Supporting Embedded Software Development

Doing the Heavy Lifting with Tcl/Tk

Andrew Mangogna
amangogna@modelrealization.com

17th Annual Tcl/Tk Conference
October 13-15, 2010

Copyright

Abstract

Software development on micro-controller based computing platforms is very challenging due to minimal computing resources and I/O capabilities. These micro-controller platforms are deployed widely in areas such as home automation, consumer electronics, automotive and medical devices. This paper discusses a set of software tools, implemented in Tcl / Tk, that support software code generation for small embedded micro-controller based systems. The core tool is a program called, **pycca**, which is a domain specific language for specifying data structures, data relationships and finite state machine behavior. The focus of the paper is not on the functionality of **pycca** but rather the underlying Tcl technology used to implement the tools. Although it is not possible to run a Tcl interpreter on small micro-controller based systems, Tcl is a valuable tool used to support the development and testing of this class of system. All the programs discussed are available as part of an open source project.

1. Introduction

This paper is a about a program called, **pycca**. **Pycca** is an acronym for Pass Your "C" Code Along[1]. **Pycca** is coded in Tcl and implements a domain specific language that supports generating "C" code that is useful in building software for highly embedded micro-controller based computing platforms.

The remainder of this section discusses the background and motivation behind **pycca**. **Pycca** is a supporting program for an execution architecture that is appropriate for many types of applications fielded on small computing platforms. The next section shows how **pycca** fits into the development workflow and shows a simple example.

The remainder of the paper is devoted to the internals of **pycca** highlighting the Tcl / Tk technology that is used in its implementation. Language parsing, internal data structures and semantic analysis are discussed. The use of relationally structured data is emphasized. The output from **pycca** is generated by template expansion and an example of the expansion is shown. Finally, other programs that use the parsed data that **pycca** can save are discussed. These programs provide another set of supporting functionality to help in debugging and visualizing the resulting software.

1.1. Resources

The software described in this paper is available as an open source project that is licensed in the same manner as Tcl / Tk itself[2].

1.2. An Execution Architecture for Embedded Systems

One very difficult aspect of developing software on a micro-controller is the *bare metal* aspect of running that software. There is often no operating system to protect against errant execution. Many times the computer architecture is so primitive that there is no notion of supervisory execution. It is important to keep strict control over the data and execution sequencing. To that end, a well defined execution architecture which controls all the essential aspects of program execution needs to be in place. The idea is to factor away from the application into a common code base all the policy decisions about how data is handled and how execution is sequenced.

One can view the execution architecture as a different model of computation that has a higher level of abstraction. Rather than the traditional view of sequential lines of code where control is transferred explicitly, the computation model is more along the lines seen in a conventional GUI program. An event loop, which is not generally seen by the programmer, handles the control sequencing as events are generated. Events are used to cause transitions in finite state machines. This is different from the usual event loop that simply maps events to a callback function. State machines can generate events to other state machines and one very powerful aspect of the execution model derives from the sequencing of interactions of what are usually small state machines.

Such an execution architecture is generally applicable to a range of application subject matters and is only dependent upon the demands the applications make on the computing environment rather than the details of the application semantics. This is an important distinction. The

---

[1] Rest assured that despite a name beginning with *py*, Python is not involved.

[2] http://tcl-cm3.sourceforge.net
Although this project is described as a set of tools for the ARM® Cortex-M3®, the scope is substantially larger than a single processor platform.

execution environment tailored for small systems is not universally suitable as it will scale to the computational requirements of the system only within certain bounds. However, it is not dependent upon the particular subject matter of any application.

1.3. STSA

We describe here a Single Threaded Software Architecture (STSA). As the name implies there is only a single thread of execution, although interrupts are allowed to preempt that single thread. This execution architecture encompasses all the policy decisions on how data is managed and how execution is sequenced.

Although the small memory footprint of micro-controller based platforms means the quantity of data in these types of systems is small, the data can be of a rather complex organization. For many of these applications, no system heap is used and all data is managed from fixed, worst case sized pools. Safety critical systems in particular avoid a global system pool since they tend to be long running[3] and the concern is that the heap will fragment over time and the system may not be able to allocate a critical data structure. Individual memory pools also offer easier allocation and management but, because the worst case allocation must be accommodated, sometimes do not give the most efficient memory usage.

The STSA also manages the the sequencing of execution. Although conventional multi-tasking operating environments are sometimes available, they do incur additional unwelcome overhead of context switching and frequently negatively affect interrupt service latencies. The usual requirements for small system applications can also be met by having a means of terminating a computation and then resuming that computation in response to some other happening. This can be done using finite state machines techniques that do not require keeping a virtual CPU context and switching between multiple contexts. Many applications for small systems function perfectly well using a single threaded, event driven approach to execution sequencing. This model of execution is analogous to the Tcl event loop with two significant differences.

- The STSA dispatches events to Moore type state machines.[4] By contrast, the Tcl event loop binds arbitrary Tcl scripts to the event. STSA has a much tighter set of rules for how execution is sequenced and supports the execution semantics of Executable UML.[5]
- The STSA binds the events to application functionality at link time whereas Tcl, naturally enough, accomplishes this binding at run time. STSA accomplishes the binding by being completely data driven, requiring a set of data structures be provided by the application. This has two advantages that are significant in small systems. First, it eliminates a large fraction of initialization code. For long running systems initialization code is run only once yet still occupies valuable memory space. Second, the paths through the state space of the code are fixed giving much tighter control over the system execution. Indeed, the entire state space of the application is determined before run time. This is considered a substantial benefit since tight control is more desirable than run-time flexibility for these types of systems.

---

[3] An example of a long running system is one that is battery powered, begins to run when the battery is installed and continues to do so until the battery fails to provide enough energy.

[4] http://en.wikipedia.org/wiki/Moore_machine

[5] http://en.wikipedia.org/wiki/Executable_UML

See also:

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

and

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

The arguments usually used for event driven versus thread based programming in Tcl apply to the small system context as well. Preemptive multi-tasking with shared state in a small system carries with it the same difficulties that it does in larger systems, namely, the extreme, if not impossible, difficulty of insuring that there is no critical section violation of the shared state information. Such violations create small timing based windows where errors can occur and such errors are very difficult to reproduce and therefore correct. Event driven, single threaded execution models eliminate many of these potential errors *by design*. Of course, not all applications have computational demands that can be easily satisfied by and event driven approach. If the worst case computation time for a particular thread of control is longer than the response latency time required of the application, then some way to defer the long running computation must be devised. But most often the applications running on small embedded targets are usually completely driven by responses to happenings in the environment in which they reside and the computation of those responses is well within the overall response latency time requirements of the system. This is especially true given that interrupts can be used to handle most of the hard real time response requirements.

An application is bound to the STSA execution environment at link time and STSA is completely data driven. Applications supply instances of a well defined data structure (as an initialized "C" variable) as part of the interface to STSA and that data structure contains all the information STSA needs to manage data memory and execution sequencing. This arrangement has several advantages such as the tight and early binding of application functionality and the fact that most of the data required by STSA is constant and therefore can be placed in read-only memory which is often more available than RAM. The disadvantage is that the data that must be supplied to STSA is tedious to specify and arrange into "C" variables. For example, part of the data required by STSA is the state transition matrix for the state machines. In the case of the Moore type machines that are supported by STSA, the transition matrix is usually implemented as a two dimensional array indexed by a state number and event number with each entry containing the new state for the transition. Although very convenient and efficient for the implementation, it is tedious and error prone to manually encode states and events into sequential integers that are suitable for use as indices into the transition matrix. This is especially the case in the face of changes in the state machine graph that inevitably occur during development.

Pycca was written specifically to help this problem. **Pycca** accepts declarations of data structures and data relationships and of state machines and other procedural operations. All algorithmic actions are specified in "C" which is wrapped as a function and ordered in the output to match the "C" compiler's needs, but is otherwise passed through unchanged. What **pycca** generates from the specification data are the data structures required by STSA to bind together the state transitions with the passed through "C" code producing a single code file and a single header file. Hence the name, **pycca**, an acronym for Pass Your "C" Code Along.

