Relation Oriented Programming with Raloo

What Happens When ::ral Meets ::oo?

Andrew Mangogna

15th Annual Tcl/Tk Conference
October 22-24, 2008

Copyright

Abstract

Raloo is an objected-oriented extension to Tcl that combines the relational data structuring capability of TclRAL with an object-oriented programming style as provided by TclOO. This paper introduces **Raloo** and describes how the capabilities of **Raloo** may be used to capture the semantics of a software problem in a more declarative manner. **Raloo** supports problem decomposition into domains, with domains containing classes, relationships and domain functions. Both synchronous processing and asynchronous processing via state machines are provided. The relationship of **Raloo** to formal software methods is also discussed.

1. Introduction

This paper is about **Raloo**, an object-oriented Tcl extension that combines the relational algebra as implemented by **TclRAL**[1] with object orientation provided by **TclOO**. The general term I apply to this combination is *relation-oriented programming* which distinguishes it from the more familiar object-oriented programming approach. **Raloo** solutions consist of three distinct projections of a software problem: relationally structured data, finite state machines and object-oriented Tcl code. The remainder of this section gives a brief overview of some background information. The necessity of space means that the overview is too brief. Next we follow some examples that show how data is structured and how asynchronous event based processing is accomplished. Finally, we discuss the relationship of **Raloo** with formal software methods.

1.1. Relational Model of Data

The literature on the Relational Model of Data is vast. Because the Relational Model underlies the view of Relational Database Management Systems (RDMS) and because of the commercial importance of such systems, much research, refinement and formalization has been devoted to the relational model in the years since Codd[2] first developed it. Space limitations and more immediate concerns allow only the briefest introduction to relational concepts.

In this paper we generally follow the formalization of the Relational Model as given by more recent writings of Date[3] and Darwen[4]. We are not concerned with Database Management Systems in this paper. The Relational Model is a complete, coherent model of data and is not necessarily dependent upon a database for its realization. Relations exist as logical entities outside of the database arena. The properties of the Relational Model we are most interested in here are:

- A complete, consistent means to organize data for any problem.
- The close association between relations and predicate logic.
- A well formulated algebra for evaluating expressions on relation values.
- The ability to specify functional relationships between relation variables which include declarative constraints on the cardinality of those relationships.

1.1.1. Relation Values

A relation value is defined as:

- A heading consisting of a set of distinctly named attributes. Each attribute is associated with a data type.
- A body consisting of tuples that match the heading of the relation. Each attribute of the tuple holds a value from the set of values defined by the data type of the attribute.
- At least one identifier[5]. An identifier is a subset (possibly improper) of the attributes of the relation heading with the property that no two tuples in the relation body have the same values for the attributes of the identifier and no subset of the identifier is also an identifier of the relation. The attributes of an identifier may have a non-empty intersection with another identifier, but may not be a subset of another identifier (*i.e.* at least one attribute must not be in common). It is common for relations to have only a single identifier consisting of a single attribute.

[1] Both TclRAL and Raloo are free software and are available at http://sourceforge.net/projects/tclral.

[2] Codd, E.F., *A Relational Model of Data for Large Shared Data Banks*, CACM 13, No. 6 (June 1970).

[3] Date, C.J., *An Introduction to Database Systems*, 8th Ed, Pearson, 2004, ISBN: 0-321-19784-4

[4] Date, C.J. and Hugh Darwen, *Databases, Types, and the Relational Model: The Third Manifesto*, Addison-Wesley, 2007, ISBN: 0-321-39942-0

[5] In database contexts, identifiers are usually referred to as *keys* or *candidate keys* and one of which may be considered a *primary key*. We prefer the term, identifier, as clearer and indicative of the role that is being satisfied.

1.1.2. Relation Variables

Since **TclRAL** supplies a native Tcl relation type, relation values may be stored in ordinary Tcl variables and may be the result of command evaluation. It is important to distinguish between relation values that may be held in ordinary Tcl variables or may be intermediate command results and those that are held in special relation variables or *relvars*. **TclRAL** allows the definition of relvars as distinct from ordinary Tcl variables that happen to hold a relation value. The reason for this distinction is that relvars allow declarative constraints to be placed upon them. **Raloo** uses the relvars of **TclRAL**, by associating each **Raloo** class with a **TclRAL** relvar and associating each relationship between classes with a **TclRAL** relvar constraint.

1.1.3. Tabular View of Relations

Frequently, relation values are displayed as tables. It is also possible to talk about relations using tabular words such as *columns* and *rows*. There is no real problem in taking the tabular view of relations as long as it is firmly understood that there is no inherent order in either the attributes or tuples of a relation. Clearly when a relation is displayed some order of presentation must be chosen. The same is true of the implementation of a relation in a program. Computer memory has a natural order and storing a relation in computer memory will result in it being stored in some order. But that order is a property of the *implementation* and not a property of the *model* of relations. So if we display a **Dog** relation as the table,

Name	Breed	Age
Buffy	Poodle	3
Rover	Retriever	5

we must realize that the table,

Age	Name	Breed
5	Rover	Retriever
3	Buffy	Poodle

is a display of the same relation value. Indeed all the permutations of tuples and attribute orders are different tabular views of the same relation value. The important point here is that access to a tuple of a relation is only made via the *value of the attributes* and access to an attribute of a tuple is only by the *name of the attribute*. This means of access to a tuple allows for data independence, *i.e.* the programmed access is independent of the method used by the implementation to store the data physically. It is sometimes tempting to think of a relation as some kind of super array from which we can extract the third row or the second column. Resist that temptation. With that warning, we will use tabular notation in the rest of the paper, confident that the reader will realize that a tabular representation of a relation is not unique and implies nothing about how the elements of a relation are accessed.

1.1.4. Relations and Logic

There is a close association between relations and predicate logic.[6] It is this close association that allows us to interpret the contents of relation values as propositions about our subject. For example, we interpret the **Dog** table above to imply that there exists in our world a Dog named Rover, that is of the Retriever breed and is aged 5 years. This relation also states that there exists a Dog named Buffy that is of the Poodle breed and is 3 years old. We insist that each tuple of a relation be a *true* proposition. This point of view allows us to view the relation as a predicate with each tuple being a true proposition obtained by binding values to the attributes. Further, we

[6]Date, C.J., *Logic and Databases: The Roots of Relational Theory*, Trafford, 2007, ISBN: 1-4251-2290-6

assume a **closed world** such that the value of a relation at any time contains “all and only those tuples that correspond to true propositions.”[7] This correspondence between predicate logic and relations gives us the fundamental ability to construct a valid system of reasoning to evaluate answers that are consistent with the logic of the problem as we have encoded it into relations.

