is acceptable to the `::ral::tuple` command. Those are:

- string
- int
- long
- wideInt
- bignum
- double
- boolean
- list
- dict
- bytearray
- Tuple
- Relation

option value
A set of *option/value* pairs that define the characteristics of the attribute. Valid options are:

-default ***value***
Specifies a default value that will be supplied if an instance is created and no value is provided for the attribute.

-check ***expr***
Check that values of the attribute satisfy *expr*. *Expr* must be suitable for the `::expr` command and may reference variables that are the same name as the attributes of the class. Checks are made on a tuple-by-tuple basis during any insert or update operation for the class instance.

-id ***number***
Specifies that the attribute is part of the identifier given by *number*. All classes must have at least one identifier. Classes may have multiple identifiers and each is given a separate number. Typically, *number* is a small integer such as 1, 2 or 3.

A Note About Types

You will notice that the `attribute` command requires that you declare a type for the attribute. That type is one of the supported internal Tcl types along with Tuple and Relation types. It may seem strange to demand type information for a language like Tcl where the prevailing mantra is *everything is a string*. [a] TclRAL demands the type information and does so as another form of data integrity checking. If you declare an attribute to be, say a `double`, then you are stating that at some time the attribute may be used in a context where a `double` is expected. For example the attribute might be added or multiplied. Given the type of "duck typing" that Tcl uses, any string that can be reasonably interpreted as a `double` and converted into one is what is expected in that context. Otherwise, an error is thrown. So, if you set the value of such an attribute declared to be `double` to the value of, "foo", then when the arithmetic operation is performed, an error would be thrown. The type declaration is really a declaration of the context in which the value may be placed and we want to insure that any values assigned to the attribute will not cause an error to be thrown later. Since `string` is the universal Tcl type, an attribute defined to be of `string` type accepts any value.

Note that attributes may be of Tuple or Relation type. TclRAL allows attributes to be typed according to most of the built in Tcl data types. We would consider it very suspect for the base classes of a domain to contain relation or tuple valued attributes. However, relation valued attributes are not uncommon in relational algebraic expressions, have some very convenient uses, are perfectly valid values and we have seen several instances of them already when the package commands were discussed.

[a] Well, really, it's more like "everything looks like a string at the script level", but that is just too many words.

A Note About Identifiers

Integer numbers are used to distinguish identifiers. The most common case is for a class to have a single identifier and for that identifier to have a single attribute. Typically, one attribute of the class will be marked as `-id 1`. Identifiers must also be minimal identifiers, *i.e.* one identifier's attributes must not be a subset (proper or improper) of the attributes of another identifier. The rule of minimal identifiers is *not* enforced by the class configuration UML model but TclRAL will insist upon it when the underlying relvar is created.

The implementation of the `attribute` command simply creates tuples in the relvars to hold the attribute characteristics.

Implementation

```
<<class config commands>>=
proc attribute {name type args} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME attribute
    }
    if {[string range $name 0 1] eq "__"} {
        tailcall DeclError RESERVED_NAME $name
    }
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    relvar insert ::rosea::Config::Attribute [list\
        Domain  $DomainName\
        Class   $ClassName\
        Name    $name\
        Type    $type\
    ]

    foreach {option value} $args {
        switch -exact -- $option {
            -default {
```

```
                relvar insert ::rosea::Config::DefaultValue [list\
                    Domain      $DomainName\
                    Class       $ClassName\
                    Attribute   $name\
                    Value       $value\
                ]
            }
            -id {
                relvar uinsert ::rosea::Config::Identifier [list\
                    Domain      $DomainName\
                    Class       $ClassName\
                    Number      $value\
                ] ; # ❶
                relvar insert ::rosea::Config::IdentifyingAttribute [list\
                    Domain      $DomainName\
                    Class       $ClassName\
                    Attribute   $name\
                    Number      $value\
                ]
            }
            -check {
                relvar insert ::rosea::Config::ValueCheck [list\
                    Domain      $DomainName\
                    Class       $ClassName\
                    Attribute   $name\
                    Expression  $value\
                ]
            }
            default {
                tailcall DeclError UNKNOWN_OPTION attribute $option
            }
        }
    }
}
```

❶ Using `uinsert` here insures we don't get an error when an identifiers has multiple attributes. The `uinsert` command performs an insert with union semantics, *i.e.* duplicates are simply ignored.

```
<<error code formats>>=
RESERVED_NAME {names beginning with two underscore characters are reserved,\
        "%s"}
```

Tests

```
<<config command tests>>=
test configure-2.0 {
    Define domain and classes
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                attribute a2 string -default 20
            }
            class c2 {
```

```
                attribute a1 string -id 1
                attribute a2 string -default 40
            }
        }
    }
    relation cardinality [relvar set ::rosea::Config::Class]
} -result {2}
```

```
<<config command tests>>=
test configure-2.1 {
    Define domain -- bad attribute option
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -foo 1 -id 1
            }
        }
    }
} -result {encountered 2 configuration script errors} -returnCodes error
```

Reference

One of the more complicated aspects of XUML is understanding class relationships and how the fundamental notions of referential integrity constraints from relational algebra are implemented. For our purposes, we have chosen to specify relationships in two pieces:

1. Specifying that some attributes are *referential*. Attribute references are the *formalism* by which relationships are realized. They define attributes whose values are equal to the value of a corresponding attribute in the related class.
2. Specifying the relationship characteristics such as the classes involved and the multiplicity and conditionality of the relationship. Relationships stand for real world associations between classes and those associations have properties of multiplicity and conditionality that are independent of the manner in which the relationship is realized.

The reason we have taken this approach is that it corresponds closely to the UML graphic that we are translating. The referential nature of an attribute is denoted in the graphic as a characteristic of the attribute and relationships have a separate graphical representation. The result of this decision is that the most common case of simple associations can be specified with the least amount of text. The trade-off is that the most general case of a reflexive, class-based association requires more specification to resolve an inherent ambiguity.

The `reference` command states that a class contains attributes that refer to another class and are the means by which a relationship between the classes is realized. It is with the `reference` command that we can state the correspondence between attributes in the referring class and attributes in the referenced class.

`reference` *relationship class ?option value . . .?*

`relationship`
The name of the relationship to which the reference applies.

`class`
The name of the class to which the reference refers.

`option value`
A set of *option/value* pairs that define the characteristics of the reference. Valid options are:

`-link` ***attrmap***
The `-link` option specifies an attribute reference. The *attrmap* value is a one or two element list. The first element is the name of an attribute in the class being defined. The second element is the name of an attribute in the class given by the *class* argument. The `-link` option specifies that an attribute in the class being defined is a reference to (*i.e.* will have the same value as) an attribute in the *class* class. If the second element of the *attrmap* is missing, then the name of the referenced attribute in *class* is assumed to be the same as the referring attribute. The link options must define a one-to-one correspondence between referring attributes in the class being defined and an identifier in *class*. At least one `link` option must be given. Multiple `-link` options may be given for *relationship* when the referenced identifier of *class* has more than one attribute.

`-refid` ***number***
The `refid` option gives the *number* of the identifier to which the reference is directed. If missing, then `-refid` defaults to 1. Typically, references in the class being defined refer to the primary identifier in *class*. However, if they refer to another identifier, then the `-refid` option is necessary. All the attributes given by the `-link` options, as a group, must reference a specific identifier in *class*.

Implementation

```
<<class config commands>>=
proc reference {relname rclass args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    set refnum 1
    set linkopts [list]

    foreach {option value} $args {
        switch -exact -- $option {
            -refid {
                set refnum $value
            }
            -link {
                if {[llength $value] < 1 || [llength $value] > 2} {
                    tailcall DeclError ARG_FORMAT $value
                }
                lappend linkopts $value
            }
            default {
                tailcall DeclError UNKNOWN_OPTION reference $option
            }
        }
    }
```

```
    # We defer processing the link options until the end so as not
    # to impose an order on the -refid option.
    foreach linkopt $linkopts {
        lassign $linkopt referring referenced
        if {$referenced eq {}} {
            set referenced $referring
        }
        relvar insert ::rosea::Config::AttributeReference [list\
            Domain              $DomainName\
            ReferringClass      $ClassName\
            ReferringAttribute  $referring\
            ReferringRole       source\
            ReferencedClass     $rclass\
            ReferencedAttribute $referenced\
            ReferencedRole      target\
            ReferencedIdNumber  $refnum\
            Relationship        $relname
        ] ; # ❶
    }
}
```

❶ For class based associations, the values inserted here are *wrong*. It is only after the association has been defined that we know it is class based and can then update the ReferringRole and ReferencedRole properly.

```
<<error code formats>>=
ARG_FORMAT      {options and values must come in pairs, got "%s"}
UNKNOWN_OPTION  {unknown %s command option, "%s"}
```

Tests

```
<<config command tests>>=
test configure-2.2 {
    Define class reference
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    catch {
        rosea configure {
            domain foo {
                class c1 {
                    attribute a1 string -id 1
                    attribute a2 string -default 20
                }
                class c2 {
                    attribute a1 string -id 1
                    attribute a2 string -default 40
                    reference R1 c1 -link a1
                }
                set ::rosea::test::arefs [relation cardinality\
                    $::rosea::Config::AttributeReference]
            }
        }
    }
    set arefs
} -result {1}
```

Defining Operations

Operations are a means of factoring code into reusable modules. We make the distinction between class based operations and instance based operations.

For instance based operations, there is an implicit `self` argument defined. Class based operations have no `self` argument. Otherwise, there is little difference between the two.

To prevent naming conflicts, we insist that all the operations for a class, be they class based or instance based, have distinct names. Further, we have the notion of *system supplied* versus *user supplied* operations. The ensemble commands for domain classes have both the system and user supplied commands and so we insist that user supplied operation names not conflict with those supplied by the system.

To support these ideas, we define the following relvars. These relvars are part of the configuration information for domains that was discussed above.

```
<<config data>>=
relvar create Operation {
    Domain  string
    Class   string
    Name    string
} {Domain Class Name}

relvar association R8\
    Operation {Domain Class} * Class {Domain Name} 1

relvar create InstanceOperation {
    Domain  string
    Class   string
    Name    string
} {Domain Class Name}

relvar create ClassOperation {
    Domain  string
    Class   string
    Name    string
} {Domain Class Name}

relvar partition R7 Operation {Domain Class Name}\
    InstanceOperation {Domain Class Name}\
    ClassOperation {Domain Class Name}

relvar create UserInstanceOperation {
    Domain      string
    Class       string
    Name        string
    Parameters  string
    Body        string
} {Domain Class Name}

relvar create SystemInstanceOperation {
    Domain      string
    Class       string
    Name        string
} {Domain Class Name}

relvar partition R9 InstanceOperation {Domain Class Name}\
    UserInstanceOperation {Domain Class Name}\
    SystemInstanceOperation {Domain Class Name}

relvar create UserClassOperation {
    Domain  string
```

```
    Class   string
    Name    string
    Parameters  string
    Body        string
} {Domain Class Name}

relvar create SystemClassOperation {
    Domain  string
    Class   string
    Name    string
} {Domain Class Name}

relvar partition R10 ClassOperation {Domain Class Name}\
    UserClassOperation {Domain Class Name}\
    SystemClassOperation {Domain Class Name}

relvar create SuppliedOperation {
    Name                string
    Command             string
    RequiresStateModel  boolean
} Name

relvar create SuppliedClassOperation {
    Name    string
} Name

relvar create SuppliedInstanceOperation {
    Name    string
} Name

relvar partition R14 SuppliedOperation Name\
    SuppliedClassOperation Name\
    SuppliedInstanceOperation Name

relvar association R15\
    SystemClassOperation Name *\
    SuppliedClassOperation Name 1

relvar association R16\
    SystemInstanceOperation Name *\
    SuppliedInstanceOperation Name 1
```

Classop

Since we make the distinction between user supplied and system supplied class based operations, we need to define those that are to be supplied by the system. Here we populate the **Supplied Operation** and **Supplied Class Operation** relvars to match the set of class commands we discussed above.

```
<<config data>>=
relvar eval {
    relvar insert SuppliedOperation {
        Name                findAll
        Command             ::rosea::ClassCmds::findAll
        RequiresStateModel  false
    } {
        Name                findById
        Command             ::rosea::ClassCmds::findById
        RequiresStateModel  false
    } {
        Name                findWhere
```

```
        Command             ::rosea::ClassCmds::findWhere
        RequiresStateModel  false
    } {
        Name                create
        Command             ::rosea::ClassCmds::create
        RequiresStateModel  false
    } {
        Name                createin
        Command             ::rosea::ClassCmds::createin
        RequiresStateModel  true
    } {
        Name                createasync
        Command             ::rosea::ClassCmds::createasync
        RequiresStateModel  true
    } {
        Name                update
        Command             ::rosea::ClassCmds::update
        RequiresStateModel  false
    }

    relvar insert SuppliedClassOperation {
        Name    findAll
    } {
        Name    findById
    } {
        Name    findWhere
    } {
        Name    create
    } {
        Name    createin
    } {
        Name    createasync
    } {
        Name    update
    }
}
```

`classop` *name params body*

name
The name of the class based operation.

params
A list of the formal parameters of the operation. Parameters are specified in the same manner as for the `::proc` command.

body
A Tcl script containing code to be executed when the class operation is invoked. Class based operations will be made part of the ensemble of commands for the class being defined.

Implementation

```
<<class config commands>>=
proc classop {name params body} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME "class operation"
    } elseif {[string first :: $name] != -1} {
        tailcall DeclError NS_QUALIFIERS $name
```

```
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    set sysop [relvar restrictone ::rosea::Config::Operation\
        Domain $DomainName Class $ClassName Name $name]
    if {[relation isnotempty $sysop]} {
        tailcall DeclError DUP_OP_NAME $name
    }

    set optuple [list Domain $DomainName Class $ClassName Name $name]
    relvar insert ::rosea::Config::Operation $optuple
    relvar insert ::rosea::Config::ClassOperation $optuple

    lappend optuple Parameters $params Body $body
    relvar insert ::rosea::Config::UserClassOperation $optuple
}
```

```
<<error code formats>>=
DUP_OP_NAME     {operation call, "%s", already exists}
NS_QUALIFIERS   {name, "%s", contains namespace qualifiers}
```

Instop

As for class based operations, we define here the instance based operations provided by the system. These are the same operations discussed above. For each class, an ensemble command named, `instop`, is created and all the instance operations, both system supplied and user supplied, are made part of that ensemble.

Below we have the data for the system supplied instance operations.

```
<<config data>>=
relvar eval {
    relvar insert SuppliedOperation {
        Name                findRelated
        Command             ::rosea::InstCmds::findRelated
        RequiresStateModel  false
    } {
        Name                findUnrelated
        Command             ::rosea::InstCmds::findById
        RequiresStateModel  false
    } {
        Name                findRelatedWhere
        Command             ::rosea::InstCmds::findRelatedWhere
        RequiresStateModel  false
    } {
        Name                findUnrelatedWhere
        Command             ::rosea::InstCmds::findUnrelatedWhere
        RequiresStateModel  false
    } {
        Name                updateAttribute
        Command             ::rosea::InstCmds::updateAttribute
        RequiresStateModel  false
    } {
        Name                readAttribute
        Command             ::rosea::InstCmds::readAttribute
        RequiresStateModel  false
    } {
        Name                delete
        Command             ::rosea::InstCmds::delete
```

```
        RequiresStateModel  false
    } {
        Name                signal
        Command             ::rosea::InstCmds::signal
        RequiresStateModel  true
    } {
        Name                delaysignal
        Command             ::rosea::InstCmds::delaysignal
        RequiresStateModel  true
    } {
        Name                canceldelayed
        Command             ::rosea::InstCmds::canceldelayed
        RequiresStateModel  true
    } {
        Name                delayremaining
        Command             ::rosea::InstCmds::delayremaining
        RequiresStateModel  true
    } {
        Name                deRef
        Command             ::rosea::InstCmds::deRef
        RequiresStateModel  false
    } {
        Name                isEmptyRef
        Command             ::rosea::InstCmds::isEmptyRef
        RequiresStateModel  false
    } {
        Name                isNotEmptyRef
        Command             ::rosea::InstCmds::isNotEmptyRef
        RequiresStateModel  false
    } {
        Name                refMultiplicity
        Command             ::rosea::InstCmds::refMultiplicity
        RequiresStateModel  false
    } {
        Name                isRefSingular
        Command             ::rosea::InstCmds::isRefSingular
        RequiresStateModel  false
    } {
        Name                isRefEqual
        Command             ::rosea::InstCmds::isRefEqual
        RequiresStateModel  false
    } {
        Name                forAllRefs
        Command             ::rosea::InstCmds::forAllRefs
        RequiresStateModel  false
    }

    relvar insert SuppliedInstanceOperation {
        Name    findRelated
    } {
        Name    findUnrelated
    } {
        Name    findRelatedWhere
    } {
        Name    findUnrelatedWhere
    } {
        Name    updateAttribute
    } {
        Name    readAttribute
    } {
        Name    delete
    } {
```

```
        Name    signal
    } {
        Name    delaysignal
    } {
        Name    canceldelayed
    } {
        Name    delayremaining
    } {
        Name    deRef
    } {
        Name    isEmptyRef
    } {
        Name    isNotEmptyRef
    } {
        Name    refMultiplicity
    } {
        Name    isRefSingular
    } {
        Name    isRefEqual
    } {
        Name    forAllRefs
    }
}
```

`instop` *name params body*

name
The name of the instance based operation.

params
A list of the formal parameters of the operation. Parameters are specified in the same manner as for the `::proc` command. The system will add an additional formal parameter named `self` and invocations of instance based operations must provide as the first argument an instance reference.

body
A Tcl script containing code to be executed when the instance operation is invoked. Class based operations will be made part of the ensemble of commands for the class being defined.

Implementation

```
<<class config commands>>=
proc instop {name params body} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME "instance operation"
    } elseif {[string first :: $name] != -1} {
        tailcall DeclError NS_QUALIFIERS $name
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    set sysop [relvar restrictone ::rosea::Config::Operation\
        Domain $DomainName Class $ClassName Name $name]
    if {[relation isnotempty $sysop]} {
        tailcall DeclError DUP_OP_NAME $name
    }

    set optuple [list Domain $DomainName Class $ClassName Name $name]
```

```
    relvar insert ::rosea::Config::Operation $optuple
    relvar insert ::rosea::Config::InstanceOperation $optuple

    lappend optuple Parameters $params Body $body
    relvar insert ::rosea::Config::UserInstanceOperation $optuple
}
```

Defining Relationships

To define the various types of relationships, we have a rather extensive data model to encode all the rules about class relationships. The figure below shows the class model for relationships.

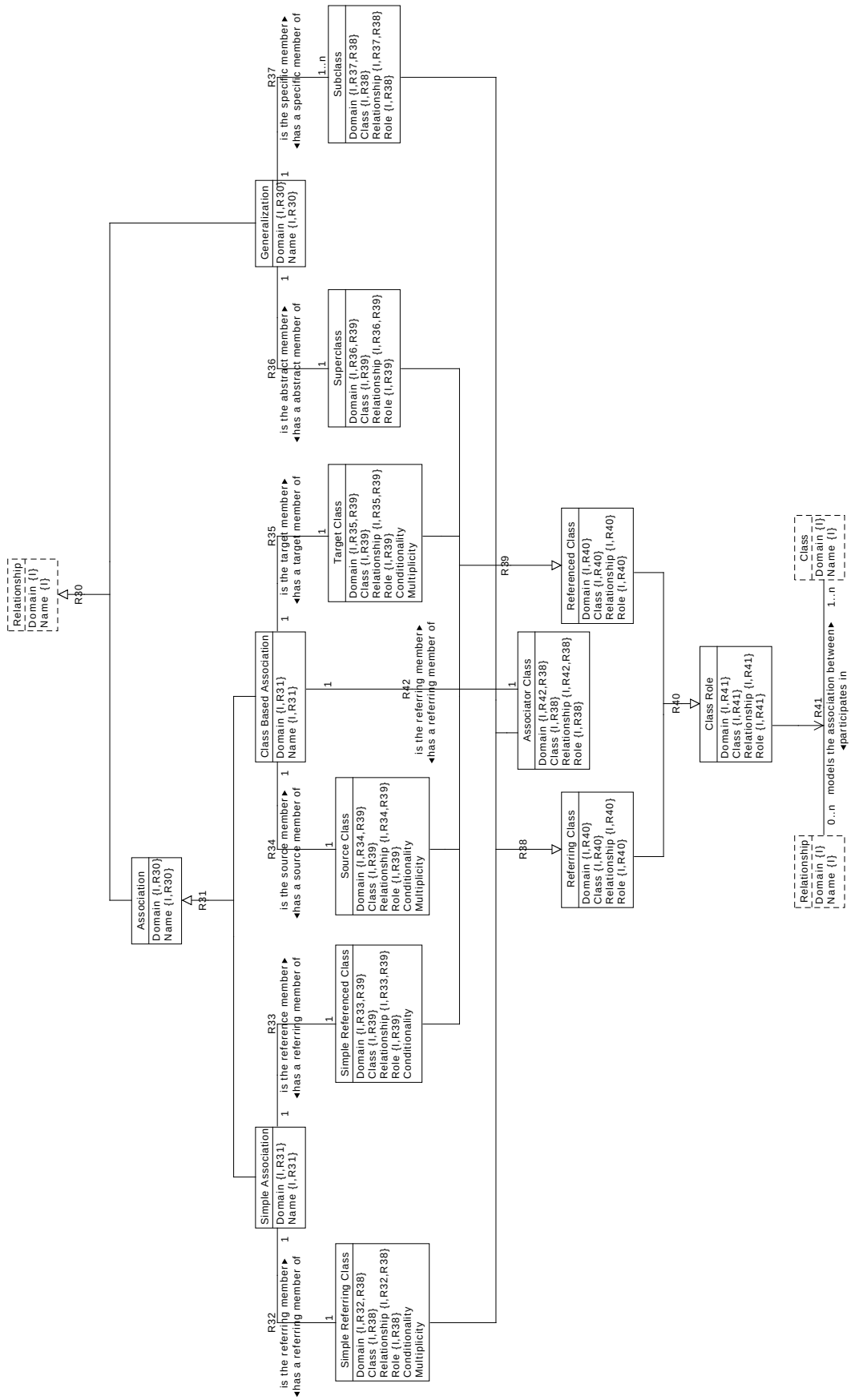

Figure 5.3: Relationship Configuration Class Diagram

A **Relationship** is either an **Association** or a **Generalization** (**R30**). A **Generalization** has one **Superclass** (**R36**) and one or more **Subclass** (**R36**). In practice we will insist upon two or more subclasses.

An **Association** also comes in two types, a **Simple Association** or a **Class Based Association** (**R31**). For a **Simple Association**, there are two participants (**R32** and **R33**). The **Simple Referring Class** is the class that contains referential attributes and the **Simple Referenced Class** contains the identifying attributes that are referenced.

A **Class Based Association** uses an **Associator Class** (**R42**) as the referring class. Its referential attributes refer to identifiers in the **Source Class** (**R34**) and **Target Class** (**R35**).

All these distinctions are variations on the **Class Role** that is played by a **Class** when participating in a **Relationship** (**R41**). A class is either a **Referring Class** or a **Referenced Class** (**R40**). Classes that serve the referring role are the **Simple Referring Class**, **Associator Class** and the **Subclass** (**R38**). The referenced role is played by **Simple Referenced Class**, **Source Class**, **Target Class** and **Superclass** (**R39**).

Following our familiar pattern, we express the data model for relationships in relvars as follows.

```
<<config data>>=
relvar create Association {
    Domain      string
    Name        string
} {Domain Name}
relvar create Generalization {
    Domain      string
    Name        string
} {Domain Name}
relvar partition R30 Relationship {Domain Name}\
    Association {Domain Name}\
    Generalization {Domain Name}
relvar create SimpleAssociation {
    Domain      string
    Name        string
} {Domain Name}
relvar create ClassBasedAssociation {
    Domain      string
    Name        string
} {Domain Name}
relvar partition R31 Association {Domain Name}\
    SimpleAssociation {Domain Name}\
    ClassBasedAssociation {Domain Name}
relvar create SimpleReferringClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
    Conditionality  boolean
    Multiplicity    boolean
} {Domain Class Relationship Role}
relvar association R32\
    SimpleReferringClass {Domain Relationship} 1\
    SimpleAssociation {Domain Name} 1
relvar create SimpleReferencedClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
    Conditionality  boolean
} {Domain Class Relationship Role}
relvar association R33\
    SimpleReferencedClass {Domain Relationship} 1\
    SimpleAssociation {Domain Name} 1
relvar create SourceClass {
    Domain          string
```

```
    Class           string
    Relationship    string
    Role            string
    Conditionality  boolean
    Multiplicity    boolean
} {Domain Class Relationship Role}
relvar association R34\
    SourceClass {Domain Relationship} 1\
    ClassBasedAssociation {Domain Name} 1
relvar create TargetClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
    Conditionality  boolean
    Multiplicity    boolean
} {Domain Class Relationship Role}
relvar association R35\
    TargetClass {Domain Relationship} 1\
    ClassBasedAssociation {Domain Name} 1
relvar create AssociatorClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
} {Domain Class Relationship Role}
relvar association R42\
    AssociatorClass {Domain Relationship} 1\
    ClassBasedAssociation {Domain Name} 1
relvar create Superclass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
} {Domain Class Relationship Role}
relvar association R36\
    Superclass {Domain Relationship} 1\
    Generalization {Domain Name} 1
relvar create Subclass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
} {Domain Class Relationship Role}
relvar association R37\
    Subclass {Domain Relationship} +\
    Generalization {Domain Name} 1
relvar create ReferringClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
} {Domain Class Relationship Role}
relvar partition R38 ReferringClass {Domain Class Relationship Role}\
    SimpleReferringClass {Domain Class Relationship Role}\
    AssociatorClass {Domain Class Relationship Role}\
    Subclass {Domain Class Relationship Role}
relvar create ReferencedClass {
    Domain          string
    Class           string
    Relationship    string
    Role            string
```

```
} {Domain Class Relationship Role}
relvar partition R39 ReferencedClass {Domain Class Relationship Role}\
    SimpleReferencedClass {Domain Class Relationship Role}\
    SourceClass {Domain Class Relationship Role}\
    TargetClass {Domain Class Relationship Role}\
    Superclass {Domain Class Relationship Role}
relvar create ClassRole {
    Domain          string
    Class           string
    Relationship    string
    Role            string
} {Domain Class Relationship Role}
relvar partition R40 ClassRole {Domain Class Relationship Role}\
    ReferringClass {Domain Class Relationship Role}\
    ReferencedClass {Domain Class Relationship Role}
relvar correlation R41 ClassRole\
    {Domain Relationship} + Relationship {Domain Name}\
    {Domain Class} * Class {Domain Name}
relvar correlation R11 ReferencedIdAttribute\
    {Domain Class Relationship Role} +\
        ReferencedClass {Domain Class Relationship Role}\
    {Domain Class Attribute Number} *\
        IdentifyingAttribute {Domain Class Attribute Number}
relvar correlation R12 AttributeReference\
    {Domain ReferringClass Relationship ReferringRole} +\
        ReferringClass {Domain Class Relationship Role}\
    {Domain ReferencedClass Relationship ReferencedRole ReferencedAttribute\
            ReferencedIdNumber} +\
        ReferencedIdAttribute {Domain Class Relationship Role Attribute Number}
```

Association

The `association` command is used to define both simple and class based associations depending upon arguments.

`association` *name source spec target ?option value ... ?*

`name`
The name of the association. Conventionally, relationships names are of the form **R<d>** where <*d*> is one or more decimal digits but any non-empty string can be used.

`source`
The name of the class that defines the starting class for a forward traversal of the relationship.

`spec`
The specifier of the relationship conditionality and multiplicity. The *spec* argument is one of the following strings. We allow two forms for some of the specifiers. The first form follows UML notation and the second form follows TclRAL notation.

`1--1`
The association is *one to one*.

`0..1--1` or `?--1`
The association is *at most one to one*.

`0..1--0..1` or `?--?`
The association is *at most one to at most one*.

`1..*--1` or `+--1`
The association is *one or more to one*.

`0..*--1` or `*--1`
The association is *zero or more to one*.

`1..*--0..1` or `+--?`
The association is *one or more to at most one*.

`0..*--0..1` or `*--?`
The association is *zero or more to at most one*.

`1..*--1..*` or `+--+`
The association is *one or more to one or more*.

`0..*--1..*` or `*--+`
The association is *zero or more to one or more*.

`1..*--0..*` or `+--*`
The association is *one or more to zero or more*.

`0..*--0..*` or `*--*`
The association is *zero or more to zero or more*.

`target`
The name of the class that defines the ending class for a forward traversal of the relationship.

`option value ...`
Options to the `association` command are given as argument option / value pairs: Valid options and their values are:

`-associator` *`class`*
The `-associator` option define the association to be **class based** and specifies *class* as the association class.

`-path` *`attribute_list`*
The `-path` option defines the direction of reference for class based reflexive associations. The *attribute_list* argument is a list of attribute name pairs that define the attribute correspondence when traversing the reflexive association in the forward direction.

In the command, we are using the *spec* string to encode several distinct pieces of information. Each different association specifier implies the conditionality, multiplicity and other properties of the association. We encode those properties in data whose identifier is the *spec* string itself.

```
<<config data>>=
relvar create Config_AssocSpec {
    Spec                string
    NeedsAssociator     boolean
    ReflexiveAllowed    boolean
    ReferringCond       boolean
    ReferringMult       boolean
    ReferencedCond      boolean
    ReferencedMult      boolean
} Spec

relvar insert Config_AssocSpec {
    Spec 1--1 NeedsAssociator false ReflexiveAllowed true
        ReferringCond false ReferringMult false
        ReferencedCond false ReferencedMult false
} {
    Spec 0..1--1 NeedsAssociator false ReflexiveAllowed false
        ReferringCond true ReferringMult false
        ReferencedCond false ReferencedMult false
} {
    Spec 0..1--0..1 NeedsAssociator false ReflexiveAllowed true
        ReferringCond true ReferringMult false
        ReferencedCond true ReferencedMult false
} {
    Spec 1..*--1 NeedsAssociator false ReflexiveAllowed true
        ReferringCond false ReferringMult true
        ReferencedCond false ReferencedMult false
} {
    Spec 0..*--1 NeedsAssociator false ReflexiveAllowed false
        ReferringCond true ReferringMult true
        ReferencedCond false ReferencedMult false
} {
    Spec 1..*--0..1 NeedsAssociator false ReflexiveAllowed false
        ReferringCond false ReferringMult true
        ReferencedCond true ReferencedMult false
} {
    Spec 0..*--0..1 NeedsAssociator false ReflexiveAllowed true
        ReferringCond true ReferringMult true
        ReferencedCond true ReferencedMult false
} {
    Spec 1..*--1..* NeedsAssociator true ReflexiveAllowed true
        ReferringCond false ReferringMult true
        ReferencedCond false ReferencedMult true
} {
    Spec 0..*--1..* NeedsAssociator true ReflexiveAllowed false
        ReferringCond true ReferringMult true
        ReferencedCond false ReferencedMult true
} {
    Spec 1..*--0..* NeedsAssociator true ReflexiveAllowed false
        ReferringCond false ReferringMult true
        ReferencedCond true ReferencedMult true
} {
    Spec 0..*--0..* NeedsAssociator true ReflexiveAllowed true
        ReferringCond true ReferringMult true
        ReferencedCond true ReferencedMult true
} {
    Spec ?--1 NeedsAssociator false ReflexiveAllowed false
        ReferringCond true ReferringMult false
        ReferencedCond false ReferencedMult false
} {
    Spec ?--? NeedsAssociator false ReflexiveAllowed true
        ReferringCond true ReferringMult false
```

```
        ReferencedCond true ReferencedMult false
} {
    Spec +--1 NeedsAssociator false ReflexiveAllowed true
        ReferringCond false ReferringMult true
        ReferencedCond false ReferencedMult false
} {
    Spec *--1 NeedsAssociator false ReflexiveAllowed false
        ReferringCond true ReferringMult true
        ReferencedCond false ReferencedMult false
} {
    Spec +--? NeedsAssociator false ReflexiveAllowed false
        ReferringCond false ReferringMult true
        ReferencedCond true ReferencedMult false
} {
    Spec *--? NeedsAssociator false ReflexiveAllowed true
        ReferringCond true ReferringMult true
        ReferencedCond true ReferencedMult false
} {
    Spec +--+ NeedsAssociator true ReflexiveAllowed true
        ReferringCond false ReferringMult true
        ReferencedCond false ReferencedMult true
} {
    Spec *--+ NeedsAssociator true ReflexiveAllowed false
        ReferringCond true ReferringMult true
        ReferencedCond false ReferencedMult true
} {
    Spec +--* NeedsAssociator true ReflexiveAllowed false
        ReferringCond false ReferringMult true
        ReferencedCond true ReferencedMult true
} {
    Spec *--* NeedsAssociator true ReflexiveAllowed true
        ReferringCond true ReferringMult true
        ReferencedCond true ReferencedMult true
}
```

The consequence of the decision to specify separately the realization of relationships as attribute references from the specification of the participants, conditionality and multiplicity of the association is that we will have to hold, temporarily, some data about associations until all of the classes are defined. Then we will bind the attribute references to the relationship characteristics. This is done below. For now we show the data that must be kept aside to perform the binding of the attribute references to the corresponding associations.

```
<<config data>>=
relvar create Config_ClassAssoc {
    Domain              string
    Relationship        string
    AssocClass          string
    SourceClass         string
} {Domain Relationship AssocClass SourceClass}
relvar create Config_NonReflexiveAssoc {
    Domain              string
    Relationship        string
    AssocClass          string
    SourceClass         string
    TargetClass         string
} {Domain Relationship AssocClass SourceClass}
relvar create Config_ReflexiveAssoc {
    Domain              string
    Relationship        string
    AssocClass          string
    SourceClass         string
} {Domain Relationship AssocClass SourceClass}
relvar create Config_ReflexivePath {
```

```
    Domain              string
    Relationship        string
    AssocClass          string
    SourceClass         string
    SourceAttr          string
    TargetAttr          string
} {Domain Relationship AssocClass SourceClass SourceAttr}
relvar partition Config1\
    Config_ClassAssoc\
        {Domain Relationship AssocClass SourceClass}\
    Config_NonReflexiveAssoc\
        {Domain Relationship AssocClass SourceClass}\
    Config_ReflexiveAssoc\
        {Domain Relationship AssocClass SourceClass}
relvar association Config2\
    Config_ReflexivePath {Domain Relationship AssocClass SourceClass} +\
    Config_ReflexiveAssoc {Domain Relationship AssocClass SourceClass} 1
```

Implementation

Like most of the commands in the configuration language, the bulk of the code is involved with populating the relvars holding the essential data provided in the command arguments. For the `association` command, there is some argument parsing to determine whether the association is simple or class based. Then it is a matter of filling in the correct information based on the type of the association.

```
<<domain config commands>>=
proc association {name source spec target args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME association
    }
    if {[string index $name 0] eq "~"} {
        tailcall DeclError TILDE_NAME $name
    }

    # Obtain references to the domain and class context.
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    <<association: parse arguments>>

    try {
        relvar insert ::rosea::Config::DomainElement [list\
            Domain  $DomainName\
            Element $name\
        ]
    } trap {RAL relvar insert DUPLICATE_TUPLE} {result opts} {
        tailcall DeclError DUP_ELEMENT_NAME $name
    }

    # Many relvars have tuples with the same heading, so we construct it
    # once here.
    set reltuple [list\
        Domain  $DomainName\
        Name $name\
    ]

    # Populate the data for a Relationship and Association since that
```

```
    # is what this command defines.
    relvar insert ::rosea::Config::Relationship $reltuple
    relvar insert ::rosea::Config::Association $reltuple

    # Populate the type of association we are dealing with.
    if {$associator eq {}} {
        <<association: populate simple association>>
    } else {
        <<association: populate class based association>>
    }
}
```

```
<<error code formats>>=
TILDE_NAME {names beginning with the tilde character are not allowed\
    in this context, "%s"}
```