1.4. STSA Concepts

Below we list the *concepts* behind STSA. This will define terms that are further discussed below. It is beyond the scope of this paper to cover each of these concepts in detail, but it is sufficient to say that **pycca** allows each concept to be specified to the STSA.

(1) Software is organized into Domains. Domains represent a coherent subject matter. Domains interface to each other by a set of Domain Operations. All other aspects of a domain are not visible outside of the domain.

(2) Domains consist of a set of Classes. Classes, in turn, consist of attributes. Classes hold the data of a Domain. The set of Classes of the Domain represents the time invariant aspect of the Domain semantics as they are encoded into data.

(3) Classes may have relationships between them that model the subject matter associations that class instances have.

(4) A Class may be instantiated an arbitrary number of times. Each Instance of a Class has the same data and behavior.

(5) An Instance may be created in one of three ways.
(a) An initial instance. Initial instances come into existence, conceptually, when the system is started.
(b) Synchronous creation. One instance may request another instance to be created as part of its behavior.
(c) Asynchronous creation. One Instance may request that a Creation Event be posted, that when dispatched will result in an Class Instance being created.

(6) A Class may define Class Operations. Class Operations are common processing associated to a particular Class.

(7) A Class may define Instance Operations. Instance Operations are common processing defined by a Class and implicitly associated with a particular Instance of the Class. Instance Operations perform the same operations on all Instances of the Class.

(8) A Class may define a State Model. The state model is a Moore state machine. Every Instance of such a Class has its own implicit state variable and Instances change state independently.

(9) Each state of the State Model may contain an Action. The Action of a state may execute arbitrary processing.

(10) An Instance transitions from its current state to another state (possibly the same as the current state) as a result of receiving an Event. Events may carry additional parameters and these values are available to the Action that is executing when the transition occurs.

(11) Instances may send Events to other Instances including themselves. Any Event sent by an Instance to itself is received before any Event sent by a different Instance. Events are not lost and the order of Events sent by an Instance to a particular receiving Instance is preserved. However, the sender of an Event does not know when the Event is actually dispatched.

(12) Instances may request that an Event be sent after some delay. No more than one Delayed Event of a given type may be outstanding between a given source Instance and a given target Instance (where source and target may be the same Instance). This has same effect as sending the Event, except that STSA manages the timing. Delayed events may be canceled and it is possible to inquire as to the amount of delay remaining for a particular delayed event.

(13) For one particular type of relationship, a Class (the super-type) may be completely partitioned into disjoint sub-Classes (the subtypes). In this case the super-type Class may define state machine events that are re-mapped at run-time into a corresponding Event of the subtype to which the super-type is currently related. Such an event is called a Polymorphic Event.

(14) An Instance of a Class may be deleted in one of two ways:
(a) Synchronous deletion One instance may request that another instance is deleted.
(b) Asynchronous deletion A state of a State Model may be designated as a Final state. Instances entering such a state are deleted automatically at the end of the execution of the final state Action.

2. Pycca

Pycca is a domain specific language. The domain of interest here is specifying data structures, relationships between data and lifecycles of processing modeled as Moore type finite state machines. The semantics are those implemented by STSA as described above. **Pycca** can be thought of as a partial code generator for STSA based applications. In truth, **pycca** generates no

code other than trivial instrumentation and functions wrappers. The hard work of expressing computation is accomplished in "C" with the "C" code being passed through to a compiler. Consequently, **pycca** is very much a declarative language since it understands nothing about the "C" code that is passed along. This outlook goes well with focus of **pycca** generated data structures required by STSA.

2.1. Example

For the purposes of this paper, we will use a simple running example. This example is about controlling a very simple automatic washing machine. Undoubtedly real washing machines are not controlled in the manner implied by this example and any reader who knows how washing machines actually are controlled would probably be horrified by this example. It is not the purpose here to build a real washing machine control system. The purpose here is to choose a subject matter that most people will know something about and avoid a long explanation of the subject matter itself.

With that caveat, the figure below shows the class diagram of our washing machine.

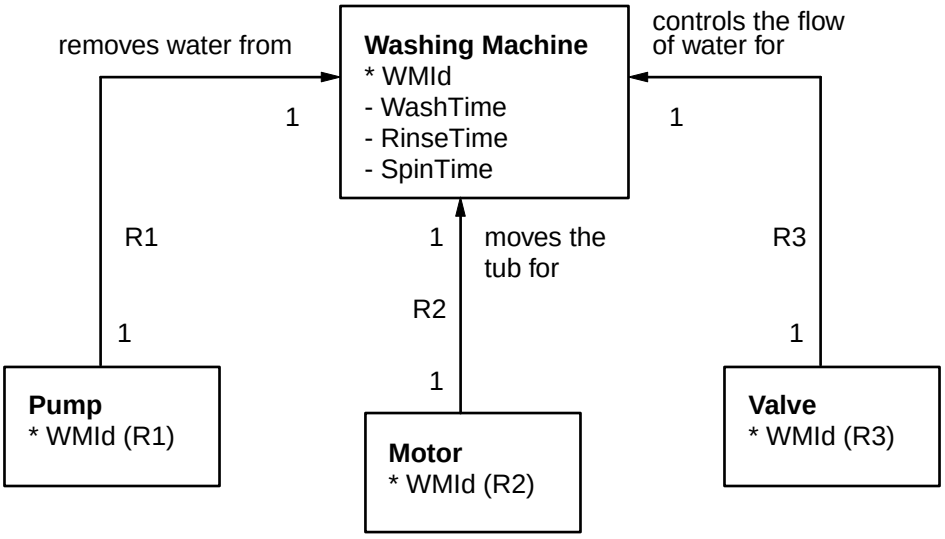

Figure 1. Washing Machine Class Diagram

The details of the graphical notation are explained below. For now, it is sufficient to view this as a picture of a relational schema and a set of referential integrity constraints.

One particular class in the diagram also exhibits interesting life-cycle behavior which is modeled as a state machine. That figure is shown below.