1.2. Object Orientation

The subject of object oriented programming is also vast and the limited space here cannot do justice to the topic. Object oriented technology is particularly useful as programs become larger and more difficult to keep track of. The close association between data and its operations and the emphasis on interfaces that remain constant greatly simply the task of dealing with large programming projects. We will content ourselves here with summarizing those characteristics of object orientation that are most pertinent to **Raloo.**

Relatively recently the Tcl community decided to include object oriented constructs into the core of the language. Before this, several objected oriented extensions, both script and "C" based, were used to achieve various forms of object orientation. Indeed the variety of different object oriented approaches has been quite amazing to consider. Because of the ease by which Tcl can be extended, several different object oriented approaches have been constructed and have accumulated significant usage.

TclOO gives a well designed set of primitives upon which higher order object oriented extensions may be built. Object orientation in Tcl is usually associated with generating a different command for each object. The first argument to object commands is taken as a method name and that method is dispatched. The dispatched method has access to both the object data and any additional arguments to the method. Much of the complexity of object orientation in Tcl is in determining how the methods of an object are specified and resolved at run time to an actual code body. Given the dynamic nature of Tcl, neither aspect of method dispatch need be static in time.

2. The Raloo Approach

Raloo takes a distinctly structured approach to solving software problems.

(1) The problem is divided into a set of **Domains** which represent coherent subject matters contained in the problem. Domains interface to each other via as set of *domain operations*

(2) Each domain consists of a class model that is a relationally normalized schema consisting of **Classes** and the **Relationships** between Classes.

(3) Certain classes may have a *lifecycle* that is defined as a Moore type state model associated with the class. Asynchronous processing is accomplished by the state models of the active classes interacting with each other by sending events.

(4) The synchronous processing of state actions and other algorithmic type processing is given by object oriented Tcl code.

In the sections below we will look at this approach using a simple example and show **Raloo** code sequences that implement this approach.

3. Domains

As programs become larger, it becomes necessary just from human understandability considerations to partition the problem into components. Raloo defines that component to be a **domain**. A domain is intended to be its own world with its own set of rules and policies that forms a coherent subject matter. In the end, that is a rather vague definition and the contents of a domain are

[7] Ibid., p. 99

truly an element of the design of the program. The design goal is to obtain domains with a consistent level of abstraction and that are potentially reusable. Domains can also be associated with the notion of *cross-cutting concerns* as is defined by aspect oriented programming.[8]

One intent of domain based design is to avoid the problems that functional decomposition often yields by abstracting and factoring common components to be used by the entire application. For example, many industrial control applications use a common means of dealing with exceptional conditions, usually referred to as *alarms*. Alarm policies and procedures can be quite extensive, but the notion of an alarm and how it is handled can be described separately from the condition or situation in the application that is considered exceptional.

We will not discuss domains further in this paper since the examples given are trivial enough not to require extensive decomposition. However, keep in mind that any realistic application most likely will need multiple domains to capture the distinct rules of its component parts.

4. Capturing Problems in Data

Taken together, a set of variables with relation values (relvars) can be seen as fundamental statement of the predicates of a program. But programs also must deal with the relationship between data entities. Relations allow you to specify many of the rules and constraints of a problem in the structure of the data. Fundamentally, the relationships between relvars are functions or partial functions defining a set of associations between the relvars. We think of this as exploiting a *declarative* approach to the structure of the program logic as contrasted with a more *procedural* view of program design. The difference usually amounts to what is specified and controlled in data and what is accounted for in the code of the program. What relational algebraic systems, like **TclRAL**, represent is factoring into common code a set of rules that allow you to specify in data most of the real-world rules that a program must enforce.

In this section we explore the three primary ways to describe the relationships between data entities.

- Simple relationships.
- Generalization relationships.
- Associative relationships.

Finally, we get to see some **Raloo** code.

4.1. Simple Relationships

Let’s consider a small example about modern clothes washing machines. Here we are interested in capturing a few facts about the way we make washing machines, namely:

- Every washing machine is identified by a serial number.
- Every washing machine is of some particular model.
- There are different models of washing machine.
- It the model of the washing machine that determines its properties.

The diagram below shows a graphic that represents these rules.[9]

[8] http://en.wikipedia.org/wiki/Aspect-oriented_programming

[9] There are many different graphical conventions in use for software ideas. Commonly, UML is used. I avoid it here because UML has far too many graphical constructs and the extra cruft only serves to confuse the situation. In this very simple graphical representation, rectangles represent classes with their attributes named inside. Relationship are represented by directed line segments with annotation for the name and cardinality of the relationship. Additional annotation giving verb phrases to each direction of traversal of the relationship is also helpful in associating the relationship to the problem semantics.

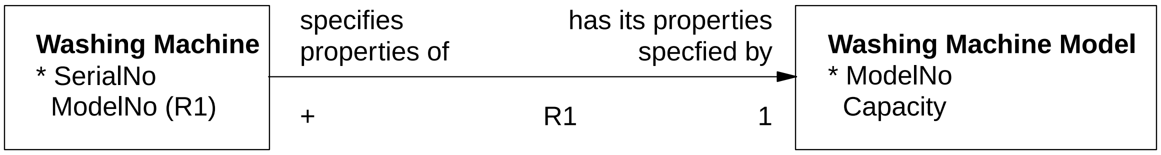

Figure 1. Washing Machine Models

We could also represent things as two tables and we have filled in some example rows.