Argument parsing uses a simple `switch` command arrangement. We use two variables to hold the option values.

```
<<association: parse arguments>>=
set associator {}
set path {}

foreach {option value} $args {
    switch -exact -- $option {
        -associator {
            set associator $value
        }
        -path {
            set path $value
            if {[llength $path] % 2 != 0} {
                tailcall DeclError ARG_FORMAT $path
            }
        }
        default {
            tailcall DeclError UNKNOWN_OPTION association $option
        }
    }
}
<<association: check arguments>>
```

Here we use the data that we supplied above to make sure the `spec` association specifier and the supplied arguments make sense together. Certain forms of reflexiveness are not allowed and certain specifiers imply that you must define a class based association. Such rules are easier to express in data rather than long sequences of checking code.

```
<<association: check arguments>>=
set cas [relvar restrictone ::rosea::Config::Config_AssocSpec Spec $spec]
if {[relation isempty $cas]} {
    tailcall DeclError BAD_RELATIONSHIP_SPEC $spec
}
set needsassoc [relation extract $cas NeedsAssociator]
if {$needsassoc && $associator eq {}} {
    tailcall DeclError NEED_ASSOCIATOR $spec
}
if {$source eq $target} {
    if {![relation extract $cas ReflexiveAllowed]} {
        tailcall DeclError REFLEXIVE_NOT_ALLOWED $spec
    }
    if {$needsassoc && $path eq {}} {
        tailcall DeclError NEED_REFLEXIVE_PATH $name $source $spec $target
    }
}
```

```
<<error code formats>>=
BAD_RELATIONSHIP_SPEC  {bad relationship specifier, "%s"}
NEED_ASSOCIATOR {relationship of type, "%s", requires associative class}
NEED_REFLEXIVE_PATH {traversal path must be specified for reflexive\
        association, %s, %s %s %s}
REFLEXIVE_NOT_ALLOWED   {associations of type, "%s", cannot be reflexive}
```

For simple associations, the `association` command argument supply what we need and we perform a series of inserts into the configuration language data model.

```
<<association: populate simple association>>=
relvar insert ::rosea::Config::SimpleAssociation $reltuple
relvar insert ::rosea::Config::SimpleReferringClass [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
    Conditionality  [relation extract $cas ReferringCond]\
    Multiplicity    [relation extract $cas ReferringMult]\
]
relvar insert ::rosea::Config::ReferringClass [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
]
relvar insert ::rosea::Config::ClassRole [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
]
relvar insert ::rosea::Config::SimpleReferencedClass [list\
    Domain          $DomainName\
    Class           $target\
    Relationship    $name\
    Role            target\
    Conditionality  [relation extract $cas ReferencedCond]\
]
relvar insert ::rosea::Config::ReferencedClass [list\
    Domain          $DomainName\
    Class           $target\
    Relationship    $name\
    Role            target\
]
relvar insert ::rosea::Config::ClassRole [list\
    Domain          $DomainName\
    Class           $target\
    Relationship    $name\
    Role            target\
]
```

For class based associations, the situation is more complex. We need to hold information about whether the relationship is reflexive. This is used later to match up the referential attributes that are used to realize the relationship to the class participants. Other than this complication, it is more of inserting the command argument data into the proper set of configuration language relvars.

```
<<association: populate class based association>>=
relvar insert ::rosea::Config::Config_ClassAssoc [list\
    Domain              $DomainName\
    Relationship        $name\
```

```
    AssocClass          $associator\
    SourceClass         $source\
]
if {[llength $path] == 0} {
    relvar insert ::rosea::Config::Config_NonReflexiveAssoc [list\
        Domain              $DomainName\
        Relationship        $name\
        AssocClass          $associator\
        SourceClass         $source\
        TargetClass         $target\
    ]
} else {
    relvar insert ::rosea::Config::Config_ReflexiveAssoc [list\
        Domain              $DomainName\
        Relationship        $name\
        AssocClass          $associator\
        SourceClass         $source\
    ]
    foreach {sourceattr targetattr} $path {
        relvar insert ::rosea::Config::Config_ReflexivePath [list\
            Domain              $DomainName\
            Relationship        $name\
            AssocClass          $associator\
            SourceClass         $source\
            SourceAttr          $sourceattr\
            TargetAttr          $targetattr\
        ]
    }
}

relvar insert ::rosea::Config::ClassBasedAssociation $reltuple
relvar insert ::rosea::Config::SourceClass [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
    Conditionality  [relation extract $cas ReferringCond]\
    Multiplicity    [relation extract $cas ReferringMult]\
]
relvar insert ::rosea::Config::ReferencedClass [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
]
relvar insert ::rosea::Config::ClassRole [list\
    Domain          $DomainName\
    Class           $source\
    Relationship    $name\
    Role            source\
]
relvar insert ::rosea::Config::TargetClass [list\
    Domain          $DomainName\
    Class           $target\
    Relationship    $name\
    Role            target\
    Conditionality  [relation extract $cas ReferencedCond]\
    Multiplicity    [relation extract $cas ReferencedMult]\
]
relvar insert ::rosea::Config::ReferencedClass [list\
    Domain          $DomainName\
    Class           $target\
```

```
    Relationship    $name\
    Role            target\
]
relvar insert ::rosea::Config::ClassRole [list\
    Domain          $DomainName\
    Class           $target\
    Relationship    $name\
    Role            target\
]
relvar insert ::rosea::Config::AssociatorClass [list\
    Domain          $DomainName\
    Class           $associator\
    Relationship    $name\
    Role            associator\
]
relvar insert ::rosea::Config::ReferringClass [list\
    Domain          $DomainName\
    Class           $associator\
    Relationship    $name\
    Role            associator\
]
relvar insert ::rosea::Config::ClassRole [list\
    Domain          $DomainName\
    Class           $associator\
    Relationship    $name\
    Role            associator\
]
```

After defining the association properties, we can determine if we have a class based association. In the case of a class based association, we need to patch up the roles that the classes have. In the `reference` command, we simply assumed that all associations were simple. This assumption lets us separate the idea of defining the association and its characteristics from the idea of specifying the attribute references that realize the association. We do this separation to make translation a bit easier and more directly related to the nature of the UML graphical representation, *i.e.* we have separate commands for the association line and the attribute annotation.

The code below is executed at the end of the `domain` command to correct the **AttributeReference** relvar values for any associations that were defined to be class based. We present it here, to show how the data stored away during the `association` command is used.

First, we consider the non-reflexive case. Here we must patch the role that the association class plays in attribute references and the role that the source class plays.

```
<<domain: bind association references>>=
variable Config_NonReflexiveAssoc

relation foreach nra $Config_NonReflexiveAssoc {
    relation assign $nra
    relvar update ::rosea::Config::AttributeReference ar {
        [tuple extract $ar Domain] eq $Domain &&
        [tuple extract $ar Relationship] eq $Relationship &&
        [tuple extract $ar ReferringClass] eq $AssocClass
    } {tuple update $ar ReferringRole associator}

    relvar update ::rosea::Config::AttributeReference ar {
        [tuple extract $ar Domain] eq $Domain &&
        [tuple extract $ar Relationship] eq $Relationship &&
        [tuple extract $ar ReferringClass] eq $AssocClass &&
        [tuple extract $ar ReferencedClass] eq $SourceClass
    } {tuple update $ar ReferencedRole source}
}
```

Considering the reflexive case, we have to resolve the inherent ambiguity that results from the fact that the associative class refers to two instances of the same class. The `-path` option shows which of the attribute references is considered the source of a forward traversal.

```
<<domain: bind association references>>=
variable Config_ReflexivePath

relation foreach rp $Config_ReflexivePath {
    relation assign $rp
    relvar update ::rosea::Config::AttributeReference ar {
        [tuple extract $ar Domain] eq $Domain &&
        [tuple extract $ar Relationship] eq $Relationship &&
        [tuple extract $ar ReferringClass] eq $AssocClass
    } {tuple update $ar ReferringRole associator}

    relvar update ::rosea::Config::AttributeReference ar {
        [tuple extract $ar Domain] eq $Domain &&
        [tuple extract $ar Relationship] eq $Relationship &&
        [tuple extract $ar ReferringClass] eq $AssocClass &&
        [tuple extract $ar ReferringAttribute] eq $SourceAttr
    } {tuple update $ar ReferencedRole source}
}
```

Finally, we have all the information necessary to create the required instances of **ReferencedIdAttribute**. We do that by evaluating a preformulated query.

```
<<domain: bind association references>>=
variable refIdQuery

eval $refIdQuery
```

The purpose of the `refIdQuery` is to create the instances of **ReferenceIdAttribute** that correlate identifying attributes to the referenced class in which they reside. Recall, that when a class refers to another class to realize a relationship, the referential attributes of the referring class have the same values as a set of identifying attributes in the referenced class. The combination of the `reference` and `association` commands have given us the necessary information about those references. What we do here is create instances that correlate those references back to identifiers of the referenced class. The result will be to make sure that the attribute names were correctly specified. If they weren't, then the constraint check that occurs at the end of the domain configuration will fail.

All the data we need to create the instances of **ReferencedIdAttribute** is contained in the instances of **AttributeReference** that we have been so carefully creating. The query itself takes the **AttributeReference** instances and simply projects out the part we need, renames the attributes appropriately and adds in the tuples to the **ReferencedIdAttribute** relvar value.

```
<<config data>>=
pipe {
    relvar set ::rosea::Config::AttributeReference |
    relation project ~ Domain ReferencedClass Relationship ReferencedRole\
            ReferencedAttribute ReferencedIdNumber |
    relation rename ~ ReferencedClass Class ReferencedRole Role\
            ReferencedAttribute Attribute ReferencedIdNumber Number |
    relvar union ::rosea::Config::ReferencedIdAttribute
} refIdQuery
```

Tests

```
<<config command tests>>=
test configure-3.0 {
    Define simple association
} -setup {
```

```
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class x {
                attribute a1 string -id 1
                attribute a2 string -default 20
            }
            class y {
                attribute a1 string -id 1
                reference R1 x -link a1
            }
            association R1 y 1--1 x
        }
    }
    relation cardinality [relvar set ::rosea::Config::SimpleAssociation]
} -result {1}
```

```
<<config command tests>>=
test configure-3.1 {
    Define reflexive simple association
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class x {
                attribute a1 string -id 1
                attribute prev string -id 2
                attribute a2 string -default 20
                reference R1 x -link {prev a1}
            }
            association R1 x 1--1 x
        }
    }
    relation cardinality [relvar set ::rosea::Config::SimpleAssociation]
} -result {1}
```

```
<<config command tests>>=
test configure-3.2 {
    Define class based association
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class a {
                attribute x1 string -id 1
                attribute y1 string -id 1
                reference R1 x -link x1
                reference R1 y -link y1
            }
            class x {
                attribute x1 string -id 1
                attribute x2 string -default 20
            }
            class y {
                attribute y1 string -id 1
```

```
            }
            association R1 x 1..*--1..* y -associator a
        }
    }
    relation cardinality [relvar set ::rosea::Config::ClassBasedAssociation]
} -result {1}
```

```
<<config command tests>>=
test configure-3.3 {
    Define reflexive class based association
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class a {
                attribute x1 string -id 1
                attribute next string -id 1
                reference R1 x -link x1 -link {next x1}
            }
            class x {
                attribute x1 string -id 1
                attribute x2 string -default 20
            }
            association R1 x 1..*--1..* x -associator a -path {x1 next}
        }
    }
    relation cardinality [relvar set ::rosea::Config::ClassBasedAssociation]
} -result {1}
```

```
<<config command tests>>=
test configure-3.4 {
    Simple association -- bad class
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            association R1 y 1--1 x
        }
    }
} -result {encountered 2 configuration script errors} -returnCodes error
```

```
<<config command tests>>=
test configure-3.5 {
    Simple association -- need associator
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            association R1 y 1..*--1..* x
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error
```

Generalization

Defining a generalization relationship is quite a bit simpler than the effort we just saw in defining associations. For generalizations, it is necessary to specify the superclass and the set of subclasses.

`generalization` *name super sub1 sub2 ?... ?*

`name`
The name of the generalization. Conventionally, relationships names are of the form **R<d>** where *<d>* is one or more decimal digits but any non-empty string can be used.

`super`
The name of the class that serves as the superclass of the generalization.

`subN`
The names of the classes that serve as the subclasses of the generalization. You must have at least two subclasses to define a generalization.

Implementation

We show the implementation of the `generalization` command with little additional commentary. It consists primarily of inserting the values of the command arguments into the configuration relvars that define generalization relationship semantics.

```
<<domain config commands>>=
proc generalization {name super args} {
    if {[llength $args] < 2} {
        tailcall DclError TOO_FEW_SUBCLASSES [llength $args]
    }
    if {$name eq {}} {
        tailcall DclError EMPTY_NAME generalization
    }
    if {[string index $name 0] eq "~"} {
        tailcall DclError TILDE_NAME $name
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName

    try {
        relvar insert ::rosea::Config::DomainElement [list\
            Domain  $DomainName\
            Element $name\
        ]
    } trap {RAL relvar insert DUPLICATE_TUPLE} {result opts} {
        tailcall DclError DUP_ELEMENT_NAME $name
    }

    set reltuple [list\
        Domain  $DomainName\
        Name $name\
    ]
    relvar insert ::rosea::Config::Relationship $reltuple
    relvar insert ::rosea::Config::Generalization $reltuple

    set supertuple [list\
        Domain          $DomainName\
        Class           $super\
        Relationship    $name\
        Role            target\
    ]
```

```
    relvar insert ::rosea::Config::Superclass $supertuple
    relvar insert ::rosea::Config::ReferencedClass $supertuple
    relvar insert ::rosea::Config::ClassRole $supertuple

    set subtuple [dict create\
        Domain          $DomainName\
        Relationship    $name\
        Role            source\
    ]
    foreach sub $args {
        dict set subtuple Class $sub
        relvar insert ::rosea::Config::Subclass $subtuple
        relvar insert ::rosea::Config::ReferringClass $subtuple
        relvar insert ::rosea::Config::ClassRole $subtuple
    }
}
```

```
<<error code formats>>=
TOO_FEW_SUBCLASSES  {at least 2 subclasses must be specified, got %d}
```

Tests

```
<<config command tests>>=
test configure-4.0 {
    Define generalization
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class super {
                attribute a1 string -id 1
                attribute a2 string -default 20
            }
            class sub1 {
                attribute a1 string -id 1
                reference R1 super -link a1
            }
            class sub2 {
                attribute a1 string -id 1
                reference R1 super -link a1
            }
            generalization R1 super sub1 sub2
        }
    }
    relation cardinality [relvar set ::rosea::Config::Class]
} -result {3}
```

Defining Class State Models

State models in XUML are used to specify the sequence of computations associated with the life cycle of class instances. All instances of a class have the behavior, yet each instance has it own notion of current state and so may progress through its life cycle independently of any other instance.

Traditionally, Moore type state models are used in XUML to define the life cycle behavior. The other alternative is a Mealy type state model. They are mathematically equivalent in the sense that any problem that can be solved by one formulation can also be

solved by the other. Individuals have their preferences over which formulation is better and we will not indulge in that discussion here. We only point out that hierarchical state models are *not* supported in the this translation scheme. Hierarchical state models are an unnecessary abomination.

As usual, we need a namespace in which to hold the state model definition commands.

```
<<state model config namespace layout>>=
namespace eval StateModelDef {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace import ::rosea::Config::ConfigEvaluate
    namespace import ::rosea::Helpers::DeclError

    <<state model config commands>>
}
```

Statemodel

`statemodel` *body*

body
A Tcl Script that is evaluated in a context to allow the definition of the properties of the class state model.

Implementation

The implementation of the `statemodel` command follows the usual pattern. We evaluate `body` in the proper context and then insert the argument data into the proper relvars that are used to collect the state model specifications. Those relvar are described below.

```
<<class config commands>>=
proc statemodel {body} {
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    variable ClassName

    namespace upvar StateModelDef\
        InitialState InitialState\
        DefaultTrans DefaultTrans\
        Terminals Terminals

    set InitialState {}
    set DefaultTrans {}
    set Terminals [list]

    try {
        ConfigEvaluate [namespace current]::StateModelDef $body

        if {$DefaultTrans eq {}} {
            set DefaultTrans CH
        }
        relvar insert ::rosea::Config::StateModel [list\
            Domain          $DomainName\
            Model           $ClassName\
            InitialState    $InitialState\
            DefaultTrans    $DefaultTrans\
        ]
```

```
        relvar insert ::rosea::Config::InstanceStateModel [list\
            Domain          $DomainName\
            Class           $ClassName\
        ]

        foreach terminal $Terminals {
            relvar updateone ::rosea::Config::State stup [list\
                    Domain $DomainName Model $ClassName Name $terminal] {
                tuple update $stup IsFinal true
            }
        }
    } on error {result opts} {
        log::error $result
        upvar #0 ::rosea::Config::errcount errcount
        incr errcount
    }
}
```

The figure below shows the way that state models are specified.

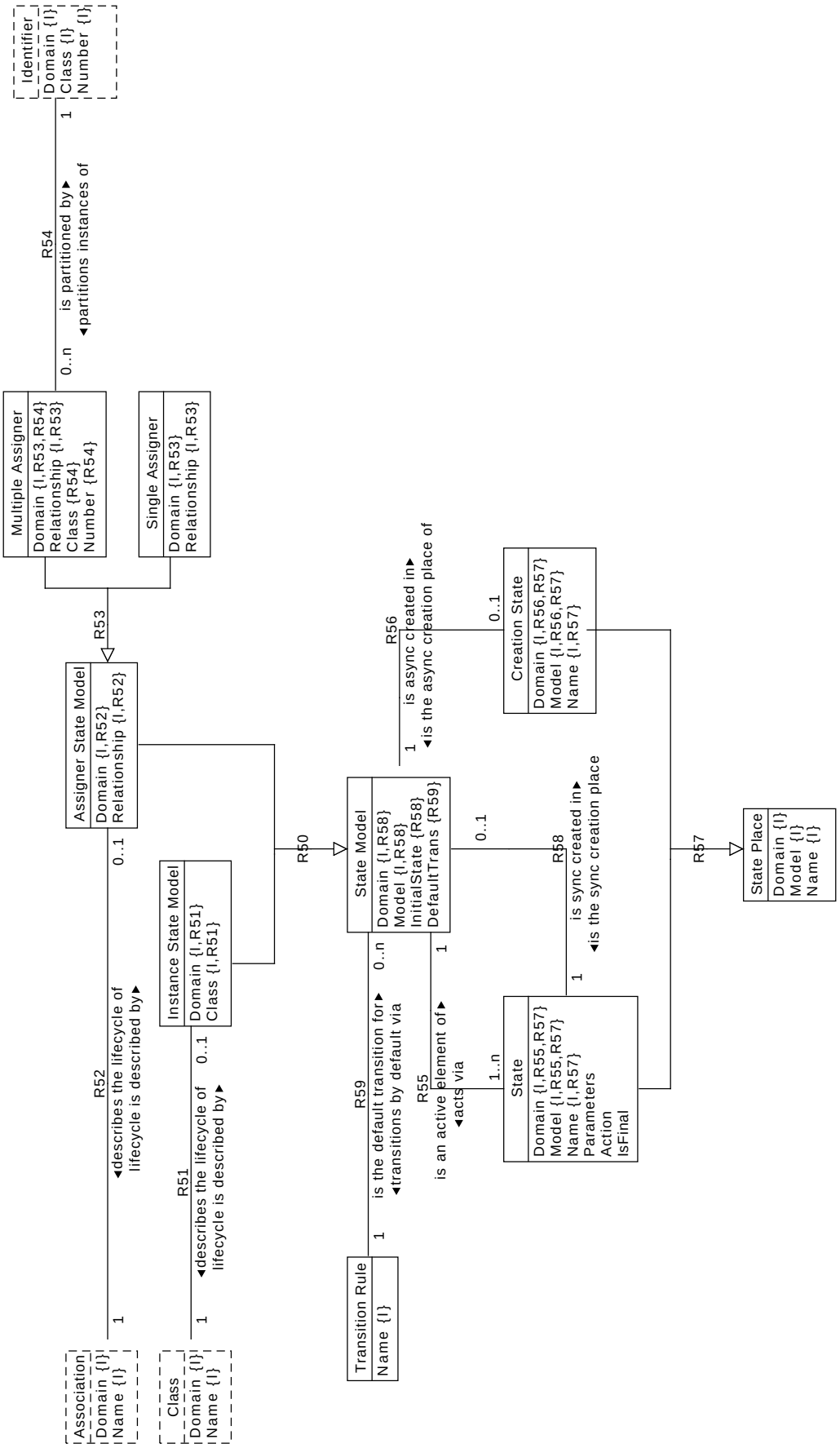

Figure 5.4: State Model Configuration Class Diagram

There are two types of **StateModel** (**R50**). The **Instance State Model** is used for classes (**R51**) and the **Assigner State Model** is defined for associations (**R52**). There are both **Single Assigner** and **Multiple Assigner** state models (**R53**). A **Multiple Assigner** is partitioned by the identifier of a class (**R54**).

A **State Model** is characterized by an **InitialState** for synchronous creation (**R58**) and possibly a **Creation State** used for asynchronous creation (**R56**). Every **State Model** has a set of **State** (**R55**) which are characterized by an **Action** and its **Parameters**. A **State** may also be marked as **IsFinal**. Together the **State** and **Creation State** form a set of **State Place** (**R57**) from which transitions may occur.

Expressing the graphic in TclRAL relvars we obtain:

```
<<config data>>=
relvar create StateModel {
    Domain          string
    Model           string
    InitialState    string
    DefaultTrans    string
} {Domain Model}

relvar create InstanceStateModel {
    Domain          string
    Class           string
} {Domain Class}

relvar association R51\
    InstanceStateModel {Domain Class} ?\
    Class {Domain Name} 1

relvar create AssignerStateModel {
    Domain          string
    Relationship    string
} {Domain Relationship}

relvar association R52\
    AssignerStateModel {Domain Relationship} ?\
    Association {Domain Name} 1

relvar partition R50 StateModel {Domain Model}\
    InstanceStateModel {Domain Class}\
    AssignerStateModel {Domain Relationship}

relvar create SingleAssigner {
    Domain          string
    Relationship    string
} {Domain Relationship}

relvar create MultipleAssigner {
    Domain          string
    Relationship    string
    Class           string
    Number          int
} {Domain Relationship}

relvar partition R53 AssignerStateModel {Domain Relationship}\
    SingleAssigner {Domain Relationship}\
    MultipleAssigner {Domain Relationship}

relvar association R54\
    MultipleAssigner {Domain Class Number} ?\
    Identifier {Domain Class Number} 1

relvar create State {
    Domain          string
```

```
    Model           string
    Name            string
    Parameters      string
    Action          string
    IsFinal         boolean
} {Domain Model Name}

relvar association R55\
    State {Domain Model} +\
    StateModel {Domain Model} 1

relvar association R58\
    StateModel {Domain Model InitialState} ?\
    State {Domain Model Name} 1

relvar create TransitionRule {
    Name    string
} Name

relvar insert TransitionRule {Name IG} {Name CH}

relvar association R59\
    StateModel DefaultTrans *\
    TransitionRule Name 1

relvar create CreationState {
    Domain          string
    Model           string
    Name            string
} {Domain Model Name}

relvar create StatePlace {
    Domain          string
    Model           string
    Name            string
} {Domain Model Name}

relvar partition R57 StatePlace {Domain Model Name}\
    State {Domain Model Name}\
    CreationState {Domain Model Name}
```

State

`state` *name parameters body*

name
The name of the state. Name must not be the empty string or one of the reserved names of @, **CH** or **IG**.

parameters
A list of the formal parameters of the state. Parameters are specified in the same manner as for the `::proc` command.

body
A Tcl script that is to be executed when the class instance enters this state.

Implementation

```
<<state model config commands>>=
proc state {name params body} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME state
    }
    if {$name in {@ CH IG}} {
        tailcall DeclError PSEUDO_STATE $name state
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    variable InitialState

    if {$InitialState eq {}} {
        set InitialState $name
    }

    relvar insert ::rosea::Config::State [list\
        Domain          $DomainName\
        Model           $ClassName\
        Name            $name\
        Parameters      $params\
        Action          $body\
        IsFinal         false\
    ]
    relvar insert ::rosea::Config::StatePlace [list\
        Domain          $DomainName\
        Model           $ClassName\
        Name            $name\
    ]
    return
}
```

Before we can discuss state model transitions, we need to deal with events. The figure below shows the model of events. This model is complicated by the need to handle the rules of polymorphic events. For most state models, events are defined locally and the model responds to signaling those local events. However, polymorphic event can be inherited down one or more hierarchies and the state models that are the leaf classes in a generalization hierarchy respond to the set of both inherited and local events.

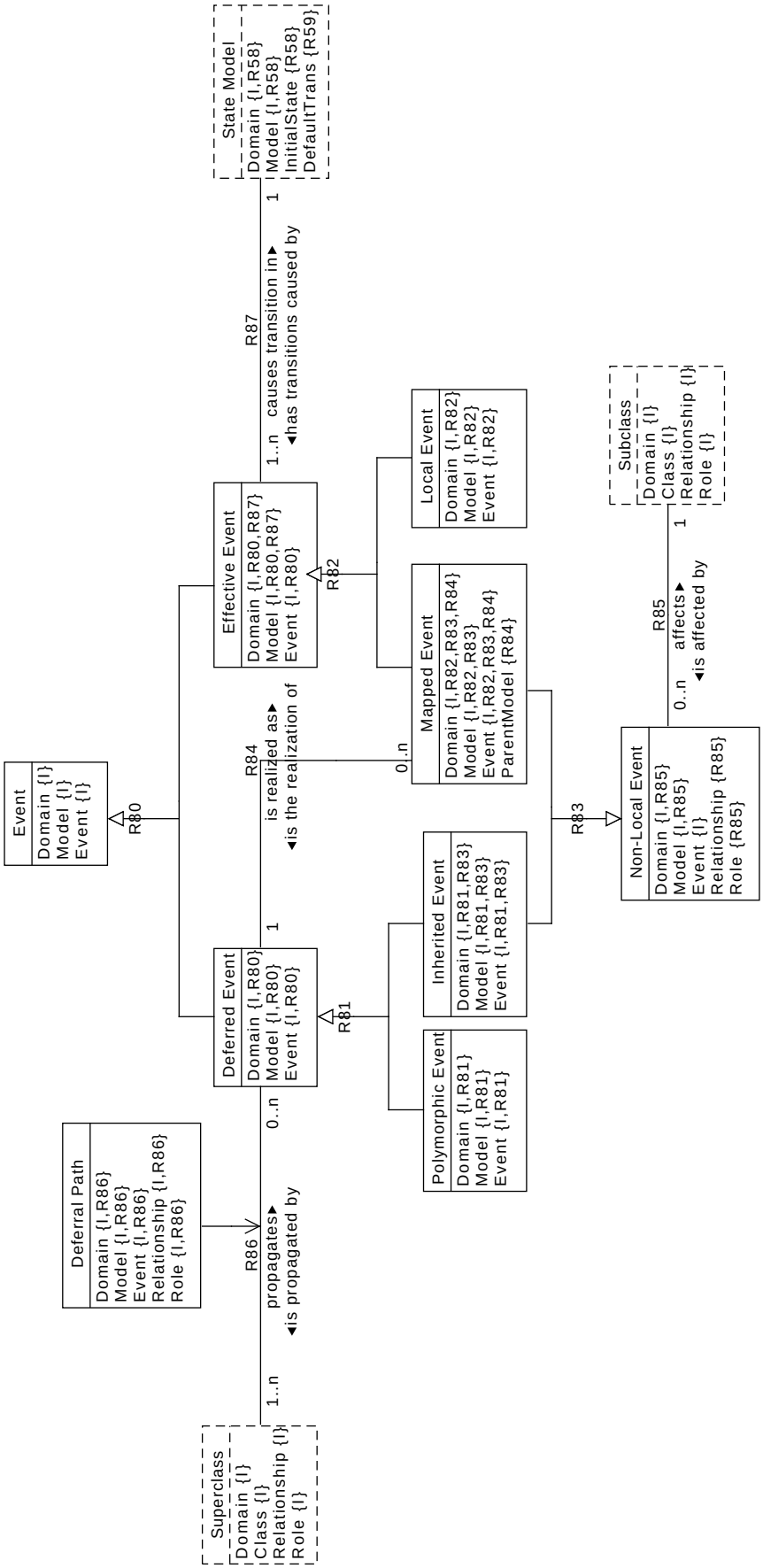

Figure 5.5: Event Configuration Class Diagram

An **Event** is of two types (**R80**), **Effective Event** and **Deferred Event**. Transitions in state models occur only on an **Effective Event**. The concept of a **Deferred Event** is used to handle the polymorphism property. The typing implied by **R80** insures that for a given state model, polymorphic and local events must have distinct names.

An **Effective Event** is either a **Local Event**, defined specifically for the state model, or a **Mapped Event** that has be subjected to a polymorphic mapping (**R82**). **Effective Events** are those which drive transitions in a **State Model** (**R87**). A **Deferred Event** is an event that must eventually be polymorphically mapped (**R84**) and consists of the **Polymorphic Event** defined for a superclass or the **Inherited Event** (**R81**) should the class be subjected to repeated specialization. In all cases a **Deferred Event** is deferred along a **Deferral Path** (**R86**) that originates at a **Superclass**.

Whether a polymorphic event is consumed at the leaf as a **Mapped Event** or passed further down the generalization hierarchy as an **Inherited Event**, both are forms of **Non-Local Event** (**R83**). A **Non-Local Event** can only affect a **Subclass** (**R85**) by the definition of polymorphic events.

These rules are represented in TclRAL relvars and constraints as:

```
<<config data>>=
relvar create Event {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}

relvar create DeferredEvent {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}

relvar create EffectiveEvent {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}

relvar partition R80 Event {Domain Model Event}\
    DeferredEvent {Domain Model Event}\
    EffectiveEvent {Domain Model Event}

relvar create DeferralPath {
    Domain          string
    Model           string
    Event           string
    Relationship    string
    Role            string
} {Domain Model Event Relationship Role}

relvar correlation R86 DeferralPath\
    {Domain Model Event} + DeferredEvent {Domain Model Event}\
    {Domain Model Relationship Role} * Superclass\
        {Domain Class Relationship Role}

relvar create PolymorphicEvent {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}

relvar create InheritedEvent {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}
```

```
relvar partition R81 DeferredEvent {Domain Model Event}\
    PolymorphicEvent {Domain Model Event}\
    InheritedEvent {Domain Model Event}

relvar create MappedEvent {
    Domain      string
    Model       string
    Event       string
    ParentModel string
} {Domain Model Event}

relvar association R84\
    MappedEvent {Domain ParentModel Event} *\
    DeferredEvent {Domain Model Event} 1

relvar create LocalEvent {
    Domain  string
    Model   string
    Event   string
} {Domain Model Event}

relvar partition R82 EffectiveEvent {Domain Model Event}\
    MappedEvent {Domain Model Event}\
    LocalEvent {Domain Model Event}

relvar create NonLocalEvent {
    Domain          string
    Model           string
    Event           string
    Relationship    string
    Role            string
} {Domain Model Event}

relvar partition R83 NonLocalEvent {Domain Model Event}\
    MappedEvent {Domain Model Event}\
    InheritedEvent {Domain Model Event}

relvar association R85\
    NonLocalEvent {Domain Model Relationship Role} *\
    Subclass {Domain Class Relationship Role} 1

relvar association R87\
    EffectiveEvent {Domain Model} +\
    StateModel {Domain Model} 1
```

Compared to events, the data needed to dispatch state model transitions is much simpler. The figure below shows the class model for transitions.

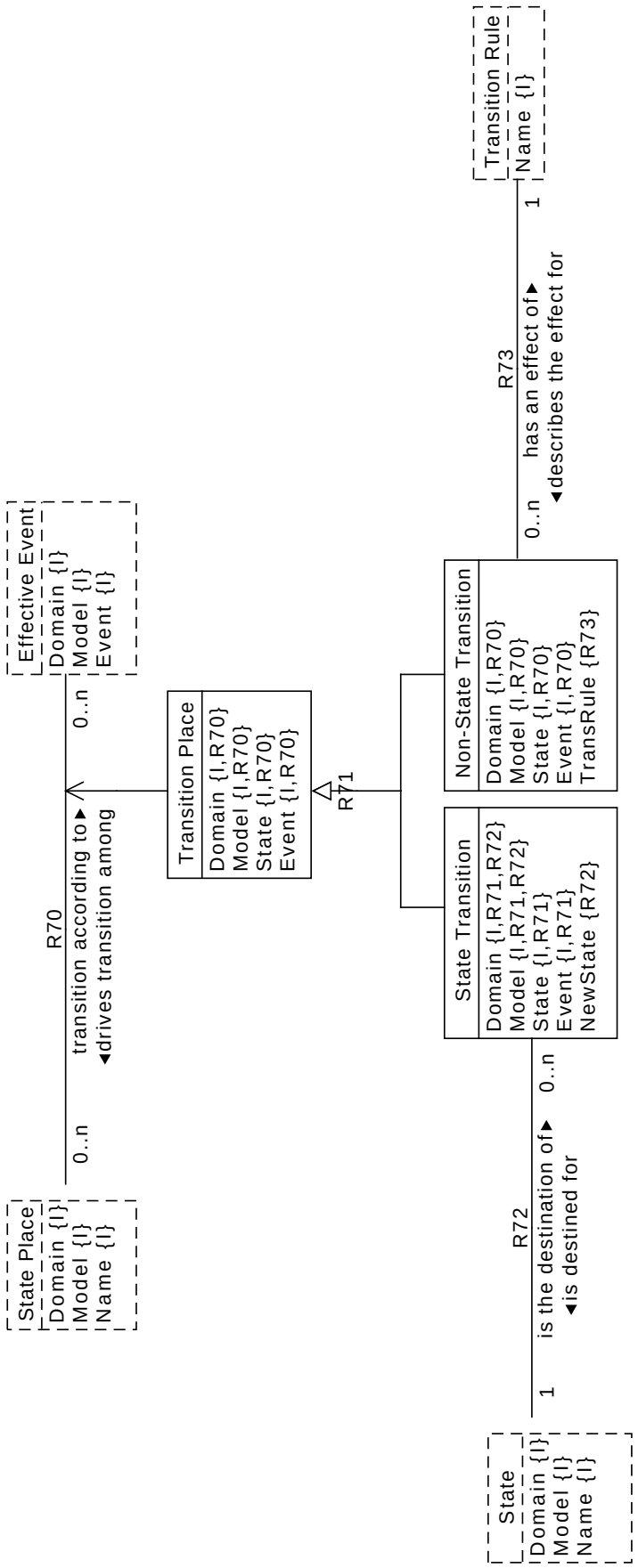

Figure 5.6: Transitions Configuration Class Diagram

The **Transition Place** models the cell of a state transition matrix as it is the correlation between **State Place** and **Effective Event** (**R70**). A **Transition Place** can be a **State Transition** or a **Non-State Transition** (**R71**). This depends upon whether we transition to a **State** (**R72**) or are one of the non-transitioning actions given by a **Transition Rule** (**R73**).

```
<<config data>>=
relvar create TransitionPlace {
    Domain  string
    Model   string
    State   string
    Event   string
} {Domain Model State Event}

relvar correlation R70 TransitionPlace\
    {Domain Model State} * StatePlace {Domain Model Name}\
    {Domain Model Event} * EffectiveEvent {Domain Model Event}

relvar create StateTransition {
    Domain      string
    Model       string
    State       string
    Event       string
    NewState    string
} {Domain Model State Event}

relvar create NonStateTransition {
    Domain      string
    Model       string
    State       string
    Event       string
    TransRule   string
} {Domain Model State Event}

relvar partition R71 TransitionPlace {Domain Model State Event}\
    StateTransition {Domain Model State Event}\
    NonStateTransition {Domain Model State Event}

relvar association R72\
    StateTransition {Domain Model NewState} *\
    State {Domain Model Name} 1

relvar association R73\
    NonStateTransition TransRule *\
    TransitionRule Name 1
```

Transition

`transition` *source – event –> target*

`source`
The name of a state in the state model being defined or the special reserved name, `@`.