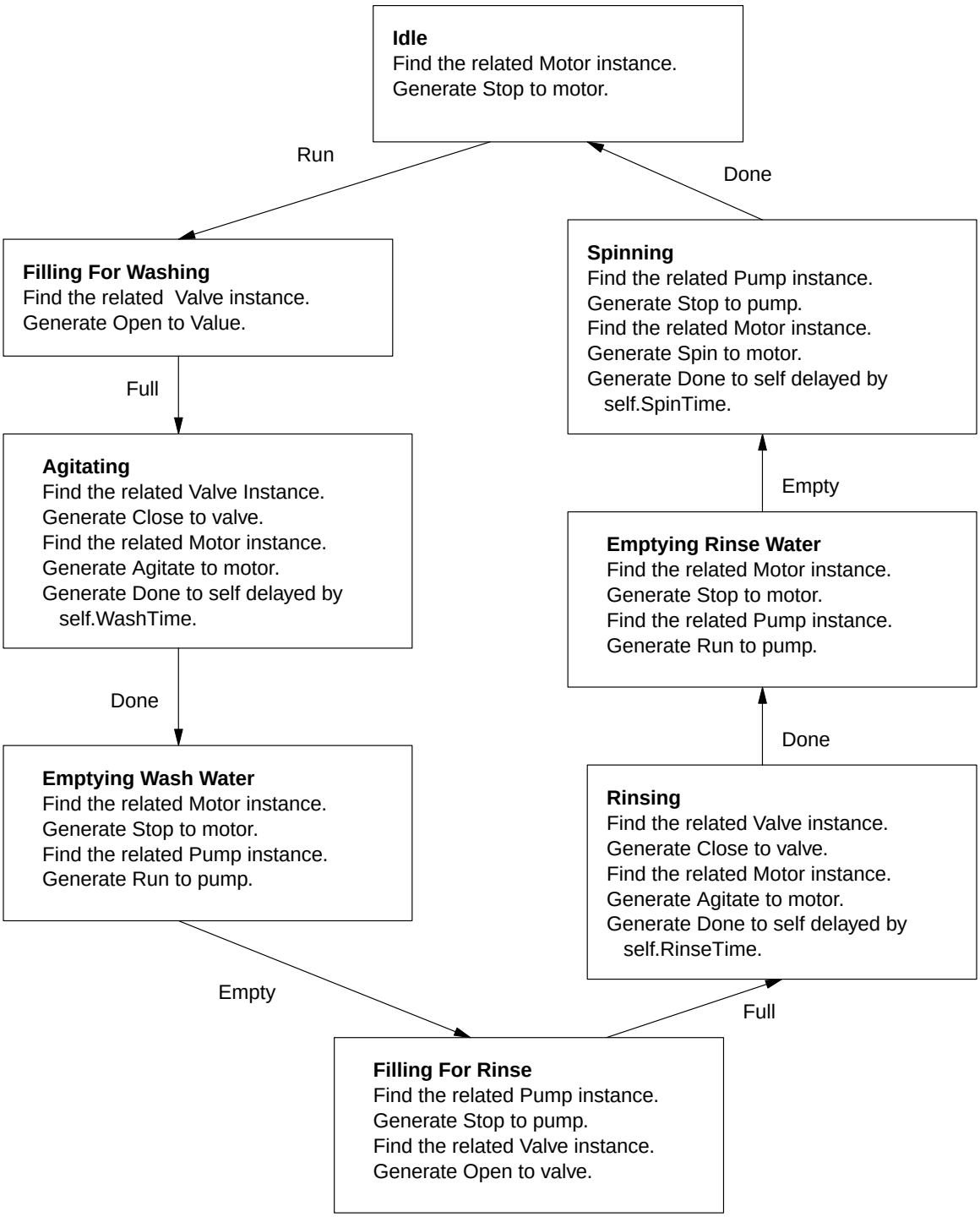

Figure 2. State Model for Washing Machine Class

In this diagram, each rectangle is a state and transitions between states are represented by a directed line segment labeled with the name of the event that causes the transition. This state machine is in the Moore formulation of state models which means that the action of the state is

performed when the state is entered. So, when in machine is in the **Idle** state, receiving the **Run** event will cause a transition to the **Filling For Washing** state and will perform the logic contained in the rectangle of that state.

2.2. Example Pycca Source

As is usually the case, even small, simple examples often end up needing more source code than is convenient to include in a paper of this type. So we will include only part of the **pycca** source for the example. Here we show the Washing Machine class definition. This should be sufficient for you to get the general idea.

```
class WashingMachine
    attribute (unsigned WashTime) default {10}
    attribute (unsigned RinseTime) default {10}
    attribute (unsigned SpinTime) default {10}
    reference R1 -> Pump
    reference R2 -> Motor
    reference R3 -> Valve

    machine
        state Idle() {
        //# Find the related Motor instance.
        //# Generate Stop to motor.

            PYCCA_generate(Stop, Motor, self->R2, self) ;
        }
        transition Idle - Run -> FillingForWashing

        state FillingForWashing() {
        //# Find the related Valve instance.
        //# Generate Open to Value.

            PYCCA_generate(Open, Valve, self->R3, self) ;
        }
        transition FillingForWashing - Full -> Agitating

        state Agitating () {
        //# Find the related Valve Instance.
        //# Generate Close to valve.
        //# Find the related Motor instance.
        //# Generate Agitate to motor.
        //# Generate Done to self delayed by self.WashTime.

            PYCCA_generate(Close, Valve, self->R3, self) ;
            PYCCA_generate(Agitate, Motor, self->R2, self) ;
            PYCCA_generateDelayedToSelf(Done, self->WashTime) ;
        }
        transition Agitating - Done -> EmptyingWashWater

        state EmptyingWashWater () {
        //# Find the related Motor instance.
        //# Generate Stop to motor.
        //# Find the related Pump instance.
        //# Generate Run to pump.

            PYCCA_generate(Stop, Motor, self->R2, self) ;
            PYCCA_generate(Run, Pump, self->R1, self) ;
        }
        transition EmptyingWashWater - Empty -> FillingForRinse

        state FillingForRinse() {
        //# Find the related Pump instance.
        //# Generate Stop to pump.
        //# Find the related Valve instance.
```

```
        //# Generate Open to valve.

            PYCCA_generate(Stop, Pump, self->R1, self) ;
            PYCCA_generate(Open, Valve, self->R3, self) ;
        }
        transition FillingForRinse - Full -> Rinsing

        state Rinsing() {
        //# Find the related Valve instance.
        //# Generate Close to valve.
        //# Find the related Motor instance.
        //# Generate Agitate to motor.
        //# Generate Done to self delayed by self.RinseTime.

            PYCCA_generate(Close, Valve, self->R3, self) ;
            PYCCA_generate(Agitate, Motor, self->R2, self) ;
            PYCCA_generateDelayedToSelf(Done, self->RinseTime) ;
        }
        transition Rinsing - Done -> EmptyingRinseWater

        state EmptyingRinseWater() {
        //# Find the related Motor instance.
        //# Generate Stop to motor.
        //# Find the related Pump instance.
        //# Generate Run to pump.

            PYCCA_generate(Stop, Motor, self->R2, self) ;
            PYCCA_generate(Run, Pump, self->R1, self) ;
        }
        transition EmptyingRinseWater - Empty -> Spinning

        state Spinning() {
        //# Find the related Pump instance.
        //# Generate Stop to pump.
        //# Find the related Motor instance.
        //# Generate Spin to motor.
        //# Generate Done to self delayed by self.SpinTime.

            PYCCA_generate(Stop, Pump, self->R1, self) ;
            PYCCA_generate(Spin, Motor, self->R2, self) ;
            PYCCA_generateDelayedToSelf(Done, self->SpinTime) ;
        }
        transition Spinning - Done -> Idle
    end
end
```

Figure 3. Pycca Source for Washing Machine Class

Compare the source text above to the graphic of Figure 1 and the state graph of Figure 2. There are three attributes of the Washing Machine class, corresponding to the times that are used in controlling the cycle. There are three references to the Pump, Motor and Valve classes. The full text of the source has definitions for those classes. The state machine definition includes

definitions for each state and the transitions between states.

It is worth noting that there is no order dependency in **pycca**. Classes may be defined in any order. State machine definitions are also order independent in the sense that you may define all the states or all the transitions and in any order. The style of the example is to define states and follow them by their outbound transitions, but that is strictly a coding convention and not a requirement of **pycca** syntax. Note that there is no separate definition of the events, they are gleaned from the **transition** statements. The body of the state definitions is "C" language code. In the above example it consist entirely of macros that are used to generate events. **Pycca** inserts a set of "C" pre-processor macros into the output that are very useful for hiding the naming conventions and interfacing with the STSA execution architecture. Indeed the "C" code for a **pycca** state machine is often very stylized since the macros provide the interface to a large part of the ordinary processing that state actions perform. Clearly, this particular example is dominated by state behavior, but other types of applications would have more of what would be considered normal "C" statements.

2.3. Pycca Workflow

The figure below shows the general workflow when using **pycca**.

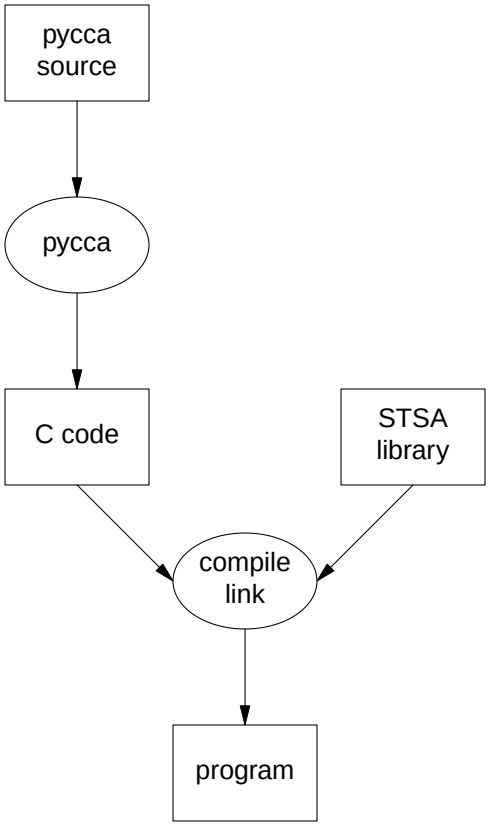

Figure 4. Building a Program with pycca

Pycca source is processed by **pycca** to produce a single "C" language code file and a single "C" header file. **Pycca** can process several source files together as if they were one file. The "C" output must be compiled and linked against the STSA library. Of course, other code files may be

linked into the program and the program may have many **pycca** generated files. **Pycca** is able to insert line numbers into the generated code file that references the original source or to leave out those numbers if they should confuse your debugger.