Washing Machine	
SerialNo	ModelNo
AAC77753	WA1
AAC77996	WA1
BAC77893	WA2

Washing Machine Model	
ModelNo	Capacity
WA1	55
WA2	45

In **Raloo** we would code this situation as:

```
package require raloo
namespace import ::raloo::Domain

Domain create wmMgmt {
    Class WashingMachine {
        Attribute {
            *SerialNo string
            ModelNo string
        }
    }
    Class WashingMachineModel {
        Attribute {
            *ModelNo string
            Capacity int
        }
    }
    Relationship R1 WashingMachine +-->1 WashingMachineModel
}

wmMgmt transaction {
    WashingMachine insert SerialNo AAC77753 ModelNo WA1
    WashingMachine insert SerialNo AAC77996 ModelNo WA1
    WashingMachine insert SerialNo BAC77893 ModelNo WA2

    WashingMachineModel insert ModelNo WA1 Capacity 55
    WashingMachineModel insert ModelNo WA2 Capacity 45
}
```

Now, let's look closely at this code sequence and see what is happening. We create a domain called, **wmMgmt**, that contains two classes and one relationship. Since this is Tcl code, **Domain** must be a command and its *create* subcommand will create a new domain as given by the next argument. Indeed, *Domain*, is a TclOO class with its usual **create** method. The last argument is a script that contains command invocations that define the contents of the domain.

In this case, we define two classes by invoking the **Class** command. The classes are named **WashingMachine** and **WashingMachineModel**. The **Class** command takes two arguments, the name of the class and a definition script that defines the properties of the class. Attributes of a class are defined by invoking the **Attribute** command. Attributes are given as a list of name/type pairs. Type names are any valid Tcl data type with **string** being Tcl's universal type. Attribute names that begin with an asterisk (*) indicate that the attribute is an identifier. Every class must have at least one identifier and an identifier may consist of more than one attribute. In the example, both classes have only a single identifier that consists of only a single attribute. This is a common occurrence. No two tuples in a relation may have the same values for their identifiers. Remember, relations are sets and as sets do not have duplicate members.

The last command in the domain definition script for **wmMgmt** creates a relationship between the **WashingMachine** and **WashingMachineModel** classes. The relationship is named **R1**. Any string may be used to name a relationship, but long held convention uses the letter, **R**, followed by a number. A relationship describes an association between classes. In the example, R1, states that attributes in **WashingMachine** refer to identifying attributes in **WashingMachineModel**. In this simple case since **WashingMachineModel** has only a single identifier and since the attribute in **WashingMachine** that refers to **WashingMachineModel** has the same name as the single identifying attribute, **Raloo** can figure this out and no additional specification of the referential attribute mapping is required.

A relationship constrains the values that attributes may have. In the example, R1, states that for every tuple in WashingMachine, the value of WashingMachine.ModelNo must match the value of exactly one tuple in WashingMachineModel. Conversely, for every tuple in WashingMachineModel, the value of WashingMachineModel.ModelNo must match the value of ModelNo in one or more tuples of WashingMachine. This is indicated by the use of "+" and "1" on the relationship specification.[10] Notice particularly how the **R1** relationship precisely states a rule of the problem, namely that individual washing machines are always of some model and every washing machine model describes at least one washing machine. The relationship constrains the allowed set of tuples in the two classes and **Raloo** enforces this constraint with no additional programming. The simple declaration of the relationship gives **Raloo** sufficient knowledge to insure the relationship constraints are not violated.

The last part of the example is a transaction where some instances are inserted into the classes. At the end of the transaction commands, **Raloo** evaluates the constraints imposed by the relationships and insures that they are consistent. To see this in action, suppose we follow the above script with:

```
wmMgmt transaction {
    WashingMachine insert SerialNo BAC77899 ModelNo WA3
}
```

This command attempts to insert an instance of **WashingMachine** for which there is no corresponding instance of **WashingMachineModel** and we get the following error message:

```
for association ::wmMgmt::R1(::wmMgmt::WashingMachine [+] ==> [1]\
::wmMgmt::WashingMachineModel), in relvar ::wmMgmt::WashingMachine
tuple {SerialNo BAC77897 ModelNo WA3} references no tuple

    while executing
"relvar eval $script"
    (class "::raloo::Domain" method "transaction" line 2)
    invoked from within
"wmMgmt transaction {
    WashingMachine insert SerialNo BAC77897 ModelNo WA3
}"
```

Which is a long winded way to say that there is no "WA3" washing machine model. Any processing that affects the values of the class instances is done in the context of a transaction on the class population and at the end of that transaction the constraints imposed by the relationships must be consistent. If constraint checks fail, the data values are restored to what they were before the transaction is executed. This is done without any additional code in the application. Indeed that is the intent, namely to factor out the processing that can be inferred by declarative statements such as the relationship definition.

The time evolution of a program can be viewed as moving the data values from one coherent state to another. Processing affects the number of tuples in the classes and the values of attributes, but at the end of each processing transaction the data content of the classes must be consistent with the rules defined by the relvar and relationship constraints.

[10] The use of "1", "?", "*" and "+" to specify cardinalities of exactly one, at most one, zero or more and one or more, respectively, is intended to be mnemonic of the usage of these characters in regular expression syntax.

4.2. Generalization Relationships

Some problem requirements are best described by placing strict categories on items. Stating that some thing must be in some category is a form of set partitioning.

Continuing with our example, suppose in our world we wish to keep track of how our washing machine models are used in production. After some time, we discontinue models in favor of new designs, but we need to keep track of the old designs for reasons of maintenance of washing machines we previously sold. We might show that graphically as:

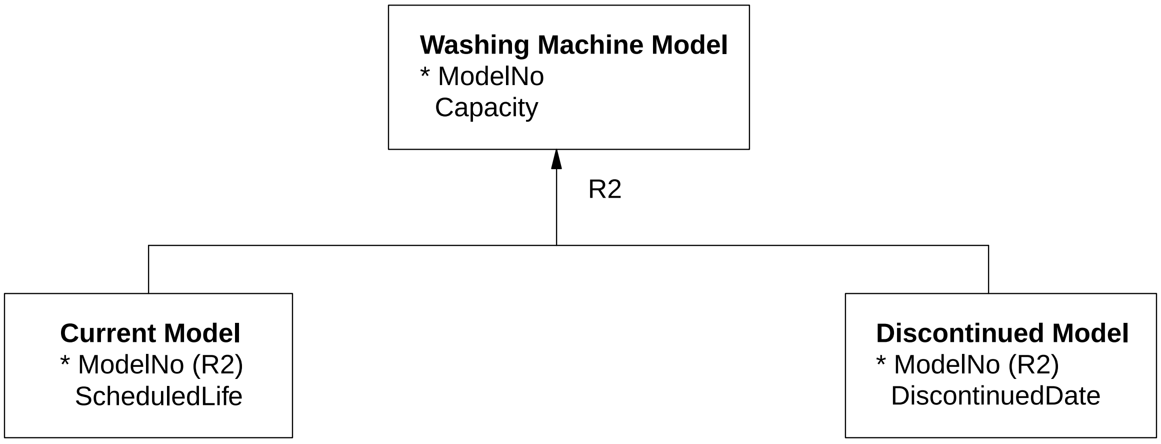

Figure 2. Washing Machine Model Types

In this graphic, **Washing Machine Model** is the *supertype* and is related to two *subtypes*, **Current Model** and **Discontinued Model**. The rules of this arrangement are that each tuple in each of the subtypes must refer to exactly one tuple in the supertype and each tuple in the supertype must be referred to by exactly one tuple in exactly one of the subtypes. This is equivalent to saying that **Washing Machine Model** is completely partitioned into two disjoint subsets, **Current Model** and **Discontinued Model**.

This type of relationship is called many things. **Raloo** uses the term, *generalization.* Unfortunately, there is some confusion surrounding this idea and the name, generalization. In UML (sadly), a generalization relationship does not necessarily have to be complete and disjoint. This relationship is also sometimes seen as an instance of *inheritance*. In our usage, there is no inheritance, only set partitioning. Sometimes this arrangement is also equated to a set of one-to-one relationships that are conditional with respect to the mapping of the supertype onto the subtypes. Such an arrangement fails to enforce the complete aspect of the set partitioning.

Again, we may use tables to demonstrate this.

Washing Machine Model	
ModelNo	Capacity
WA1	55
WA2	45
WC1	30
WD2	45

Current Model	
ModelNo	ScheduledLife
WA1	24
WA2	36

Discontinued Model	
ModelNo	DiscontinuedDate
WC1	1July2007
WD2	1May2008

In **Raloo**, we capture the situation as:

```
package require raloo
namespace import ::raloo::Domain

Domain create wmMgmt {
    Class WashingMachineModel {
        Attribute {
            *ModelNo string
            Capacity int
        }
    }
    Class CurrentModel {
        Attribute {
            *ModelNo string
            ScheduledLife int
        }
    }
    Class DiscontinuedModel {
        Attribute {
            *ModelNo string
            DiscontinuedDate string
        }
    }
    Generalization R2 WashingMachineModel {
        SubType CurrentModel
        SubType DiscontinuedModel
    }
}

wmMgmt transaction {
    WashingMachineModel insert ModelNo WA1 Capacity 55
    WashingMachineModel insert ModelNo WA2 Capacity 45
    WashingMachineModel insert ModelNo WC1 Capacity 30
    WashingMachineModel insert ModelNo WD2 Capacity 45

    CurrentModel insert ModelNo WA1 ScheduledLife 24
    CurrentModel insert ModelNo WA2 ScheduledLife 36

    DiscontinuedModel insert ModelNo WC1 DiscontinuedDate 1July2007
    DiscontinuedModel insert ModelNo WD2 DiscontinuedDate 1May2008
}
```

In this code, three classes are defined. The **WashingMachineModel** class is the supertype and the **Generalization** command defines the subtypes of **WashingMachineModel** to be **CurrentModel** and **DiscontinuedModel**. The generalization, **R2**, states that the instance of **WashingMachineModel** will be partitioned into two disjoint sets. Necessarily, the sum of the number of instances of the subtypes must equal the number of instances of the supertype.

If we try to add an unrelated supertype, such as in:

```
wmMgmt transaction {
    WashingMachineModel insert ModelNo WD3 Capacity 50
}
```

we would be told:

```
for partition ::wmMgmt::R2(::wmMgmt::WashingMachineModel is partitioned
[::wmMgmt::CurrentModel | ::wmMgmt::DiscontinuedModel]),
in relvar ::wmMgmt::WashingMachineModel
tuple {ModelNo WD3 Capacity 50} is not referred to by any tuple

    while executing
"relvar eval $script"
    (class "::raloo::Domain" method "transaction" line 2)
    invoked from within
"wmMgmt transaction {
    WashingMachineModel insert ModelNo WD3 Capacity 50
}"
```

which tells that we have a supertype instance that is not related to any subtype instance.

4.3. Associative Relationships

Continuing with our simple example, let us suppose that we are interested in tracking the features that our washing machines have relative to their model. So for example, Model WA1 has separate wash and rinse temperatures and an automatic timer whereas Model WA2 has a bleach dispenser but no timer. In general, a given model of washing machine has many features and multiple models share many of the same features. We insist that a feature is not a feature unless it is designed into at least one washing machine and that all washing machine models have one or more features.

In this case we will require another class to capture these rules. Some associations between classes require yet another class to mediate the association. These types of relationships are called *Associative Relationships*. Associative classes arise in two distinct situations. In one case the cardinality of the relationship requires that we have another class to hold all the necessary referential attributes. In the other case, we define attributes that apply to the relationship itself and therefore those attributes are properly included in the associative class and not in the participating classes.

Graphically, we represent this situation as:

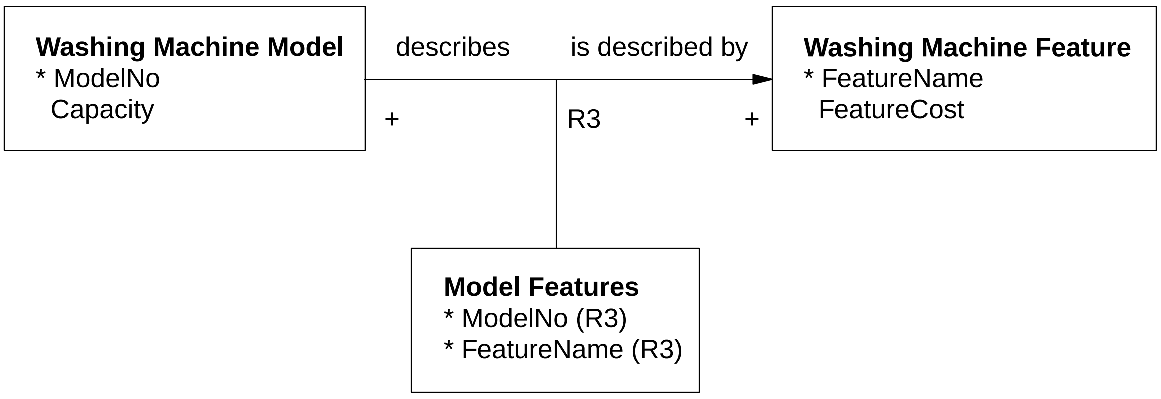

Figure 3. Washing Machine Model Features

Again, we can have a tabular view with example rows.