`event`
The name of an event that causes the transition.

`target`
The name of a state in the state model being defined or one of the special non-transitioning states, `IG` or `CH`.

Implementation

The implementation of the `transition` command consists mainly of inserting tuples into the event relvars using the data from the command arguments. Note however, that all the events defined by invoking `transition` are deemed to be a **LocalEvent**. This certainly may not be true for classes that are leaf subclass of a generalization hierarchy. When the domain configuration is completed we will propagate the polymorphic events down the generalization hierarchies and in that process recategorize any inherited events properly.

One other minor concern is dealing with the initial pseudo-state, `@`. This state is where an instance resides if it has been created asynchronously. One may not define the `@` state in a `state` command (it cannot have an activity) and the only valid place it can appear is as the `source` state in a `transition` command.

```
<<state model config commands>>=
proc transition {source - event -> target} {
    if {$event eq {}} {
        tailcall DeclError EMPTY_NAME event
    }
    if {$source in {CH IG}} {
        tailcall DeclError PSEUDO_STATE $name "transition source state"
    }
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    if {$source eq "@"} {
        if {$target in {CH IG}} {
            tailcall DeclError BAD_CREATION_TARGET $target
        }
        set cstuple [list\
            Domain          $DomainName\
            Model           $ClassName\
            Name            @\
        ]
        relvar uinsert ::rosea::Config::CreationState $cstuple
        relvar uinsert ::rosea::Config::StatePlace $cstuple
    }

    set eventtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        Event   $event\
    ]
    relvar uinsert ::rosea::Config::Event $eventtuple ; # ❶
    relvar uinsert ::rosea::Config::EffectiveEvent $eventtuple
    relvar uinsert ::rosea::Config::LocalEvent $eventtuple

    set tranrule [relvar restrictone ::rosea::Config::TransitionRule\
        Name $target]
    set transtuple [list\
        Domain  $DomainName\
        Model   $ClassName\
        State   $source\
        Event   $event\
    ]
    relvar insert ::rosea::Config::TransitionPlace $transtuple

    if {[relation isnotempty $tranrule]} {
        lappend transtuple TransRule $target
        set tpsubtype NonStateTransition
    } else {
        lappend transtuple NewState $target
        set tpsubtype StateTransition
    }
```

```
    relvar insert ::rosea::Config::$tpsubtype $transtuple
    return
}
```

1. When we insert event tuples into the relvars we use the `uinsert` operation which does not raise an error on duplicates. This just lets us glean any event names from all the invocations of `transition` without having to be particularly careful if we have seen the event before.

```
<<error code formats>>=
BAD_CREATION_TARGET {the target of a creation event must be a state,\
        got "%s"}
```

Initialstate

By default, instances created from classes that have a state model are placed in the first state that was defined for the model. The `initialstate` command is used to specify explicitly the inital state for newly created instances.

`initialstate` *name*

name
The name of at state in the state model being defined. Instances of the class that are created synchronously using the `create` class command will be placed in this state.

Implementation

```
<<state model config commands>>=
proc initialstate {name} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME initialstate
    }
    if {$name in {@ CH IG}} {
        tailcall DeclError PSEUDO_STATE $name initialstate
    }
    variable InitialState $name
    return
}
```

Defaulttrans

It is customary to write `transition` commands only for outgoing transition that appear on the state model graphic. For any entries in transition matrix that are not set by a `transition` command, a default value is supplied. That default is either `IG` or `CH` depending upon the argument to the `defaulttrans` command. If no `defaulttrans` command is invoked when a state model is defined, then the default transition will be `CH`.

`defaulttrans` *trans*

trans
Either the string **IG** or **CH**. For all transitions not explicitly mentioned in a `transition` command, the default transition is defined as *trans*. If `defaulttrans` is not invoked during a state model definition then the default transition is **CH**.

Implementation

```
<<state model config commands>>=
proc defaulttrans {name} {
    if {$name ni {CH IG}} {
        tailcall DeclError EXPECTED_PSEUDO_STATE $name
    }
    variable DefaultTrans $name
    return
}
```

Terminal

Terminal states are those where the class instance is deleted after the state activity is executed. This allows for asynchronous deletion of class instances.

`terminal` *?state ...?*

`state`
The name of a state in the state model that will be marked as a terminal state. If a state machine transitions into a terminal state, the associated instance is deleted after the state activity is executed.

Implementation

```
<<state model config commands>>=
proc terminal {args} {
    variable Terminals
    ::struct::set add Terminals $args
    return
}
```

Tests

```
<<config command tests>>=
test configure-5.0 {
    Define statemodel
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                statemodel {
                    state s1 {a b} {
                        puts $a $b
                    }
                    transition s1 - e1 -> s2

                    state s2 {} {
                        puts "in s2"
                    }
                    transition s2 - e1 -> s1
```

```
                }
            }
        }
    }
    relation cardinality [relvar set ::rosea::Config::StateModel]
} -result {1}
```

Polymorphic

Polymorphic events are those defined in a superclass and mapped at run time to an event in a subclass.

`polymorphic` *?event ... ?*

`event`
The name of an event that is to be marked as polymorphic. This command should only be invoked when defining a class that will serve as the superclass for a generalization relationship.

Implementation

```
<<class config commands>>=
proc polymorphic {args} {
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::ClassDef ClassName ClassName

    foreach polyevent $args {
        set eventtuple [list\
            Domain $DomainName\
            Model $ClassName\
            Event $polyevent\
        ]
        relvar insert ::rosea::Config::PolymorphicEvent $eventtuple
        relvar insert ::rosea::Config::DeferredEvent $eventtuple
        relvar insert ::rosea::Config::Event $eventtuple
    }
    return
}
```

Previously, we indicated that polymorphic event rules require that we perform some other processing once the configuration script for a domain has been executed. Now we are prepared to show that processing. It will be helpful to refer to the Event Configuration Class Diagram above to follow the description.

The `polymorphic` command above inserts the argument event names into the configuration relvars simply as a deferred event that is polymorphic. We must resolve two issues.

1. For classes that are super classes for mutiple generalizations, we must have corresponding instances of **Deferral Path** for each of the generalization hierarchies.
2. For classes that are sub classes, they either consume the polymorphic event in a state model or it continues to be inherited by any sub classes further down the generalization hierarchy.

To solve the first issue we know that relationship, **R86**, specifies how a polymorphic event may be propagated along multiple generalizations when it is signaled to a given superclass. After the configuration scripts have been executed, we will have all the **Superclass** instances and all the **Deferred Event** instances and can now make up the correlation between them.

```
<<domain: propagate polymorphic events>>=
# Create Deferral Path instances

variable PolymorphicEvent
variable DeferredEvent
variable Superclass
variable Subclass
variable DeferralPath

set dpaths [pipe {
    relation restrictwith $PolymorphicEvent {$Domain eq $name} |
    relation semijoin ~ $DeferredEvent |
    relation join ~ $Superclass -using {Domain Domain Model Class}
}]
relvar insert ::rosea::Config::DeferralPath {*}[relation body $dpaths]
```

Starting with the **Polymorphic Event** class we can navigate **R81** to find the corresponding **Deferred Event** and the join to the **Superclass**. This gives the set of paths along which the polymorphic event must be propagated.

To solve the second issue, we must walk the generalization hierarchy and determine if events are being consummed or inherited down the hierarchy. We want to start the walk only on the ultimate super classes, *i.e.* those super classes that are *not* themselves the sub class of some other generalization.

```
<<domain: propagate polymorphic events>>=
set supers [relation semiminus $Subclass $Superclass\
    -using {Domain Domain Class Class}]

relation foreach super $supers {
    PropagatePolyEvents $super
}
```

So we find all **Superclass** instances that have no corresponding **Subclass** role and propagate any polymorphic events down the generalization hierarchy.

The problem we are trying to solve in the propagation of polymorphic events arises from the way in which we attempt to minimize the amount of input from the user when specifying the events. Recall that the `polymorphic` command simply inserts instances into the **Polymorphic Event** class (and corresponding instances in **Deferred Event** and **Event** classes). Events found when defining a state model are simply created as instances of the **Local Event** class (and again the corresponding **Effective Event** and **Event** instances). At the end of the configuration process we can now deduce which events were inherited down the generalization hierarchy and which were truly local events.

For sub classes that are leafs of generalization hierarchy, polymorphic events inherited from the super class are migrated to be **Mapped Events**. For sub classes that are intermediate in the hierarchy, polymorhphic events are migrated to instances of **Inherited Event**. Both **Inherited Event** and **Mapped Event** are types of **Non-Local Event** and **R85** insures that they affect only **Subclass** instances.

The `PropagatePolyEvents` procedure below accomplishes this reclassification operation. By doing it this way, we do not have to burden the user with all the subtle characteristics of polymorphic events. We need only specify which events are polymorphic and which events cause a state model transition and then we can deduce the intent to inherit down the generalization hierarchy.

The `PropagatePolyEvents` procedure takes a singular relation value that is from the **Superclass** relvar and tracks the polymorphic event inheritance down the hierarchy.

```
<<helper commands>>=
proc PropagatePolyEvents {super} {
    namespace upvar ::rosea::Config\
        DeferredEvent DeferredEvent\
        LocalEvent LocalEvent\
        Superclass Superclass\
        Subclass Subclass\
        DeferralPath DeferralPath\
```

```
        Generalization Generalization ; # ❶

    set supername [relation extract $super Class]

    <<PropagatePolyEvents: find sub classes>>

    relation foreach sub $subs {
        <<PropagatePolyEvents: determine deferred events>>
        <<PropagatePolyEvents: find multiple generalization classes>>

        if {[relation isempty $multigens]} {
            <<PropagatePolyEvents: migrate for single generalization>>
        } else {
            <<PropagatePolyEvents: migrate for multiple generalization>>
        }

        <<PropagatePolyEvents: add non-local events>>
        <<PropagatePolyEvents: continue propagation>>
    }
}
```

❶ There are a number of configuration relvars that we will need, so we bring them into scope here.

We can find the subclasses that particpate in a particular generalization by navigating **R36** and **R37**.

```
<<PropagatePolyEvents: find sub classes>>=
set subs [pipe {
    relation semijoin $super\
        $Generalization -using {Domain Domain Relationship Name}\
        $Subclass -using {Domain Domain Name Relationship}
}]
```

The instances of **Deferred Event** for the superclass have corresponding instances for the subclass. We need to create those instances. Conveniently, we can accomplish that by navigating **R86** and then updating the **Model** attribute to be the name of the subclass. Since **R86** is an associative relationships, we have to traverse to the associator class and then on to the **Deferred Event** class.

```
<<PropagatePolyEvents: determine deferred events>>=
set defrdevents [pipe {
    relation semijoin $super $DeferralPath\
        -using {Domain Domain Class Model Relationship Relationship}\
        $DeferredEvent |
    relation update ~ deftup {1} {
        tuple update $deftup Model [relation extract $sub Class]
    }
}]
```

We need to know if a given subclass is a leaf subclass in the generalization hierarchy or if it is part of a multiple generalization arrangements. This tells us if we must consume any inherited events or if they will be allowed to be inherited further down the hierarchy. Subclasses that are part of a multiple generation will have instances of **Superclass** to show their role in that generalization.

```
<<PropagatePolyEvents: find multiple generalization classes>>=
set multigens [relation semijoin $sub $Superclass\
    -using {Domain Domain Class Class}]
```

In the case of a leaf subclass, all the events must be **Effective Event** instances since there must be a state model in the leaf subclass to consume any inherited polymorphic events. The events to which the state model responds are inherited ones and any newly introduced local ones. The inherited events that appear in `transition` statements will have been created as **Local Event** during the configuration script execution. So we want to subtract out any local events that are mapped to the leaf subclass by deferral from a superclass. The deferred events are then created as **Mapped Event** instances.

```
<<PropagatePolyEvents: migrate for single generalization>>=
relvar minus ::rosea::Config::LocalEvent $defrdevents
relvar union ::rosea::Config::Event $defrdevents
relvar union ::rosea::Config::EffectiveEvent $defrdevents
relvar union ::rosea::Config::MappedEvent [relation extend\
    $defrdevents metuple ParentModel string {$supername}]
```

For the case where a subclass participates in a multiple generalization, the deferred events become instances of **Inherited Event**.

```
<<PropagatePolyEvents: migrate for multiple generalization>>=
relvar union ::rosea::Config::Event $defrdevents
relvar union ::rosea::Config::DeferredEvent $defrdevents
relvar union ::rosea::Config::InheritedEvent $defrdevents
```

Because we are creating new instances of **Deferred Event**, we must also add instances of **Deferral Path** so as to know which superclass is passing the polymorphic event along.

```
<<PropagatePolyEvents: migrate for multiple generalization>>=
relvar union ::rosea::Config::DeferralPath [relation join\
    $defrdevents $Superclass -using {Domain Domain Model Class}]
```

In both cases, inherited polymorphic events, whether consumed or passed along, are a form of **Non-Local Event**.

```
<<PropagatePolyEvents: add non-local events>>=
relvar union ::rosea::Config::NonLocalEvent [relation extend\
    $defrdevents nletuple\
    Relationship string {[relation extract $sub Relationship]}\
    Role string {[relation extract $sub Role]}]
```

For the multiple generalization subclasses, we recursively continue to propagate the polymorphic events.

```
<<PropagatePolyEvents: continue propagation>>=
relation foreach multigen $multigens {
    PropagatePolyEvents $multigen
}
```

Tests

```
<<config command tests>>=
test configure-6.0 {
    Define polymorphic events
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1
                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
```

```
                    transition s1 - e2 -> s1
                }
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R1 S X Y
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 4}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 4}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 0}
} -result {1}
```

```
<<config command tests>>=
test configure-6.1 {
    Define polymorphic events -- inherit across one level
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1
                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                reference R1 S -link id
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R1 S X Y

            class A {
                attribute id string -id 1
                reference R2 X -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
```

```
                }
            }
            class B {
                attribute id string -id 1
                reference R2 X -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R2 X A B
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 8}\
        {[relation cardinality $::rosea::Config::InheritedEvent] == 2}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 6}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 0}
} -result {1}
```

```
<<config command tests>>=
test configure-6.2 {
    Define polymorphic events -- new event in leaf subclass
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1

                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                reference R1 S -link id

                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    state s2 {} {
                        puts "in s2"
                    }
                    transition s1 - e1 -> s2
                    transition s2 - e2 -> s1
                    transition s2 - e3 -> s2 ; # New local event!
                }
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id

                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    state s2 {} {
```

```
                        puts "in s2"
                    }
                    transition s1 - e1 -> s2
                    transition s2 - e2 -> s1
                }
            }
            generalization R1 S X Y
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 4}\
        {[relation cardinality $::rosea::Config::InheritedEvent] == 0}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 4}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 1}
} -result {1}
```

```
<<config command tests>>=
test configure-6.3 {
    Define polymorphic events -- inject polymorphic event mid-level
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1
                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                reference R1 S -link id
                polymorphic e3
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R1 S X Y

            class A {
                attribute id string -id 1
                reference R2 X -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            class B {
                attribute id string -id 1
                reference R2 X -link id
                statemodel {
```

```
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R2 X A B
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 10}\
        {[relation cardinality $::rosea::Config::InheritedEvent] == 2}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 8}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 0}
} -result {1}
```

```
<<config command tests>>=
test configure-6.4 {
    Define polymorphic events -- multiple hierarchies
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1
                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R1 S X Y

            class A {
                attribute id string -id 1
                reference R2 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
```

```
                    transition s1 - e2 -> s1
                }
            }
            class B {
                attribute id string -id 1
                reference R2 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                }
            }
            generalization R2 S A B
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 8}\
        {[relation cardinality $::rosea::Config::InheritedEvent] == 0}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 8}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 0}
} -result {1}
```

```
<<config command tests>>=
test configure-6.5 {
    Define polymorphic events -- common subclass
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class S {
                attribute id string -id 1
                polymorphic e1 e2
            }
            class X {
                attribute id string -id 1
                attribute zid string -id 2
                reference R1 S -link id
                reference R2 Z -link zid
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                    transition s1 - e2 -> s1
                    transition s1 - e3 -> s1
                }
            }
            class Y {
                attribute id string -id 1
                reference R1 S -link id
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                }
            }
            generalization R1 S X Y
```

```
            class Z {
                attribute zid string -id 1
                polymorphic e3
            }
            class A {
                attribute zid string -id 1
                reference R2 Z -link zid
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                }
            }
            generalization R2 Z A X
        }
    }
    testConditions\
        {[relation cardinality $::rosea::Config::NonLocalEvent] == 6}\
        {[relation cardinality $::rosea::Config::InheritedEvent] == 0}\
        {[relation cardinality $::rosea::Config::MappedEvent] == 6}\
        {[relation cardinality $::rosea::Config::LocalEvent] == 0}
} -result {1}
```

Defining Assigners

The assigner concept is not frequently used in XUML models but is an essential concept in modeling competitive relationships. Some associations model competitive behavior such as allocating resources. For such situations, instances of the relationships must be created and deleted, *i.e.* linked and unlinked, according to some protocol and in some fashion that serializes the relationship lifecycle among the competing accessors. Lifecycles behavior of associations is accomplished in XUML models by associating a state model and having the state model activities insure the proper creation of relationship instances. Such state models are known as *assigners*.

Note that we associate the assigner with the relationship. We do *not* require factoring the relationships into a class based association and then attach the assigner to the associator class. In `rosea`, the relationship may or may not be class based and if it is that class may or may not have a state model. The assigner state model is independent of any class and its state model. It is, after all, a state model to deal with the application semantics of forming and dissolving the relationship.

Also note that only associative relationships may have an assigner. Assigners are not meaningful for generalization relationships given the disjoint union nature of a generalization.

There is a further complication. Usually, there is a single state machine associated with the association assigner. However, some assigners have more complicated competitive protocols. In those cases the association may have multiple assigners that are partitioned by another class. The archetypal example is that of assigning department store clerks to customers. If any clerk can service any customer, then a single assigner is used to sequence the life cycle of clerks and customers to insure that clerks are not over allocated or customers over served. If a customer can only be served by a clerk working in a particular department, then there will be as many assigners as there are departments and the identifier of the department class serves to identify the assigner instances needed to insure that clerks, within the department, are not over allocated and that customers, visiting the department, are not over served. We will see below how multiple assigner are specified.

Assigners and multi-assigners fall into the XUML semantics category of not that common but essential to express the proper execution semantics. As we will see, the specifications required for assigners is almost the same as that for ordinary class state models.

We will create an `assigner` command and, as usual, allocate a namespace in which to evaluate the definition of an assigner.

```
<<assigner config namespace layout>>=
namespace eval AssignerDef {
    logger::initNamespace [namespace current]

    <<tclral imports>>
```

```
    namespace import ::ral::relvar
    namespace import ::rosea::Config::ConfigEvaluate
    namespace import ::rosea::Helpers::DeclError

    <<assigner config commands>>
}
```

Assigner

`assigner` *association body*

association
The name of an association relationships to which the assigner is bound. Conventionally, relationships names are of the form **R<d>** where <*d*> is one or more decimal digits but any non-empty name is acceptable.

body
A Tcl Script that is evaluated in a context to allow the definition of the properties of the assigner.

Implementation

```
<<domain config commands>>=
proc assigner {rname body} {
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar AssignerDef\
        RelationshipName RelationshipName\
        multiClass multiClass\
        multiIdNum multiIdNum\
        InitialState InitialState\
        DefaultTrans DefaultTrans

    set RelationshipName $rname
    set multiClass {}
    set multiIdNum 1
    set InitialState {}
    set DefaultTrans {}

    try {
        ConfigEvaluate [namespace current]::AssignerDef $body

        if {$DefaultTrans eq {}} {
            set DefaultTrans CH
        }
        relvar insert ::rosea::Config::StateModel [list\
            Domain          $DomainName\
            Model           $rname\
            InitialState    $InitialState\
            DefaultTrans    $DefaultTrans\
        ]
        set assigntuple [list\
            Domain          $DomainName\
            Relationship    $rname\
        ]
        relvar insert ::rosea::Config::AssignerStateModel $assigntuple
        if {$multiClass eq {}} {
            relvar insert ::rosea::Config::SingleAssigner $assigntuple
        } else {
```

```
            lappend assigntuple Class $multiClass Number $multiIdNum
            relvar insert ::rosea::Config::MultipleAssigner $assigntuple
        }
    } on error {result opts} {
        log::error $result
        upvar #0 ::rosea::Config::errcount errcount
        incr errcount
    }
}
```

State

```
<<assigner config commands>>=
proc state {name params body} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME state
    }
    if {$name in {CH IG}} {
        tailcall DeclError PSEUDO_STATE $name state
    }

    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::AssignerDef\
        RelationshipName RelationshipName

    variable InitialState

    if {$InitialState eq {}} {
        set InitialState $name
    }

    relvar insert ::rosea::Config::State [list\
        Domain          $DomainName\
        Model           $RelationshipName\
        Name            $name\
        Parameters      $params\
        Action          $body\
        IsFinal         false\
    ]
    relvar insert ::rosea::Config::StatePlace [list\
        Domain          $DomainName\
        Model           $RelationshipName\
        Name            $name\
    ]
    return
}
```

Transition

```
<<assigner config commands>>=
proc transition {source - event -> target} {
    if {$event eq {}} {
        tailcall DeclError EMPTY_NAME event
    }
    if {$source in {CH IG}} {
        tailcall DeclError PSEUDO_STATE $name "transition source state"
    }
```

```
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName
    namespace upvar ::rosea::Config::DomainDef::AssignerDef\
        RelationshipName RelationshipName

    set eventtuple [list\
        Domain  $DomainName\
        Model   $RelationshipName\
        Event   $event\
    ]
    relvar uinsert ::rosea::Config::Event $eventtuple
    relvar uinsert ::rosea::Config::EffectiveEvent $eventtuple
    relvar uinsert ::rosea::Config::LocalEvent $eventtuple

    set tranrule [relvar restrictone ::rosea::Config::TransitionRule\
        Name $target]
    set transtuple [list\
        Domain  $DomainName\
        Model   $RelationshipName\
        State   $source\
        Event   $event\
    ]
    relvar insert ::rosea::Config::TransitionPlace $transtuple

    if {[relation isnotempty $tranrule]} {
        lappend transtuple TransRule $target
        set tpsubtype NonStateTransition
    } else {
        lappend transtuple NewState $target
        set tpsubtype StateTransition
    }
    relvar insert ::rosea::Config::$tpsubtype $transtuple
    return
}
```

Initialstate

```
<<assigner config commands>>=
proc initialstate {name} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME initialstate
    }
    if {$name in {CH IG}} {
        tailcall DeclError PSEUDO_STATE $name initialstate
    }
    variable InitialState $name
    return
}
```

```
<<error code formats>>=
PSEUDO_STATE    {the transition action, "%s", is not valid as %s}
```

Defaulttrans

```
<<assigner config commands>>=
proc defaulttrans {name} {
    if {$name ni {CH IG}} {
        tailcall DeclError EXPECTED_PSEUDO_STATE $name
```

```
    }
    variable DefaultTrans $name
    return
}
```

```
<<error code formats>>=
EXPECTED_PSEUDO_STATE    {expected CH or IG, got "%s"}
```

Identifyby

The `identifyby` command is used to indicate that the assigner being defined is a multiple assigner. The name of the class used to partition and identify the assigner instances is given.

Implementation

```
<<assigner config commands>>=
proc identifyby {name args} {
    if {[llength $args] % 2 != 0} {
        tailcall DeclError ARG_FORMAT $args
    }

    variable multiClass $name

    foreach {option value} $args {
        switch -exact -- $option {
            -id {
                variable multiIdNum $value
            }
            default {
                tailcall DeclError UNKNOWN_OPTION identifyby $option
            }
        }
    }

    return
}
```

Tests

```
<<config command tests>>=
test configure-7.0 {
    Define assigner
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class Customer {
                attribute Name string -id 1
            }
            class Clerk {
                attribute Name string -id 1
                attribute Customer string
                reference R1 Customer -link {Customer Name}
            }
```

```
            association R1 Clerk 0..1--0..1 Customer
            assigner R1 {
                state s1 {a b} {
                    puts $a $b
                }
                transition s1 - e1 -> s2

                state s2 {} {
                    puts "in s2"
                }
                transition s2 - e1 -> s1
            }
        }
    }
    relation cardinality [relvar set ::rosea::Config::SingleAssigner]
} -result {1}
```

```
<<config command tests>>=
test configure-7.1 {
    Define mulit assigner
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class Customer {
                attribute Name string -id 1
            }
            class Clerk {
                attribute Name string -id 1
                attribute Customer string
                attribute Department string
                reference R1 Customer -link {Customer Name}
                reference R2 Department -link {Department Name}
            }
            class Department {
                attribute Name string -id 1
            }
            association R1 Clerk 0..1--0..1 Customer
            association R2 Clerk 1..*--1 Department
            assigner R1 {
                identifyby Department
                state s1 {a b} {
                    puts $a $b
                }
                transition s1 - e1 -> s2

                state s2 {} {
                    puts "in s2"
                }
                transition s2 - e1 -> s1
            }
        }
    }
    relation cardinality [relvar set ::rosea::Config::MultipleAssigner]
} -result {1}
```

Defining Domain Operations

`operation` *name params body*

name
The *name* of the domain operation.

params
The *params* argument is a list of parameters to the operation and is specified in the same manner as for the `::proc` command.

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

```
<<domain config commands>>=
proc operation {name params body} {
    if {$name eq {}} {
        tailcall DeclError EMPTY_NAME operation
    }
    namespace upvar ::rosea::Config::DomainDef DomainName DomainName

    try {
        relvar insert ::rosea::Config::DomainElement [list\
            Domain      $DomainName\
            Element     $name\
        ]
    } trap {RAL relvar insert DUPLICATE_TUPLE} {result opts} {
        tailcall DeclError DUP_ELEMENT_NAME $name
    }
    relvar insert ::rosea::Config::DomainOperation [list\
        Domain      $DomainName\
        Name        $name\
        Parameters  $params\
        Body        $body\
    ]
}
```

Handling Configuration Errors

One of the design elements of the configuration DSL is the use of a set of relvar and relvar constraints as the data structures to hold the configuration information. This design approach allows us to encode the rules for what is a valid domain configuration into the constraints. Any attempt to define domain elements that violate those rules will be caught at the end of the relvar transaction that end the `configure` command.

The problem with this approach is that the error messages the are returned refer to the relvars of the domain meta-model not those of the domain we were attempting to define. For example, if a domain configuration script fails to define any identifiers for a class, the error that is generated by TclRAL describes the failure of the constraint on **R4** between **Class** and **Identifier**. An example of such as message is as follows:

```
for association ::rosea::Config::R4(::rosea::Config::Identifier [+] ==> [1] :: ↩
    rosea::Config::Class), in relvar ::rosea::Config::Class
tuple {Domain d1 Name c2} is not referenced by any tuple
```

The first line describes the constraint violation and that is followed by one or more lines that name the tuple values and which aspect of the constraint is violated. For our example, in domain, `d1`, there was a tuple in the **Class** class named, `c2`, which was not referenced by any tuples in the in the **Identifier** class as is required by the **R4** association. You can see this on the class diagram for configuring classes.

All of these terms are meaningless to someone trying to run a `configure` script and finding out which class is lacking the identifier means looking at the values of tuples of the **Class** relvar for the value of the **Name** attribute.

We will mitigate this problem by capturing the result returned by TclRAL and turning it into error messages that are more meaningful to the task of configuring a domain. Unfortunately, the messages produced by TclRAL are intended to be human readable, so we will have to parse them into a form that is easier to deal with programmatically. Fortunately, the messages are very regular in structure so the parsing code need not be very sophisticated.

Our strategy is to extract only the essential information from the error message and then map the information onto a more meaningful error message. Part of what we want to include in the error message is the values from the tuples failing the constraint. These values contain information the user entered rather than the abstractions of the meta-model.

The essential information mapping is shown below. Naturally enough, we hold it in a relvar.

```
<<config data>>=
relvar create Config_DataError {
    Relationship        string
    RefClass            string
    RefType             string
    Format              string
} {Relationship RefClass RefType}
```

The **Relationship** attribute is the name of the relationship in the metamodel that failed the constraint check. The **RefClass** attribute is the meta-model class that has the tuples that failed the constraint check. The **RefType** attribute defines the way in which the constraint failed. The **Format** attribute is a string that contains the text of the error message that we want to display. The text in the **Format** attribute may also contain variable references to attributes in the tuple of **RefClass** that failed. The variable references in the **Format** will be substituted with the values from the failing tuples in order to give specifics of the error back to the user.

Implementation

The design of the `HandleConfigError` procedure is shown below. It is a nested iteration over the lines of the error result, pulling off the constraint violation details and then iterating over the tuples that were found in the violation.

```
<<config commands>>=
proc HandleConfigError {result} {
    set lines [split [string trimright $result] \n]
    set nlines [llength $lines]
    set lineno 0
    upvar #0 ::rosea::Config::errcount errcount
    while {$lineno < $nlines} {
        <<HandleConfigError: examine one failure>>
    }

    return
}
```

The information in the TclRAL error message is simple enough and well structured enough that a regular expression can be used to match and extract the interesting parts.

```
<<HandleConfigError: examine one failure>>=
set line [lindex $lines $lineno]
incr lineno
if {[regexp {^for[^:]+([^(]+)\(.+\), in relvar (.+)$} $line\
        match rnum refclass]} {
    set rnum [namespace tail $rnum]
    set refclass [namespace tail $refclass]

    # Now iterate over the "tuple" lines that follow the constraint message.
    while {$lineno < $nlines} {
        set tupline [lindex $lines $lineno]
```

```
        if {[regexp {^tuple {(.+)} (.+)$} $tupline match tuple phrase]} {
            incr lineno
            incr errcount
            <<HandleConfigError: examine one tuple>>
            <<HandleConfigError: format error message>>
        } else {
            break
        }
    }
} else {
    log::error "unknown configuration error, \"$result\""
}
```

We match the phrase in the tuple message to create an enumeration of the types of constraint violations. These are just a bit simpler to handle and look up.

```
<<HandleConfigError: examine one tuple>>=
if {[string match {is not referenced*} $phrase]} {
    set reftype notrefed
} elseif {[string match {references no*} $phrase]} {
    set reftype refnone
} elseif {[string match {*to by multiple*} $phrase]} {
    set reftype multrefed
} else {
    log::error "unknown constraint phrasing, \"$phrase\""
    continue
}
```

Finally, we look up the format information and generate an error message. Using the `dict with` command allows us to take the tuple value from the error message, treat it like a dictionary and get the values into Tcl variable. The `subst` command then will perform the variable substitutions in the format string.

```
<<HandleConfigError: format error message>>=
set cde [relvar restrictone Config_DataError Relationship $rnum RefClass\
        $refclass RefType $reftype]
if {[relation isnotempty $cde]} {
    dict with tuple {
        log::error [subst -nocommands [relation extract $cde Format]]
    }
} else {
    log::error "$line\n$tupline"
}
```

We now need to enumerate all the meta-model constraints that can be violated and supply messages that provide less abstract and more useful user error messages. Because of the way the configuration data is stored into the meta-model relvars, not all violations are even possible.

```
<<config data>>=
relvar insert Config_DataError {
    Relationship        R3
    RefClass            Class
    RefType             notrefed
    Format              {in domain, \"$Domain\", class, \"$Name\",\
                        has no attributes}
} {
    Relationship        R4
    RefClass            Class
    RefType             notrefed
    Format              {in domain, \"$Domain\", class, \"$Name\",\
                        has no identifiers}
} {
```

```
    Relationship        R11
    RefClass            ReferencedClass
    RefType             notrefed
    Format              {in domain, \"$Domain\", for relationship,\
                        \"$Relationship\", no class has defined a reference\
                        to class, \"$Class\"}
} {
    Relationship        R11
    RefClass            ReferencedIdAttribute
    RefType             refnone
    Format              {in domain, \"$Domain\", relationship,\
                        \"$Relationship\", references attribute,\
                        \"$Attribute\", in class, \"$Class\", which is\
                        not an identifying attribute, does not exist\
                        or \"$Relationship\" does not exist}
} {
    Relationship        R12
    RefClass            AttributeReference
    RefType             refnone
    Format              {in domain, \"$Domain\", the attribute,\
                        \"$ReferringAttribute\", in class, \"$ReferringClass\",\
                        refers to, \"$ReferencedAttribute\", in class,\
                        \"$ReferencedClass\", but the class, attribute or\
                        relationship does not exist}
} {
    Relationship        R13
    RefClass            AttributeReference
    RefType             refnone
    Format              {in domain, \"$Domain\", the attribute,\
                        \"$ReferringAttribute\", in class, \"$ReferringClass\",\
                        does not exist, but is used as a referential attribute}
} {
    Relationship        R41
    RefClass            ClassRole
    RefType             refnone
    Format              {in domain, \"$Domain\", relationship,\
                        \"$Relationship\", references class, \"$Class\",\
                        which does not exist}
} {
    Relationship        R52
    RefClass            AssignerStateModel
    RefType             refnone
    Format              {in domain, \"$Domain\", an assigner state model is\
                        defined on relationship, \"$Relationship\", which is\
                        not an association type relationship}
} {
    Relationship        R72
    RefClass            StateTransition
    RefType             refnone
    Format              {in domain, \"$Domain\", the state model for,\
                        \"$Model\", contains the transition,\
                        \"$State - $Event -> $NewState\",\
                        but state \"$NewState\" does not exist}
} {
    Relationship        R81
    RefClass            DeferredEvent
    RefType             multrefed
    Format              {in domain, \"$Domain\", in the state model for,\
                        \"$Model\", event, \"$Event\", is both an inherited\
                        polymorphic event and a locally defined one}
} {
    Relationship        R86
```

```
    RefClass            DeferredEvent
    RefType             notrefed
    Format              {in domain, \"$Domain\", in the state model for,\
                        \"$Model\", event, \"$Event\", is defined as\
                        polymorphic but \"$Model\" is not a superclass\
                        of a generalization}
} {
    Relationship        R87
    RefClass            EffectiveEvent
    RefType             refnone
    Format              {in domain, \"$Domain\", event, \"$Event\" has been\
                        inherited, but no state model exists for, \"$Model\"}
}
```

Tests

```
<<config command tests>>=
test configure-8.0 {
    configure error -- no attributes defined for class
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
            }
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", class, "c1", has no attributes*} -match glob
```

```
<<config command tests>>=
test configure-8.1 {
    configure error -- no identifier defined for class
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string
            }
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", class, "c1", has no identifiers*} -match glob
```

```
<<config command tests>>=
test configure-8.2 {
    configure error -- no reference
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
```

```
                attribute a1 string -id 1
            }
            class c2 {
                attribute a2 string -id 1
            }
            association R1 c1 1--1 c2
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", for relationship, "R1", no class has defined a reference to  ↩
    class, "c2"*}\
-match glob
```

```
<<config command tests>>=
test configure-8.3 {
    configure error -- bad reference
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                reference R1 c3 -link a1
            }
            class c2 {
                attribute a2 string -id 1
            }
            association R1 c1 1--1 c2
        }
    }
} -result {encountered 3 configuration script errors} -returnCodes error\
-output {*in domain, "foo", relationship, "R1", references attribute, "a1", in class, "c3", ↩
     which is not an identifying attribute, does not exist or "R1" does not exist*}\
-match glob
```

```
<<config command tests>>=
test configure-8.4 {
    configure error -- bad referential attribute linkage
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                reference R1 c2 -link {a1 a3}
            }
            class c2 {
                attribute a2 string -id 1
            }
            association R1 c1 1--1 c2
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", relationship, "R1", references attribute, "a3", in class, "c2", ↩
     which is not an identifying attribute, does not exist or "R1" does not exist*}\
-match glob
```

```
<<config command tests>>=
test configure-8.5 {
    configure error -- transition refers to non-existent state
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s2
                }
            }
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", the state model for, "c1", contains the transition, "s1 - e1 -> ↩
     s2", but state "s2" does not exist*}\
-match glob
```

```
<<config command tests>>=
test configure-8.6 {
    configure error -- duplicated polymorphic events
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class super {
                attribute a1 string -id 1
                polymorphic e1
            }
            class sub1 {
                attribute a1 string -id 1
                reference R1 super -link a1

                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                }
            }
            class sub2 {
                attribute a1 string -id 1
                reference R1 super -link a1
                polymorphic e1
            }
            class ssub1 {
                attribute a1 string -id 1
                reference R2 sub2 -link a1
            }
            class ssub2 {
                attribute a1 string -id 1
                reference R2 sub2 -link a1
```

```
            }
            generalization R1 super sub1 sub2
            generalization R2 sub2 ssub1 ssub2
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", in the state model for, "sub2", event, "e1", is both an  ↩
    inherited polymorphic event and a locally defined one*}\
-match glob
```

```
<<config command tests>>=
test configure-8.7 {
    configure error -- no leaf state models
} -setup {
} -cleanup {
    cleanupConfigData
} -body {
    rosea configure {
        domain foo {
            class super {
                attribute a1 string -id 1
                polymorphic e1
            }
            class sub1 {
                attribute a1 string -id 1
                reference R1 super -link a1

                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                }
            }
            class sub2 {
                attribute a1 string -id 1
                reference R1 super -link a1
            }
            class ssub1 {
                attribute a1 string -id 1
                reference R2 sub2 -link a1
                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                }
            }
            class ssub2 {
                attribute a1 string -id 1
                reference R2 sub2 -link a1
            }
            generalization R1 super sub1 sub2
            generalization R2 sub2 ssub1 ssub2
        }
    }
} -result {encountered 1 configuration script errors} -returnCodes error\
-output {*in domain, "foo", event, "e1" has been inherited, but no state model exists for,  ↩
    "ssub2"*}\
-match glob
```

Chapter 6

Generating Domains

After specifying the domain using the configuration DSL, the second step in translating a model is to generate all the architectural data values and class and relationship ensemble commands.