In general, the difficulties of working with **pycca** come more from the translation of the design ideas into source. From there on, the workflow is quite traditional.

3. Pycca Internals

Up to now, there has been a lot of background and not much Tcl. However, **pycca** is implemented in Tcl. Now we will consider the design of the internals of **pycca** and attempt to highlight the Tcl technology that was used.

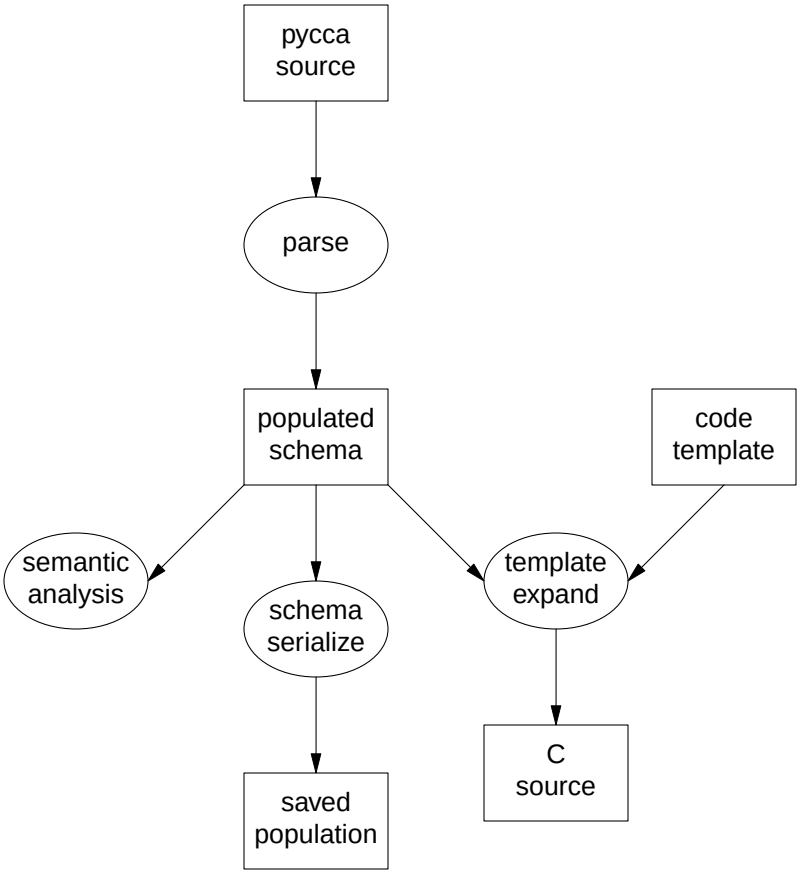

Figure 5. Pycca Internal Data Flow

Figure 5 above shows a high level view of the design of the internals of **pycca**. The **pycca** source is first parsed. The parsed components are stored in a normalized relational schema. Like most language processing programs, after the syntax is verified, it is necessary to do some semantic analysis to insure that the language statements are meaningful. Unlike most computer languages, **pycca** is not expression oriented. Rather the **pycca** language itself is declarative in nature. Algorithmic computation is left to "C" code statements that are passed along to a compiler. This means that the generation of output is amenable to using template expansion techniques. **Pycca** expands a template by querying the data from the populated schema to produce the desired "C" code file. Optionally, **pycca** can save the populated schema by serializing it. We

will find that useful later.

In the following sections we will examine the details of each of these steps, again looking at the Tcl technology used to implement them. What we will see is that **pycca** looks very much like a traditional database application and less like a language compiler than might initially be recognized.

3.1. Parsing Techniques

There are many options in Tcl for parsing domain specific languages. The usual advice to those with less experience in Tcl is use the Tcl parser itself to parse a language. Being a command language with a very simple syntax, Tcl is well suited to apply to domain specific languages. The language syntax can be defined in terms of Tcl commands with parameters that pass the required information. It is relatively easy to support a hierarchy of definitions by evaluating a script body in a namespace that exposes a limited set of commands suitable to the context. However, exposing some Tcl syntax is unavoidable with this approach and that may not be acceptable in all circumstances.

In the case of **pycca** it was decided that a more conventional syntax would be better for the target audience. **Fickle** and **tackle** are Tcl analogs to the venerable lex / yacc (or flex / bison) lexical analyzer and parser generators. They use an input file syntax that very close to that of lex and yacc and generate a lexical analyzer or parser in Tcl code. Unfortunately, the generated parser is not as robust as one might desire. Detection of language grammar issues is very limited. A number of small problems with the **tackle** code itself were found. In the end, the generated parser is sufficient for the purposes, but a future version of **pycca** will probably return to using Tcl syntax and the Tcl parser.

3.2. Parsing Results

In conventional language processing programs, as language statements are recognized the parsed components are used to build symbol tables and syntax trees. This is especially true of languages that are generating code for statements and expressions. Since the usual infix notation provides a convenient way to specify a tree in a linear fashion, grammar reductions are usually used to construct an expression as a syntax tree that is much easier to manipulate programmatically.

However, the **pycca** language is not used for code generation, *per se*. As we discussed above, what **pycca** does is to pass along "C" code, somewhat repackaged and reorganized, and to generate data structures that are used as the interface to the execution architecture. Symbol tables and syntax trees do not help in this type of processing. Rather, what **pycca** does is to populate a normalized relational schema. This relational schema constitutes a *meta-model* of the domain for which **pycca** is the domain specific language.

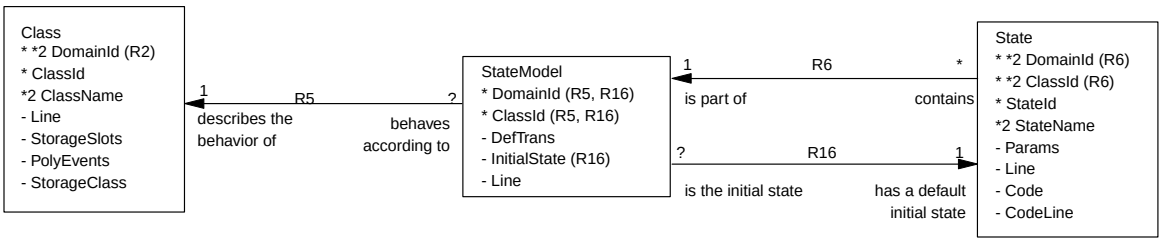

Figure 6. Part of the pycca Meta-model

Figure 6 above shows a graphical representation of very small part of the relational schema that constitutes the **pycca** meta-model. This is only a small part of the schema that we will use for an example and the full schema graphic is available with the project source. In this graphic, rectangles represent relation variables with their attributes given in a list. Attributes that begin with an asterisk (*) are identifiers. If there are multiple identifiers, the asterisk is followed by a number, *e.g.* *2. An identifier may consists of several attributes and a given attribute may be part of multiple identifiers. All relation variables have at least one identifier and all tuples of the relation must be unique in the values of all identifiers. Thus relation variables hold sets. Attributes that have a leading hyphen (-) are descriptive attributes that define a variable in the predicate that the relation variable intends to model.

Arrows in the diagram indicate a relationship between the relation variables. Relationships are implemented via referential attributes and constraints on the values of those referential attributes, *i.e.* attributes in one relation variable whose values must match the values of attributes in another relation variable. The direction of the relationship is from the referring attributes to the referred-to attributes. Relationships are given a moniker, *e.g.* **R16**, and attributes that are used to implement that relationship have a parenthesized list of relationship monikers following their name. Relationships are functions or partial functions between the participating relation variable sets. The cardinality of the relationship is also given by a single character that is mnemonic of the use of that character in regular expression notation. So, **1** means exactly one, **?** means at most one, **+** means one or more and ***** means zero or more. The relationship is traditionally annotated with the semantic meaning of the set mapping implied by the relationship function.