Washing Machine Model	
ModelNo	Capacity
WA1	55
WA2	45

Washing Machine Feature	
FeatureName	Feature Cost
Dual Temp Control	20
Automatic Timer	45
Bleach Dispenser	15

Model Feature	
ModelNo	**FeatureName**
WA1	Dual Temp Control
WA1	Automatic Timer
WA2	Dual Temp Control
WA2	Bleach Dispenser

And the corresponding **Raloo** code is:

```
package require raloo
namespace import ::raloo::Domain

Domain create wmMgmt {
    Class WashingMachineModel {
        Attribute {
            *ModelNo string
            Capacity int
        }
    }
    Class WashingMachineFeature {
        Attribute {
            *FeatureName string
            FeatureCost int
        }
    }
    Class ModelFeature {
        Attribute {
            *ModelNo string
            *FeatureName string
        }
    }
    AssocRelationship R3\
        WashingMachineModel +--ModelFeature-->+ WashingMachineFeature
}

wmMgmt transaction {
    WashingMachineModel insert ModelNo WA1 Capacity 55
    WashingMachineModel insert ModelNo WA2 Capacity 45

    WashingMachineFeature insert FeatureName "Dual Temp Control"\
            FeatureCost 20
    WashingMachineFeature insert FeatureName "Automatic Timer"\
            FeatureCost 45
    WashingMachineFeature insert FeatureName "Bleach Dispenser"\
            FeatureCost 15

    ModelFeature insert ModelNo WA1 FeatureName "Dual Temp Control"
    ModelFeature insert ModelNo WA1 FeatureName "Automatic Timer"
    ModelFeature insert ModelNo WA2 FeatureName "Dual Temp Control"
    ModelFeature insert ModelNo WA2 FeatureName "Bleach Dispenser"
}
```

The only new construct in this code is the definition of **R3** by using the **AssocRelationship** command. The syntax of the relationship cardinality is similar to before, but now the name of the associative class is mentioned emphasizing its mediating role in implementing the relationship. Note that the direction of the relationship is somewhat arbitrary. It is the associative class that holds the referential attributes and so the direction could have been chosen to be the opposite of that given. Choose the relationship direction for associative relationships based on what makes the semantics of the problem clearest. There is usually one direction that is semantically more significant.

Raloo enforces the cardinality of associative relationships and since **R3** is defined as many to many and unconditional, there may not be any **Washing Machine Models** or **Washing Machine**

Features that are not related in some way. So, for example, if we attempt to add an unrelated washing machine model such as:

```
wmMgmt transaction {
    WashingMachineModel insert ModelNo WA7 Capacity 45
}
```

we will be given the error:

```
for correlation ::wmMgmt::R3(::wmMgmt::WashingMachineModel <== [+]
 ::wmMgmt::ModelFeature [+] ==> ::wmMgmt::WashingMachineFeature),
 in relvar ::wmMgmt::WashingMachineModel
tuple {ModelNo WA7 Capacity 45} is not referenced by any tuple

    while executing
"relvar eval $script"
    (class "::raloo::Domain" method "transaction" line 2)
    invoked from within
"wmMgmt transaction {
    WashingMachineModel insert ModelNo WA7 Capacity 45
}"
```

which tells us that there was no instance of **ModelFeature** that referred to an instance of **WashingMachineModel** that is identified as **WA7**.

5. Lifecycle Behavior

The data of an application is only part of the story. Many classes of an application will exhibit behavior that varies over time. The variations in behavior usually depend upon what has gone on in the past. The natural means of modeling such behavior is via a state machine. **Raloo** supports making any class *active* by associating a Moore type state machine with that class.[11] Each state has a body of code associated with it that is executed upon entry into the state.

To demonstrate we will continue in our washing machine example and look at a very simplified version of how a washing machine might run a cycle to wash clothes. Our washing machine consists of the familiar upright tub that is attached to a motor that can either agitate the tub back and forth or spin the tub at high speed. In addition we have a pump to remove water from the tub and a valve to allow water to flow into the tub. Also we have sensors that can detect when the tub is full or empty. Clearly, a real-world automatic washing machine is much more sophisticated than this simple model.

First we must encode the problem semantics in a class model.

[11] There has been much discussion over which style of state machine, Moore or Mealy, is the best. Moore machines execute their actions upon entry into a state and therefore the actions are usually associated with the state itself. Mealy machines execute their actions upon exit from a state and therefore the actions are usually associated with the transition between states. It can be proven that they are equivalent in their expressive power so to some extent the passions of the debate are overblown. In the end, I prefer Moore machines because they yield simpler implementation data structures and I prefer the direct association between state and processing.

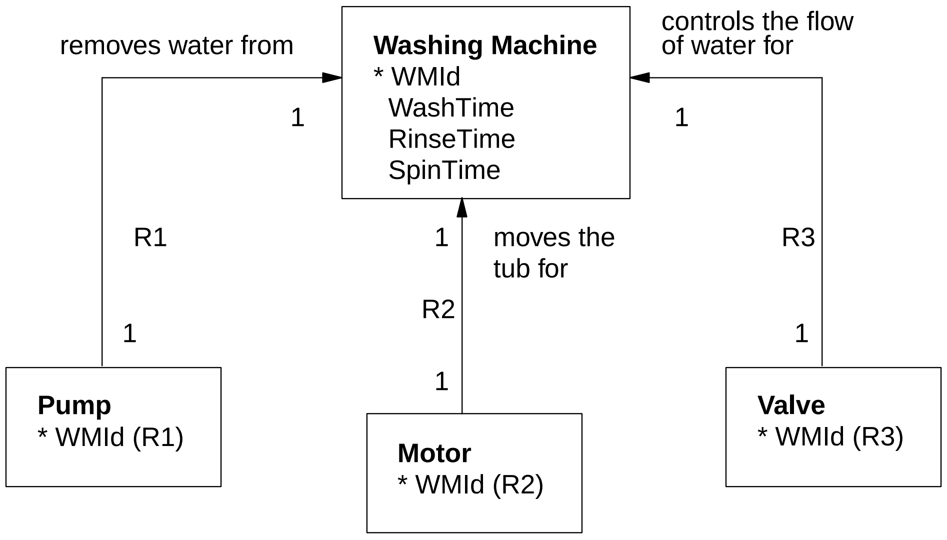

Figure 4. Washing Machine Class Model

This model is very simple and just states that every washing machine has a pump, motor and valve. Also there are no pumps, motors, or valves that are not attached to some washing machine.[12] The diagram below shows a state model for the washing machine cycle. The initial state is **Idle**. Again the state model is overly simplistic for a real-world situation. For example, there is no way to stop a cycle once it has been started. Also, note that no pump is run while in the spinning part of the cycle begging the question of where the extracted water is to go. After the state model diagram we give the **Raloo** code.