Let's take this opportunity to recap again our intent. The run time commands that map XUML concepts onto Tcl use a set of data structures to accomplish that mapping. Part of what generating a domain implies is to populate those data structures. We do that by querying data that was accumulated during the configuration phase for the domain. This data was inserted in to a set of relvars by the configuration DSL commands. We used a set of relvars to hold the DSL command data because the integrity constraints on those relvars enforce the rules required to properly specify the properties of the domain such as its classes and relationships. As an added benefit, using relvars to hold the configuration data and placing that data in a transaction eliminates most of the order dependencies in the DSL.

Another aspect of domain generation is to create the other structural mappings from the domain onto Tcl. This takes the form of namespace ensemble commands for classes, relationships and assigners that provide a convenient mapping of the run time procedures into ensembles that reference a particular class or relationship, *etc.*

Generating a domain should happen after all the configuration for the domain is complete. It is allowable to invoke the `configure` command multiple times and domain configurations are cumulative in the sense that a configuration script may refer to the same domain many times and the additional configuration is simply added to any previous one.

But at some point configuration is finished and that must be followed by an invocation of `generate`. Although the `generate` command may be invoked on a *per domain* basis (and this is useful during initial debugging of the configuration script), a common workflow is to just invoke it once and generate everything for all the domains of a program in a single pass.

After generation is completed, there are still two more steps to complete the translation of the XUML model, namely population and bridging. We will discuss those steps below.

Generation is a rather complicated undertaking so we will be dividing it up into many smaller parts to make clearer what we are trying to accomplish. Given the relational schema used to accumulate the configuration script information, you can anticipate that generating the domain involves a set of queries on that information. When generating code for conventional compiled languages that are not as dynamic as Tcl, *e.g.* when `pycca` generates "C" source, the results of those queries are used to generate programming language text files which are then handed over to a language compiler. But, since this is Tcl, we will avoid all the unnecessary intermediate files and just directly execute the generated Tcl commands. This is much more in keeping with the dynamic style of Tcl.

`rosea generate` *pattern nsqual*

`pattern`
A pattern of the form used by the `::string match` command. All domains whose names match *pattern* are generated. If the *pattern* argument is missing, then its default value is "*" which will match all domain names.

`nsqual`
A namespace qualifier that indicates the relative position where the domain commands and data will be placed. If *nsqual* is not fully qualified, then it is qualified to the namespace of the caller. If the *nsqual* argument is missing or the empty string, then the domain is placed in the global namespace (_i.e. "::").

The generate command generates all the commands and data for the domains whose names match *pattern*. The domain is located in the namespace given by *nsqual*::*domainname*.

Implementation

Although the implementation of `generate` is long, it is structurally straight forward. After some preliminaries to set up local variables that reference the required data, we simply iterate over each domain that matches the *pattern* argument and perform the generation for that domain.

```
<<rosea exports>>=
namespace export generate

<<rosea commands>>=
proc generate {{pattern *} {prefix {}}} {
    <<generate: fix up prefix>>
    <<generate: reference config data>>

    set genDomain [relation restrict $Domain dom {
        [string match $pattern [tuple extract $dom Name]]
    }]
    relation foreach domain $genDomain {
        <<generate: generate one domain>>
    }
}
```

Fix Up Prefix

We have to examine the *prefix* argument to check if it is fully qualified. If not we find the namespace of the caller. In the end we want a value for `prefix` that can be simply tacked on to the beginning of names to get to the correct namespace for things.

```
<<generate: fix up prefix>>=
if {$prefix ne {}} {
    if {[string range $prefix 0 1] ne "::"} {
        set prefix [uplevel 1 {namespace current}]::$prefix
    }
    set prefix [string trimright $prefix :]
} elseif {$prefix eq "::"} {
    set prefix {} ; # ❶
}
```

❶ Since we intend to concatenate a "::" to the prefix before adding the domain name, we need to treat the global namespace differently so that we don't end up with ":::" as the leading characters of a namespace.

Reference Config Data

Generating a domain involves querying the data that was stored away during the configuration of the domain. The relvars holding the data are only read by this procedure. Since each relvar has a backing read-only Tcl variable, we bring them into scope so that we can refer to the relvar contents using simple variable syntax. Turns out there are a lot of these relvars.

```
<<generate: reference config data>>=
namespace upvar Config\
    Domain Domain\
    DomainElement DomainElement\
    Class Class\
    Attribute Attribute\
    DefaultValue DefaultValue\
    ValueCheck ValueCheck\
    Identifier Identifier\
    IdentifyingAttribute IdentifyingAttribute\
    Relationship Relationship\
    Association Association\
    SimpleAssociation SimpleAssociation\
    SimpleReferringClass SimpleReferringClass\
    SimpleReferencedClass SimpleReferencedClass\
    ClassBasedAssociation ClassBasedAssociation\
    AssociatorClass AssociatorClass\
    SourceClass SourceClass\
    TargetClass TargetClass\
    Generalization Generalization\
    Superclass Superclass\
    Subclass Subclass\
    AttributeReference AttributeReference\
    ReferencedIdAttribute ReferencedIdAttribute\
    ReferringClass ReferringClass\
    ReferencedClass ReferencedClass\
    DomainOperation DomainOperation\
    SuppliedOperation SuppliedOperation\
    UserClassOperation UserClassOperation\
    SystemClassOperation SystemClassOperation\
    UserInstanceOperation UserInstanceOperation\
    SystemInstanceOperation SystemInstanceOperation\
    InstanceStateModel InstanceStateModel\
    StateModel StateModel\
    StatePlace StatePlace\
    State State\
    EffectiveEvent EffectiveEvent\
    DeferredEvent DeferredEvent\
    TransitionPlace TransitionPlace\
    StateTransition StateTransition\
    NonStateTransition NonStateTransition\
    AssignerStateModel AssignerStateModel\
    SingleAssigner SingleAssigner\
    MultipleAssigner MultipleAssigner\
    PolymorphicEvent PolymorphicEvent
```

Generate One Domain

Generating the contents for one domain is decomposed into the sequence of generating the various elements that make up the domain.

```
<<generate: generate one domain>>=
<<generate: create domain namespace>>
<<generate: create domain operations>>
```

```
<<generate: populate state model data>>
<<generate: create class data and commands>>
<<generate: create relationship data and commands>>
<<generate: create assigner data and commands>>
```

Create Domain Namespace

Each domain sits in a namespace given by the *prefix* argument concatenated to the domain name (with the required "::" separator). We update that prefix value into the Domain relvar as it will be needed later during the population phase. Some often used `ral` package commands are imported into the domain namespace. Here we also create the empty relvars for the run time architecture data that we are about to populate.

```
<<generate: create domain namespace>>=
set domainName [relation extract $domain Name]
relvar updateone Config::Domain dtup [list Name $domainName] {
    tuple update $dtup Location $prefix
}

set domns ${prefix}::$domainName
namespace eval $domns {
    <<tclral imports>>
    <<arch relvar definitions>>
}
namespace eval $domns namespace path ::rosea::InstCmds
```

Create Domain Operations

Each domain operation is converted into an ordinary Tcl procedure. Those procedures are exported from the domain namespace and accumulated into a namespace ensemble command that has the same name as the domain. Domain operations can be thought of as the "API" for the domain and constitute the public invocable interface.

```
<<generate: create domain operations>>=
set domops [relation semijoin $domain $DomainOperation -using {Name Domain}]
relation foreach domop $domops {
    relation assign $domop\
        {Name name} {Parameters parameters} {Body body}
    proc ${domns}::$name $parameters [list ::ral relvar eval $body] ; # ❶
}

namespace eval $domns [list\
    namespace export {*}[relation list $domops Name]\
]
namespace eval $domns namespace ensemble create
```

❶ Note that all domain operations are run as a `relvar` transaction. This allows the operation to modify class model data with impunity as long as it is left consistent by the end of the operation.

Populate State Model Data

To support event dispatch, the run time uses the relvars defined above. It may be helpful to refer back to that figure when reading the code below. Now we query the configuration data to populate these relvars. We can do this for the domain as a whole since there is nothing particularly class dependent in this data.

First we obtain all the state models for the domain. Then we simply populate one by one the relvars that are concerned with event dispatch. Because there are constraints between the relvars, we have to create the instances in a transaction.

```
<<generate: populate state model data>>=
set statemodels [relation semijoin $domain $StateModel\
    -using {Name Domain}]

relvar eval {
    <<generate: populate InitialState>>
    <<generate: populate State>>
    <<generate: populate TerminalState>>
    <<generate: populate Event>>
    <<generate: populate Transition>>
    <<generate: populate PolymorphicEvent>>
}
```

Populate InitialState

The InitialState relvar is populated from the projection of attributes from the StateModel configuration and with some renaming.

```
<<generate: populate InitialState>>=
relvar set ${domns}::__Arch_InitialState [pipe {
    relation project $statemodels Model InitialState |
    relation rename ~ Model Class InitialState State
}]
```

Populate State

The State relvar data is contained in the StatePlace configuration relvar.

```
<<generate: populate State>>=
set states [relation semijoin $domain $StatePlace -using {Name Domain}]
relvar set ${domns}::__Arch_State [pipe {
    relation project $states Model Name |
    relation rename ~ Model Class Name State
}]
```

Populate TerminalState

Terminal states are those that were marked as final during the configuration.

```
<<generate: populate TerminalState>>=
relvar set ${domns}::__Arch_TerminalState [pipe {
    relation semijoin $domain $State -using {Name Domain} |
    relation restrict ~ termtuple {[tuple extract $termtuple IsFinal]} |
    relation project ~ Model Name |
    relation rename ~ Model Class Name State
}]
```

Populate Event

The Event relvar is populated from the EffectiveEvent configuration relvar. This contains both the local event for the state model as well as any inherited polymorphic events.

```
<<generate: populate Event>>=
set events [relation semijoin $domain $EffectiveEvent\
    -using {Name Domain}]
relvar set ${domns}::__Arch_Event [pipe {
```

```
    relation project $events Model Event |
    relation rename ~ Model Class
}]
```

Populate Transition

The Transition relvar requires a bit more work. The instances of this relvar need to form the Cartesian product of the states and events so that every state / event combination is represented. Any transitions not explicitly defined during the domain configuration have to be set to the default transition.

The strategy is to compute all the transitions by joining the states and events. The transitions that were explicitly set during the domain configuration are tuples in the TransitionPlace relvar. The difference between all the transitions and those in the TransitionPlace are the ones to which we must assign the default transition. There are a few other twists along the way as we see below.

First, we get all the transitions. The attribute renaming will make the header the same as the TransitionPlace header.

```
<<generate: populate Transition>>=
set alltrans [pipe {
    relation join $states $events |
    relation rename ~ Name State
}] ; # ❶
```

❶ Recall that the relational join creates a new tuple where the commonly named attributes match in value. In this case the common attribute names are **Domain** and **Model**. The net effect is to produce the Cartesian product of states and events within the same domain and state model.

State transition are actually of two types. Those that actually transition to a new state or one of the non-transition actions of “IG” or “CH”. We have to treat the two cases separately. First the transitions that take you to a new state.

```
<<generate: populate Transition>>=
set statetrans [relation semijoin $domain $StateTransition\
    -using {Name Domain}]
```

For the non-transitioning case, we need to do some attribute renaming to make the header match.

```
<<generate: populate Transition>>=
set nontrans [pipe {
    relation semijoin $domain $NonStateTransition -using {Name Domain} |
    relation rename ~ TransRule NewState
}]
```

Computing the default transitions is a bit more involved.

```
<<generate: populate Transition>>=
set deftrans [pipe {
    relation minus $alltrans $TransitionPlace |
    relation join ~ $StateModel |
    relation eliminate ~ InitialState |
    relation rename ~ DefaultTrans NewState |
    relation update ~ dttup {[tuple extract $dttup State] eq "@"} {
        tuple update $dttup NewState CH}
}]
```

We will discuss computing the default transition step by step.

1. The difference between all the transitions and the explicitly defined ones is the set of transitions where we need to supply the default transition.

2. Join to the StateModel value since that is where the default transition is recorded.
3. We don't want the InitialState attribute. It's not in the header of the result we want.
4. We rename the DefaultTrans attribute, again to make the headings match what we are after.
5. Finally, we do some trickery with creation events. The "@" state is the pseudo initial state used used for creation events. We insist that any events not explicitly specified as causing an outbound transition from the pseudo-initial state be "CH" transitions. We want to avoid the circumstance where an instance is created asynchronously but using an event for which no transition out of the pseudo-initial state was specified. If the default transition happens to be "IG", we would end up with an instance forever sitting in the pseudo-initial state and the semantics of asynchronous creation would be violated. Asynchronous event creation must either cause a transition or create an error. So regardless of the default transition given for the state model, unspecified creation events cause an error and we force that in this step.

Finally, if we union our three groups of transitions, discard the Domain attribute and juggle the attribute names, we get to the Transition relvar population we are looking for.

```
<<generate: populate Transition>>=
relvar set ${domns}::__Arch_Transition [pipe {
    relation union $statetrans $nontrans $deftrans |
    relation eliminate ~ Domain |
    relation rename ~ Model Class
}]
```

```
<<generate command tests>>=
test generate-5.0 {
    Classes with state models
} -setup {
    rosea configure {
        domain foo {
            class Dog {
                attribute Name string -id 1
                attribute Breed string

                statemodel {
                    state Born {} {
                        puts "In Born"
                    }
                    state Grown {} {
                        puts "In Grown"
                        delaysignal 200 $self Time
                    }
                    state Old {} {
                        puts "In Old"
                        delaysignal 200 $self Time
                    }
                    state Dead {} {
                        puts "In Dead"
                        set ::done 1
                    }
                    terminal Dead

                    transition Born - Time -> Grown
                    transition Grown - Time -> Old
                    transition Old - Time -> Dead
                }
            }
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
```

```
    removeDomain ::foo
} -body {
    testConditions\
        {[relation cardinality [relvar set ::foo::__Arch_Transition]] == 4}\
        {[relation cardinality [relvar set ::foo::__Arch_State]] == 4}\
        {[relation cardinality [relvar set ::foo::__Arch_Event]] == 1}\
} -result {1}
```

Populate PolymorphicEvent

Populating the PolymorphicEvent relvar is also straight forward. It is just the contents of the DeferredEvent configuration relvar projected for this domain and renamed to match the attribute naming in the header. The hard work here was done during the propagation of the polymorphic events that happens at the end of the domain configuration.

```
<<generate: populate PolymorphicEvent>>=
relvar set ${domns}::__Arch_PolymorphicEvent [pipe {
    relation semijoin $domain $DeferredEvent\
        -using {Name Domain} |
    relation project ~ Model Event |
    relation rename ~ Model Class
}]
```

Create Class Data and Commands

When we create the domain classes, we will iterate over each class to perform the queries. As before, we will factor out the queries that are used inside the iteration loop to save the command rewriting that the `pipe` command would otherwise perform on each iteration. We bring the variables containing the queries into scope but show the queries below, lexically close to the code that uses them.

```
<<generate: create class data and commands>>=
<<class query variables>>

set classes [relation semijoin $domain $DomainElement -using {Name Domain}\
        $Class -using {Domain Domain Element Name}]
relation foreach class $classes {
    <<generate: create class namespace>>
    <<generate: create class relvar>>
    <<generate: create state model>>
    <<generate: create class operations>>
    <<generate: create instance operations>>
}
```

Create Class Namespace

For each class in the domain, we create a child namespace. That namespace is used to hold specifics of the class such as its class based and instance based operations. For convenience, we set up a command resolution path to the instance commands and the to the parent domain namespaces.

```
<<generate: create class namespace>>=
set className ${domns}::[relation extract $class Name]
namespace eval $className {
    <<tclral imports>>
}
namespace eval $className [list namespace path\
        [list ::rosea::InstCmds $domns]]
```

Create Class Relvar

With the namespace defined, we can now create the relvar that holds the instance data. Each class of the domain has a relvar that holds its instances. To create the relvar we need to know the heading and the identifiers.

The heading of a class consists of its attributes names and types. We will want that in the form of a dictionary which the `relation dict` command conveniently provides.

```
<<rosea data>>=
pipe {
    relation semijoin $class $Attribute -using {Domain Domain Name Class} |
    relation project ~ Name Type |
    relation dict ~ Name Type
} headingQuery

<<class query variables>>=
variable headingQuery
```

The other thing you need when creating the relvar for a class is the set of identifiers and the attributes that make up those identifiers. In the query below, the `relation group` command will create a relation valued attribute named `Attributes` in this case. That relation valued attribute will have one tuple for each attribute of each identifier.

```
<<rosea data>>=
pipe {
    relation semijoin $class $Identifier -using {Domain Domain Name Class}\
            $IdentifyingAttribute |
    relation group ~ Attributes Attribute
} idQuery

<<class query variables>>=
variable idQuery
```

The heading of the result of this query will be:

Table 6.1: Heading of Identifier Query

Domain	**Class**	**Number**	**Attributes**
string	string	int	Relation
			Attribute
			string

We need to transform the result of the `idQuery` into the set of identifying attributes lists that the `relvar create` command requires. This is done by iterating across all the identifiers of the class, extracting the grouped attributes for the identifier and then turning the attribute name tuples into a list. The required operations are all provided by `ral`. When dealing with these meta-model situations, we often need to convert the rows of some relation value in a list.

```
<<generate: create class relvar>>=
set heading [eval $headingQuery]
set idset [list]
relation foreach id [eval $idQuery] -ascending Number {
    lappend idset [relation list\
            [relation extract $id Attributes] Attribute]
}

relvar create $className $heading {*}$idset

# ❶
```

```
<<generate: setup value checks>>
<<generate: setup default values>>
```

❶ Order is important here. Since we will be using relvar tracing to implement value checks and default values, we want to the default value trace to execute first. In TclRAL, relvar traces are executed most recent trace done first.

Tests

As we test domain generation we will need something to clean up after ourselves and remove the domain that is generated by the test case.

```
<<test utility procs>>=
proc removeDomain {domns} {
    relvar constraint delete {*}[relvar constraint names ${domns}::*]
    relvar unset {*}[relvar names ${domns}::*]
    namespace delete $domns
    set ::rosea::Dispatch::event_queue [list]
}
```

```
<<generate command tests>>=
test generate-1.0 {
    Generate domain relvars
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute x2 string -default 20
            }
            class Y {
                attribute y1 string -id 1
            }
        }
    }
    rosea generate
    set rvnames [relvar names ::foo::*]
    expr {"::foo::X" in $rvnames && "::foo::Y" in $rvnames}
} -result {1}
```

```
<<generate command tests>>=
test generate-1.1 {
    Generate domain relvars -- multiple identifiers
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute y1 string -id 1
                attribute z1 int -id 2
                attribute x2 string
            }
```

```
        }
    }
    rosea generate
    relvar identifiers ::foo::X
} -result {{x1 y1} z1}
```

Setup Default Values

We will support the concept of a default attribute value by creating a relvar trace that is triggered when a tuple is inserted into the class relvar. Relvars support tracing in a manner analogous to Tcl variable tracing.

First, let's examine the command that will be invoked as the trace. This command, `DefValueTrace`, is shown below. The *op*, *relvar* and *tuple* arguments are added by the relvar tracing mechanism and are not interesting to us for this usage. The *defheading* and *defvalues* arguments are part of the command prefix that we must generate when the relvar trace is added. The *defheading* argument is a dictionary of attribute name / type pairs for those attributes that have defined default values. It is that subset of the heading that contains attributes with defined defaults. The *defvalues* argument is a dictionary of attribute name / value pairs that are the corresponding defined default values.

The return value of a relvar insert trace command must be a tuple value whose heading matches that of the relvar on which the insert is performed. In the `InsertTuple` procedure below, we construct a new tuple whose heading is the merge of the default values heading and the heading of the tuple as it was inserted. Similarly, the value of the newly constructed tuple is the merge of the default values and the tuple value as it was inserted. Note that `dict merge` does just what we want when we merge the tuple onto the default heading and values. Any dictionary entries that are in the inserted tuple will take precedence over any defaults. If the defaults are missing from what is inserted, then the merge will put them into place.

```
<<helper commands>>=
proc DefValueTrace {defheading defvalues op relvar tuple} {
    tuple create\
        [dict merge $defheading [tuple heading $tuple]]\
        [dict merge $defvalues [tuple get $tuple]]
}
```

To insert the trace, we must find the attributes of the class that have been configured to have a default value. That query is shown below. After finding all the attributes of the class, we join DefaultValue to obtain the set of attributes that have defined default values.

```
<<rosea data>>=
pipe {
    relation semijoin $class $Attribute -using {Domain Domain Name Class} |
    relation join ~ $DefaultValue -using {Domain Domain Class Class Name Attribute}
} defaultValuesQuery

<<class query variables>>=
variable defaultValuesQuery
```

We test to make sure that we need to add a trace at all. The `relation dict` command takes the default values relation and extracts the needed columns and returns just the form we need to create the command prefix using the `DefValueTrace` command.

```
<<generate: setup default values>>=
set defaultvalues [eval $defaultValuesQuery]

if {[relation isnotempty $defaultvalues]} {
    relvar trace add variable $className insert [list\
        ::rosea::Helpers::DefValueTrace\
        [relation dict $defaultvalues Name Type]\
        [relation dict $defaultvalues Name Value]\
    ]
}
```

Tests

```
<<generate command tests>>=
test generate-1.2 {
    Generate domain relvars, default values
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute y1 string -id 1
                attribute x2 string -default foo
                attribute x3 int -default 20
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red y1 truck]
    rosea tunnel $ref readAttribute x2 x3
} -result {foo 20}
```

```
<<generate command tests>>=
test generate-1.3 {
    Generate domain relvars, override default values
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute y1 string -id 1
                attribute x2 string -default foo
                attribute x3 int -default 20
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red y1 truck x2 bar x3 40]
    rosea tunnel $ref readAttribute x2 x3
} -result {bar 40}
```

Setup Value Checks

Attributes may have a value check expression associated with them. This allows an easy tuple-by-tuple check on attribute values and is useful when an attribute can only assume a subset of values of its data type. For example, we might have an `int` attribute but would like to insure that it always is in the range of 10 to 20.

Like default values, value checks are implemented using relvar tracing. We just want the trace to fail if the supplied expression is false. We follow the same pattern that we used for default values. We will establish a helper command that does the expression evaluation. If the expression fails, then an error is thrown. Otherwise the input tuple is returned unmodified.

```
<<helper commands>>=
proc CheckValueTrace {attrchecks op relvar args} {
```

```
    if {$op eq "insert"} {
        set tuple [lindex $args 0]
        EvalAttrCheck $attrchecks $tuple
        return $tuple
    } elseif {$op eq "update"} {
        set tuple [lindex $args 1]
        EvalAttrCheck $attrchecks $tuple
        return $tuple
    } elseif {$op eq "set"} {
        set relvalue [lindex $args 0]
        relation foreach inst $relvalue {
            set tuple [relation tuple $inst]
            EvalAttrCheck $attrchecks $tuple
        }
        return $relvalue
    } else {
        tailcall DeclError RELVAR_TRACE_OP $op
    }
}
```

We factor all the real work into another helper procedure.

```
<<helper commands>>=
proc EvalAttrCheck {attrchecks tuple} {
    dict for {attrname checkexpr} $attrchecks {
        tuple assign $tuple
        set result [expr $checkexpr]
        if {!$result} {
            tailcall DeclError ATTR_CHECK_FAILED $attrname [tuple get $tuple]\
                $checkexpr $result
        }
    }
}
```

```
<<error code formats>>=
RELVAR_TRACE_OP     {unexpected relvar trace operation, "%s"}
ATTR_CHECK_FAILED   {check for attribute, "%s", failed:\
                    instance value was, "%s": "%s" evaluated to "%s"}
```

The query for obtaining those attributes which have a check expression is very similar to that for default values.

```
<<rosea data>>=
pipe {
    relation semijoin $class $Attribute -using {Domain Domain Name Class} |
    relation join ~ $ValueCheck -using {Domain Domain Class Class Name Attribute}
} checkValuesQuery

<<class query variables>>=
variable checkValuesQuery
```

And finally we set up the relvar trace for the checks. Note that the expressions to be run are passed in a dictionary along with the attribute name. This gives us all the information we need to construct a reasonable error message if the check fails and allows us to combine all the attribute check expression evaluation into a single trace callback.

```
<<generate: setup value checks>>=
set checkvalues [eval $checkValuesQuery]

if {[relation isnotempty $checkvalues]} {
    relvar trace add variable $className {insert update set} [list\
        ::rosea::Helpers::CheckValueTrace\
        [relation dict $checkvalues Name Expression]\
```

```
    ]
}
```

Tests

```
<<generate command tests>>=
test generate-1.6 {
    Generate domain relvars, check values on create
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute x2 int -check {$x2 > 1 && $x2 < 10}
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red x2 5]
    rosea tunnel $ref readAttribute x2
} -result {5}
```

```
<<generate command tests>>=
test generate-1.7 {
    Generate domain relvars, check values on update
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute x2 int -check {$x2 > 1 && $x2 < 10}
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red x2 5]
    rosea tunnel $ref updateAttribute x2 7
    rosea tunnel $ref readAttribute x2
} -result {7}
```

```
<<generate command tests>>=
test generate-1.8 {
    Generate domain relvars, value check fails
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
```

```
                attribute x1 string -id 1
                attribute x2 int -check {$x2 > 1 && $x2 < 10}
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red x2 5]
    rosea tunnel $ref updateAttribute x2 17
} -result {check for attribute, "x2", failed: instance value was, "x1 red x2 17": "$x2 > 1  ↩
    && $x2 < 10" evaluated to "0"}\
    -returnCodes error
```

```
<<generate command tests>>=
test generate-1.9 {
    Generate domain relvars, default value and a check expression
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute x2 int -default 5 -check {$x2 > 1 && $x2 < 10}
            }
        }
    }
    rosea generate
    set ref [::foo::X create x1 red]
    rosea tunnel $ref readAttribute x2
} -result {5}
```

Create State Model

To determine if a class has a state model, we use the following query.

```
<<rosea data>>=
pipe {
    relation semijoin $class $StateModel -using {Domain Domain Name Model} |
    relation isnotempty
} hasSMQuery

<<class query variables>>=
variable hasSMQuery
```

For classes that have a state model, we must create a relvar to hold the current state of the instances. Further, we will create a child namespace to hold the state activities and the procedures for the activities themselves.

```
<<generate: create state model>>=
set hasStateModel [eval $hasSMQuery]
if {$hasStateModel} {
    <<generate: create current state relvar>>
    <<generate: create activity namespace>>
    <<generate: create activity procedures>>
}
```

Create Current State Relvar

We store the current state for classes instances that have a state model in a separate relvar. This makes sure that the current state is not accidentally tampered with during normal access to the class model. Since this is Tcl, nothing is really hidden or inaccessible. If you know the name, you can get the data. However, separating the current state into its own relvar does prevent the worst case accidents and abuses.

The heading for the relvar that holds the current state is different for each class. It is composed of the identifying attributes and one other attribute named, `__State`. We use the attributes of the lowest numbered identifier (*i.e.* 1) since that is the identifier that is used to construct instance references. The relvar is named in the form, `__<class name>__STATEINST` to prevent any name conflicts.

```
<<generate: create current state relvar>>=
set instid [lindex $idset 0]
set instheading [list]
foreach attr $instid {
    lappend instheading $attr [dict get $heading $attr] ; # ❶
}
lappend instheading __State string
relvar create ${domns}::__[relation extract $class Name]__STATEINST\
    $instheading $instid
```

❶ In addition to the attribute name, we need the type information to construct the relvar header.

Create Activity Namespace

We want to keep the state activity procedures in their own namespace to avoid any naming conflicts with class operations. Also, a separate namespace allows us to set the command resolution path to something convenient so that when the state activities execute the common procedures they will use can be invoked using unqualified names.

```
<<generate: create activity namespace>>=
set actns ${className}::__Activity
namespace eval $actns {
    <<tclral imports>>
}
namespace eval $actns [list\
    namespace path [list\
        ::rosea::InstCmds\
        $className\
        $domns\
    ]\
]
```

Create Activity Procedures

The state activities are realized as ordinary procedures in the `__Activity` namespace of the class. We obtain the state names and code using the query below.

```
<<rosea data>>=
pipe {
    relation semijoin $class $State -using {Domain Domain Name Model} |
    relation project ~ Name Parameters Action
} statesQuery

<<class query variables>>=
variable statesQuery
```

Iterating across the queried relation value, we invoke `::proc` to create the state activity procedures. Note the insertion of `self` as the first argument of the activity procedure.

```
<<generate: create activity procedures>>=
relation foreach state [eval $statesQuery] {
    relation assign $state\
        {Name stateName}\
        {Parameters params}\
        {Action stateAction}
    proc ${actns}::$stateName [linsert $params 0 self] $stateAction
}
```

Create Class Operations

For each class in the domain, we will create an ensemble command that is the same name as the fully qualified name of the class. There are two sources of class operations, those supplied by the system and those defined when the class was configured. Further, some of the system supplied operations are only valid for classes that have a state model. It is also the case, that the same considerations apply regarding instance based operations. Some are system supplied others are user supplied and some of the system instance operations only make sense if there is a state model associated with the class. Consequently, we will factor out some helper procedures to make these queries for us.

We start with a query to obtain system supplied operations. The procedure below obtains the system supplied operations, filters out those that meet the criteria for a state model and then joins to select the desired type, either instance or class based.

```
<<helper commands>>=
proc QuerySystemOperations {which requiresSM} {
    return [pipe {
        relvar set ::rosea::Config::SuppliedOperation |
        relation restrictwith ~ {$RequiresStateModel == $requiresSM} |
        relation join [relvar set ::rosea::Config::$which] ~
    }]
}
```

We can specialize this query for class based operations by supplying the variable parts. Here we need to augment the Command attribute by including the name of the class (it is fully qualified here). This supplies the first argument to all class based operations, the name of the class itself. The return value is a dictionary that we will use in creating the ensemble command.

```
<<helper commands>>=
proc QuerySystemClassOperations {className requiresSM} {
    return [pipe {
        QuerySystemOperations SystemClassOperation $requiresSM |
        relation update ~ sco 1 {
            tuple update $sco Command\
                [concat [tuple extract $sco Command]\
                [list $className]]
        } |
        relation dict ~ Name Command
    }]
}
```

For instance based operations, the instance reference is supplied by the caller and so we can just return the dictionary we find.

```
<<helper commands>>=
proc QuerySystemInstanceOperations {requiresSM} {
    return [pipe {
        QuerySystemOperations SystemInstanceOperation $requiresSM |
        relation dict ~ Name Command
    }]
}
```

To create the class based operations, we use the `::proc` command to create the ones configured for the class. We keep track of a mapping dictionary as we go along. This mapping dictionary is used when the class ensemble command is created to provide

for the mapping of ensemble subcommand onto the procedures that implement them. That mapping is an essential feature of namespace ensemble commands and we make use of the feature here. The system supplied operations do depend upon whether the class has a state model.

```
<<generate: create class operations>>=
set opmap [dict create]
set classops [relation semijoin $class $UserClassOperation\
    -using {Domain Domain Name Class}]
relation foreach classop $classops {
    relation assign $classop\
        {Name opname}\
        {Parameters params}\
        {Body opbody}

    proc ${className}::$opname $params $opbody
    dict set opmap $opname ${className}::$opname
}

set sysmap [QuerySystemClassOperations $className false]
if {$hasStateModel} {
    set sysmap [dict merge $sysmap [QuerySystemClassOperations $className true]]
}

namespace ensemble create\
    -command $className\
    -map [dict merge $opmap $sysmap]
```

Tests

```
<<generate command tests>>=
test generate-1.4 {
    generate class based operations
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                attribute a2 string

                classop echo {id} {
                    return $id
                }
            }
        }
    }
    rosea generate
    ::foo::c1 echo 27
} -result {27}
```

Create Instance Operations

Similar logic applies when creating the instance based operations for a class. It is somewhat simpler in that the instance reference is a caller supplied

```
<<generate: create instance operations>>=
set opmap [dict create]
set instops [relation semijoin $class $UserInstanceOperation\
    -using {Domain Domain Name Class}]
relation foreach instop $instops {
    relation assign $instop\
        {Name opname}\
        {Parameters params}\
        {Body opbody}

    proc ${className}::$opname [linsert $params 0 self] $opbody
    dict set opmap $opname ${className}::$opname
}
set sysmap [QuerySystemInstanceOperations false]
set hasPolyEvents [relation isnotempty\
    [relation semijoin $class $PolymorphicEvent\
        -using {Domain Domain Name Model}]\
]
if {$hasStateModel || $hasPolyEvents} {
    set sysmap [dict merge $sysmap [QuerySystemInstanceOperations true]]
}

namespace ensemble create\
    -command ${className}::Instance\
    -parameters instref\
    -map [dict merge $opmap $sysmap]
```

Tests

```
<<generate command tests>>=
test generate-1.5 {
    generate instance based operations
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class c1 {
                attribute a1 string -id 1
                attribute a2 string

                instop echo {id} {
                    return $id
                }
            }
        }
    }
    rosea generate
    set ref [::foo::c1 create a1 bar a2 baz]
    rosea tunnel $ref echo 27
} -result {27}
```

Create Relationship Data and Commands

For relationships, we need to similar work as for classes. We have to population the architecture relvars that drive the run time processing of relationship traversal and linking. Each relationship has an ensemble command associated with it. There there is

the nuance of assigners that might be defined on the relationship.