The graphic describes a relational schema that in turn describes a set of logical predicates that we insist be true. Using the **State** class as an example, we can make the following statements:

- A **State** is identified by the Domain in which it resides (DomainId), the Class to which it belongs (ClassId) and an arbitrary number (StateId).
- A **State** is also identified by the Domain in which it resides (DomainId), the Class to which it belongs (ClassId) and its name (StateName).
- A **State** is characterized by a set of parameters (Params), the number of the line in the pycca file on which the State definition begins (Line), a body of "C" code (Code) and the line on which the state code first appears (CodeLine).

Considering the relationship, R16, we can say:

- A **StateModel** *has* ***exactly one*** *default initial* **State**.
- A **State** *is the default initial state for* ***at most one*** **StateModel**.

All of these statements define rules that must be true for the domain for which **pycca** is the domain specific language. The actual number of rules is quite large. The **pycca** domain is a bit abstract since it is involved with defining classes and state models, but does have a well defined set of semantics and there is a well defined set of rules that is captured by the relational schema.

3.3. Storing the Parsing Results

As we stated above, the parsing results of **pycca** are not stored in the conventional manner of symbol tables and syntax trees. Rather, it is stored as set of relation variables. There are many ways in Tcl to store relational information. One obvious choice is to use a Database Management System (DBMS). Most DBMS are programmed in some dialect of SQL and with careful SQL programming the relational qualities of the meta-model may be preserved.

Another alternative is to use **TclRAL**. TclRAL is a "C" based Tcl extension that provides a complete relational algebra. It does this by extending the internal type system of Tcl to define the new types of *Tuple* and *Relation* and adds a new shadow variable system specifically to hold relation values. TclRAL is analogous to a data base system that only has in-memory tables and is programmed in Tcl rather than SQL. It is convenient to use TclRAL as we do not have to deal with other languages like SQL, there is no *impedance mismatch* between the query language and the programming language as they are both Tcl, it interacts well with other Tcl constructs such as the existing Tcl data types, interpreters, threads and starpacks and the amount of data that **pycca** handles easily fits within program memory.

Here we present the commands to create the relation variables that correspond to the graphic above (here we assume that `::ral::*` has been imported into the namespace).

```
relvar create Class {
    DomainId int
    ClassId int
    ClassName string
    Line int
    StorageSlots int
    PolyEvents list
    StorageClass string
} {DomainId ClassId} {DomainId ClassName}

relvar create StateModel {
    DomainId int
    ClassId int
    DefTrans string
    InitialState string
    Line int
} {DomainId ClassId}

relvar association R5\
    StateModel {DomainId ClassId} ?\
    Class {DomainId ClassId} 1

relvar create State {
    DomainId int
    ClassId int
    StateId int
    StateName string
    Params list
    Line int
    Code string
    CodeLine int
} {DomainId ClassId StateId} {DomainId ClassId StateName}

relvar association R6\
    State {DomainId ClassId} *\
    StateModel {DomainId ClassId} 1

relvar association R16\
    StateModel {DomainId ClassId InitialState} ?\
    State {DomainId ClassId StateName} 1
```

Figure 7. Pycca Meta-model Expressed in TclRAL

Figure 7 shows the translation of the above graphic into TclRAL. The close correspondence between the graphics and the TclRAL script is by design. The `relvar create` command creates a relation variable that can hold a relation value of the type given by the heading. The heading consists of both attribute names and data types, the data type names being any valid Tcl data type. The identifiers of the relvar are given as additional arguments that are lists of attribute names that constitute each identifier.

It may seem unusual to have to specify type information in a language like Tcl. This is actually a data validation constraint. TclRAL will insist that any value stored in an attribute be able to be

coerced into the type of that attribute. This avoids any problems in future usage of the attribute. Since `string` is the universal Tcl type, any attribute declared as a string type will accept any value.

The code segment below is taken from the `tackle` source of the **pycca** parser and shows the action taken when a state is defined and when an initial state is declared.

```
machineProp
    :   STATE NAME varList CODE {
            newState [dict get $2 value] $3 [dict get $1 line]\
                [dict get $4 value] [dict get $4 line]
        }
    |   INITIAL STATE NAME {
            setInitialState [dict get $3 value]
        }
    •••
    ;
```

Figure 8. Grammar Reduction for a State

As we can see from above, when a state declaration is recognized then the procedure `newState` is invoked. That procedure is shown below.

```
proc ::newState {name params line code codeline} {
    catchDuplicate {
        relvar insert State [list\
            DomainId $::DomainId\
            ClassId $::ClassId\
            StateName $name\
            Params $params\
            Line $line\
            Code $code\
            CodeLine $codeline\
        ]
    } $name
}
```

Figure 9. Relvar Population when a State is Recognized

The `newState` procedure simply performs an insert into the State relvar (the `catchDuplicate` procedure is a control structure that emits a friendlier error message should the state already be defined).

Similarly, defining an initial state results in updating the appropriate attribute in the StateModel relvar as shown below.

```
proc ::setInitialState {state} {
    relvar updateone StateModel sm\
            [list DomainId $::DomainId ClassId $::ClassId] {
        tuple update $sm InitialState $state
    }
}
```

Figure 10. Relvar Population when an Initial State is Specified

In this procedure the global variables `::DomainId` and `::ClassId` are set by other grammar reductions to provide a context for statements that are organized hierarchically in a particular domain or class. The body of the procedure simply updates the InitialState attribute of a single StateModel tuple.

The above is intended to illustrate the point of how **pycca** stores the results of parsing by populating a relational schema. Viewed another way, the **pycca** language is then simply a more convenient textual representation for the schema population. The schema could have been populated, like any other relational schema, by directly providing tuples to be inserted into the relation variables. Although applicable to any schema, such population methods require knowledge of the relation variables in the schema and such generic ways of populating the schema are devoid of the semantics of the problem that the domain specific language is intended to help.

As we shall see, populating a relational schema provides a very powerful method of manipulating and querying the data. We now begin to exploit that property of well organized data.

3.4. Semantic Analysis

Every computer language allows many statements that are syntactically correct but otherwise meaningless. In this section we examine how **pycca** evaluates its input to determine if all the language statements can be given a precise meaning.

3.5. Semantics Enforced by Referential Integrity

As we saw above, **pycca** uses a relational schema to store the parsed data. That schema has many integrity constraints associated with it. Previously we had looked at one of those constraints that defined the relationship between a StateModel and a State that was specified to be the default initial state. For R16 we said that:

- A **StateModel** *has* ***exactly one*** *default initial* **State**.
- A **State** *is the default initial state for* ***at most one*** **StateModel**.

So at the end of populating the schema, R16 will insure that somewhere along the way exactly one state was specified to be the default initial state and that the state is part of the StateModel being specified. TclRAL will enforce the association constraint that was defined as R16 to insure that the data entry is performed correctly.

That's really rather nice. For no extra coding, we used referential integrity constraints to enforce the semantics of our underlying meta-model. This is one of the reasons integrity constraints are so important. Other than specifying the constraint to TclRAL, we have no other coding to do and the **system** (*i.e.* TclRAL) will insure that state of the stored data always satisfies the constraint.