[12] Remember, the predicates represented by the relation variables need only be true and not necessarily profound. The fact that there are no pumps that are not somehow bound up in washing machines is blatantly obvious but still true and allowing for the opposite to be true would indeed be strange for this problem. But relation variable predicates should also relevant to the problem. It is a common mistake to include items in a class model that are true from some point of view but not particularly relevant to the abstractions that motivate the solution of the problem at hand. Just because we know that the motor in the washing machine is colored black does not mean we need a color attribute for motors.

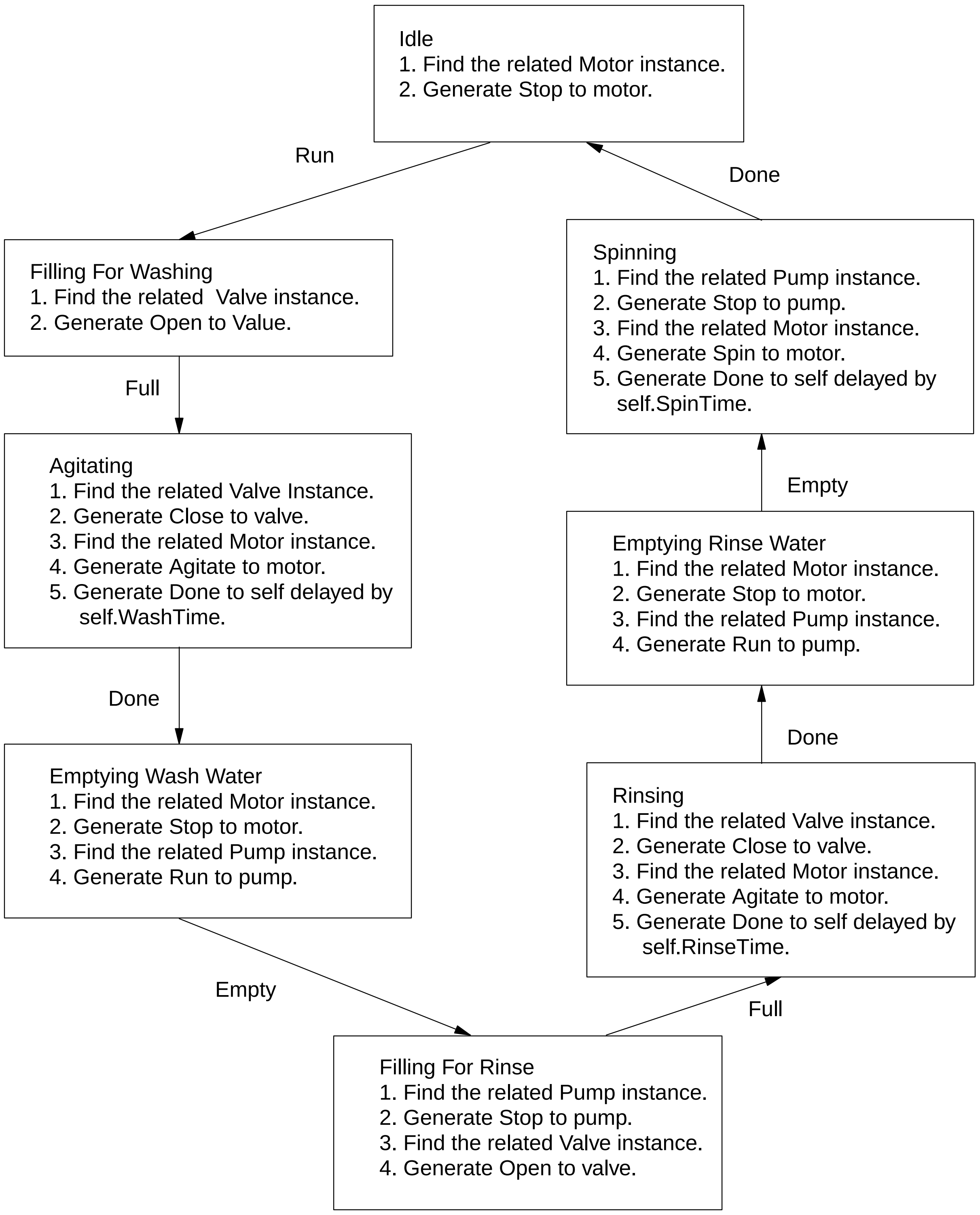
Idle
1. Find the related Motor instance.
2. Generate Stop to motor.
Run
Done
Filling For Washing
1. Find the related Valve instance.
2. Generate Open to Value.
Spinning
1. Find the related Pump instance.
2. Generate Stop to pump.
3. Find the related Motor instance.
4. Generate Spin to motor.
5. Generate Done to self delayed by self.SpinTime.
Full
Agitating
1. Find the related Valve Instance.
2. Generate Close to valve.
3. Find the related Motor instance.
4. Generate Agitate to motor.
5. Generate Done to self delayed by self.WashTime.
Empty
Emptying Rinse Water
1. Find the related Motor instance.
2. Generate Stop to motor.
3. Find the related Pump instance.
4. Generate Run to pump.
Done
Done
Emptying Wash Water
1. Find the related Motor instance.
2. Generate Stop to motor.
3. Find the related Pump instance.
4. Generate Run to pump.
Rinsing
1. Find the related Valve instance.
2. Generate Close to valve.
3. Find the related Motor instance.
4. Generate Agitate to motor.
5. Generate Done to self delayed by self.RinseTime.
Empty
Full
Filling For Rinse
1. Find the related Pump instance.
2. Generate Stop to pump.
3. Find the related Valve instance.
4. Generate Open to valve.