We will categorize relationships as simple, class based or generalizations and the require processing will be different for each category.

```
<<generate: create relationship data and commands>>=
set rships [relation semijoin $domain $DomainElement -using {Name Domain}\
        $Relationship -using {Domain Domain Element Name}]

<<generate: simple associations>>
<<generate: class based associations>>
<<generate: generalizations>>
```

Simple Associations

For simple associations, we traverse the R30 and R31 generalizations in the configuration model. Continuing on we can come up with a set of SimpleReferringClass and SimpleReferencedClass instances.

```
<<generate: simple associations>>=
set sassocs [relation semijoin $rships $Association\
        $SimpleAssociation]
set refing [relation semijoin $sassocs $SimpleReferringClass\
        -using {Domain Domain Name Relationship}]
set refed [relation semijoin $sassocs $SimpleReferencedClass\
        -using {Domain Domain Name Relationship}]
```

We now undertake a query on the configuration relvars to gather up all the information required for the later parts of the generation of simple relationships. Gathering all the information in one query makes that query rather complicated. Much of the complexity arises from needing the same information in slightly different forms in order to satisfy the interface requirements of the procedures that create underlying `ral` entities. The main goal of the query is to obtain the class and attributes that play a referring role in a relationship and the corresponding class and attributes that are referenced in the relationship. Setting up the correspondence between referring and referenced is the basis for defining the relvar constraints and the run time data that is used for navigation and linking. We present the entire query and then explain each step.

```
<<rosea data>>=
pipe {
    relation join $AttributeReference $refing\
        -using {Domain Domain ReferringClass Class\
        Relationship Relationship ReferringRole Role} |
    relation rename ~ Conditionality ReferringCond\
            Multiplicity ReferringMult |
    relation join ~ $refed -using {Domain Domain\
        ReferencedClass Class Relationship\
        Relationship ReferencedRole Role} |
    relation project ~ Domain Relationship\
        ReferringClass ReferringAttribute\
        ReferencedClass ReferencedAttribute\
        ReferringCond ReferringMult Conditionality |
    relation group ~ ReferringAttrs\
            ReferringAttribute ReferencedAttribute |
    relation extend ~ rfa\
        TagReferringAttrs {Relation {RefOrder int ReferringAttribute string\
            ReferencedAttribute string}} {
                [relation tag [tuple extract $rfa ReferringAttrs] RefOrder]
            } |
    relation extend ~ wfa\
        SrcAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferringAttribute -ascending RefOrder]
        }\
```

```
        DstAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferencedAttribute -ascending RefOrder]
        } |
    relation extend ~ ram\
        ForwAttrs list {
            [Interleave [tuple extract $ram SrcAttrs]\
                [tuple extract $ram DstAttrs]]
        }\
        RevAttrs list {
            [Interleave [tuple extract $ram DstAttrs]\
                [tuple extract $ram SrcAttrs]]
        } |
    relation eliminate ~ TagReferringAttrs
} referencesQuery

<<generate: simple associations>>=
variable referencesQuery
set references [eval $referencesQuery]
```

Going command by command we can explain the query.

1. By joining the Simple Referring Class instances to the Attribute Reference, we can obtain the information on those attributes that refer to the other class in the association and which attributes are referenced.
2. We need to rename some attributes to prevent any name conflicts that would occur in the next command.
3. Joining to the Simple Reference Class instances, we obtain the information about the Conditionality of the association.
4. We project out only the attributes that we intend to use.
5. By grouping the ReferringAttribute and ReferencedAttribute we create a relation valued attribute expressly shows the correspondence between referring and referenced. Each tuple of the resulting ReferringAttrs attribute maps a referring attribute to a referenced attribute. There will be as many tuples in the value as there are attribute in the identifier being referenced. As it turns out, the ReferringAttrs value is what is needed to support run time linking of the association.
6. The form of correspondence we generated in the last step will need to be manipulated into different lists and alternating lists for the benefit of some of the interfaces of procedures we will need to call. We must be careful to keep the correspondence correct. To that end we will `tag` the ReferringAttrs value. A new attribute is created by extending the relationship with yet another relation value attribute that has a tag value added as a new column. The `relation tag` command will put sequential integers into this new tagging attribute.
7. In this step we generate the attribute representation that we need in order to create the relvar constraint. The `relvar association` command need a list of referring attributes and a list of referenced attributes and expects the corresponding items in each list to form the constraint reference. So in this query, we extend out two more attributes to be what is needed by `relvar association`. Notice that we create the lists in ascending order of the tag, thereby insuring the correspondence between the two lists.
8. The next we generate the attribute reference representation needed to support run time traversal of the association. Here we need to supply a list of attributes for each direction of the traversal. Ultimately, these lists will be used as an argument to the `relation semijoin` command. The command expects a list consisting of alternating referring and referenced attribute names. So we extend a couple of more attributes and set their values by interleaving the lists we created in the previous step. The interleaving process forms a new list by taking an element from each list alternately.
9. Lastly, we discard the tagged attributes as they are no longer needed.

The `Interleave` procedure used above creates a list by alternating elements from two lists.

```
<<helper commands>>=
proc Interleave {l1 l2} {
    set result [list]
```

```
    foreach a $l1 b $l2 {
        lappend result $a $b
    }
    return $result
}
```

The results of the query, as held in the `references` variable, can now be projected out and placed into the architecture relvars used at run time and can be used to create the relationship ensemble commands. The processing performed here is a good example of why the the run time procedures were designed to operate on much simpler data structures. The data used by the run time is the results of a rather complicated query, pulled apart for specific operations. This prevents having to perform such complicated queries at run time. You can also think of the run time data as a *simplified view* of the configuration data that is accomplished by pushing the computation of the view to the time when the domain is *generated* since class structure and relationships definitions do not change during the running of the domain.

Continuing on, we need to populate the run time data that deals with simple associations.

```
<<generate: simple associations>>=
relvar eval {
    <<generate: populate Link for simple associations>>
    <<generate: populate AssociationLink for simple associations>>
    <<generate: populate RefLink for simple associations>>
}
```

The Link and AssociationLink relvars hold the run time data for navigating the association relationships. We first take the referring class information and associate it with the forward direction for navigation. The reverse direction, *i.e.* that direction given by the `~RX` notation originates at the referenced class. For simple associations, we add two tuples to Link,one for each direction of traversal. The operations just project out the required attributes and then juggle the names to make the headers match.

```
<<generate: populate Link for simple associations>>=
set flink [pipe {
    relation project $references Relationship ReferringClass |
    relation rename ~ Relationship Name ReferringClass SrcClass
}]
set blink [pipe {
    relation project $references Relationship ReferencedClass |
    relation rename % Relationship Name ReferencedClass SrcClass |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%] ; # ❶

relvar union ${domns}::__Arch_Link $flink $blink
```

❶ For the reverse direction we need to tack on the tilde (~) character to the association name. It turns out that tilde has a special meaning to the `pipe` command and so we have to ask `pipe` to use percent (`%`) as its special character to represent where the previous command result is substituted.

The manipulations for populating the AssociationLink relvar are similar.

```
<<generate: populate AssociationLink for simple associations>>=
set flink [pipe {
    relation project $references Relationship ReferringClass\
            ReferencedClass ForwAttrs |
    relation rename ~ Relationship Name ReferringClass SrcClass\
            ReferencedClass DstClass ForwAttrs Attrs |
    relation extend ~ al PrevSrcClass string {{}}
}] ; # ❶
set blink [pipe {
    relation project $references Relationship ReferringClass\
            ReferencedClass RevAttrs |
    relation rename % Relationship Name ReferencedClass SrcClass\
```

```
            ReferringClass DstClass RevAttrs Attrs |
    relation extend % al PrevSrcClass string {{}} |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]

relvar union ${domns}::__Arch_AssociationLink $flink $blink
```

❶ We need to supply a PrevSrcClass attribute as the empty string and this is accomplished using `relation extend`.

For the case of RefLink, the ReferringAttrs attribute matches exactly and so a simple projection of attributes is all that is necessary.

```
<<generate: populate RefLink for simple associations>>=
relvar union ${domns}::__Arch_RefLink [relation project $references\
    Relationship ReferringClass ReferencedClass ReferringAttrs]
```

The last step for simple associations is to create the relvar constraint and relationship ensemble commands. For that we will iterate across the `references` relation since we must invoke the necessary commands one by one. It is usually necessary to iterate across relation values when needing to interface to scalar oriented procedures. Otherwise, we prefer to use the set-at-a-time operations of relational algebra.

```
<<generate: simple associations>>=
relation foreach reference $references {
    relation assign $reference\
        {Relationship relationship}\
        {ReferringClass referringClass}\
        {ReferencedClass referencedClass}\
        {ReferringCond referringCond}\
        {ReferringMult referringMult}\
        {Conditionality referencedCond}\
        {SrcAttrs srcAttrs}\
        {DstAttrs dstAttrs}

    # Create the relvar constraint.
    relvar association ${domns}::$relationship\
         ${domns}::$referringClass $srcAttrs\
        [MapCondMultToConstraint $referringCond $referringMult]\
        ${domns}::$referencedClass $dstAttrs\
        [MapCondMultToConstraint $referencedCond false]

    # All simple associations get a link and unlink command in their ensemble.
    set ensemblemap [dict create\
        link [list ::rosea::RelCmds::linkSimple $relationship]\
        unlink [list ::rosea::RelCmds::unlinkSimple $relationship]\
    ]

    # Determine if we have any assigners assocated  with the relationship.
    # Single assigners get different commands from multiple assigners.
    set rpath ${domns}::$relationship
    set assigner [relation semijoin $reference $SingleAssigner\
        -using {Domain Domain Relationship Relationship}]
    if {[relation isnotempty $assigner]} {
        dict set ensemblemap signal\
            [list ::rosea::RelCmds::signalAssigner $rpath]
    } else {
        set assigner [relation semijoin $reference $MultipleAssigner\
            -using {Domain Domain Relationship Relationship}]
        if {[relation isnotempty $assigner]} {
            dict set ensemblemap signal [list\
                ::rosea::RelCmds::signalMultiAssigner $rpath]
```

```
            dict set ensemblemap create [list\
                ::rosea::RelCmds::createMultiAssigner $rpath]
        }
    }
    # Create the relationship ensemble command.
    namespace ensemble create -command $rpath -map $ensemblemap
}
```

The configuration data holds the conditionality and multiplicity of the associations as booleans. We need to translate that into the regular expression like syntax that TclRAL uses.

```
<<helper commands>>=
proc MapCondMultToConstraint {cond mult} {
    if {$cond} {
        return [expr {$mult ? "*" : "?"}]
    } else {
        return [expr {$mult ? "+" : "1"}]
    }
}
```

Tests

```
<<generate command tests>>=
test generate-2.0 {
    generate constraints for simple associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class x {
                attribute a1 string -id 1
                attribute a2 string -default 20
            }
            class y {
                attribute a1 string -id 1
                reference R1 x -link a1
            }
            association R1 y 1--1 x
        }
    }
    rosea generate
    relvar constraint info ::foo::R1
} -result {association ::foo::R1 ::foo::y a1 1 ::foo::x a1 1}
```

```
<<test utility procs>>=
proc testConditions {args} {
    set result 1
    foreach exp $args {
        set passed [uplevel 1 [list expr $exp]]
        if {!$passed} {
            log::error "\"$exp\" failed"
            set result 0
        }
    }
    return $result
}
```

```
<<generate command tests>>=
test generate-2.1 {
    association run time data
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class X {
                attribute x1 string -id 1
                attribute x2 string -default 20
            }
            class Y {
                attribute y1 string -id 1
                reference R1 X -link {y1 x1}
            }
            association R1 Y 1--1 X
        }
    }
    rosea generate

    set forwattrs [pipe {
        relvar restrictone ::foo::__Arch_AssociationLink Name R1 SrcClass Y |
        relation extract ~ Attrs
    }]

    set revattrs [pipe {
        relvar restrictone ::foo::__Arch_AssociationLink Name ~R1 SrcClass X |
        relation extract % Attrs
    } {} |%]

    testConditions\
        {[relation cardinality $::foo::__Arch_Link] == 2}\
        {[relation cardinality $::foo::__Arch_RefLink] == 1}\
        {[llength [info commands ::foo::R1]] == 1}\
        {$forwattrs eq {y1 x1}}\
        {$revattrs eq {x1 y1}}

} -result {1}
```

```
<<generate command tests>>=
test generate-2.2 {
    generate constraints for simple reflexive associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class x {
                attribute a1 string -id 1
                attribute prev string -id 2
                attribute a2 string -default 20
                reference R1 x -link {prev a1}
            }
            association R1 x 1--1 x
        }
    }
```

```
    rosea generate
    relvar constraint info ::foo::R1
} -result {association ::foo::R1 ::foo::x prev 1 ::foo::x a1 1}
```

```
<<generate command tests>>=
test generate-2.3 {
    traversal for simple reflexive associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class x {
                attribute a1 string -id 1
                attribute prev string -id 2
                attribute a2 string -default 20
                reference R1 x -link {prev a1}
            }
            association R1 x ?--? x
        }
    }
    rosea generate
    relvar eval {
        set f [::foo::x create a1 fred prev {} a2 10]
        set j [::foo::x create a1 john prev fred a2 20]
    }
    set r [rosea tunnel $f findRelated ~R1]
    rosea tunnel $r readAttribute a2
} -result {20}
```

```
<<generate command tests>>=
test generate-2.4 {
    navigate associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    rosea configure {
        domain meta {
            class Class {
                attribute Domain string -id 1
                attribute Name string -id 1
            }

            class Attribute {
                attribute Domain string -id 1
                attribute Class string -id 1
                attribute Name string -id 1
                attribute Type string

                reference R3 Class -link Domain -link {Class Name}
            }
            association R3 Attribute 1..*--1 Class

            operation showAttrs {class} {
                set c1 [Class findWhere\
                        {$Domain eq "foo" && $Name eq $class}]
                set attrs [instop $c1 findRelated ~R3]
```

```
                return [instop $attrs deRef]
            }
        }
    }
    rosea generate
    relvar eval {
        meta::Class create Domain foo Name c1
        meta::Attribute create Domain foo Class c1 Name a1 Type string
        meta::Attribute create Domain foo Class c1 Name a2 Type string
    }
    meta::showAttrs c1
} -result {{Domain string Class string Name string Type string} {{Domain foo Class c1 Name a1 Type string} {Domain foo Class c1 Name a2 Type string}}}\
    -match relation
```

```
<<generate command tests>>=
test generate-2.5 {
    link instances across association
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 1..*--1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    relvar eval {
        set c1 [meta::C1 create Id foo A1 20]
        meta::C2 create Id bar C1Id {}
        meta::C2 create Id baz C1Id {}
        set c2 [meta::C2 create Id fub C1Id {}]
        meta::R1 link $c1 [meta::C2 findAll]
    }
    rosea tunnel $c2 readAttribute C1Id
} -result {foo}
```

```
<<generate command tests>>=
test generate-2.6 {
    unlink instances across association
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
```

```
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    relvar eval {
        set c1 [meta::C1 create Id foo A1 20]
        meta::C2 create Id bar C1Id foo
        meta::C2 create Id baz C1Id foo
        set c2 [meta::C2 create Id fub C1Id foo]
    }
    set before [rosea::InstCmds::refMultiplicity\
        [rosea::InstCmds::findRelated $c1 ~R1]]
    meta::R1 unlink [meta::C2 findWhere {[string match b* $Id]}]
    set after [rosea::InstCmds::refMultiplicity\
        [rosea::InstCmds::findRelated $c1 ~R1]]
    testConditions\
        {$before == 3}\
        {$after == 1}
} -result {1}
```

```
<<generate command tests>>=
test generate-2.7 {
    unlink using referred to instance
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    relvar eval {
        set c1 [meta::C1 create Id foo A1 20]
        meta::C2 create Id bar C1Id foo
        meta::C2 create Id baz C1Id foo
        meta::C2 create Id fub C1Id foo
    }
    set before [rosea::InstCmds::refMultiplicity\
        [rosea::InstCmds::findRelated $c1 ~R1]]
    meta::R1 unlink $c1
    set after [rosea::InstCmds::refMultiplicity\
```

```
        [rosea::InstCmds::findRelated $c1 ~R1]]
    testConditions\
        {$before == 3}\
        {$after == 0}
} -result {1}
```

```
<<generate command tests>>=
test generate-2.8 {
    unlink and violate referential integrity
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 1..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    catch {relvar eval {
        set c1 [meta::C1 create Id foo A1 20]
        meta::C2 create Id bar C1Id foo
        meta::C2 create Id baz C1Id foo
        meta::C2 create Id fub C1Id foo
        meta::R1 unlink $c1}
    } result
    set result
} -result {for association*::meta::R1*} -match glob
```

Class Based Associations

For class based associations, we will follow a similar strategy as for simple associations. The use of an association class does cause some additional complexity as we shall see.

We have to traverse the configuration model down the path to the Class Based Association relvar. From there we obtain the source and target classes that participate in the association.

```
<<generate: class based associations>>=
set cassocs [relation semijoin $rships $Association\
        $ClassBasedAssociation]
set sourceclass [relation semijoin $cassocs $SourceClass\
    -using {Domain Domain Name Relationship}]
set targetclass [relation semijoin $cassocs $TargetClass\
    -using {Domain Domain Name Relationship}]
set bothclass [relation union $sourceclass $targetclass] ; # ❶
```

❶ The headings of Source Class and Target Class are the same so this union is defined.

As with simple associations, we make one large query to accumulate the information we need. We present the query below, followed by a step by step explanation.

```
<<rosea data>>=
pipe {
    relation semijoin $cassocs $AssociatorClass\
            -using {Domain Domain Name Relationship}\
        $ReferringClass $AttributeReference\
            -using {Domain Domain Class ReferringClass\
            Relationship Relationship Role ReferringRole} |
    relation project ~ Domain Relationship ReferringClass ReferringAttribute\
            ReferencedClass ReferencedAttribute ReferencedRole |
    relation join ~ $bothclass -using {Domain Domain Relationship Relationship\
            ReferencedClass Class ReferencedRole Role} |
    relation group ~ ReferringAttrs ReferringAttribute ReferencedAttribute |
    relation extend ~ rfa\
        TagReferringAttrs {Relation {RefOrder int ReferringAttribute string\
            ReferencedAttribute string}} {
                [relation tag [tuple extract $rfa ReferringAttrs] RefOrder]
            } |
    relation extend ~ wfa\
        SrcAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferringAttribute -ascending RefOrder]
        }\
        DstAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferencedAttribute -ascending RefOrder]
        } |
    relation extend ~ ram\
        ForwAttrs list {
            [Interleave [tuple extract $ram SrcAttrs]\
                [tuple extract $ram DstAttrs]]
        }\
        RevAttrs list {
            [Interleave [tuple extract $ram DstAttrs]\
                [tuple extract $ram SrcAttrs]]
        } |
    relation eliminate ~ TagReferringAttrs
} assocrefsQuery

<<generate: class based associations>>=
variable assocrefsQuery
set assocrefs [eval $assocrefsQuery]
```

1. Traverse from the Associator Class to the Referring Class and on to the Attribute Reference class. This is like the query for the simple associations but we take a different path from the Relationship class to get there. This query is really all about working out the attribute references.
2. Project the attributes we find useful.
3. Joining the Source Class and Target Class tuples now gives us a relation with references that the associator class makes to both the Source and Target Classes. The ReferencedRole attribute distinguishes the two cases.
4. The `group` operation gives a relation valued attribute that contains the correspondence between referential attributes in the associator class and the referenced attributes in the source or target class.
5. Like in the simple associations case, we need to take the attribute references to be able to preserve the correspondence in later operations.
6. Compute the form of the attribute references needed for creating the relvar constraint.

7. Compute the form of the attribute references needed for relationship traversal.
8. The TagReferringAttrs attribute is no longer needed.

With the query done, we can now populate the run time data that is needed for class based associations. We have to populate Link and AssociationLink to support relationship navigation and ASsocRef to support relationship linkage.

```
<<generate: class based associations>>=
relvar eval {
    <<generate: populate Link for class based associations>>
    <<generate: populate AssociationLink for class based associations>>
    <<generate: populate AssocRef for class based associations>>
}
```

For class based association, the Link relvar contains tuples for the implied two traversals needed to go from the source to the associator and then to the target and *vice versa*. For the associative class, there both a forward and reverse path that only differ by the syntax convention of naming the reverse path with a tilde.

```
<<generate: populate Link for class based associations>>=
set aforw [pipe {
    relation project $assocrefs Relationship ReferringClass |
    relation rename ~ Relationship Name ReferringClass SrcClass
}]
set arev [relation update $aforw lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}]
```

For the classes that are sources, there is a forward path to the associative class.

```
<<generate: populate Link for class based associations>>=
set sources [pipe {
    relation restrict $assocrefs aref {
        [tuple extract $aref ReferencedRole] eq "source"} |
    relation eliminate ~ ReferencedRole
}]
set sforw [pipe {
    relation project $sources Relationship ReferencedClass |
    relation rename ~ Relationship Name ReferencedClass SrcClass
}]
```

And for classes that are targets, there is a reverse path from the target to the associative class.

```
<<generate: populate Link for class based associations>>=
set targets [pipe {
    relation restrict $assocrefs aref {
        [tuple extract $aref ReferencedRole] eq "target"} |
    relation eliminate ~ ReferencedRole
}]
set trev [pipe {
    relation project $targets Relationship ReferencedClass |
    relation rename % Relationship Name ReferencedClass SrcClass |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]
```

The set of link instances that we need is then just the union of the four sets we just computed.

```
<<generate: populate Link for class based associations>>=
relvar union ${domns}::__Arch_Link $aforw $arev $sforw $trev
```

For each Link tuple we need a corresponding AssociationLink tuple. To describe the traversal from Source Class to Associator to Target Class requires entries that chain together the individual steps and so will have a non-empty value for the PrevSrcClass attribute. The start of the traversal from either end, *i.e.* forward from Source to Associator and reverse from Target to Associator, has an empty value for the PrevSrcClass attribute. We compute those first.

```
<<generate: populate AssociationLink for class based associations>>=
set sforw [pipe {
    relation project $sources Relationship ReferringClass ReferencedClass\
        RevAttrs |
    relation rename ~ Relationship Name ReferringClass DstClass\
        ReferencedClass SrcClass RevAttrs Attrs |
    relation extend ~ al PrevSrcClass string {{}}
}]
set trev [pipe {
    relation project $targets Relationship ReferringClass ReferencedClass\
        RevAttrs |
    relation rename % Relationship Name ReferringClass DstClass\
        ReferencedClass SrcClass RevAttrs Attrs |
    relation extend % al PrevSrcClass string {{}} |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]
```

To compute the other two tuples in AssociationLink, we need the correct value for the PrevSrcClass attribute. When traversing from the associator class to target class that is the name of the source class and *vice versa* for the opposite direction. To get all this information in one relation value we will `join` the sources to the targets. To do that, we need to rename the attributes (they are the same name in both the `sources` and `targets` relations). We give the attributes associated with the SourceClass the prefix `One` and the attributes associated with the TargetClass the prefix `Other`. This is just a naming convention to try to keep things straight and minimize the confusion for something that is already difficult enough.

```
<<generate: populate AssociationLink for class based associations>>=
set onerefs [relation rename $sources ReferencedClass OneClass\
        Conditionality OneCond Multiplicity OneMult\
        ReferringAttrs OneReferringAttrs SrcAttrs OneSrcAttrs\
        DstAttrs OneDstAttrs ForwAttrs OneForwAttrs RevAttrs OneRevAttrs]
set otherrefs [relation rename $targets ReferencedClass OtherClass\
        Conditionality OtherCond Multiplicity OtherMult ReferringAttrs\
        OtherReferringAttrs SrcAttrs OtherSrcAttrs DstAttrs OtherDstAttrs\
        ForwAttrs OtherForwAttrs RevAttrs OtherRevAttrs]
```

Now we can `join` the two relation and get one big relation that contains all the information needed determine the other two tuples for the AssociationLink relvar.

```
<<generate: populate AssociationLink for class based associations>>=
set bothrefs [relation join $onerefs $otherrefs\
    -using {Domain Domain ReferringClass ReferringClass\
        Relationship Relationship}]
```

So now we go back to the pattern of projecting out the needed data. The navigation from the associative class to the target (aka OtherClass) is in the forward direction and preceded by the source (aka OneClass).

```
<<generate: populate AssociationLink for class based associations>>=
set aforw [pipe {
    relation project $bothrefs Relationship ReferringClass OtherClass\
        OtherForwAttrs OneClass |
    relation rename ~ Relationship Name ReferringClass SrcClass\
            OtherClass DstClass OtherForwAttrs Attrs OneClass PrevSrcClass
}]
```

The navigation from the association class to the source (aka OneClass) is in the reverse direction and preceded by the target (aka OtherClass). And we have change the relationship name to include the tilde.

```
<<generate: populate AssociationLink for class based associations>>=
set arev [pipe {
    relation project $bothrefs Relationship ReferringClass OneClass\
```

```
            OneForwAttrs OtherClass |
    relation rename % Relationship Name ReferringClass SrcClass\
            OneClass DstClass OneForwAttrs Attrs OtherClass PrevSrcClass |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]
```

Finally, we can combine the four types of AssociationLink relations and install the result into the relvar.

```
<<generate: populate AssociationLink for class based associations>>=
relvar union ${domns}::__Arch_AssociationLink $sforw $trev $aforw $arev
```

The AssocRef relvar holds the information needed to link and unlink class based association. All the information is present, we need only project it out and juggle the naming. The relation value attribute shown in the graphic is obtained using `relation group`.

```
<<generate: populate AssocRef for class based associations>>=
relvar union ${domns}::__Arch_AssocRef [pipe {
    relation project $assocrefs Relationship ReferringClass\
        ReferencedClass ReferencedRole ReferringAttrs |
    relation rename ~ ReferringClass AssocClass ReferencedClass Participant\
        ReferencedRole Role |
    relation group ~ References Participant Role ReferringAttrs
}]
```

Again, following the pattern from simple associations, we will need to iterate over the class based association so that we can create the `relvar correlation` constraint and the relationship ensemble command. We will iterate over the tuples in the `bothrefs` variable since it has all the information about the relationship in one place.

```
<<generate: class based associations>>=
relation foreach bothref $bothrefs {
    relation assign $bothref\
        {Relationship relationship}\
        {ReferringClass associator}\
        {OneClass sourceClass}\
        {OneCond sourceCond}\
        {OneMult sourceMult}\
        {OneSrcAttrs sourceAssocAttrs}\
        {OneDstAttrs sourceDstAttrs}\
        {OtherClass targetClass}\
        {OtherCond targetCond}\
        {OtherMult targetMult}\
        {OtherSrcAttrs targetAssocAttrs}\
        {OtherDstAttrs targetDstAttrs}

    # Create the correlation constraint
    relvar correlation ${domns}::$relationship ${domns}::$associator\
        $sourceAssocAttrs [MapCondMultToConstraint $targetCond $targetMult]\
        ${domns}::$sourceClass $sourceDstAttrs\
        $targetAssocAttrs [MapCondMultToConstraint $sourceCond $sourceMult]\
        ${domns}::$targetClass $targetDstAttrs ; # ❶

    set ensemblemap [dict create\
        link [list ::rosea::RelCmds::linkAssoc $relationship]\
        unlink [list ::rosea::RelCmds::unlinkAssoc $relationship]\
    ]
    # Determine if we have any assigners
    set rpath ${domns}::$relationship
    set assigner [relation semijoin $bothref $SingleAssigner\
        -using {Domain Domain Relationship Relationship}]
    if {[relation isnotempty $assigner]} {
        dict set ensemblemap signal\
```

```
            [list ::rosea::RelCmds::signalAssigner $rpath]
    } else {
        set assigner [relation semijoin $bothref $MultipleAssigner\
            -using {Domain Domain Relationship Relationship}]
        if {[relation isnotempty $assigner]} {
            dict set ensemblemap signal [list\
                ::rosea::RelCmds::signalMultiAssigner $rpath]
            dict set ensemblemap create [list\
                ::rosea::RelCmds::createMultiAssigner $rpath]
        }
    }
    namespace ensemble create -command $rpath -map $ensemblemap
}
```

❶ How the conditionality and multiplicity are set needs some explanation. The configuration DSL uses syntax that mimics the UML graphic. This specifies the multiplicity and conditionality from the point of view of the two participating classes. However, the `relvar correlation` command specifies multiplicity and conditionality from the point of view of the associative class. If you work through that difference in perspective, it means you have to invert the multiplicity and conditionality from source and target for the benefit of TclRAL. Both ways of specifying this work well in the respective worlds, but when you cross the boundary you have to remember the switch.

Tests

```
<<generate command tests>>=
test generate-3.0 {
    generate constraints for class based associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class a {
                attribute ax1 string -id 1
                attribute ax2 string -id 1
                attribute ay1 string -id 1
                attribute ay2 string -id 1
                reference R1 x -link {ax1 x1} -link {ax2 x2}
                reference R1 y -link {ay1 y1} -link {ay2 y2}
            }
            class x {
                attribute x1 string -id 1
                attribute x2 string -id 1
                attribute x3 string -default 20
            }
            class y {
                attribute y1 string -id 1
                attribute y2 string -id 1
            }
            association R1 x 0..*--1..* y -associator a
        }
    }
    rosea generate
    relvar constraint info ::foo::R1
} -result {correlation ::foo::R1 ::foo::a {ax1 ax2} + ::foo::x {x1 x2} {ay1 ay2} * ::foo::y ↩
    {y1 y2}}
```

```
<<generate command tests>>=
test generate-3.1 {
    class based association when simple would do
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class a {
                attribute x1 string -id 1
                attribute y1 string
                reference R2 x -link x1
                reference R2 y -link y1
            }
            class x {
                attribute x1 string -id 1
                attribute x2 int -default 20
            }
            class y {
                attribute y1 string -id 1
                attribute y2 string -default bar
            }
            association R2 x 0..*--1 y -associator a
        }
    }
    rosea generate
    relvar constraint info ::foo::R2
} -result {correlation ::foo::R2 ::foo::a x1 1 ::foo::x x1 y1 * ::foo::y y1}
```

```
<<generate command tests>>=
test generate-3.2 {
    generate constraints for reflexive class based associative relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class A {
                attribute xid string -id 1
                attribute next_xid string -id 1
                reference R1 X -link xid -link {next_xid xid}
            }
            class X {
                attribute xid string -id 1
                attribute value string -default 20
            }
            association R1 X 0..*--0..* X -associator A -path {xid next_xid}
        }
    }
    rosea generate
    relvar constraint info ::foo::R1
} -result {correlation ::foo::R1 ::foo::A xid * ::foo::X xid next_xid * ::foo::X xid}
```

```
<<generate command tests>>=
test generate-3.3 {
    traversal for reflexive class based associative relationship
} -setup {
```

```
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class Element {
                attribute ElemId string -id 1
                attribute ElemNumber int
            }
            class Group {
                attribute PrevElem string -id 1
                attribute NextElem string -id 1
                reference R1 Element\
                    -link {PrevElem ElemId} -link {NextElem ElemId}
            }
            association R1 Element *--* Element\
                -associator Group\
                -path {PrevElem NextElem}
        }
    }
    rosea generate
    relvar eval {
        ::foo::Element create ElemId fred ElemNumber 10
        ::foo::Element create ElemId john ElemNumber 20
        set s [::foo::Element create ElemId sally ElemNumber 30]
        ::foo::Group create PrevElem fred NextElem john
        ::foo::Group create PrevElem fred NextElem sally
    }
    set r [rosea tunnel $s findRelated ~R1]
    rosea tunnel $r readAttribute ElemNumber
} -result {10}
```

```
<<generate command tests>>=
test generate-3.4 {
    link class bases association instances
} -setup {
    rosea configure {
        domain foo {
            class A {
                attribute ax1 string -id 1
                attribute ax2 string -id 1
                attribute ay1 string -id 1
                attribute ay2 string -id 1
                reference R1 X -link {ax1 x1} -link {ax2 x2}
                reference R1 Y -link {ay1 y1} -link {ay2 y2}
            }
            class X {
                attribute x1 string -id 1
                attribute x2 string -id 1
                attribute x3 int
            }
            class Y {
                attribute y1 string -id 1
                attribute y2 string -id 1
                attribute y3 int
            }
            association R1 X 1..*--1..* Y -associator A
        }
    }
    rosea generate
} -cleanup {
```

```
    cleanupConfigData
    removeDomain ::foo
} -body {
    relvar eval {
        ::foo::X create x1 x1-1 x2 x2-1 x3 27
        ::foo::X create x1 x1-2 x2 x2-2 x3 47
        ::foo::Y create y1 y1-1 y2 y2-1 y3 57
        ::foo::Y create y1 y1-2 y2 y2-2 y3 67

        set assoc [::foo::R1 link [::foo::X findAll] [::foo::Y findAll]]
    }
    rosea tunnel $assoc refMultiplicity
} -result {4}
```

```
<<generate command tests>>=
test generate-3.5 {
    unlink class based association instances.
} -setup {
    rosea configure {
        domain foo {
            class A {
                attribute ax1 string -id 1
                attribute ax2 string -id 1
                attribute ay1 string -id 1
                attribute ay2 string -id 1
                reference R1 X -link {ax1 x1} -link {ax2 x2}
                reference R1 Y -link {ay1 y1} -link {ay2 y2}
            }
            class X {
                attribute x1 string -id 1
                attribute x2 string -id 1
                attribute x3 int
            }
            class Y {
                attribute y1 string -id 1
                attribute y2 string -id 1
                attribute y3 int
            }
            association R1 X 0..*--0..* Y -associator A
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    relvar eval {
        ::foo::X create x1 x1-1 x2 x2-1 x3 27
        ::foo::X create x1 x1-2 x2 x2-2 x3 47
        ::foo::Y create y1 y1-1 y2 y2-1 y3 57
        ::foo::Y create y1 y1-2 y2 y2-2 y3 67
    }
    set assoc [::foo::R1 link [::foo::X findAll] [::foo::Y findAll]]
    ::foo::R1 unlink [::foo::X findAll]
    relation cardinality [relvar set ::foo::A]
} -result {0}
```

Generalizations

The last type of relationship we must generate data and commands for is generalization. We will following the same pattern as earlier. We formulate a query to gather all the information we need and then proceed to parcel out the information into the run time data structures and to create the needed relvar constraints and relationship commands.

```
<<rosea data>>=
pipe {
    relation semijoin $rships $Generalization\
            $Subclass -using {Domain Domain Name Relationship}\
            $ReferringClass $AttributeReference\
                -using {Domain Domain Class ReferringClass\
                Relationship Relationship Role ReferringRole} |
    relation project ~ Domain Relationship ReferringClass ReferringAttribute\
            ReferencedClass ReferencedAttribute |
    relation group ~ ReferringAttrs\
            ReferringAttribute ReferencedAttribute |
    relation extend ~ rfa\
        TagReferringAttrs {Relation {RefOrder int ReferringAttribute string\
            ReferencedAttribute string}} {
                [relation tag [tuple extract $rfa ReferringAttrs] RefOrder]
            } |
    relation extend ~ wfa\
        SrcAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferringAttribute -ascending RefOrder]
        }\
        DstAttrs list {
            [relation list [tuple extract $wfa TagReferringAttrs]\
                ReferencedAttribute -ascending RefOrder]
        } |
    relation extend ~ ram\
        ForwAttrs list {
            [Interleave [tuple extract $ram SrcAttrs]\
                [tuple extract $ram DstAttrs]]
        }\
        RevAttrs list {
            [Interleave [tuple extract $ram DstAttrs]\
                [tuple extract $ram SrcAttrs]]
        } |
    relation eliminate ~ TagReferringAttrs
} subrefsQuery

<<generate: generalizations>>=
variable subrefsQuery
set subrefs [eval $subrefsQuery]
```

Populating the run time data for generalization is a variation on what we have seen above. In addition to the Link and AssociationLink relvars, generalizations bring in the PartitionLink and PartitionDst relvars.

```
<<generate: generalizations>>=
relvar eval {
    <<generate: populate Link for generalizations>>
    <<generate: populate AssociationLink for generalizations>>
    <<generate: populate PartitionLink for generalizations>>
    <<generate: populate PartitionDst for generalizations>>
    <<generate: populate migrate data>>
}
```

For the Link relvar data, the references from the subclass to the superclass are in the forward direction. The references from the superclass to the subclass are in the reverse direction. We separate those two cases with the necessary renaming of the

relationship in the reverse direction. We follow the established pattern of projecting out the information we need from the query we performed and then juggling names to make the headings match.

```
<<generate: populate Link for generalizations>>=
set sublinks [pipe {
    relation project $subrefs Relationship ReferringClass |
    relation rename ~ Relationship Name ReferringClass SrcClass
}]
set superlinks [pipe {
    relation project $subrefs Relationship ReferencedClass |
    relation rename % Relationship Name ReferencedClass SrcClass |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]

relvar union ${domns}::__Arch_Link $sublinks $superlinks
```

The traversal from the subclass to the superclass is treated as an ordinary association type traversal, so the subclass references are used to populate the AssociationLink.

```
<<generate: populate AssociationLink for generalizations>>=
relvar union ${domns}::__Arch_AssociationLink [pipe {
    relation project $subrefs Relationship ReferringClass ReferencedClass\
            ForwAttrs |
    relation rename ~ Relationship Name ReferringClass SrcClass\
            ReferencedClass DstClass ForwAttrs Attrs |
    relation extend ~ al PrevSrcClass string {{}}
}]
```

The superclass references then wind up as PartitionLink tuples since they have to be dealt with differently.

```
<<generate: populate PartitionLink for generalizations>>=
relvar union ${domns}::__Arch_PartitionLink $superlinks
```

We must also insert tuples into PartitionDst for each subclass giving the attribute references.

```
<<generate: populate PartitionDst for generalizations>>=
relvar union ${domns}::__Arch_PartitionDst [pipe {
    relation project $subrefs Relationship ReferencedClass\
            ReferringClass RevAttrs |
    relation rename % Relationship Name ReferencedClass SrcClass\
            ReferringClass DstClass RevAttrs Attrs |
    relation update % lnk {1} {
        tuple update $lnk Name ~[tuple extract $lnk Name]}
} {} |%]
```

To support subclass migration, the SuperLink and SubLink relvars are populated.

```
<<generate: populate migrate data>>=
relvar union ${domns}::__Arch_SuperLink [pipe {
    relation project $subrefs Relationship ReferencedClass |
    relation rename ~ ReferencedClass SuperClass
}]
relvar union ${domns}::__Arch_SubLink [pipe {
    relation project $subrefs Relationship ReferringClass ReferringAttrs |
    relation rename ~ ReferringClass SubClass
}]
```

Generalization require creating `relvar partition` constraints and a relationship ensemble command. As before, we will iterate across a relation value to handle invoking the needed commands.