This naturally brings up the question of whether or not all the semantic analysis can be accomplished this way. That is an open question. Certainly, **pycca** does not accomplish all the semantic analysis by using relvar constraints. The underlying meta-model used by **pycca** is not that sophisticated containing only 18 relvar constraints. There is a certain trade-off between the

complexity of the data model (and consequently the amount of the semantics that are encoded in the referential integrity constraints) and the simplicity of doing some semantic analysis in code. **Pycca** uses both techniques as we will see below. Also there seem to be some problem semantics that are difficult or impossible to compose entirely into data constraints. In particular, constraining a directed graph to be acyclic may be impossible to do with referential integrity constraints alone.[6]

3.6. Semantics Enforced by Schema Query

In addition to referential integrity constraints, **pycca** also verifies 18 different semantic aspects of the source by querying the schema and examining the results. Here we look at one example, the declaration of isolated states. It is possible to declare a State in a StateModel that has no inbound state transition and no outbound state transitions. Such a state cannot be reached and therefore any code that is associated with that state will never be executed. This is the state machine equivalent of *dead code*. **Pycca** considers it an error to declare an isolated state in a state machine.

```
# Find isolated states, i.e. states that have no outgoing or
# incoming transitions
set noIns [relation semiminus $::Transition $::State\
        -using {DomainId DomainId ClassId ClassId NewState StateName}]
set noOuts [relation semiminus $::NormalTrans $::State\
        -using {DomainId DomainId ClassId ClassId StateName StateName}]
set isoStates [relation intersect $noIns $noOuts]
relation foreach isolated $isoStates {
    relation assign $isolated Line StateName
    reporterror "state has no incoming or outgoing transitions"\
        $Line $StateName
}
```

Figure 11. Finding Isolated States

The query in Figure 11 above shows how isolated states are detected. The strategy is to find all the states that have no incoming transitions and all the states that have no outgoing transitions. Then simple set intersection of those two relations then gives the set of isolated states for which an error needs to be reported.

Finding the set of states without any incoming transitions is done with the *semiminus* operation. Semiminus gives, roughly speaking, the set of tuples in one relation that are *not* related to those in another relation. In the above example, we find those tuples of State that are *not* related to a Transition using the NewState attribute. The NewState attribute gives the destination of a transition, so those States whose StateName values are not associated with the NewState attribute have no incoming transitions. Similar reasoning holds for the semiminus operation with respect to the NormalTrans relvar where the StateName attribute holds the source state of a transition. Once those two sets are determined, it is then just a simple matter of taking the intersection and iterating through the tuples to report the errors.

What is most noticeable about the above code sequence is the lack of any looping construct. Relational algebraic operation always apply the the entire set represented by the relation value. The only loop is that implied by the `relation foreach` command which is used to access the

[6] Private communications from Paul Higham for whom this is a favorite problem.

individual attributes for error reporting. It is common in the relational realm only to use tuple iteration at the end of the computation where it is necessary to interface to other non-relational constructs.

3.7. Template Expansion

Since the primary purpose of **pycca** is to pass along "C" code as opposed to actually compiling a source language to a target language, it does not use the usual language processing strategy of building syntax trees and flow graphs. The bulk of the work is building data structures, encapsulating code fragments and then emitting "C" code and header files. Since "C" source code requires substantial type annotation and "C" compilers insist upon a certain ordering in the code, **pycca** uses template expansion as a means of generating its output. The **textutil::expander** package in **tcllib** is well suited to this purpose. Using the expander is simple enough. You define a template, generate an instance of the expander and request the expander to do its job. Embedded in the template are Tcl scripts that are extracted and executed with the result being inserted into the generated text.

The example below is the template that **pycca** uses to emit its code file. We have instantiated the expander as:

```
textutil::expander ex
ex setbrackets %< >%
```

The default of square brackets ([]) for enclosing the embedded Tcl script is not convenient for "C" code. Below is the template string for the generated "C" code file. It is generated by invoking:

```
ex expand $code_body
```

```
set code_body {
%<domain_name>%
%<instrument_define>%
%<emit_auxiliary Impl Prolog>%

#ifdef INSTRUMENT
#   ifndef INSTR_FUNC
#       define INSTR_FUNC(s) printf("%s: %s %d0, s, __FILE__, __LINE__)
#   endif /* INSTR_FUNC */
#endif /* INSTRUMENT */

%<state_defines>%
%<event_defines>%
%<subcode_defines>%
%<inst_name_defines>%
%<class_struct_decl>%
%<class_struct_define>%
%<event_param_struct>%

%<operation_decl>%
%<ctor_decl>%
%<dtor_decl>%
%<action_decl>%
%<domain_op_decl false>%
%<initial_inst_ctor_decl>%
%<storage_decl>%

%<iab_define>%
%<odb_define>%
%<pdb_define>%
%<class_define>%

%<multi_refs>%
%<inst_define>%
%<operation_define>%
%<ctor_define>%
%<dtor_define>%
%<action_define>%
%<domain_op_define>%
%<initial_inst_ctor_define>%

%<emit_auxiliary Impl Epilog>%
}
```

Figure 12. Pycca Code Template

Each command between the %< and >% brackets is a query on the relational schema that was populated during the parsing phase.

As an example, we will consider part of what happens to the code of a state action. Each action is turned into a function by wrapping a signature around the code. These functions are file static in scope and, since the order of reference is not known, it is necessary to place forward

declarations to them into the code. This is accomplished by the `action_decl` command.

Before we examine the query for the action declarations, we need some preliminaries. We'd like to emit the declarations on a class by class basis, so it is useful to be able to have a relation value that relates the name of a class to all of its state machine information. The `getStatesByClass` command does this.

```
proc ::getStatesByClass {} {
    return [pipe {
        relation semijoin $::currentDomain $::State\
                -using {DomainId DomainId} |
        relation group ~ States StateId StateName Params Line Code CodeLine |
        relation join ~ $::Class -using {DomainId DomainId ClassId ClassId} |
        relation project ~ ClassName States
    }]
}
```

Lets examine this query. First the `pipe` command is a control structure that allows you to write a nested set of commands in a sequential fashion. It has nothing to do with relational algebra *per se*, although it is particularly useful in that context. In the above example, `pipe` allows us to write the four commands of the query is a clearer order. Commands are separated by a vertical bar (|) and the result of the previous command is substituted for the tilde (~) character argument in the subsequent command. `Pipe` is supplied as part of the **ralutil** package.

There are four steps in this query.

(1) Find all the states in the current domain. The **relation semijoin** operation returns a relation that has the same heading as the State relvar with those tuples that are related to the current domain. The `::currentDomain` variable is an ordinary Tcl variable that is global in scope and assigned a relation value by other processing. The heading of the State relation is:

DomainId	ClassId	StateId	StateName	Params	Line	Code	CodeLine
int	int	int	string	list	int	string	int

The result of this operation is the set of all states in the current domain.

(2) The **relation group** operation creates a relation valued attribute from attributes in the original relation. In this case we are grouping all the attributes that pertain to the states of a class into a single attribute. The result will have a new attribute named **States** that is a relation valued attribute[7]. The resulting heading is:

DomainId	ClassId	States					
int	int	StateId	StateName	Params	Line	Code	CodeLine
		int	string	list	int	string	int

(3) The **relation join** operation is used to relate the ClassId attribute to is corresponding name. In this case we are interested in the name of the class rather than its numeric identifier that is used internally.

(4) Finally, the **relation project** operation gives us only the attributes that we are interested in, namely the ClassName and its corresponding set of states. Note that the States

[7] Readers steeped in SQL will find relation valued attributes unusual, but they have many legitimate uses as well as a host of abusive uses.

attribute is a relation itself and contains the complete set of states that correspond to ClassName. The heading of the result relation is:

ClassName	States					
string	StateId	StateName	Params	Line	Code	CodeLine
	int	string	list	int	string	int

With the preliminaries out of the way, we proceed to the `action_decl` command that generates a set of forward function declarations.

```
proc ::action_decl {} {
    append result [comment "Forward Declarations of State Action Functions"]
    relation foreach state [getStatesByClass] -ascending ClassName {
        relation assign $state
        relation foreach stateinfo $States -ascending StateName {
            relation assign $stateinfo
            append result [emit_action_signature $ClassName $StateName] " ;\n"
        }
    }

    return $result
}

proc ::emit_action_signature {className stateName} {
    append result\
        "static void ${className}_${stateName}\("\
        "void *const s_, "\
        "void *const p_)"
}
```

Figure 13. State Action Declaration Query

It is a relatively simple matter to use the **relation foreach** command to iterate through each class and then to iterate through each state generating a static "C" function declaration. The **relation assign** command takes a singleton relation value and creates a Tcl variable for each attribute with the variable named the same as the attribute. for each attribute that has the same name as the attribute.