Figure 5. Washing Machine State Model

```
package require raloo
namespace import ::raloo::Domain

package require logger

Domain create washer {
    Class WashingMachine {
        Attribute {
            *WMId int
            WashTime int
            RinseTime int
            SpinTime int
        }
        Lifecycle {
            State Idle {} {
                set motor [my selectRelated ˜R2]
                $motor generate Stop
            }
            State FillingForWashing {} {
                set valve [my selectRelated ˜R3]
                $valve generate Open
            }
            State Agitating {} {
                set valve [my selectRelated ˜R3]
                $valve generate Close
                set motor [my selectRelated ˜R2]
                $motor generate Agitate
                my generateDelayed [expr {[my readAttr WashTime] * 1000}] Done
            }
            State EmptyingWashWater {} {
                set motor [my selectRelated ˜R2]
                $motor generate Stop
                set pump [my selectRelated ˜R1]
                $pump generate Run
            }
            State FillingForRinse {} {
                set pump [my selectRelated ˜R1]
                $pump generate Stop
                set valve [my selectRelated ˜R3]
                $valve generate Open
            }
            State Rinsing {} {
                set valve [my selectRelated ˜R3]
                $valve generate Close
                set motor [my selectRelated ˜R2]
                $motor generate Agitate
                my generateDelayed [expr {[my readAttr RinseTime] * 1000}] Done
            }
            State EmptyingRinseWater {} {
                set motor [my selectRelated ˜R2]
                $motor generate Stop
                set pump [my selectRelated ˜R1]
                $pump generate Run
            }
            State Spinning {} {
                set pump [my selectRelated ˜R1]
                $pump generate Stop
                set motor [my selectRelated ˜R2]
                $motor generate Spin
                my generateDelayed [expr {[my readAttr SpinTime] * 1000}] Done
            }
            Transition Idle - Run -> FillingForWashing
            Transition FillingForWashing - Full -> Agitating
            Transition Agitating - Done -> EmptyingWashWater
            Transition EmptyingWashWater - Empty -> FillingForRinse
            Transition FillingForRinse - Full -> Rinsing
            Transition Rinsing - Done -> EmptyingRinseWater
            Transition EmptyingRinseWater - Empty -> Spinning
            Transition Spinning - Done -> Idle
        }
    }
    Class Pump {
        Attribute {
            *WMId int
        }
        Lifecycle {
            State Off {} {
                log::debug "pump [my readAttr WMId] is off"
            }
            State Running {} {
                log::debug "pump [my readAttr WMId] is running"
                # To simulate the tub sensor, call the tub
                # empty after a second.
                [my selectRelated R1] generateDelayed 1000 Empty
            }
            Transition Off - Run -> Running
            Transition Running - Stop -> Off
        }
    }
    Class Motor {
        Attribute {
            *WMId int
        }
        Lifecycle {
            State Off {} {
                log::debug "motor [my readAttr WMId] is off"
            }
            State Agitating {} {
                log::debug "motor [my readAttr WMId] is agitating"
            }
            State Spinning {} {
                log::debug "motor [my readAttr WMId] is spinning"
            }
            Transition Off - Agitate -> Agitating
            Transition Agitating - Stop -> Off
            Transition Off - Spin -> Spinning
            Transition Spinning - Stop -> Off
        }
    }
    Class Valve {
        Attribute {
            *WMId int
        }
        Lifecycle {
            State Closed {} {
                log::debug "valve [my readAttr WMId] is closed"
            }
            State Open {} {
                log::debug "valve [my readAttr WMId] is open"
                # To simulate the tub sensor, call the tub
                # full after a second.
                [my selectRelated R3] generateDelayed 1000 Full
            }
            Transition Closed - Open -> Open
            Transition Open - Close -> Closed
        }
    }
    Relationship R1 Pump 1-->1 WashingMachine
    Relationship R2 Motor 1-->1 WashingMachine
    Relationship R3 Valve 1-->1 WashingMachine
    # A domain operation to kick things off
    DomainOp run {wmid} {
        set wm [WashingMachine selectOne WMId $wmid]
        if {[$wm isnotempty]} {
            $wm generate Run
        }
    }
}
# Populate the domain with one washer
washer transaction {
    # Set ridiculously short times for testing.
    WashingMachine insert WMId 1 WashTime 5 RinseTime 3 SpinTime 2
    Pump insert WMId 1
    Motor insert WMId 1
    Valve insert WMId 1
}
logger::initNamespace ::washer debug
# Run a cycle
washer run 1
# Enter the event loop to run the state machine.
vwait forever
```

Executing this code will yield:

```
[Sun Sep 14 16:39:38 PDT 2008] [washer] [debug] ’valve 1 is open’
[Sun Sep 14 16:39:39 PDT 2008] [washer] [debug] ’valve 1 is closed’
[Sun Sep 14 16:39:39 PDT 2008] [washer] [debug] ’motor 1 is agitating’
[Sun Sep 14 16:39:44 PDT 2008] [washer] [debug] ’motor 1 is off’
[Sun Sep 14 16:39:44 PDT 2008] [washer] [debug] ’pump 1 is running’
[Sun Sep 14 16:39:45 PDT 2008] [washer] [debug] ’pump 1 is off’
[Sun Sep 14 16:39:45 PDT 2008] [washer] [debug] ’valve 1 is open’
[Sun Sep 14 16:39:46 PDT 2008] [washer] [debug] ’valve 1 is closed’
[Sun Sep 14 16:39:46 PDT 2008] [washer] [debug] ’motor 1 is agitating’
[Sun Sep 14 16:39:49 PDT 2008] [washer] [debug] ’motor 1 is off’
[Sun Sep 14 16:39:49 PDT 2008] [washer] [debug] ’pump 1 is running’
[Sun Sep 14 16:39:50 PDT 2008] [washer] [debug] ’pump 1 is off’
[Sun Sep 14 16:39:50 PDT 2008] [washer] [debug] ’motor 1 is spinning’
[Sun Sep 14 16:39:52 PDT 2008] [washer] [debug] ’motor 1 is off’
```

The new construct here is given by the **Lifecycle** command. This command associates a state model with a class. The states are defined using a **State** command. Events in **Raloo** may carry parameters and those parameters are delivered to the state code as ordinary arguments. Hence the **State** command has the same interface as **proc**. The **Transition** command defines the transitions between states. The arguments to **Transition** give the current state, the event that causes the transition and the new state. Conceptually the code associated with the state is executed when the state is entered. The two arguments, - and -> are required syntactic sugar.