One difficulty we have to deal with is the arguments to `relation partition`. You supply subclass names and referential attribute lists as a variable number of arguments. To get the things into that form requires some additional processing on the `subrefs` relation.

```
<<generate: generalizations>>=
set partrefs [pipe {
    relation project $subrefs Relationship ReferencedClass DstAttrs\
            ReferringClass SrcAttrs |
    relation extend ~ prs SubAttrMap list {[list\
            ${domns}::[tuple extract $prs ReferringClass]\
            [tuple extract $prs SrcAttrs]]} |
    relation eliminate ~ ReferringClass SrcAttrs |
    relation group ~ SubClassRefs SubAttrMap |
    relation extend ~ spt SubParts list {
            [concat {*}[relation list [tuple extract $spt SubClassRefs]]]} |
    relation eliminate ~ SubClassRefs
}]

relation foreach partref $partrefs {
    relation assign $partref\
        {Relationship relationship}\
        {ReferencedClass superclass}\
        {DstAttrs superattrs}\
        {SubParts subparts}

    relvar partition ${domns}::$relationship ${domns}::$superclass\
            $superattrs {*}$subparts
    namespace ensemble create\
        -command ${domns}::$relationship\
        -map [dict create migrate [list ::rosea::RelCmds::migrate\
                $relationship]]
}
```

Tests

```
<<generate command tests>>=
test generate-4.0 {
    generate constraints for generalization relationship
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea configure {
        domain foo {
            class super {
                attribute s1 string -id 1
                attribute s2 string -id 1
                attribute a2 string -default 20
            }
            class sub1 {
                attribute sb1 string -id 1
                attribute sb2 string -id 1
                reference R1 super -link {sb1 s1} -link {sb2 s2}
            }
            class sub2 {
                attribute sb3 string -id 1
                attribute sb4 string -id 1
                reference R1 super -link {sb3 s1} -link {sb4 s2}
            }
            generalization R1 super sub1 sub2
        }
    }
```

```
    rosea generate
    relvar constraint info ::foo::R1
} -result {partition ::foo::R1 ::foo::super {s1 s2} ::foo::sub2 {sb3 sb4} ::foo::sub1 {sb1 ↩
    sb2}}
```

```
<<generate command tests>>=
test generate-4.1 {
    migrate subclass
} -setup {
    rosea configure {
        domain foo {
            class S {
                attribute s1 string -id 1
                attribute s2 string -id 1
                attribute a2 int -default 20
            }
            class X {
                attribute x1 string -id 1
                attribute x2 string -id 1
                reference R1 S -link {x1 s1} -link {x2 s2}
            }
            class Y {
                attribute y1 string -id 1
                attribute y2 string -id 1
                reference R1 S -link {y1 s1} -link {y2 s2}
            }
            generalization R1 S X Y
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    relvar eval {
        foo::S create s1 a s2 b
        foo::X create x1 a x2 b
    }
    set ref [foo::X findWhere {$x1 eq "a"}]
    foo::R1 migrate $ref Y
} -result {::foo::Y {{y1 string y2 string} {{y1 a y2 b}}}} -match ref
```

```
<<generate command tests>>=
test generate-4.2 {
    repeated specialization
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::grocer
} -body {
    rosea configure {
        domain grocer {
            class Fruit {
                attribute Name string -id 1
                attribute Color string
            }
            class SeededFruit {
                attribute Name string -id 1
                reference R1 Fruit -link Name
            }
            class UnseededFruit {
```

```
                attribute Name string -id 1
                reference R1 Fruit -link Name
            }
            generalization R1 Fruit SeededFruit UnseededFruit

            class StoneFruit {
                attribute Name string -id 1
                reference R2 SeededFruit -link Name
            }
            class PitFruit {
                attribute Name string -id 1
                attribute PitSize int -default 10
                reference R2 SeededFruit -link Name
            }
            generalization R2 SeededFruit StoneFruit PitFruit
        }
    }
    rosea generate
    relvar eval {
        set ref [::grocer::Fruit create Name apple Color red]
        ::grocer::SeededFruit create Name apple
        ::grocer::PitFruit create Name apple PitSize 30
    }
    set pit [::rosea::InstCmds::findRelated $ref\
            {~R1 SeededFruit} {~R2 PitFruit}]
    ::rosea::InstCmds::readAttribute $pit PitSize
} -result {30}
```

```
<<generate command tests>>=
test generate-4.3 {
    compound generalization
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::grocer
} -body {
    rosea configure {
        domain grocer {
            class Fruit {
                attribute Name string -id 1
                attribute Color string
            }
            class SeededFruit {
                attribute Name string -id 1
                reference R1 Fruit -link Name
            }
            class UnseededFruit {
                attribute Name string -id 1
                reference R1 Fruit -link Name
            }
            generalization R1 Fruit SeededFruit UnseededFruit

            class RedFruit {
                attribute Name string -id 1
                reference R2 Fruit -link Name
            }
            class YellowFruit {
                attribute Name string -id 1
                reference R2 Fruit -link Name
            }
            generalization R2 Fruit RedFruit YellowFruit
        }
```

```
    }
    rosea generate
    relvar eval {
        set aref [::grocer::Fruit create Name apple Color red]
        ::grocer::SeededFruit create Name apple
        ::grocer::RedFruit create Name apple

        set bref [::grocer::Fruit create Name banana Color yellow]
        ::grocer::UnseededFruit create Name banana
        ::grocer::YellowFruit create Name banana
    }
    set apple [rosea tunnel $aref findRelated {~R1 SeededFruit}]
    set banana [rosea tunnel $bref findRelated {~R2 YellowFruit}]
    testConditions\
        {[rosea tunnel $apple readAttribute Name] eq "apple"}\
        {[rosea tunnel $banana readAttribute Name] eq "banana"}
} -result {1}
```

```
<<generate command tests>>=
test generate-4.4 {
    signal a polymorphic event to a compound generalization
} -setup {
} -cleanup {
    cleanupConfigData
    removeDomain ::grocer
} -body {
    rosea configure {
        domain grocer {
            class Fruit {
                attribute Name string -id 1
                attribute Color string

                polymorphic grow
            }
            class SeededFruit {
                attribute Name string -id 1
                reference R1 Fruit -link Name

                statemodel {
                    state seed {} {
                        puts "In Seed"
                    }
                    transition seed - grow -> sprouted

                    state sprouted {} {
                        puts "In SeededFruit"
                        lappend ::rosea::test::states SeededFruit
                    }
                }
            }
            class UnseededFruit {
                attribute Name string -id 1
                reference R1 Fruit -link Name

                statemodel {
                    state seed {} {
                        puts "In Seed"
                    }
                    transition seed - grow -> sprouted

                    state sprouted {} {
                        puts "In UnseededFruit"
```

```
                        lappend ::rosea::test::states UnseededFruit
                    }
                }
            }
            generalization R1 Fruit SeededFruit UnseededFruit

            class RedFruit {
                attribute Name string -id 1
                reference R2 Fruit -link Name

                statemodel {
                    state seed {} {
                        puts "In Seed"
                    }
                    transition seed - grow -> sprouted

                    state sprouted {} {
                        puts "In RedFruit"
                        lappend ::rosea::test::states RedFruit
                    }
                }
            }
            class YellowFruit {
                attribute Name string -id 1
                reference R2 Fruit -link Name

                statemodel {
                    state seed {} {
                        puts "In Seed"
                    }
                    transition seed - grow -> sprouted

                    state sprouted {} {
                        puts "In YellowFruit"
                        lappend ::rosea::test::states YellowFruit
                    }
                }
            }
            generalization R2 Fruit RedFruit YellowFruit
        }
    }
    rosea generate
    relvar eval {
        set aref [::grocer::Fruit create Name apple Color red]
        ::grocer::SeededFruit create Name apple
        ::grocer::RedFruit create Name apple
    }

    set ::rosea::test::states [list]
    rosea tunnel $aref signal grow
    vwait ::rosea::test::states

    testConditions\
        {"SeededFruit" in $::rosea::test::states}\
        {"RedFruit" in $::rosea::test::states}
} -result {1}
```

Create Assigner Data and Commands

We did some of the generation work for assigners when we considered association relationships. There we included in the relationship ensemble command any additional commands that perform assigner operations for those association that had defined assigners.

Here, we complete the work on assigners. We must create the `__Activity` namespace and the state action procedures. Also we must create the instance relvars for the assigners.

```
<<generate: create assigner data and commands>>=
set assigners [relation semijoin $domain $AssignerStateModel\
        -using {Name Domain}]
relation foreach assigner $assigners {
    set asgnns\
        ${domns}::[relation extract $assigner Relationship]::__Activity
    namespace eval $asgnns {
        <<tclral imports>>
    }
    namespace eval $asgnns [list\
        namespace path [list\
            ::rosea::InstCmds\
            $domns\
        ]\
    ]
    set states [pipe {
        relation semijoin $assigner $State\
            -using {Domain Domain Relationship Model} |
        relation project ~ Name Parameters Action
    }]
    relation foreach state $states {
        relation assign $state\
            {Name stateName}\
            {Parameters params}\
            {Action stateAction}
        set params [linsert $params 0 self]
        proc ${asgnns}::$stateName $params $stateAction
    }
}
```

single assigners have an instance relvar. We just make up an identifier for it.

```
<<generate: create assigner data and commands>>=
relation foreach sassigner\
        [relation semijoin $domain $SingleAssigner\
        -using {Name Domain}] {
    relation assign $sassigner {Relationship relationship}
    relvar create ${domns}::__${relationship}__STATEINST {
        Id      int
        __State string
    } Id
    CreateInInitialState $domns $relationship {Id 0} ; # ❶
}
```

Multiple assigners have an identifier that matches that of some class so we have to search down the proper identifying attributes.

```
<<generate: create assigner data and commands>>=
relation foreach massigner\
        [relation semijoin $domain $MultipleAssigner\
        -using {Name Domain}] {
    relation assign $massigner {Relationship relationship}
    set assignvar ${domns}::__${relationship}__STATEINST
```

```
    set idattrs [pipe {
        relation semijoin $massigner $Identifier\
            $IdentifyingAttribute\
            $Attribute -using {Domain Domain Class Class\
                Attribute Name} |
        relation dict ~ Name Type
    }]
    relvar create $assignvar\
        [concat $idattrs [list __State string]]\
        [dict keys $idattrs]
}
```

❶ For single assigners, we create the single instance as part of generation. For multiple assigners, we don't know how many there are so they must be created when the domain is populated or at run time.

Tests

```
<<generate command tests>>=
test generate-6.0 {
    signal an assigner
} -setup {
    rosea configure {
        domain foo {
            class Customer {
                attribute Name string -id 1
            }
            class Clerk {
                attribute Name string -id 1
                attribute Customer string
                reference R1 Customer -link {Customer Name}
            }
            association R1 Clerk 0..1--0..1 Customer
            assigner R1 {
                state s1 {a b} {
                    puts $a $b
                }
                transition s1 - e1 -> s2

                state s2 {} {
                    puts "in s2"
                        set ::done 1
                }
                transition s2 - e1 -> s1
            }
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    ::foo::R1 signal e1
    set timer [after 1000 set ::done TIMEOUT]
    vwait ::done
    after cancel $timer
    set ::done
} -result {1}
```

```
<<generate command tests>>=
test generate-6.1 {
    signal a multi assigner
} -setup {
    rosea configure {
        domain foo {
            class Customer {
                attribute Name string -id 1
            }
            class Clerk {
                attribute Name string -id 1
                attribute Customer string
                attribute Department string
                reference R1 Customer -link {Customer Name}
                reference R2 Department -link {Department Name}
            }
            class Department {
                attribute Name string -id 1
            }
            association R1 Clerk 0..1--0..1 Customer
            association R2 Clerk 1..*--1 Department
            assigner R1 {
                identifyby Department
                state s1 {a b} {
                    puts $a $b
                }
                transition s1 - e1 -> s2

                state s2 {} {
                    puts "in s2"
                        set ::done 1
                }
                transition s2 - e1 -> s1
            }
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    ::foo::R1 create Name Shoes
    ::foo::R1 signal {Name Shoes} e1
    set timer [after 1000 set ::done TIMEOUT]
    vwait ::done
    after cancel $timer
    set ::done
} -result {1}
```

Chapter 7

Initial Instance Population

The third important step in translating an XUML domain is to establish an initial population for the instances in a domain. If you consider the evolution of the domain over time, the initial data population sets the values of attributes at time zero. This is analogous to specifying the initial state of a state machine. Then at each transaction in the processing, the domain data moves from one valid set of values to another as implied by the class attributes and relationships and enforced by the underlying relvar constraints.

A significant benefit of using TclRAL relvars as the basis of the data architecture is the instance population of the domain can be established purely in data by specifying attribute values alone. Attributes values and relationship linkage is established by setting values in a declarative way. It is possible to execute a series of `create` and `link` commands to establish the initial instance population in much the same way as class instances are dealt with at run-time. However, this implies writing potentially long sequences of code. The `populate` command will allow you to specify attribute values and the implied relationship linkages in data organized as tables that correspond exactly to the class diagram. It is not unusual to maintain many different populations for a domain. Often testing will use a different initial population than the delivered system. Population scripts can be stored in files and the system build mechanism can choose the desired initial instance population.

Following our pattern for evaluating DSL scripts, we establish a namespace where the population commands will reside so that the population script may invoke commands without requiring additional qualification.

```
<<population commands namespace>>=
namespace eval Populate {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace import ::rosea::Config::ConfigEvaluate
    namespace import ::rosea::Helpers::DeclError
    <<populate commands>>
    <<domain population namespace layout>>
}
```

`rosea populate` *script*

`script`
A Tcl script that is executed in an environment that will resolve *population configuration* commands that may be used to define the initial instance population of a domain.

Implementation

```
<<rosea exports>>=
namespace export populate
```

```
<<rosea commands>>=
proc populate {script} {
    namespace upvar Config errcount errcount configlineno configlineno
    set errcount 0
    set configlineno 1

    ConfigEvaluate ::rosea::Populate $script ; # ❶

    if {$errcount > 0} {
        tailcall DeclError CONFIG_ERRORS $errcount
    }
    return $errcount
}
```

❶ It is important that population happen as a transaction.

Like the `configure` command, we supply variations to take the data from a channel or a file.

`rosea populateFromChan` *channel*

`channel`
A handle to a Tcl channel, *e.g.* as returned from `::open`.

The `populateFromChan` command executes the `populate` command on the script that is obtained by reading the contents of *channel*.

Implementation

```
<<rosea exports>>=
namespace export populateFromChan

<<rosea commands>>=
proc populateFromChan {chan} {
    tailcall populate [::chan read -nonewline $chan]
}
```

`rosea populateFromFile` *filename*

`filename`
The name of a file containing a population script.

The `populateFromFile` command executes the `populate` command on the script that is contained in the file named, *filename*.

Implementation

```
<<rosea exports>>=
namespace export populateFromFile

<<rosea commands>>=
proc populateFromFile {filename} {
```

```
    set f [::open $filename r]
    try {
        populateFromChan $f
    } finally {
        ::chan close $f
    }
}
```

Populating Domains

Just as we did when defining the DSL to configure a domain, we define a namespace in which the commands of populating a domain will reside.

```
<<domain population namespace layout>>=
namespace eval DomainPop {
    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    namespace path ::rosea::Helpers
    <<domain populate commands>>
}
```

Domain

The `domain` command specifies the set of commands used to populate a particular domain. The domain name must have been defined in the configuration phase of constructing the domain.

`domain` *name body*

name
The name of the domain. A domain name must be a non-empty string.

body
A Tcl script containing invocation of the domain population commands to specify the initial instance population of the domain.

Implementation

```
<<populate commands>>=
proc domain {name body} {
    set domain [relvar restrictone ::rosea::Config::Domain Name $name]
    if {[relation isempty $domain]} {
        tailcall DeclError UNKNOWN_DOMAIN $domain
    }

    namespace upvar DomainPop DomainName DomainName DomainLoc DomainLoc
    relation assign $domain {Name DomainName} {Location DomainLoc}

    try {
        relvar eval {
            ConfigEvaluate [namespace current]::DomainPop $body
        }
    } on error {result opts} {
```

```
        log::error $result
        upvar #0 ::rosea::Config::errcount errcount
        incr errcount
    }
}
```

Class

The `class` command is used to define the class context for supplying attribute values for the population. The intent here is to allow the values to be specified in a tabular manner.

`class` *name heading values ...*

name
The name of the class for whom instances are to be created.

heading
A list of attributes of the class. Attributes that have been given default values need not be present. Otherwise, it is necessary to supply a value for all attributes when an instance is created.

values ...
A single value or a variable number of values to be used for populating the attributes of the class. If there is only a single *value* argument, it is treated as a list. This treatment is convenient when constant values are assigned to the attributes. For a single *value* argument or if many *values* are given, the number of values must be a multiple of the number of attributes in the *heading*. Values are assigned to the corresponding attributes. The number of instances create is the total number of values supplied divided by the number of attributes in the *heading*. If the value given an attribute is the literal string "-", then the attribute is not populated and will receive its default value. To give an attribute a value of "-" it must be escaped as "\-".

Implementation

```
<<domain populate commands>>=
proc class {class heading args} {
    # Determine if we have one big list or a bunch of values.
    set popvalues [expr {[llength $args] == 1 ? [lindex $args 0] : $args}]
    set headlen [llength $heading]
    if {[llength $popvalues] % $headlen != 0} {
        tailcall DeclError ARG_MISMATCH $headlen [llength $popvalues]
    }

    # Iterate over the attribute values taking them in chunks that are the same
    # size as the heading.
    set body [list]
    for {set vindex 0} {$vindex < [llength $popvalues]} {incr vindex $headlen} {
        set values [lrange $popvalues $vindex [expr {$vindex + $headlen - 1}]]
        # Create a tuple as a list of attribute name / value pairs.
        set tuple [list]
        foreach attr $heading value $values {
            # Check if we are skipping a value to use a default.  We must also
            # allow some way to specify the value as "-".
            if {$value eq "-"} {
                continue
            } elseif {$value eq "\\-"} {
                set value -
            }
            lappend tuple $attr $value
```

```
        }
        # Accumulate the tuples as a list to form the body that will be
        # inserted into the relvar.
        lappend body $tuple
    }

    # Insert the body into the relvar.
    namespace upvar [namespace current]\
        DomainName DomainName\
        DomainLoc DomainLoc
    set domns ${DomainLoc}::${DomainName}
    set relvar ${domns}::$class
    set insts [relvar insert $relvar {*}$body]

    # Now we have to deal with any state model that might be present.  We have
    # to set up the initial state properly if such a thing exists for this
    # class.
    set idattrs [lindex [relvar identifiers $relvar] 0]
    relation foreach inst $insts {
        CreateInInitialState $domns $class [pipe {
            relation project $inst {*}$idattrs |
            relation tuple ~ |
            tuple get ~
        }]
    }

    return
}
```

```
<<error code formats>>=
ARG_MISMATCH      {number of population values must be a multiple of %d, got %d}
```

Tests

```
<<populate command tests>>=
test populate-1.0 {
    Populate a domain
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    rosea populate {
```

```
        domain meta {
            class C1\
                {Id     A1} {
                foo     24
                bar     42
            }
            class C2\
                {Id     C1Id} {
                f1      24
                f2      24
            }
        }
    }
    relation cardinality $::meta::C1
} -result {2}
```

```
<<populate command tests>>=
test populate-1.1 {
    Populate a domain -- alternate syntax
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    rosea populate {
        domain meta {
            class C1\
                {Id     A1}\
                foo     24\
                bar     42
            class C2\
                {Id     C1Id}\
                f1      24\
                f2      24
        }
    }
    relation cardinality $::meta::C2
} -result {2}
```

```
<<populate command tests>>=
test populate-1.2 {
    Populate a domain using default value syntax
} -setup {
    rosea configure {
        domain meta {
```

```
            class C1 {
                attribute Id string -id 1
                attribute A1 int -default 20
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::meta
} -body {
    rosea populate {
        domain meta {
            class C1\
                {Id     A1} {
                foo     24
                bar     -
            }
            class C2\
                {Id     C1Id} {
                f1      24
                f2      24
            }
        }
    }
    set ref [::meta::C1 findWhere {$A1 == 20}]
    ::rosea::InstCmds::readAttribute $ref Id
} -result {bar}
```

```
<<populate command tests>>=
test populate-1.3 {
    Populate a domain with initial states
} -setup {
    rosea configure {
        domain meta {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}

                statemodel {
                    state s1 {} {
                        puts "in s1"
                    }
                    transition s1 - e1 -> s1
                }
            }
            association R1 C2 0..*--0..1 C1
        }
    }
```

```
    rosea generate * [namespace current]
} -cleanup {
    cleanupConfigData
    removeDomain [namespace current]::meta
} -body {
    rosea populate {
        domain meta {
            class C1\
                {Id     A1} {
                foo     24
                bar     42
            }
            class C2\
                {Id     C1Id} {
                f1      24
                f2      24
            }
        }
    }
    relation cardinality [relvar set [namespace current]::meta::__C2__STATEINST]
} -result {2}
```

Assigner

In the context of populating a model, the `assigner` command is used to create instances for those assigners that are multiple assigners. For singular assigners, the assigner instance is created during the generation of the domain. For multiple assigners, it is necessary to create the various assigner instances and the `assigner` command can be used to create an initial instance population for a multiple assigner.

`assigner` *association heading values ...*

`association`
The name of the association for which instances are to be created.

`heading`
A list of attribute names. These names must match the identifying attributes of the assigner, *i.e.* they must match those given by the `identifyby` command given when the assigner was specified.

`values ...`
A single value or a variable number of values to be used for populating the attributes of the class. If there is only a single *value* argument, it is treated as a list. This treatment is convenient when constant values are assigned to the attributes. For a single *value* argument or if many *values* are given, the number of values must be a multiple of the number of attributes in the *heading*. Values are assigned to the corresponding attributes. The number of instances create is the total number of values supplied divided by the number of attributes in the *heading*.

Implementation

```
<<domain populate commands>>=
proc assigner {rname heading args} {
    # Determine if we have one big list or a bunch of values.
    set popvalues [expr {[llength $args] == 1 ? [lindex $args 0] : $args}]
    set headlen [llength $heading]
    if {[llength $popvalues] % $headlen != 0} {
        tailcall DeclError ARG_MISMATCH $headlen [llength $popvalues]
    }
```

```
    namespace upvar [namespace current]\
        DomainName DomainName\
        DomainLoc DomainLoc
    set domns ${DomainLoc}::${DomainName}
    set relvar ${domns}::__${rname}__STATEINST
    set idattrs [lindex [relvar identifiers $relvar] 0]
    if {![struct::set equal $heading $idattrs]} {
        tailcall DeclError NO_IDENTIFIER $heading $idattrs
    }

    # Iterate over the attribute values taking them in chunks that are the same
    # size as the heading.
    for {set vindex 0} {$vindex < [llength $popvalues]} {incr vindex $headlen} {
        set values [lrange $popvalues $vindex [expr {$vindex + $headlen - 1}]]
        # Create a tuple as a list of attribute name / value pairs.  We assume
        # the values are in the same order as the heading -- that is the point.
        set tuple [list]
        foreach attr $heading value $values {
            lappend tuple $attr $value
        }
        CreateInInitialState $domns $rname $tuple
    }

    return
}
```

Tests

```
<<populate command tests>>=
test populate-2.0 {
    populate a multi assigner
} -setup {
    rosea configure {
        domain foo {
            class Customer {
                attribute Name string -id 1
            }
            class Clerk {
                attribute Name string -id 1
                attribute Customer string
                attribute Department string
                reference R1 Customer -link {Customer Name}
                reference R2 Department -link {Department Name}
            }
            class Department {
                attribute Name string -id 1
            }
            association R1 Clerk 0..1--0..1 Customer
            association R2 Clerk 1..*--1 Department
            assigner R1 {
                identifyby Department
                state s1 {a b} {
                    puts $a $b
                }
                transition s1 - e1 -> s2

                state s2 {} {
                    puts "in s2"
                        set ::done 1
                }
```

```
                transition s2 - e1 -> s1
            }
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
} -body {
    rosea populate {
        domain foo {
            assigner R1 Name Shoes
            assigner R1 Name Clothes
        }
    }
    ::foo::R1 signal {Name Shoes} e1
    set timer [after 1000 set ::done TIMEOUT]
    vwait ::done
    after cancel $timer
    set ::done
} -result {1}
```

Chapter 8

Serializing a Domain

Since all the data of a domain is held in relation variables, it is possible to save the entire state of a domain by using TclRAL to serialize the relvars of the domain. This will save not only the object instance data itself, but all the state information and the architectural relvars. Deserializing the domain then restores it to its exact configuration. _

`rosea save` *?*`-tclral | -sqlite` *? ?*`-async` *cmdprefix? domain filename*

`-tclral`
If this option is present or if no serialization type is specified, the serialization is accomplished using the native method of TclRAL.

`-sqlite`
If `-sqlite` is present, then serialization is done to a SQLite data base.

`-async` ***cmdprefix***
The `-async` option allows a *cmdprefix* to be specified that will be invoked when the serialization is completed. Because serialization may not happen in the middle of a transaction, after executing the `save` command, the serialization action may not be completed. If supplied, *cmdprefix* is invoked with the *domain* and *filename* argument values.

domain
The fully qualified path name to the domain to be saved.

filename
The name of the file to which the domain data is serialized.

The `save` command arranges for all the data for the given *domain* to be saved in the file given by, *filename*. Options allow for the format of the saved file to be either native TclRAL serialization (which is ASCII and very Tcl oriented) or the domain may be saved in a SQLite database. In both cases, serializing a domain uses the ability of TclRAL to serialize a set of relation variables.

Implementation

The implementation strategy for the `save` command first parses the arguments for the various options. Then a script is built that executes the serialization and, if specified, the completion command. The composed script is then executed as an idle callback. This will insure that we are at a quiet time in the execution of a domain to make sure the serialization does not take place in the middle of a transaction.

```
<<rosea exports>>=
namespace export save

<<rosea commands>>=
```

```
proc save {args} {
    set savecmd ::ral::serializeToFile
    set asynccmd {}

    set options $args
    while {1} {
        if {[string index [lindex $options 0] 0] eq "-"} {
            set options [lassign $options option]
            switch -exact -- $option {
                -sqlite {
                    set savecmd ::ral::storeToSQLite
                }
                -tclral {
                    set savecmd ::ral::serialize
                }
                -async {
                    set options [lassign $options asynccmd]
                }
                default {
                    tailcall DeclError UNKNOWN_OPTION save $option
                }
            }
        } else {
            break
        }
    }

    if {[llength $options] == 2} {
        lassign $options domain file
    } else {
        tailcall DeclError SAVE_ARG_ERROR $args
    }

    set script "try \{$savecmd $file ${domain}*\}" ; # ❶
    if {$asynccmd ne {}} {
        append script " finally \{$asynccmd $domain $file\}"
    }

    after idle [list ::apply [list {} $script]] ; # ❷
}
```

❶ Note that we append an asterisk to the domain name to make all the relvars of the domain match for the benefit of the `serialize` and `storeToSQLite` commands.

❷ Using a lamda procedure is a very clean way to deal with scripts built on the fly.

```
<<error code formats>>=
SAVE_ARG_ERROR  {wrong number of arguments: expected:\
        "save ?-sqlite | -tclral? ?-async <cmdprefix>?\
        <domain> <filename>", got: "%s"}
```

Tests

```
<<save command tests>>=
test save-1.0 {
    Serialize a domain
} -setup {
    rosea configure {
```

```
        domain serial {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
        }
    }
    rosea generate
    rosea populate {
        domain serial {
            class C1\
                {Id     A1} {
                foo     24
                bar     42
            }
            class C2\
                {Id     C1Id} {
                f1      bar
                f2      foo
            }
        }
    }
} -cleanup {
    cleanupConfigData
    removeDomain ::serial
    removeDomain ::sertest
    file delete serial.ral
} -body {
    set asynccmd [lambda@ [namespace current] {domain file} {
        syncToTest
    }]
    rosea save -async $asynccmd ::serial serial.ral
    waitForSync

    ral deserializeFromFile serial.ral ::sertest
    ral relation cardinality [ral relvar set ::sertest::C1]
} -result {2}
```

```
<<save command tests>>=
test save-2.0 {
    Serialize a domain using SQLite
} -setup {
    rosea configure {
        domain serial {
            class C1 {
                attribute Id string -id 1
                attribute A1 int
            }

            class C2 {
                attribute Id string -id 1
                attribute C1Id string
                reference R1 C1 -link {C1Id Id}
            }
            association R1 C2 0..*--0..1 C1
```

```
        }
    }
    rosea generate
    rosea populate {
        domain serial {
            class C1\
                {Id     A1} {
                foo     24
                bar     42
            }
            class C2\
                {Id     C1Id} {
                f1      foo
                f2      {}
            }
        }
    }
} -cleanup {
    cleanupConfigData
    removeDomain ::serial
    removeDomain ::sertest
    file delete serial.sqlite
} -body {
    set asynccmd [lambda@ [namespace current] {domain file} {
        syncToTest
    }]
    rosea save -sqlite -async $asynccmd ::serial serial.sqlite
    waitForSync

    ral loadFromSQLite serial.sqlite ::sertest
    ral relation cardinality [ral relvar set ::sertest::C1]
} -result {2}
```

Chapter 9

Bridging to Instance Operations

It is necessary to provide a means to break the encapsulation of a domain, albeit in a very controlled fashion. Consider, for example, the problem of interfacing Tk events into a `rosea` domain. It is frequently the case that a Tk event needs to be signaled as a state machine event to a particular object. To accomplish this, there must be a small piece of *bridge code* that is used to map the Tk event semantics onto the object semantics. The bridge code may be simple enought to be the script bound to the Tk event (using the `bind` command or a `-command` option on a widget). You could provide a domain operation to perform this task, but such code adds significant clutter to the domain's public interface and should that domain be reused in another context, is unlikely to satisfy the needs for reuse. A much more robust technique is to have a general operation that can be used in the bridge code that will need to be supplied when a particular domain is built into an overall software application.

In Tk, events are associated with windows. In Rosea, state machine events are signaled to class instances. So the bridge code will have to map windows to instances in some manner. This can be done in a variety of ways, but one choice is to map the window name to an identifier of an instance and use the `findById` operation on a class to obtain a reference to the instance. We then want to execute the `signal` operation on that instance and have that event appear to be signaled from outside of the domain, *i.e.* we want to transform a Tk event into state machine event that starts a new thread of control. The `tunnel` command provides this capability.

`rosea tunnel` *instref operation ?arg1 arg2 ... ?*

`instref`
An instance reference of the instance to which the *operation* will be directed.

`operation`
The name of an operation to be applied to the *instref*.

`arg1 arg2 ...`
The arguments of the *operation*.

The `tunnel` command invokes an instance based operation on an instance from outside of the domain's implemenation. The operation will be run as a transaction on the domain and will start a new thread of control.

Implementation

Note that the implementation of `tunnel` is almost the same as the implementation of `instop`. The difference is that the operation is performed inside of a transaction. This allows operations to manipulate the data model in limited ways. However, by providing a point of invocation that is outside of the domain, we will insure that any generated signals do not appear as if they came from inside the domain.

In particular, users should **not** violate encapsulation by executing from within a state activity, a Tk widget command like:

```
.runbutton configure -command [namespace code [list instop $self signal Run]]
```

This has the effect on injecting code into the namespace of a state activity and should not be used as it can, under certain corner cases, confuse the determination of the source of an event.

The better way directly to bridge a Tk widget event to a state machine event is:

```
.runbutton configure -command [::rosea tunnel $self signal Run]
```

The `tunnel` command can be viewed as a convenience procedure to break domain encapsulation in a controlled manner without having to code a clumsy invocation of `::rosea::InstCmds::instop` directly.

```
<<rosea exports>>=
namespace export tunnel

<<rosea commands>>=
proc tunnel {instref op args} {
    relvar eval {
        set result [[lindex $instref 0]::Instance $instref $op {*}$args]
    }
    return $result
}
```

Chapter 10

State Machine Trace

In this section we discuss the capabilities and design of tracing state machine event dispatch. We are immediately 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 `ral` package relvar tracing or with `rosea` package state machine tracing. Here we are discussing the ability of the `rosea` 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 read the code base sequentially and have a good sense of what will happen during execution. Indeed, it is much easier to consult the graphical representation to know what will happen during run time. 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. Signaling, *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.