The result of template expansion query for our running example is:

```
/*
 * Forward Declarations of State Action Functions
 */
static void Motor_Agitating(void *const s_, void *const p_) ;
static void Motor_Off(void *const s_, void *const p_) ;
static void Motor_Spinning(void *const s_, void *const p_) ;
static void Pump_Off(void *const s_, void *const p_) ;
static void Pump_Running(void *const s_, void *const p_) ;
static void Valve_Closed(void *const s_, void *const p_) ;
static void Valve_Open(void *const s_, void *const p_) ;
static void WashingMachine_Agitating(void *const s_, void *const p_) ;
static void WashingMachine_EmptyingRinseWater(void *const s_, void *const p_) ;
static void WashingMachine_EmptyingWashWater(void *const s_, void *const p_) ;
static void WashingMachine_FillingForRinse(void *const s_, void *const p_) ;
static void WashingMachine_FillingForWashing(void *const s_, void *const p_) ;
static void WashingMachine_Idle(void *const s_, void *const p_) ;
static void WashingMachine_Rinsing(void *const s_, void *const p_) ;
static void WashingMachine_Spinning(void *const s_, void *const p_) ;
```

Figure 14. State Action Declaration Example Results

So the strategy here is to use the code template to specify the order in which fragments will be presented to the "C" compiler. Those fragments are generated by expanding the template which in turn invokes a series of commands that are queries on the schema population that was built up during the parsing phase. The query commands build up a *report* that is inserted into the result of the template expansion. Here it happens to be that the generated report is "C" code.

4. Friends of Pycca

It is often difficult to determine precisely what functionality a program or suite of programs will need when first starting out. That was certainly the case with **pycca**. The original idea for **pycca** was simply to ease to burden of constructing the data structures needed by the STSA. It is easy to convince yourself that a fixed set of implementation data structures will meet the immediate needs of the program you are constructing. However, using relational data structures provides substantially more power and flexibility. Since TclRAL can serialize a set of relvars to external storage, once parsed any given **pycca** file can be easily saved for later use. One aspect that relation data structures support is the notion of *ad hoc* query. Most programmers don't deal with *ad hoc* queries as part of a program with fixed functionality. But support of these types of queries is a powerful way to extend programs or create sets of programs that work well together, bound by a common relational data schema.

These consideration lead to several auxiliary programs for **pycca**. In the section below, we discuss three of the most useful examples of programs that use the serialized **pycca** output to provide additional support for building embedded software programs. The serialized relvars that **pycca** saves contain all the information that is in the original **pycca** source but in an already parsed and organized form. This makes writing the helper program usually a simple matter of reading in the file and performing a different set of queries to produce the needed output.

4.1. Pyccaexplore

Although **pycca** allows states and events to be specified as named entities, in truth the execution architecture works strictly off of numbers. State and event names are converted by **pycca** to integers that are ultimately used as indices into an array that holds the state transition matrix. In small embedded systems, there is no memory to hold the original string names for the states and

events and **pycca** does not attempt to put those strings into the generated program. Consequently, during debugging one is often faced with trying to determine the state of a particular class instance and where it might transition under a given event. Sadly, when looking at a source level debugger the information that is available will be something to the effect of:

> Instance 0x57ae is in state 3 and is about to receive event 1 causing a transition to state 4.

This is not very helpful. What is needed is the mapping between state and event names and the numbers that **pycca** assigned to them. This is why **pyccaexplore** was written.

Pyccaexplore will read the serialized relvars saved by **pycca** and display the state transition matrix of all the classes. That display contains both the name and number of each state and event.

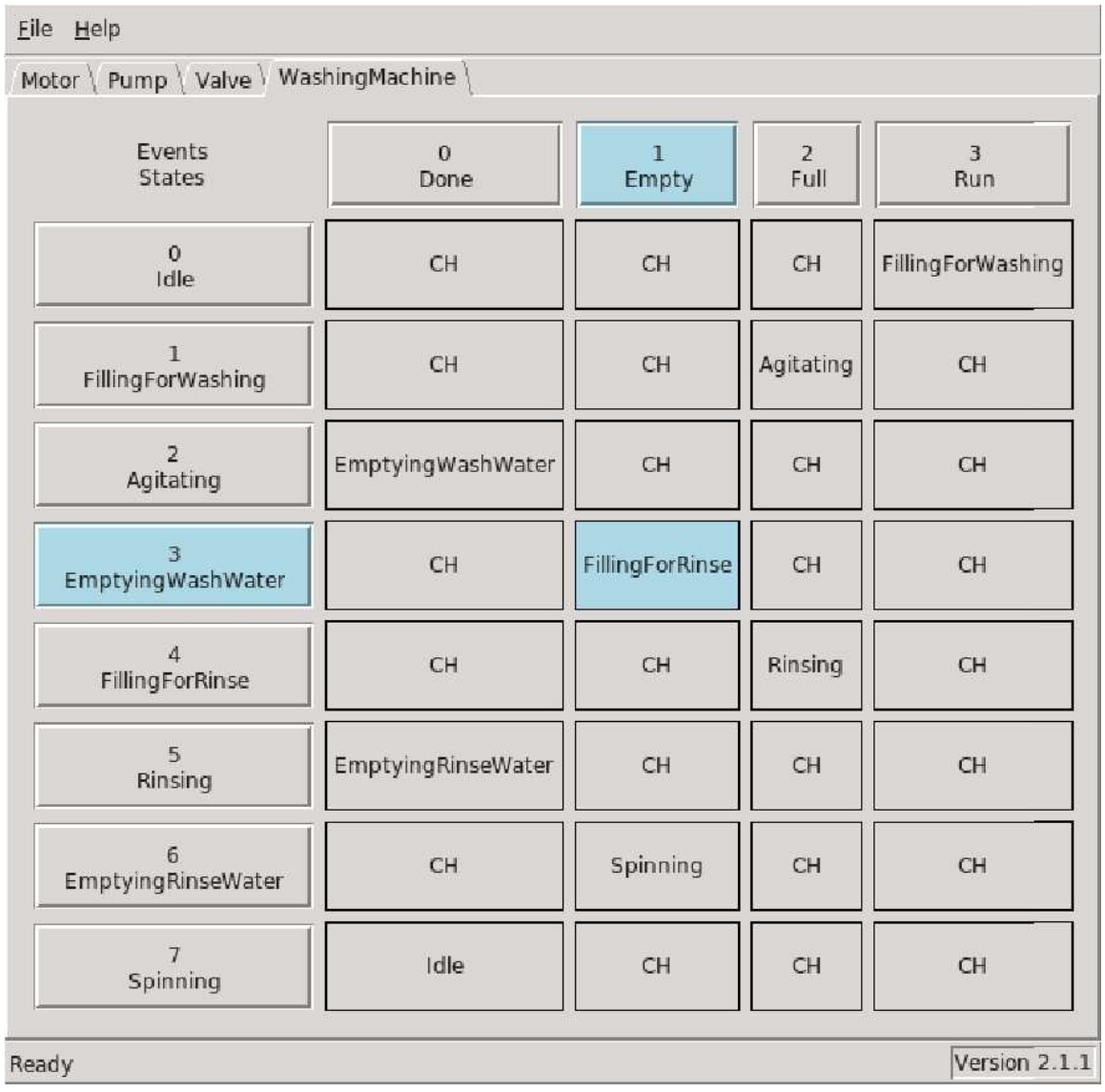

Figure 15. Screen Shot of Pyccaexplore

The figure above is a screen shot of **pyccaexplore** for the example state machine. Buttons along the left allow selection of rows that correspond to states. Buttons along the top allow selection of columns that correspond to events. When both a row and column are selected, a single cell is highlighted showing the transition that would happen when the highlighted event is received in the highlighted state. The above figure can be interpreted as indicating that when state 3 (Emptying-WashWater) receives event 1 (Empty) it transitions to the FillingForRinse state which is numbered 4.

4.2. Pycca2dot

The **pycca** source that defines a state machine is relatively easy to read and does not depend upon the ordering of state and transition declarations. But, state machines are sometimes better represented as directed graphs. Usually when designing a state machine, I do so graphically. Often, maintaining the design time graphics during the implementation does not happen. What **pycca2dot** does is produce a graphic that precisely matches the implementation. Reading in the serialized relvars, **pycca2dot** produces a file that can be fed to **dot**[8] to produce a state machine graphic. The **dot** program takes a free form text input file that defines the graph connections and appearance properties. **Dot** lays out the graph and is capable of producing a wide variety of graphical file format outputs. The figure below show how **dot** rendered the Washing Machine state model show previously in Figure 2. This is quite remarkable especially given that we have supplied no positioning information at all. **Dot** accomplishes the graph layout based solely on the graph connectivity information.