The state action code is executed as a TclOO method on an instance reference object. Instance reference objects are **Raloo** objects that reference one or more instances of a class (or equivalently one or more tuples of a relation variable). The idea of an instance reference is central to the way that **Raloo** merges the ideas of relations with object oriented techniques. Objects of **Raloo** classes are references to the tuples that are stored in a relvar that is associated with the class. The cardinality of the instance references may be empty, one or many with one being the most common case. There are a large number of methods available to instance reference objects. In this example, we focus on only two:

(1) Finding a related instance by traversing a relationship.

(2) Generating an event to an instance.

The **selectRelated** method will find a related instance by traversing a relationship. The **Idle** state uses the command

```
    set motor [my selectRelated ˜R2]
```

to find the instance of motor that is associated with the instance of WashingMachine that is executing the **Idle** action. The use of the tilde (˜) indicates that the relationship is traversed in the reverse direction and is required here since the class diagram gives the forward direction of R2 as from **Motor** to **WashingMachine**. A chain of traversals may be generated by simply giving additional relationship names to the **selectRelated** method.

The **generate** method causes an event to be generated to an instance. Thus, the **Idle** state after finding the related motor using the **selectRelated** method generates the **Stop** event to that motor using:

```
    $motor generate Stop
```

If the **Stop** event carried any parameters, then they would be listed as additional arguments to the **generate** method.

The state transitions that result from calls to the **generate** method are dispatched via the Tcl event loop. **Raloo** arranges for the event dispatch to happen as a callback script associated with an invocation of the **after** command. Consequently, **Raloo** applications must enter the Tcl event loop to cause the application to dispatch events, execute the state transitions and cause the state actions to perform the application processing. **Raloo** itself does not enter the event loop. For Tk based applications entering the event loop happens naturally, but for Tcl based applications invoking the **vwait** command is required to cause the application to run. Thus, **Raloo** applications are structured very much like Tk applications in the sense that after initialization, the remaining application is executed as callbacks driven by events that are generated from interactions with the outside world. This is very much in keeping with the Tcl event based style.

There are a few finer points worth noting here. Because **R2** is one-to-one and unconditional in its cardinality, we know that the **selectRelated** method must yield an instance reference object that refers to exactly one motor tuple.[13] Otherwise, the data would be inconsistent and **Raloo** would have thrown an error and rolled back any previous attempt to create such a population. So it is not necessary to test that the *motor* variable actually refers to anything. Given that, clearly the two statements of the **Idle** state could be written as a single statement with no **motor** variable being defined. Compare this with the **run** domain operation that is defined to start the running of a washing cycle. Here, the ID of the intended washing machine is given as an argument. The **selectOne** method will look up the **WashingMachine** that matches that ID. But it is possible to supply an ID that does not match any washing machine and therefore it is necessary to test if **selectOne** actually found any matching instance of **WashingMachine**. This is easily accomplished by the **isnotempty** method.

6. Relation to Formal Software Methods

The ideas implemented in **Raloo** are not new or original. **Raloo** is a Tcl implementation of the underlying execution semantics of a formal software methodology called, *Executable UML*. [14] [15] [16] [17] [18] This methodology was originally developed by Sally Shlaer and Stephen Mellor in the late 1980's to the early 1990's and at that time was known as the Shlaer-Mellor method. The methodology combined the concepts of the relational model of data with the previous work of Mellor and Ward[19] in using state machines and data flow diagrams to model system behavior. Over time, the graphical conventions of UML were adopted and to distinguish the particular characteristics of this approach, the name, *Executable UML*, was used. Executable UML is a proper profile of UML that includes what early forms of UML never had, namely, rigorous execution semantics.

From the beginning, Executable UML had a goal of specifying software systems in an implementation independent manner and translating[20] the resulting models into the actual implementation. Part of that translation process involves mapping the models onto a **Software Architecture**

[13] Mathematically, R2 is a bijective function from the Motor set to the WashingMachine set and therefore always invertible.

[14] Shlaer, Sally and Stephen J. Mellor, *Object Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, 1988, ISBN: 0-13-629023-X

[15] Shlaer, Sally and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1992, ISBN: 0-13-629940-7

[16] Mellor, Stephen J. and Marc J. Balcer, *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002, ISBN: 0-201-74804-5

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

[18] Starr, Leon, *How to Build Shlaer-Mellor Object Models*, Yourdon Press, 1996, ISBN: 0-13-207663-2

[19] Ward, Paul T. and Stephen J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, Vol 1, 1985, ISBN: 0-917072-51-0, Vol 2, 1985, ISBN: 0-917072-52-9, Vol 3, 1986, ISBN: 0-917072-53-7

[20] Indeed some practitioners use the name *Executable and Translatable UML* to distinguish the approach of mechanically generating the implementation since newer versions of UML do support a more rigorous set of execution semantics.

Domain. The software architecture is the domain that controls the policies and mechanisms for how data and execution are handled in the system. **Raloo** can be viewed as an Executable UML software architecture implemented in Tcl where the domain interfaces correspond directly to the execution semantics of Executable UML models.

But as we have already seen, **Raloo** can also be viewed as a Tcl object-oriented extension without any reference to formal software methods. The utility of **Raloo** as a programming approach derives from the coherent data structuring capability provided by a rigorous relational algebra with enforced referential integrity constraints, the support for declaring state models for asynchronous processing associated with class lifecycles and the use of object-oriented Tcl code both to declare the structure of the problem solution and to express the processing details of the algorithms.

7. Acknowledgements

The author wishes to acknowledge the help and guidance of his friends and colleagues that have suffered through more than their share of talking about **Raloo**. Special thanks go to Paul Higham for many coffee break discussions and for his clarity of thought and expression. Thanks also to Leon Starr for always keep the focus on understanding and solving problems rather than running computers.

Table of Contents

Table of Figures