We define a namespace in which to place the trace specific code and data.

```
<<trace commands namespace>>=
namespace eval Trace {
    logger::initNamespace [namespace current]

    <<tclral imports>>
```

```
    namespace import ::ral::relvar
    namespace path [list\
        [namespace parent]::Helpers\
        [namespace parent]::InstCmds\
    ]
    <<trace data>>
    <<trace commands>>
}
```

Trace Data

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

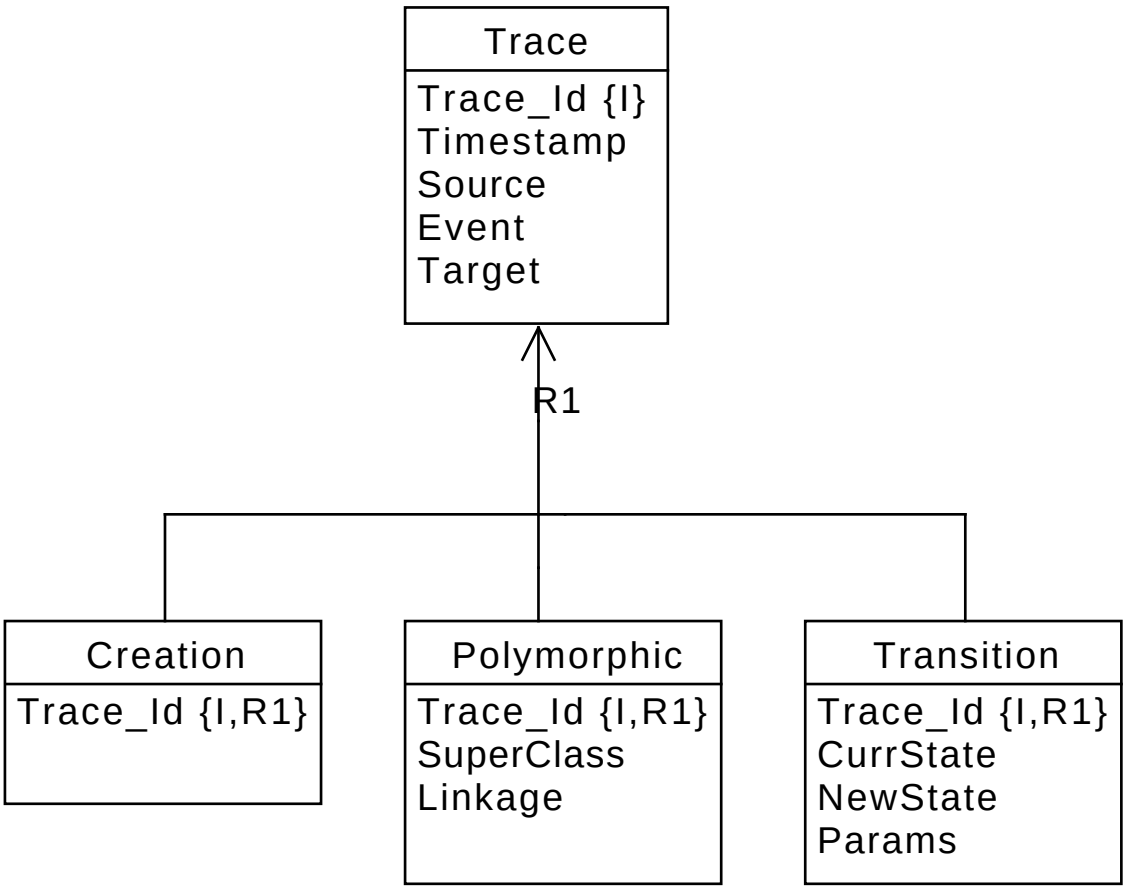

Figure 10.1: Trace Data Class Model

A trace is identified by an arbitrary 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 instance reference to which the event is directed.

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 eventually have two trace entries. **Polymorphic** events map, at run time, 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 parameters associated with the event and the outcome of the transition.

Following our established pattern, the most direct implementation of this data schema is to use the `TclRAL` package.

```
<<trace data>>=
relvar create Trace {
    Trace_Id    int
    Timestamp   bignum
    Source      list
    Event       string
    Target      list
} 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
```

We will support two queries on trace data, selecting based on the class of the target of the event dispatch and selecting based on the instances that are targets. We put together some procedures to perform those queries.

```
<<trace commands>>=
proc QueryClassTraces {classes} {
    variable Trace
    return [relation restrictwith $Trace {
        [lindex $Target 0] in $classes ||\
            [namespace tail [lindex $Target 0]] in $classes
    }] ; # ❶
}
```

❶ Note that the classes can be either fully qualified relvar names or just the tail. If multiple domains are involved in the application, using fully qualified names insures no unintended name conflict between domains.

To query specific instances as targets, we need to be able to compare the instances in the trace to an arbitrary list of instance references. An easy way to accomplish that is to construct an anonymous function which is evaluated during a `relation restrictwith` operation. We are forced to do something like this since the `relation restrictwith` command evaluates expressions and *not* scripts. Since we need a little logic to iterate over all the instances references, an anonymous function that can be invoked from the expression will work. To make coding the anonymous function easier, we will use the `lamda` package from `tcllib`.

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

We store the lambda in a variable for easy reference later.

```
<<trace data>>=
variable targetCmpFunc [lambda {instrefs target} {
    lassign $target trelvar tinst
    foreach instref $instrefs {
        lassign $instref crelvar cinst
        if {$trelvar eq $crelvar && [ral relation is $tinst subsetof $cinst]} {
            return true
        }
    }
    return false
}]
```

A few points to note here.

- We iterate over all the instance references in the argument list.
- We split out the instance reference components for the target only once.
- It is important to compare relvar names first so we can short circuit the relation comparison. If the instances refer to different relvars then the headings of the instance relation values won't match.
- We actually determine if the target is simply contained in an instance reference, *i.e.* we allow the instance reference arguments to refer to multiple instances.

```
<<trace commands>>=
proc QueryTargetTraces {targets} {
    variable Trace
    variable targetCmpFunc
    return [relation restrictwith $Trace {[{*}$targetCmpFunc $targets $Target]}]
}
```

We 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.

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

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

```
<<trace data>>=
variable traceLogCmd [::logger::init rosea]
```

Trace Procedure

Control and access to the state machine event trace data is provided by a single command called, `trace`. This command uses a series of subcommands and arguments to access the trace data.

`rosea trace` *subcmd ?arg1 arg2 ... ?*

`subcmd`
The trace control command to be invoked. Trace control commands should be one of:

- `control`
- `decode`
- `format`
- `diagram`

`arg1 arg2 ...`
The arguments to the trace control operation.

The `trace` command controls various aspects of state machine event dispatch tracing.

`rosea trace control` *?arg1 arg2 ... ?*

The `control` subcommand executes one of the following trace control operations:

- `on`
  Turn on trace collection.
- `off`
  Turn off trace collection.
- `status`
  Returns the current status of tracing.
- `clear`
  Discards all trace entries.
- `logon`
  Log traces as they occur via the `logger` package.
- `logoff`
  Turn trace logging off.
- `loglevel` *level*
  Set the logging level for traces to *level*.
- `save` *filename*
  Save all traces to a SQLite database.

`rosea trace decode` *?arg1 arg2 ... ?*

The `decode` subcommand queries the set of collected state machine traces according to the value of its argument and returns a list of dictionaries that contain the selected trace information. Valid trace selection arguments are:

- `all`
  Decode all trace entries.
- `class` *class1 class2 ...*
  Decode only those trace entries for which the class of the receiving instance matches one of the *classN* arguments.
- `target` *instref1 instref2 ...*
  Decode only those trace entries for which the receiving instance matches one of the `instrefN` arguments.

`rosea trace format` *traces*

The `format` subcommand returns a human readable formatting of the traces given as an argument. The `traces` argument is a list of dictionaries as returned from the `rosea trace decode` command.

`rosea trace diagram` *which filename ?arg1 arg2 ... ?*

The `diagram` subcommand queries a set of collected state machine traces according to the value of its `which` argument and writes to `filename` a script suitable to be used with the seqdiag program to produce a UML sequence diagram of the traces. Valid arguments depend upon the value of `which` and are listed below:

- `all filename`
  Diagram all trace entries.
- `class filename classlist` *?option value ... ?*
  Diagram only those trace entries for which the class of the receiving instance matches one of the class names in the `classlist` list. Optional *option value* pairs are passed into the generated `seqdiag` script.
- `target filename targetrefs` *?option value ... ?*
  Diagram only those trace entries for which the receiving instance matches one of the instance references in the `targe trefs` list. Optional *option value* pairs are passed into the generated `seqdiag` script.

```
<<rosea exports>>=
namespace export trace

<<rosea commands>>=
proc trace {subcmd args} {
    switch -exact -- $subcmd {
        control {
            tailcall traceControl {*}$args
        }
        decode {
            switch -exact -- [lindex $args 0] {
                all {
                    tailcall Trace::DecodeAllTraces
                }
                class {
                    tailcall Trace::DecodeClassTraces {*}[lrange $args 1 end]
                }
                target {
                    tailcall Trace::DecodeTargetTraces {*}[lrange $args 1 end]
                }
                default {
                    tailcall DeclError UNKNOWN_TRACE_CMD [lindex $args 0]
                }
            }
        }
        format {
            tailcall formatTraces {*}$args
        }
        diagram {
            switch -exact -- [lindex $args 0] {
                all {
                    tailcall Trace::DiagAllTraces {*}[lrange $args 1 end]
                }
                class {
                    tailcall Trace::DiagClassTraces {*}[lrange $args 1 end]
                }
                target {
                    tailcall Trace::DiagTargetTraces {*}[lrange $args 1 end]
                }
```

```
                default {
                    tailcall DeclError UNKNOWN_TRACE_CMD [lindex $args 0]
                }
            }
        }
        default {
            tailcall DeclError UNKNOWN_TRACE_CMD $subcmd
        }
    }
}
```

```
<<error code formats>>=
UNKNOWN_TRACE_CMD   {unknown trace subcomand, "%s"}
```

Trace Control

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

```
<<rosea commands>>=
proc traceControl {op args} {
    switch -exact -- $op {
        on {
            namespace upvar Trace traceState traceState
            set traceState on
        }
        off {
            namespace upvar Trace traceState traceState
            set traceState off
        }
        status {
            namespace upvar Trace traceState traceState
            return $traceState
        }
        clear {
            relvar eval {
                foreach rvar {Trace Creation Polymorphic Transition} {
                    relvar set Trace::$rvar\
                            [relation emptyof [relvar set Trace::$rvar]]
                }
            }
            namespace upvar Trace traceNumber traceNumber
            set traceNumber 0
        }
        logon {
            namespace upvar Trace traceLogState traceLogState
            set traceLogState on
        }
        logoff {
            namespace upvar Trace traceLogState traceLogState
            set traceLogState off
        }
        loglevel {
            namespace upvar Trace traceLogLevel traceLogLevel
            if {[llength $args] != 0} {
                set traceLogLevel [lindex $args 0]
            }
            return $traceLogLevel
        }
        save {
            if {[llength $args] == 0} {
```

```
                DeclError NO_SAVEFILE
            }
            ral storeToSQLite [lindex $args 0] ::rosea::Trace::*
        }
        default {
            DeclError BAD_TRACEOP $op
        }
    }
}
```

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

Trace Population

Each of the event dispatch procedures 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.

```
<<trace commands>>=
proc 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
    }
}
```

```
<<trace commands>>=
proc 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
    }
}
```

```
<<trace commands>>=
proc 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.

```
<<trace commands>>=
proc NewTrace {src event target} {
    variable traceNumber
    return [relvar insert Trace [list\
        Trace_Id    [incr traceNumber]\
        Timestamp   [clock microseconds]\
        Source      $src\
        Event       $event\
        Target      $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.

```
<<trace commands>>=
proc 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.

```
<<trace command tests>>=
test LogTrace-1.0 {
    log trace data
} -setup {
    rosea configure {
        domain foo {
            class c1 {
                attribute Id int -id 1
                attribute Count int -default 0
                attribute Max int -default 2
                statemodel {
                    state s1 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                        #puts [relformat [deRef $self] self]
                        lassign [readAttribute $self Count Max] count max
                        if {$count >= $max} {
                            ::rosea::test::syncToTest $count
                        }
```

```
                    }
                    transition s1 - e1 -> s2

                    state s2 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                        signal $self e2
                    }
                    transition s2 - e2 -> s1
                }
            }
        }
    }
    rosea generate
    turnOnTraceLog
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    turnOffTraceLog
} -body {
    set ref [::foo::c1 create Id 1]
    ::rosea::InstCmds::signal $ref e1
    waitForSync
} -result {2}
```

```
<<trace command tests>>=
test LogTrace-2.0 {
    log trace data for a polymorphic event
} -setup {
    rosea configure {
        domain foo {
            class lt2 {
                attribute Id int -id 1

                polymorphic *e1 *e2
            }
            generalization R1 lt2 lt2-sub1 lt2-sub2

            class lt2-sub1 {
                attribute Id int -id 1
                reference R1 lt2 -link Id

                statemodel {
                    state sub1-s1 {} {
                        ::rosea::test::syncToTest true
                    }
                    transition sub1-s1 - *e1 -> sub1-s2

                    state sub1-s2 {} {
                        delaysignal 50 $self *e2
                    }
                    transition sub1-s2 - *e2 -> sub1-s1
                }
            }

            class lt2-sub2 {
                attribute Id int -id 1
                reference R1 lt2 -link Id

                statemodel {
                    state sub1-s1 {} {
                        ::rosea::test::syncToTest true
```

```
                    }
                    transition sub1-s1 - *e1 -> sub1-s2

                    state sub1-s2 {} {
                        delaysignal 50 $self *e2
                    }
                    transition sub1-s2 - *e2 -> sub1-s1
                }
            }
        }
    }

    rosea generate

    rosea populate {
        domain foo {
            class lt2 Id 1
            class lt2-sub1 Id 1
        }
    }

    turnOnTraceLog
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    turnOffTraceLog
} -body {
    set superref [::foo::lt2 findWhere {$Id == 1}]
    ::rosea::InstCmds::signal $superref *e1
    waitForSync
} -result {true}
```

```
<<trace command tests>>=
test LogTrace-3.0 {
    log trace data for a creation event
} -setup {
    rosea configure {
        domain foo {
            class lt3 {
                attribute Id int -id 1
                statemodel {
                    transition @ - e1 -> s1

                    state s1 {value} {
                        ::rosea::test::syncToTest $value
                    }
                }
            }
        }
    }

    rosea generate

    turnOnTraceLog
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    turnOffTraceLog
} -body {
    ::foo::lt3 createasync e1 10 Id 1
    waitForSync
```

```
} -result {10}
```

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

    ::rosea trace control clear
    ::rosea trace control on
    set logcmd [::logger::servicecmd rosea]
    set prevloglevel [${logcmd}::currentloglevel]
    ${logcmd}::setlevel [::rosea trace control loglevel]
    ::rosea trace control logon
}

proc turnOffTraceLog {} {
    variable prevloglevel

    set logcmd [::logger::servicecmd rosea]
    ${logcmd}::setlevel $prevloglevel
    ::rosea trace control logoff
    ::rosea trace control off
    ::rosea trace control 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 primitive 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 query, transforming relational data into a dictionary provides a convenient interface for handling sets of trace data. Here we discuss 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 instance reference of the instance that signaled the event. If the source of the event is outside of an object, then this will be the nil reference.

event
The name of the event.

target
The instance reference of the target of the event.

class
The fully qualified relvar name for 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 class name of the superclass instance mapping the event.

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

For polymorphic type events, the `target` value is the class 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.

```
<<trace commands>>=
proc TracesToRecords {traces} {
    foreach rvname {Creation Polymorphic Transition} { # ❶
        set traces [::ralutil::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   [lindex $Target 0]\
        ] ; # 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.

```
<<trace commands>>=
proc DecodeAllTraces {} {
    variable Trace
    tailcall TracesToRecords $Trace
}
```

Decode All Traces Tests

```
<<trace command tests>>=
test DecodeAllTraces-1.0 {
    display trace data
} -setup {
    rosea configure {
        domain foo {
            class c1 {
                attribute Id int -id 1
                attribute Count int -default 0
                attribute Max int -default 2
                statemodel {
                    state s1 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                        lassign [readAttribute $self Count Max] count max
                        if {$count >= $max} {
                            ::rosea::test::syncToTest $count
                        }
                    }
                    transition s1 - e1 -> s2

                    state s2 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                    }
                    transition s2 - e2 -> s1
                }
            }
        }
    }
    rosea generate
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    ::rosea trace control off
    ::rosea trace control clear
} -body {
    ::rosea trace control on
    set ref [::foo::c1 create Id 1]
    ::rosea::InstCmds::signal $ref e1
    ::rosea::InstCmds::signal $ref e2
    waitForSync
    set traces [::rosea::Trace::DecodeAllTraces]
    dict get [lindex $traces 0] event
} -result {e1}
```

Decode Class Traces

```
<<trace commands>>=
proc DecodeClassTraces {args} {
    return [TracesToRecords [QueryClassTraces $args]]
}
```

Decode Class Traces Tests

```
<<trace command tests>>=
test DecodeClassTraces-1.0 {
```

```
    display trace data for specific class
} -setup {
    rosea configure {
        domain foo {
            class dct1 {
                attribute Id string -id 1
                attribute Power int -default 20
                reference R1 dct2 -link Id
                statemodel {
                    state Off {} {
                        signal [findRelated $self R1] Lower
                    }
                    transition Off - TurnOn -> On

                    state On {power} {
                        updateAttribute $self Power $power
                        signal [findRelated $self R1] Raise
                    }
                    transition On - TurnOff -> Off
                }
            }

            association R1 dct1 1--1 dct2

            class dct2 {
                attribute Id string -id 1
                statemodel {
                    state Down {} {
                        ::rosea::test::syncToTest\
                            [readAttribute [findRelated $self ~R1] Power]
                    }
                    transition Down - Raise -> Up

                    state Up {} {
                        delaysignal 50 [findRelated $self ~R1] TurnOff
                    }
                    transition Up - Lower -> Down
                }
            }
        }
    }

    rosea generate

    rosea populate {
        domain foo {
            class dct1 Id first
            class dct2 Id first
        }
    }

    rosea trace control clear
    rosea trace control on
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    rosea trace control off
    rosea trace control clear
} -body {
    ::rosea::InstCmds::signal [::foo::dct1 findWhere {$Id eq "first"}] TurnOn 30
    waitForSync
    # puts [::rosea trace format [::rosea trace decode all]]
```

```
    set traces [::rosea::Trace::DecodeClassTraces ::foo::dct2]
    expr {[llength $traces] == 2 &&\
            [dict get [lindex $traces 1] event] eq "Lower"}
} -result {1}
```

Decode Target Traces

```
<<trace commands>>=
proc DecodeTargetTraces {args} {
    tailcall TracesToRecords [QueryTargetTraces $args]
}
```

```
<<trace command tests>>=
test DecodeTargetTraces-1.0 {
    display trace data for specific instances
} -setup {
    rosea configure {
        domain foo {
            class c1 {
                attribute Id string -id 1
                statemodel {
                    state S1 {} {
                    }
                    transition S1 - E1 -> S1
                    transition S1 - E2 -> S2

                    state S2 {} {
                        ::rosea::test::syncToTest S2
                    }
                }
            }
        }
    }

    rosea generate

    rosea populate {
        domain foo {
            class c1 Id first
            class c1 Id second
            class c1 Id third
        }
    }

    rosea trace control clear
    rosea trace control on
} -cleanup {
    cleanupConfigData
    removeDomain ::foo
    rosea trace control off
    rosea trace control clear
} -body {
    set i1 [foo::c1 findById Id first]
    set i2 [foo::c1 findById Id second]
    set i3 [foo::c1 findById Id third]
    rosea tunnel $i1 signal E1
    rosea tunnel $i2 signal E1
    rosea tunnel $i3 signal E1
    rosea tunnel $i1 signal E1
    rosea tunnel $i1 signal E1
```

```
    rosea tunnel $i1 signal E2
    rosea tunnel $i2 signal E2
    rosea tunnel $i3 signal E2

    waitForSync
    set traces [::rosea::Trace::DecodeTargetTraces $i1 $i2]
    #puts [rosea::formatTraces $traces]
    llength $traces
} -result {6}
```

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.

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

Format Traces Tests

```
<<trace command tests>>=
test formatTraces-1.0 {
    display human readable trace data
} -setup {
    rosea configure {
        domain foo {
            class c1 {
                attribute Id int -id 1
                attribute Count int -default 0
                attribute Max int -default 2
                statemodel {
                    state s1 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                        lassign [readAttribute $self Count Max] count max
                        if {$count >= $max} {
                            ::rosea::test::syncToTest $count
                        }
                    }
                    transition s1 - e1 -> s2
```

```
                    state s2 {} {
                        updateAttribute $self Count\
                            [expr {[readAttribute $self Count] + 1}]
                    }
                    transition s2 - e2 -> s1
                }
            }
        }
    }
    rosea generate
} -cleanup {
    rosea trace control off
    rosea trace control clear
} -body {
    rosea trace control on
    set one [::foo::c1 create Id 1]
    ::rosea::InstCmds::signal $one e1
    ::rosea::InstCmds::signal $one e2
    waitForSync
    set traces [rosea trace format [rosea trace decode all]]
    # puts $traces
    llength [split $traces \n]
} -result {2}
```

Format Trace Record

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

```
<<trace commands>>=
proc FormatTraceRec {rec} {
    dict with rec {
        switch -exact -- $type {
            transition {
                append result\
                    "Transition: "\
                    "[FormatInstRef $source] - $event"\
                    [expr {[llength $params] != 0 ?\
                        "\([join $params {, }]\)" : {}}]\
                    " -> [FormatInstRef $target] ==> "\
                    "$current -> $new"
            }
            polymorphic {
                append result\
                    "Polymorphic : "\
                    "[FormatInstRef $source] - $event -> $super ==> "\
                    "$link -> [FormatInstRef $target]"
            }
            creation {
                append result\
                    "Creation: "\
                    "[FormatInstRef $source] - $event -> $class ==>\
                        [FormatInstRef $target]"
            }
            default {
                DeclError BAD_TRACETYPE $type
            }
        }
    }
    return $result
```

```
}
```

```
<<trace commands>>=
proc FormatInstRef {instref} {
    lassign $instref relvar inst
    if {$relvar eq {}} {
        return {{}}
    } else {
        return "$relvar\{[tuple get [relation tuple $inst]]\}"
    }
}
```

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

Format Time Stamp

```
<<trace commands>>=
proc 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

```
<<trace commands>>=
proc 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

```
<<trace commands>>=
proc DiagTraces {traces args} {
    set result {}
    append result "seqdiag \{\n"
    append result "    activation = none;\n"
    foreach {option value} $args {
        append result "    $option = $value;\n"
    }

    foreach rvname {Creation Polymorphic Transition} {
        set traces [::ralutil::rvajoin $traces [relvar set $rvname] $rvname]
    }

    relation foreach trace $traces -ascending Trace_Id {
        relation assign $trace
        if {[isEmptyRef $Source]} {
            set Source EXTERNAL
        } else {
            set Source [namespace tail [lindex $Source 0]]
        }
        set Target [namespace tail [lindex $Target 0]]
        if {[relation isnotempty $Transition]} {
            relation assign $Transition Params
            set evtlabel $Event[expr {[llength $Params] != 0 ?\
                "\([join $Params {, }]\)" : {}}]
        } elseif {[relation isnotempty $Polymorphic]} {
            relation assign $Transition Linkage
            set evtlabel "$Event <<Polymorphic $Linkage>>"
        } elseif {[relation isnotempty $Creation]} {
            set evtlabel "$Event <<Creation>>"
        }
        append result "    $Source ->> $Target \[label=\"$evtlabel\"\];\n"
    }

    append result "\}\n"

    return $result
}
```

Diagram All Traces

```
<<trace commands>>=
proc DiagAllTraces {filename args} {
    set chan [open $filename w]
    variable Trace
    try {
        chan puts $chan [DiagTraces $Trace {*}$args]
    } finally {
        chan close $chan
    }
    return
}
```

Diagram Class Traces

```
<<trace commands>>=
proc DiagClassTraces {filename classes args} {
```

```
    set chan [open $filename w]
    try {
        chan puts $chan [DiagTraces [QueryClassTraces $classes] {*}$args]
    } finally {
        chan close $chan
    }
    return
}
```

Diagram Target Traces

```
<<trace commands>>=
proc DiagTargetTraces {filename targets args} {
    set chan [open $filename w]
    try {
        chan puts $chan [DiagTraces [QueryTargetTraces $targets] {*}$args]
    } finally {
        chan close $chan
    }
    return
}
```

Chapter 11

An Example

In this section, we discuss an example that illustrates the usage of `rosea`. 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 familiar with model-driven translation oriented development may have many questions that remain.

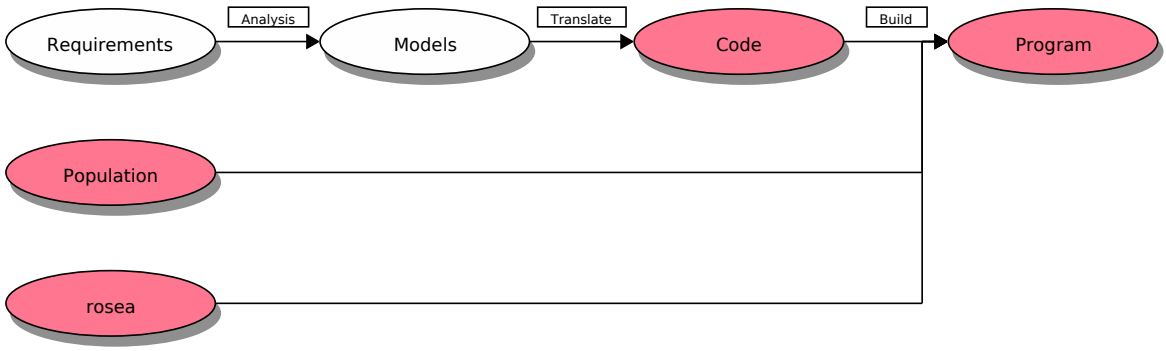

Figure 11.1: Simplified Translation Workflow

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 `rosea` package, we will produce a running example program.

The subject matter of our example is an automatic clothes washer. This is a very simple washing machine, especially compared to modern commercially available washers. The intent is to select a subject that most people would be familiar with from ordinary experience so that we don't have to devote too much time explaining the problem. One word of caution. This model is an example for pedagogical purposes and probably has little correspondence with the way *real* washing machines operate or are designed to operate. You 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 `rosea`.

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

Domain Data

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

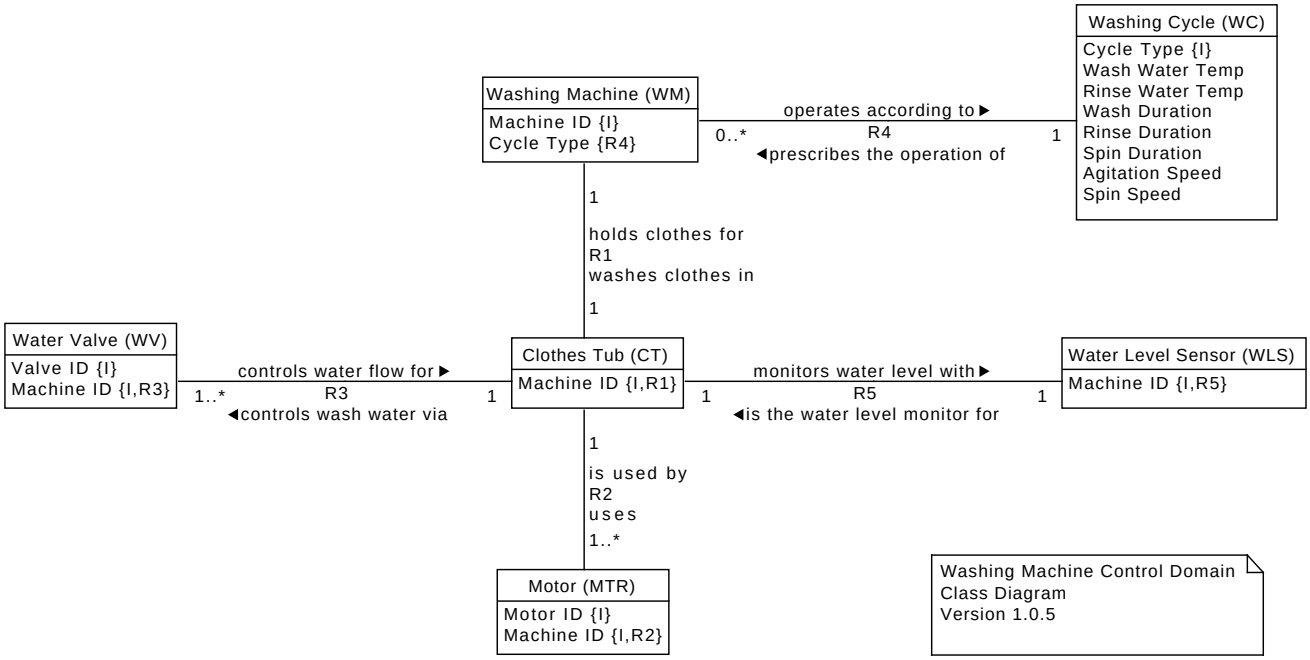

Figure 11.2: Washing Machine Class Diagram

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 class diagram facet of the model is static in the sense that at any point in time, the logical predicates you can infer from the diagram will be true. The classes hold the parameters of the domain and the relationships state how the components are associated with respect to each other. 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 `rosea` package constructs.

Washing Machine Class

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

```
<<WM class>>=
class WashingMachine {
```

```
    <<WM attributes>>
    <<WM references>>
    statemodel {
        <<WM statemodel>>
    }
}
```

Looking at the Washing Machine class on the class diagram, we see that there are two attributes. The MachineID attribute is an identifier and we make its data type a simple string. The CycleType attribute is also a simple string.

```
<<WM attributes>>=
attribute MachineID string -id 1
attribute CycleType string
```

The Washing Machine class makes a reference to realize the R4 relationship. The class diagram tells us that the CycleType attribute is involved in that reference (*i.e.* it has the `R4` annotation). It must refer to the identifier of the other participant in R4 namely, the Washing Cycle class. So we can define the reference from Washing Machine to Washing Cycle that is used to realize R4.

```
<<WM references>>=
reference R4 WashingCycle -link CycleType
```

As it happens, the referring attribute in the Washing Machine class, CycleType, has the same name as the identifying attribute in the Washing Cycle class. This means that the `-link` option need only specify the one name, *i.e.* `-link CycleType` is shorthand for `-link {CycleType CycleType}`

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

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

Washing Cycle Class

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

```
<<WC class>>=
class WashingCycle {
    <<WC attributes>>
}
```

We need only specify the attributes and their data types, marking the CycleType as the identifier (as was shown in the graphic). In order to determine the proper data types of the attributes we would need to consult the descriptive text associated with the class diagram. As we have no such text in this example, we will infer a set of data types based on how the attribute values will be used in the processing.

```
<<WC attributes>>=
attribute CycleType string -id 1
attribute WashWaterTemp string -check {$WashWaterTemp in {Hot Warm Cold}}
attribute RinseWaterTemp string -check {$RinseWaterTemp in {Hot Warm Cold}}
attribute WashDuration int
attribute RinseDuration int
attribute SpinDuration int
attribute AgitationSpeed string -check {$AgitationSpeed in {Low Medium High}}
attribute SpinSpeed string -check {$SpinSpeed in {Low Medium High}}
```

Clothes Tub Class

By now the pattern of examining the model graphics to determine the class characteristics is established. The Clothes Tub class is an active class with a state model.

```
<<CT class>>=
class ClothesTub {
    <<CT attributes>>
    <<CT references>>
    statemodel {
        <<CT statemodel>>
    }
}
```

A Clothes Tub is identified by the MachineID attribute:

```
<<CT attributes>>=
attribute MachineID string -id 1
```

and the MachineID also serves as the reference required to realize R1. It is not uncommon for attributes to serve both roles.

```
<<CT references>>=
reference R1 WashingMachine -link MachineID
```

The `R1` association itself can be transcribed from the graphic as follows.

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

Water Valve Class

The Water Valve class represents the means to move water into and out of a Clothes Tub.

```
<<WV class>>=
class WaterValve {
    <<WV attributes>>
    <<WV references>>
    <<WV operations>>
}
```

It takes two attributes to identify a Water Valve. The ValveId attribute is a string from among the set {Hot Cold Drain} and here we see a referential attribute being used as an identifying attribute again.

```
<<WV attributes>>=
attribute ValveID string -id 1 -check {$ValveID in {Hot Cold Drain}}
attribute MachineID string -id 1
```

```
<<WV references>>=
reference R3 ClothesTub -link MachineID
```

The R3 relationship represents the design of the washing machine hardware that includes three valves on each machine. Notice the we don't concern ourselves here with whether a washing machine has three or three thousand valves on it. That is dealt with when we populate the class instances. The important point here is that there is one or more valves associated with each tub.

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

Motor Class

The Motor class is similar to the Water Valve class.

```
<<MTR class>>=
class Motor {
    <<MTR attributes>>
    <<MTR references>>
    <<MTR operations>>
}
```

The class diagram graphic shows the two identifying attributes.

```
<<MTR attributes>>=
attribute MotorID string -id 1 -check {$MotorID in {Pump Agitator Spin}}
attribute MachineID string -id 1
```

Like the Water Valve class, the MachineID attributes is also referential.

```
<<MTR references>>=
reference R2 ClothesTub -link MachineID
```

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

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

Water Level Sensor

The Water Level Sensor class is also simple.

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

Because there is only one Water Level Sensor per washing machine, we do not need another identifying attribute.

```
<<WLS attributes>>=
attribute MachineID string -id 1
```

```
<<WLS references>>=
reference R5 ClothesTub -link MachineID
```

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

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

Domain Dynamics

The second facet of the model that is considered during translation is the dynamics. The model encodes the sequences of domain execution as state models attached to classes. In this model, there are two classes for which state models are defined, Washing Machine and Clothes Tub. The control is partitioned by the model between these classes. The Washing Machine class is given the responsibility for coordinating the actions needed to clean clothes according to the particulars specified by the Washing Cycle.

In this example that coordination primarily involves timing the various components of the washing cycle. The Clothes Tub class deals with sequencing the mechanics of the washer to perform a specific activity.

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

Washing Machine State Model

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

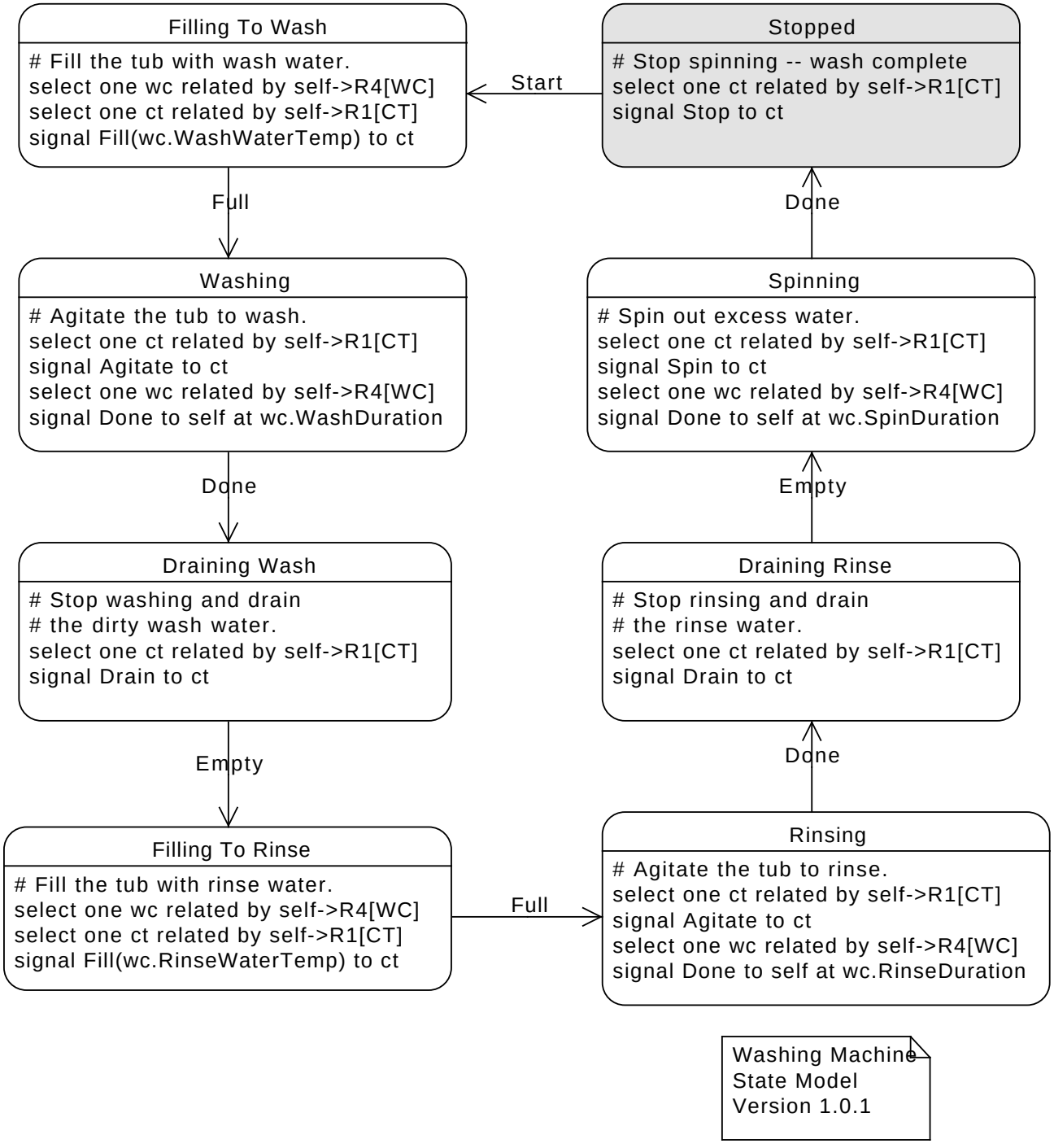

Figure 11.3: Washing Machine State Model Diagram

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

The transition table corresponding to the diagram is shown next.