[8] www.graphviz.org

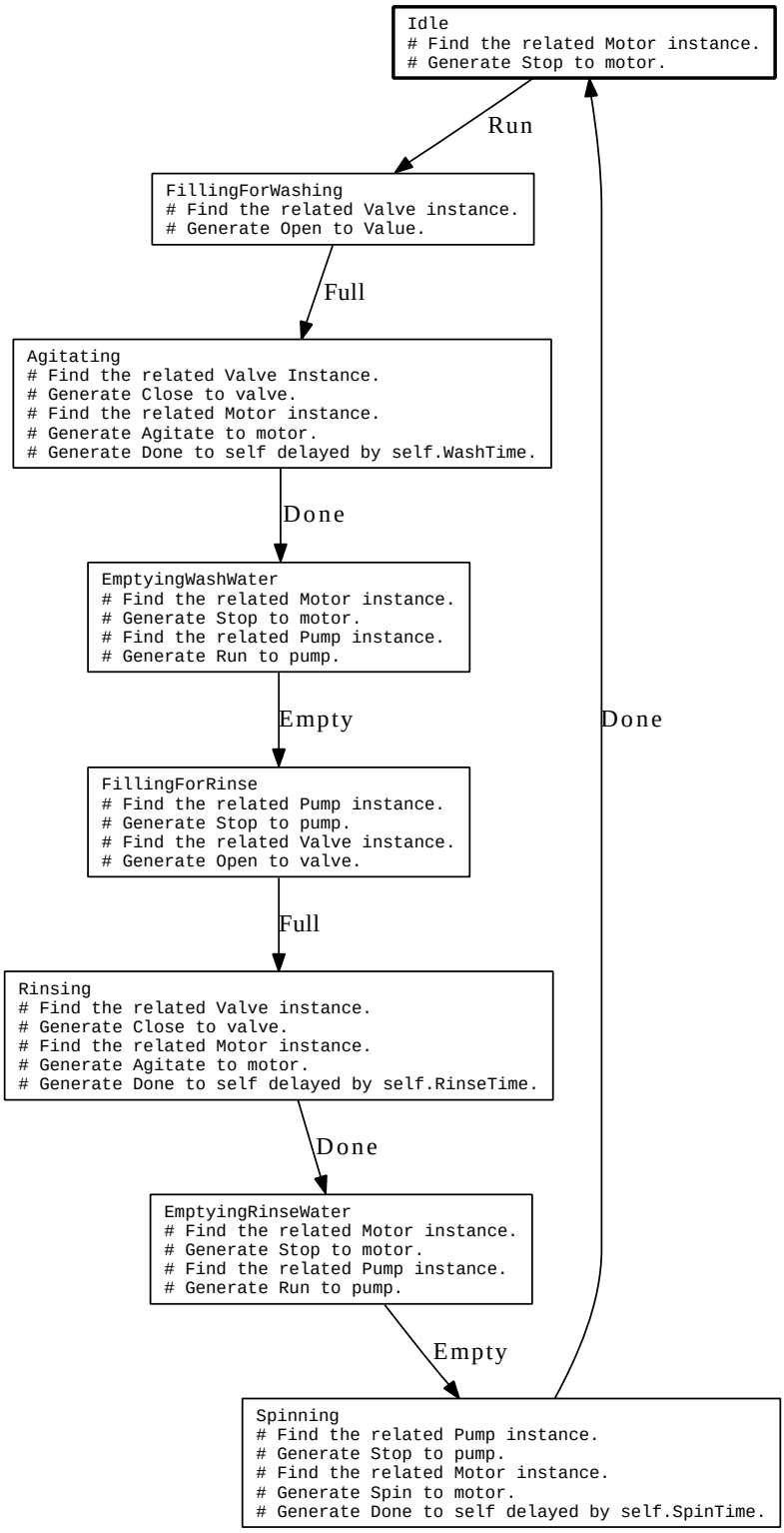

Figure 16. Washing Machine State Model from pycca2dot

4.3. Tracing State Machine Dispatch

The STSA execution architecture can be conditionally compiled to trace the dispatch of state machine events. Programs may supply a callback function that is invoked during the tracing and that function can store or transmit the trace data. However, as we have seen, there is no string information in the embedded system and all the trace data is numeric. It takes knowledge of the state and event name to number mapping as well as knowledge of the link map to translate the numerical trace information back into meaningful strings for human consumption. The **mechtrace** package reads the saved relvars from **pycca** and the ELF file of the executable and is able to translate a packet of trace information into a human readable string.

The most immediately useful consequence of this is to obtain a chronological sequence of event dispatches. This is extraordinarily useful information since the execution sequencing of the program is not easily followed just from the code alone. The STSA implements state machine callback type of execution sequencing. So all the significant code in the program ends up packaged into functions that are called as part of dispatching an event. The invoked callback may indeed generate other events, but does not directly affect the execution sequencing as other events may already be queued. The effect is very similar to typical GUI programs where the sequencing of event dispatches is centralized and not directly visible to the programmer. So a chronological sequence of state transitions can provide enormous insight into the run-time behavior of a program.

Another benefit of obtaining trace data is related to testing. Because all the execution sequencing is centralized, the sequence of trace data can be used to compute the state transition coverage for testing purposes. In well designed state machines, the state action code is small and contains little conditional logic. Indeed deeply nested conditional logic in a state action is an indication that the analysis should be revisited to look for the possibility of several classes being confused as one or that other state information is being hidden in instance variables. The **mechtrace** package includes procedures to tally the transitions that are implied by the tracing data and then to compute the transitions that were not taken in the state machines. This gives objective evidence of the testing that has been done as well as indicates any short comings in the test case coverage, at least at a state transition level. Given that state diagrams are directed graphs, then a depth-first search of the graph can be used to compute a spanning tree the arcs which are labeled with the events necessary to visit every state in the machine. Thus much of the work of determining a set of test cases that will cover the transitions of the state machine can be computed *a priori*.

5. Conclusions

Pycca is a program that implements a domain specific language. The language that **pycca** implements is used as a helper to specify data and control structures for highly embedded microcontroller based software applications.

Yet the internals of **pycca** look more like a traditional database application. **Pycca** uses language statements to populate a normalized relational schema. The language provides a more convenient syntax that is focused on the declarative aspects of the application and hides the details of the underlying schema. The output of **pycca** is generated by query on the population and the report of that query is generated by template expansion to produce "C" code. The relational data structure provide a number of distinct advantages.

(1) The algebra of relations is independent of the headings of the relations. So there is only one set of access methods that need to be learned. It is not necessary to code *ad hoc* data structures and a set of access mechanisms for them. Especially since in practice, such access layers rarely get the attention and detailed error checking that they need.

(2) Referential integrity constraints are a powerful tool to validate the schema population and enforce the rules of the application. Integrity constraints are declarative in nature and

correspond to the semantic rules of the application. Specifying the integrity constraints is usually easier than writing code to accomplish the validation and enables a substantial part of the application semantics to be checked during the data population.

(3) There are a variety of implementation choices for relational data in Tcl that can be matched against the scale of the application. For simpler, in-memory applications TclRAL is a good choice and does not involve a separate query language. If the file storage nature of the application is more important, SQLite is good choice. It is easy to use, readily available and does not involve extensive administration, albeit the queries must be programmed in a separate language from Tcl. Finally, for large scale applications it may be advisable to use one of the server based DBMS accessed via TDBC.

(4) A relational schema serves as a convenient way to bind a suite of programs together, each performing specific tasks associated with the overall application. This makes phasing in a toolbox of programs for the application much easier to accomplish and argues against monolithic programs that try to provide all the functionality of an application. Such programs inevitably become unwieldy and hard to maintain.

None of these points is particularly new or novel. Much of the motivation of creating DBMS in the past was to reap just these benefits. What has changed is the array of choices in the implementation technology that allows even simple programs that one might never consider to be database applications to take advantage of the power of relationally structured data.