Table 11.1: Washing Machine Transition Table

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

The transition table may be directly translated into the required `rosea` configuration statements. Below, we have not shown the Tcl code that executes for each state activity. We will return to the processing below. For now, we use a set of literate program chunks to represent the state activity code in order to get a clearer view of how the transition table is represented as `statemodel` configuration statements.

```
<<WM 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
```

Clothes Tub State Model

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

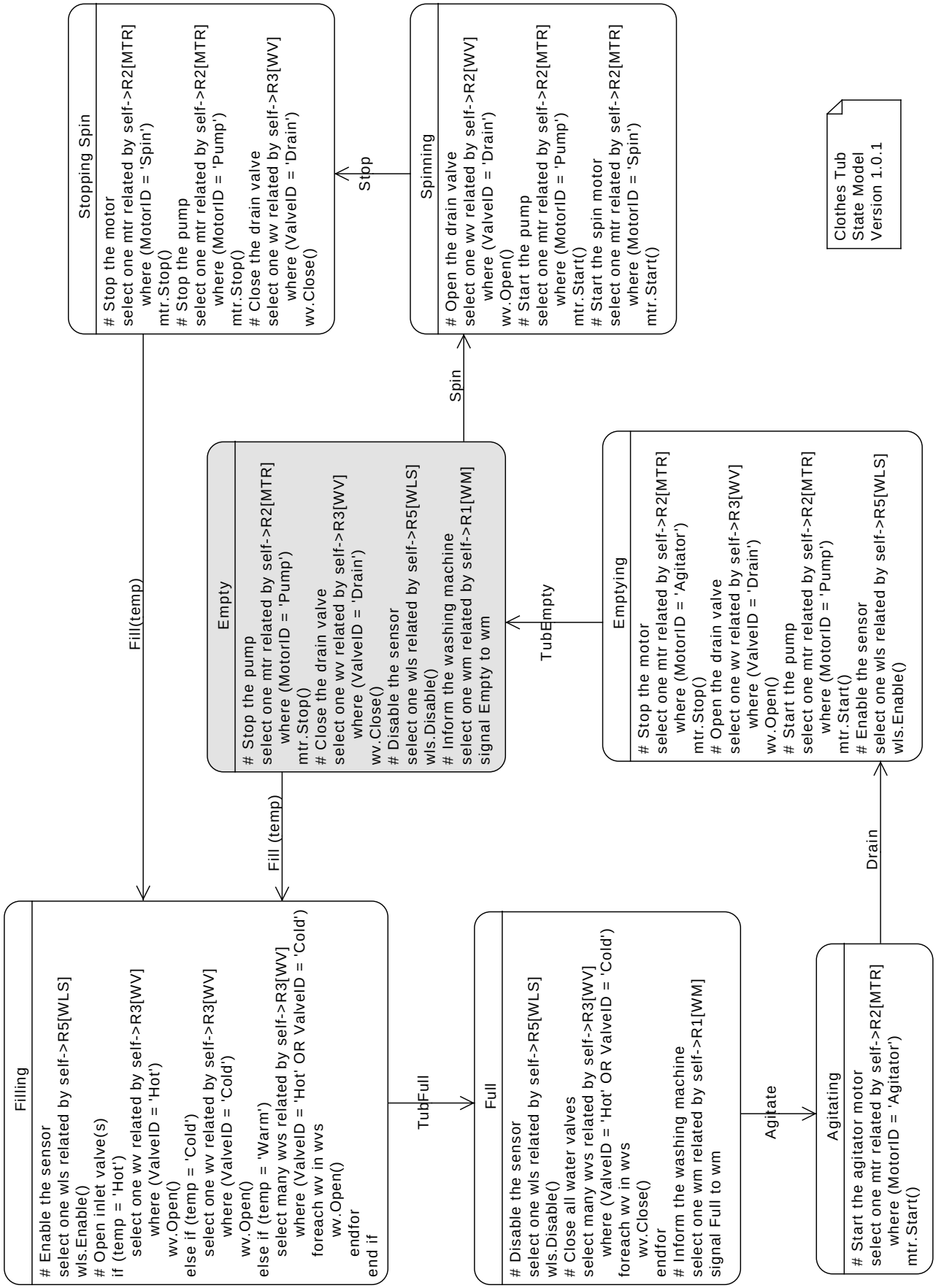

Figure 11.4: Clothes Tub State Model Diagram

The transition table follows immediately from the diagram.

Table 11.2: Clothes Tub Transition Table

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

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

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

```
<<CT 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

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
```

Domain Processing

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

For state activities, the diagrams above show the action language that is to be executed when the state is entered. In the translations below, we duplicate the action language from the state and write the translation to Tcl code immediately following. In most cases, the translation is one statement to another.

The model level actions performed by the state activities, such as access to attribute values, navigating the class diagram or signaling events is accomplished by invoking the procedures that we discussed as part of the `rosea` package.

Washing Machine State Activities

Below is the processing for the Washing Machine state activities. For the first few states, we will make several comments on the correspondence between the action language statements and the Tcl code. After the translation pattern is established, we present the remaining state with minimal explanation. In the interest of clarity, no attempt has been made to shorten or optimize the Tcl code. There will be cases where some variable assignments will seem superfluous and command nesting might seem a more natural way to express the logic. Here we are trying to emphasize the relationships between the action language statements and the Tcl code. In delivered code, we might be more succinct.

Stopped Activity

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

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

Stopped Implementation

```
<<WM stopped activity>>=
set ct [findRelated $self ~R1]
signal $ct Stop
```

The corresponding Tcl code invokes the `findRelated` instance command on the `self` instance. Recall that all state activities have an implicit `self` argument defined for them. With the Clothes Tub instance in hand, it is then signaled to Stop using the signal instance command that we have already seen. There are a couple things to note here.

1. The use of the direction syntax, *i.e.* `~R1`, obviates the need to say that we are navigating to an instance of the `CT` class in the manner that the action language statement does. Because of the syntax convention and the manner in which the association was defined, the system knows the destination class.

2. The question arises whether the instance reference returned from `findRelated` in the above traversal across `~R1` can ever be the empty reference or ever reference multiple instances. The answer is no! The `R1` association, as shown in the class diagram, is singular and unconditional on the Clothes Tub side. Every traversal from Washing Machine to Clothes Tub along `~R1` is guaranteed to return exactly one instance. Any operation in the domain that would perturb that state of affairs is rejected as violating the referential integrity of the class diagram. Consequently, no test of the multiplicity of the returned reference is necessary as it would be strictly redundant. In general, traversing unconditional relationships never

requires a test to determine if we obtained an instance reference and traversing a conditional relationship should always be followed by a test of the returned reference to see if it is empty. The fact that a relationship is conditional implies that there is to be conditional processing associated with traversing the relationship.

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 [findRelated $self R4]
set ct [findRelated $self ~R1]
signal $ct Fill [readAttribute $wc 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
```

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

Washing Implementation

```
<<WM washing activity>>=
set ct [findRelated $self ~R1]
signal $ct Agitate
set wc [findRelated $self R4]
delaysignal [expr {[readAttribute $wc WashDuration] * 1000}] $self 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.

The delayed signal is accomplished by using the `delaysignal` procedure. Notice that the event is self directed, albeit delayed by the washing cycle time. So this state sends two signals, one to the Clothes Tub to tell it to start agitating the clothes and a delayed signal to itself so it when know when the clothes have been washing long enough.

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 [findRelated $self ~R1]
signal $ct 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 [findRelated $self R4]
set ct [findRelated $self ~R1]
signal $ct Fill [readAttribute $wc 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 [findRelated $self ~R1]
signal $ct Agitate
set wc [findRelated $self R4]
delaysignal [expr {[readAttribute $wc RinseDuration] * 1000}] $self 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 [findRelated $self ~R1]
signal $ct 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 [findRelated $self ~R1]
signal $ct Spin
set wc [findRelated $self R4]
delaysignal [expr {[readAttribute $wc SpinDuration] * 1000}] $self Done
```

Clothes Tub State Activities

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

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 [findRelatedWhere $self ~R2 {$MotorID eq "Pump"}]
instop $mtr Stop

set wv [findRelatedWhere $self ~R3 {$ValveID eq "Drain"}]
instop $wv Close

set wls [findRelated $self ~R5]
instop $wls Disable

set wm [findRelated $self R1]
signal $wm 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 [findRelated $self ~R5]
instop $wls Enable
if {$temp eq "Hot"} {
    set wv [findRelatedWhere $self ~R3 {$ValveID eq "Hot"}]
    instop $wv Open
} elseif {$temp eq "Cold"} {
    set wv [findRelatedWhere $self ~R3 {$ValveID eq "Cold"}]
    instop $wv Open
} elseif {$temp eq "Warm"} {
    set wvs [findRelatedWhere $self ~R3 {$ValveID eq "Hot" ||\
            $ValveID eq "Cold"}]
    forAllRefs wv $wvs {
        instop $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 [findRelated $self ~R5]
instop $wls Disable
set wvs [findRelatedWhere $self ~R3 {$ValveID eq "Hot" || $ValveID eq "Cold"}]
forAllRefs wv $wvs {
    instop $wv Close
}
```

```
set wm [findRelated $self R1]
signal $wm 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 [findRelatedWhere $self ~R2 {$MotorID eq "Agitator"}]
instop $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 [findRelatedWhere $self ~R2 {$MotorID eq "Agitator"}]
instop $mtr Stop

set wv [findRelatedWhere $self ~R3 {$ValveID eq "Drain"}]
instop $wv Open

set mtr [findRelatedWhere $self ~R2 {$MotorID eq "Pump"}]
instop $mtr Start

set wls [findRelated $self ~R5]
instop $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 [findRelatedWhere $self ~R3 {$ValveID eq "Drain"}]
instop $wv Open

set mtr [findRelatedWhere $self ~R2 {$MotorID eq "Pump"}]
instop $mtr Start

set mtr [findRelatedWhere $self ~R2 {$MotorID eq "Spin"}]
instop $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 [findRelatedWhere $self ~R2 {$MotorID eq "Spin"}]
instop $mtr Stop

set mtr [findRelatedWhere $self ~R2 {$MotorID eq "Pump"}]
instop $mtr Stop

set wv [findRelatedWhere $self ~R3 {$ValveID eq "Drain"}]
instop $wv Close
```

Class Instance Operations

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

Water Valve Operations

```
<<WV operations>>=
instop Open {} {
    tailcall VALVE::open [readAttribute $self ValveID]
}

instop Close {} {
    tailcall VALVE::close [readAttribute $self ValveID]
}
```

Motor Class Operations

```
<<MTR operations>>=
instop Start {} {
    tailcall MOTOR::start [readAttribute $self MotorID]
}

instop Stop {} {
    tailcall MOTOR::stop [readAttribute $self MotorID]
}
```

Water Level Sensor Operations

```
<<WLS operations>>=
instop Enable {} {
    tailcall SENSOR::enable [readAttribute $self MachineID]
}

instop Disable {} {
    tailcall SENSOR::disable [readAttribute $self MachineID]
}
```

Domain Operations

In this section we show the code for the 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 even though we do not intend to supply a user interface, *per se*.

Create Washer

Since there several classes involved in creating a functioning washer, a domain operation can make that simpler.

```
<<wmctrl operations>>=
operation createWasher {washer} {
    set wm [WashingMachine create MachineID $washer CycleType Normal]
    ClothesTub create MachineID $washer
    WaterValve create ValveID Hot MachineID $washer
    WaterValve create ValveID Cold MachineID $washer
    WaterValve create ValveID Drain MachineID $washer
    Motor create MotorID Pump MachineID $washer
    Motor create MotorID Agitator MachineID $washer
    Motor create MotorID Spin MachineID $washer
    WaterLevelSensor create MachineID $washer
```

```
    return $wm
}
```

Delete Washer

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

```
<<wmctrl operations>>=
operation deleteWasher {washer} {
    set wm [WashingMachine findById MachineID $washer]
    if {[isNotEmptyRef $wm]} {
        set ct [findRelated $wm ~R1]
        delete\
            [findRelated $ct ~R3]\
            [findRelated $ct ~R2]\
            [findRelated $ct ~R5]
        delete $ct $wm
    }
    return
}
```

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.

```
<<wmctrl operations>>=
operation startWasher {washer} {
    set wm [WashingMachine findWhere {$MachineID eq $washer}]
    if {[isEmptyRef $wm]} {
        error "unknown washer, \"$washer\""
    }

    signal $wm 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 means 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.

```
<<wmctrl operations>>=
operation selectCycle {washer cycle} {
    set wm [WashingMachine findWhere {$MachineID eq $washer}]
    if {[isEmptyRef $wm]} {
        error "unknown washer, \"$washer\""
    }

    set wc [WashingCycle findWhere {$CycleType eq $cycle}]
```

```
    if {[isEmptyRef $wc]} {
        error "unknown cycle, \"$cycle\""
    }

    R4 link $wm $wc

    return
}
```

Initial Instance Population

In this section we populate the data of the domain model.

```
<<initial instance population>>=
class WashingCycle\
    {CycleType WashWaterTemp RinseWaterTemp WashDuration RinseDuration\
             SpinDuration AgitationSpeed SpinSpeed} {

    Normal Cold Cold 20 10 10 Medium Medium
    Whites Hot Cold 20 10 20 High High
    PermPress Warm Cold 15 10 15 Medium Medium
    Delicate Cold Cold 15 10 10 Low Low
}
```

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

```
<<initial instance population>>=
class WashingMachine {MachineID CycleType} {wm1 Normal}
```

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

```
<<initial instance population>>=
class ClothesTub MachineID wm1
```

```
<<initial instance population>>=
class Motor\
    {MotorID    MachineID} {
    Pump        wm1
    Agitator    wm1
    Spin        wm1
}
```

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

```
<<initial instance population>>=
class WaterValve\
    {ValveID    MachineID} {
    Hot         wm1
    Cold        wm1
    Drain       wm1
}
```

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

```
<<initial instance population>>=
class WaterLevelSensor MachineID wm1
```

Stubbing the External Operations

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

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::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::WaterValve::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 tub as being full or empty. So in some sense we must *simulate* the action of the sensor. To accomplish simulating the sensor, we will record data in a relvar identified by the washing machine id. This will allow us to simulate an arbitrary number of washing machines. 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::WaterLevelSensor::SENSOR {
    namespace import ::ral::*
    namespace import ::ralutil::*

    ::logger::initNamespace [namespace current] debug

    relvar create SensorState {
        MachineID   string
        State       string
        TimerID     string
    } MachineID

    proc enable {machine} {
        log::debug "enable sensor on machine, \"$machine\""

        set ss [relvar restrictone SensorState MachineID $machine]
        if {[relation isempty $ss]} {
            set ss [relvar insert SensorState [list\
                MachineID $machine State TubEmpty TimerID None]]
        }
        set sensorState [relation extract $ss State]

        set newState\
            [expr {$sensorState eq "TubEmpty" ? "TubFull" : "TubEmpty"}]
```

```
        set tid [after 3000 [namespace code [list trigger $machine $newState]]]
        relvar updateone SensorState sstate [list MachineID $machine] {
            tuple update $sstate State $newState TimerID $tid
        }
    }

    proc disable {machine} {
        log::debug "disable sensor on machine, \"$machine\""
        set ss [relvar restrictone SensorState MachineID $machine]
        if {[relation isnotempty $ss]} {
            after cancel [relation extract $ss TimerID]
        }
    }
```

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 Clothes Tub that matches the sensor and signal an event. We also update our notion of the current status of the sensor.

```
<<external operation stubs>>=
    proc trigger {machine value} {
        set ct [::wmctrl::ClothesTub findWhere {
            $MachineID eq $machine
        }]
        rosea tunnel $ct signal $value
        relvar updateone SensorState sstate [list MachineID $machine] {
            tuple update $sstate State $value
        }
    }
}
```

Running the Example

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 set a global variable named `::done` and invoke `::vwait` on that variable.

To actually set the `::done` variable, we will install a command trace on the Stopping Spin state activity of the Clothes Tub 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. Since state activities are turned into ordinary Tcl procs, we can use a command execution trace to assign a value to `::done` and thereby resume execution after our call to `::vwait`. Note that we are adding this synchronization solely to be able to run one washing cycle in our example and gain control after that cycle has completed. Once we come out of the event loop, we will want to do some other processing to show the execution results of the example. An actual application would most likely run forever or invoke `::exit` based on some other condition or circumstance.

```
<<running the example>>=
proc syncToStop {cmd code result op} {
    set ::done 1
}

trace add execution ::wmctrl::ClothesTub::__Activity::StoppingSpin\
        leave syncToStop
```

Finally, yes truly finally, we are in a position to drive the domain operations 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>>=
rosea trace control on
```

```
wmctrl selectCycle wm1 PermPress
puts "**** Start Run"
wmctrl startWasher wm1

vwait ::done ; # ❶

puts "**** Finish Run"
rosea trace control off

puts "**** Trace Begin"
puts [rosea trace format [rosea trace decode all]]
puts "**** Trace End"

rosea trace diagram all images/wmctrl-seqdiag.diag span_height 7 ; # ❷

rosea trace control save wmctrl-trace.sqlite ; # ❸
```

❶ Enter the Tcl event loop. State machine events will then be dispatched and the washer will operate.

❷ Draw a sequence diagram of the example run. This will allow us to compare the textual log with a graphic containing the same information.

❸ We save the traces to demonstrate that the trace data can be made available for future analysis.

Example Run Results

After running the example we obtain the following output.

Output From Running the Example

```
**** Start Run
[Tue Oct 27 14:16:46 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'enable ↩
    sensor on machine, "wm1"'
[Tue Oct 27 14:16:46 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve, ↩
    "Hot"'
[Tue Oct 27 14:16:46 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve, ↩
    "Cold"'
[Tue Oct 27 14:16:49 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'disable ↩
     sensor on machine, "wm1"'
[Tue Oct 27 14:16:49 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve, ↩
    "Hot"'
[Tue Oct 27 14:16:49 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve, ↩
    "Cold"'
[Tue Oct 27 14:16:49 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
    Agitator"'
[Tue Oct 27 14:17:04 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
    Agitator"'
[Tue Oct 27 14:17:04 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve, ↩
    "Drain"'
[Tue Oct 27 14:17:04 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
    Pump"'
[Tue Oct 27 14:17:04 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'enable ↩
    sensor on machine, "wm1"'
[Tue Oct 27 14:17:07 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
    Pump"'
[Tue Oct 27 14:17:07 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve, ↩
    "Drain"'
```

```
[Tue Oct 27 14:17:07 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'disable ↩
    sensor on machine, "wm1"'
[Tue Oct 27 14:17:07 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'enable  ↩
   sensor on machine, "wm1"'
[Tue Oct 27 14:17:07 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve,  ↩
   "Cold"'
[Tue Oct 27 14:17:10 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'disable ↩
    sensor on machine, "wm1"'
[Tue Oct 27 14:17:10 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve,  ↩
   "Hot"'
[Tue Oct 27 14:17:10 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve,  ↩
   "Cold"'
[Tue Oct 27 14:17:10 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
   Agitator"'
[Tue Oct 27 14:17:20 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
   Agitator"'
[Tue Oct 27 14:17:20 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve,  ↩
   "Drain"'
[Tue Oct 27 14:17:20 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
   Pump"'
[Tue Oct 27 14:17:20 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'enable  ↩
   sensor on machine, "wm1"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
   Pump"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve,  ↩
   "Drain"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::WaterLevelSensor::SENSOR] [debug] 'disable ↩
    sensor on machine, "wm1"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'opening valve,  ↩
   "Drain"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
   Pump"'
[Tue Oct 27 14:17:23 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'starting motor, " ↩
   Spin"'
[Tue Oct 27 14:17:38 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
   Spin"'
[Tue Oct 27 14:17:38 PDT 2015] [wmctrl::Motor::MOTOR] [info] 'stopping motor, " ↩
   Pump"'
[Tue Oct 27 14:17:38 PDT 2015] [wmctrl::WaterValve::VALVE] [info] 'closing valve,  ↩
   "Drain"'
**** Finish Run
**** Trace Begin
14:16:46.275.934:   0.000.000: Transition: {} - Start -> ::wmctrl::WashingMachine{ ↩
   MachineID wm1} ==> Stopped -> FillingToWash
14:16:46.277.209:   0.001.275: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
    - Fill(Warm) -> ::wmctrl::ClothesTub{MachineID wm1} ==> Empty -> Filling
14:16:49.315.235:   3.038.026: Transition: {} - TubFull -> ::wmctrl::ClothesTub{ ↩
   MachineID wm1} ==> Filling -> Full
14:16:49.316.474:   0.001.239: Transition: ::wmctrl::ClothesTub{MachineID wm1} -  ↩
   Full -> ::wmctrl::WashingMachine{MachineID wm1} ==> FillingToWash -> Washing
14:16:49.317.423:   0.000.949: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
    - Agitate -> ::wmctrl::ClothesTub{MachineID wm1} ==> Full -> Agitating
14:17:04.317.591:  15.000.168: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
    - Done -> ::wmctrl::WashingMachine{MachineID wm1} ==> Washing -> DrainingWash
14:17:04.318.076:   0.000.485: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
    - Drain -> ::wmctrl::ClothesTub{MachineID wm1} ==> Agitating -> Emptying
14:17:07.319.831:   3.001.755: Transition: {} - TubEmpty -> ::wmctrl::ClothesTub{ ↩
```

```
    MachineID wm1} ==> Emptying -> Empty
14:17:07.321.104:   0.001.273: Transition: ::wmctrl::ClothesTub{MachineID wm1} -  ↩
    Empty -> ::wmctrl::WashingMachine{MachineID wm1} ==> DrainingWash ->  ↩
    FillingToRinse
14:17:07.321.645:   0.000.541: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Fill(Cold) -> ::wmctrl::ClothesTub{MachineID wm1} ==> Empty -> Filling
14:17:10.322.489:   3.000.844: Transition: {} - TubFull -> ::wmctrl::ClothesTub{ ↩
    MachineID wm1} ==> Filling -> Full
14:17:10.323.555:   0.001.066: Transition: ::wmctrl::ClothesTub{MachineID wm1} -  ↩
    Full -> ::wmctrl::WashingMachine{MachineID wm1} ==> FillingToRinse -> Rinsing
14:17:10.324.224:   0.000.669: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Agitate -> ::wmctrl::ClothesTub{MachineID wm1} ==> Full -> Agitating
14:17:20.324.348:  10.000.124: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Done -> ::wmctrl::WashingMachine{MachineID wm1} ==> Rinsing -> DrainingRinse
14:17:20.324.857:   0.000.509: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Drain -> ::wmctrl::ClothesTub{MachineID wm1} ==> Agitating -> Emptying
14:17:23.326.482:   3.001.625: Transition: {} - TubEmpty -> ::wmctrl::ClothesTub{ ↩
    MachineID wm1} ==> Emptying -> Empty
14:17:23.327.715:   0.001.233: Transition: ::wmctrl::ClothesTub{MachineID wm1} -  ↩
    Empty -> ::wmctrl::WashingMachine{MachineID wm1} ==> DrainingRinse -> Spinning
14:17:23.328.403:   0.000.688: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Spin -> ::wmctrl::ClothesTub{MachineID wm1} ==> Empty -> Spinning
14:17:38.328.527:  15.000.124: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Done -> ::wmctrl::WashingMachine{MachineID wm1} ==> Spinning -> Stopped
14:17:38.329.050:   0.000.523: Transition: ::wmctrl::WashingMachine{MachineID wm1} ↩
     - Stop -> ::wmctrl::ClothesTub{MachineID wm1} ==> Spinning -> StoppingSpin
**** Trace End
```

The first section is the set of log messages that show the interaction with the motors, values and sensors. This amounts to a trace of the external side effects that the domain produces on the washing machine hardware. The second portion shows 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 of the target instance, from current state to new state, is shown after the "==⇒" symbol. Instances are shown with the set of attribute name / value pairs that form the identifier contained in the instance reference.

The figure below shows the trace information in sequence diagram form. This diagram was produced using `rosea diagram all` to generate a file suitable for the `seqdiag` program to render a graphic.

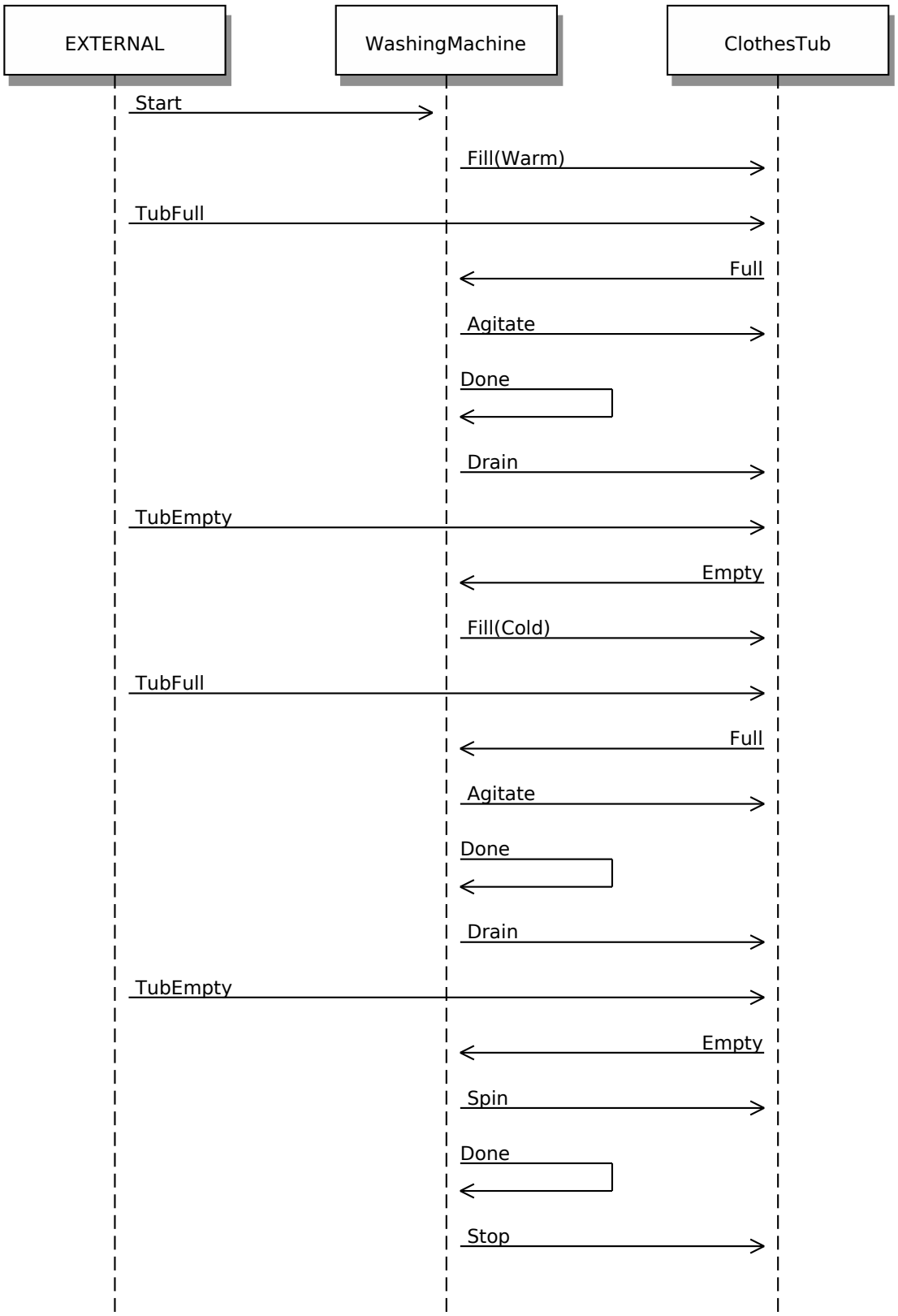

Figure 11.5: Example Sequence Diagram

The track labeled **EXTERNAL** shows events that originate outside of the domain. The example begins with a **Start** event being sent to an instance of the **WashingMachine** class. 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 **WashingMachine** and **ClothesTub** instances 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 by the state activities are often very important), it does go a long way to showing the chronological unfolding of the example's execution.

Chapter 12

Code Organization

In this section we show the organization of the files that can be tangled from the literate source.

First, this software is copyrighted. It is licensed in the same manner as Tcl itself.

```
<<copyright info>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2014-2015 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.
```

Each *root chunk* is described in a section below.

Source Code

```
<<rosea.tcl>>=
<<copyright info>>

package require Tcl 8.6

package require logger
<<required packages>>

proc ::K {x y} {set x}

namespace eval ::rosea {
    <<rosea exports>>
    namespace ensemble create

    variable version 1.3

    logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar
    <<rosea data>>
    <<rosea commands>>
    <<rosea namespace layout>>

    namespace path ::rosea::Helpers
    namespace import ::rosea::Config::ConfigEvaluate
}

package provide rosea $::rosea::version
```

Package Index

We also provide a root chunk to extract a package index file.

```
<<pkgIndex.tcl>>=
package ifneeded rosea 1.3 [list source [file join $dir rosea.tcl]]
```

Unit Tests

```
<<rosea.test>>=
<<copyright info>>

package require Tcl 8.6
package require cmdline
package require logger

<<required packages>>
```

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

logger::setlevel $options(level)

source ../src/rosea.tcl
chan puts "testing rosea version: [package require rosea]"

package require tcltest
eval tcltest::configure $argv

namespace eval ::rosea::test {
    namespace import ::tcltest::*
    ::logger::initNamespace [namespace current]

    <<tclral imports>>
    namespace import ::ral::relvar

    <<test utility procs>>
    <<helper command tests>>
    <<class command tests>>
    <<instance command tests>>
    <<dispatch command tests>>
    <<config command tests>>
    <<generate command tests>>
    <<populate command tests>>
    <<save command tests>>
    <<trace command tests>>

    cleanupTests
}
```

Example Domain

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

package require Tcl 8.6
package require logger

source rosea.tcl

rosea configure {
    domain wmctrl {
        <<wmctrl operations>>
        <<WM class>>
        <<WC class>>
        <<CT class>>
        <<WV class>>
        <<MTR class>>
        <<WLS class>>
        <<wmctrl relationships>>
    }
}

rosea generate
```

```
rosea populate {
    domain wmctrl {
        <<initial instance population>>
    }
}
```

Example Program

```
<<runwm.tcl>>=
<<wmctrl.tcl>>
<<external operation stubs>>
<<running the example>>
```

Reference Documentation

```
<<rosea.man>>=
[manpage_begin rosea n 1.3]
[comment {
<<copyright info>>
}]
[moddesc "XUML Data and Execution Architecture"]
[titledesc "Relation Oriented Software Execution Architecture"]
[copyright "2014-2015 by G. Andrew Mangogna"]
[require rosea [opt 1.0]]
[description]
<<manual description>>
[section "Package Commands"]
The commands in this section are part of the ensemble command [cmd ::rosea].
Their use is to configure, generate and populate domains
and to control state machine tracing during run time.
[list_begin definitions]
[usage "Package Commands"]
<<manual package commands>>
[list_end]
[section "Configuration Commands"]
The commands in this section is used are intended to be invoked as part
of the [arg body] script of the [cmd configure] command.
There is only one command, [cmd domain], and it is used to define
the elements of a domain.
[list_begin definitions]
[usage "Configuration Commands"]
<<manual configuration commands>>
[list_end]
[subsection "Domain Configuration Commands"]
The commands in this section are intended to be invoked as part of the
[arg body] of the [cmd domain] command.
These commands specify the details of the various elements that
are part of a domain.
[list_begin definitions]
[usage "Domain Configuration Commands"]
<<manual domain configuration commands>>
[list_end]
[subsection "Class Configuration Commands"]
The commands in this section are intended to be invoked as part of the
[arg body] script of the [cmd class] command.
```

```
These command define the characteristics a class such as its attributes.
[list_begin definitions]
[usage "Class Configuration Commands"]
<<manual class configuration commands>>
[list_end]
[subsection "State Model Configuration Commands"]
The commands in this section are intended to be invoked as part of
the [arg body] script of the [cmd statemodel] command.
These command define the states, transitions, _etc._ of the state model that is
bound to the enclosing [cmd class].
[usage "State Model Configuration Commands"]
[list_begin definitions]
<<manual statemodel configuration commands>>
[list_end]
[subsection "Assigner Configuration Commands"]
The commands in this section are intended to be invoked as part of the
[arg body] script of the [cmd assigner] command.
These commands define the state model of the assigner and are similar
to the
State Model Configuration Commands
[sectref "State Model Configuration Commands" above].
[list_begin definitions]
[usage "Assigner Configuration Commands"]
<<manual assigner configuration commands>>
[list_end]
[subsection "Population Commands"]
The commands in this section are intended to be invoked as part of the
[arg body] script of the [cmd "::rosea populate"] command.
These commands define class instance values that are to be part of the
initial instance population for the domain.
Population commands should only be invoked after the domain
has been generated.
It is only at that time that the data storage has been create to
hold the class instances.
[list_begin definitions]
[usage "Population Commands"]
<<manual populate commands>>
[list_end]
[subsection "Domain Population Commands"]
The commands in this section are intended to be invoked as part of the
[arg body] script of the [cmd domain] command which is, in turn,
part of a script for defining an initial instance population.
[list_begin definitions]
[usage "Domain Population Commands"]
<<manual domain populate commands>>
[list_end]
[section "Architecture Commands"]
The commands in this section are supplied by [cmd rosea].
They are intended to be invoked by state activities and
operations and provide the means to access model level
semantics in the implementation.
[para]
Many of the commands below accept as an argument
or return as a value an [emph "instance reference"].
An instance reference or [arg instref],
is an ordinary Tcl value whose internals should be considered opaque.
An instance reference provides a means to refer to zero or more
class instances.
The primary purpose of the instance reference is to specify which
class instances are to be subject to an operation or
to specify the results of some operation on the class model as a whole,
[emph e.g.] finding a set of instances based on some attribute value.
```

```
[subsection "Class Commands"]
Each class in a domain will have a class command defined for it.
The class command has the same name as the class and is an ensemble
command whose subcommands are given below.
The subcommands listed below are included in the class ensemble command
for all classes in the domain.
In addition,
any class based operations defined for the class will also
appear as additional subcommands in the class command ensemble.
[list_begin definitions]
[usage "Class Commands"]
<<manual architecture class commands>>
[list_end]
[subsection "Relationship Commands"]
Each relationship in a domain will have a relationship command defined
for it.
The relationship command has the same name as the relationship
and is an ensemble command whose subcommands are given below.
The subcommands of a relationship ensemble command will depend upon
the type of the relationship and not all of the subcommands below
are available to all relationships.
How each command applies to a relationship is described below.
[list_begin definitions]
[usage "Relationship Commands"]
<<manual architecture relationship commands>>
[list_end]
[subsection "Instance Commands"]
The commands below define operations on class instances.
They are intended to be invoked as part of state activities and operations
to access the data model and signal events.
The context in which a state activity runs is such that these
commands will resolve using unqualified command names.
[list_begin definitions]
[usage "Instance Commands"]
<<manual architecture instance commands>>
[list_end]
[manpage_end]
```

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.

Articles

[6] [ls-articulate] Leon Starr, How to Build Articulate UML Class Models, 2008, http://www.modelint.com/how-to-build-articulate-uml-class-models/

[7] [ls-time] Leon Starr, Time and Synchronization in Executable UML, 2008, http://www.modelint.com/time-and-synchronization-in-executable-uml/

Appendix A

Literate Programming

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

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

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

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers 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.

Index