Translating Executable Software Models with `micca`

Andrew Mangogna

Model Realization

Abstract

Micca is a program, written in Tcl, for translating executable software models into “C” code. It accepts a domain specific language formulated as a Tcl script that describes the data, dynamics, and processing for a software domain and produces “C” header and code files implementing the model logic. This paper explores the underlying Tcl technology that is used to implement the translation. Tcl features and extensions for creating DSL’s, handling relational-structured data, parsing using a PEG, and code generation using template expansion are discussed. The source code for `micca` is freely available and licensed in the same manner as Tcl.

Introduction

`Micca` is a program, written Tcl, that aids in translating Executable UML domain models into “C” code. This paper is about how the facilities of Tcl are used to implement `micca`.

The next section presents an example that is used throughout the paper. Then we discuss four aspects of Tcl that were used in `micca`. Finally, there is a summary and information about where `micca` can be obtained.

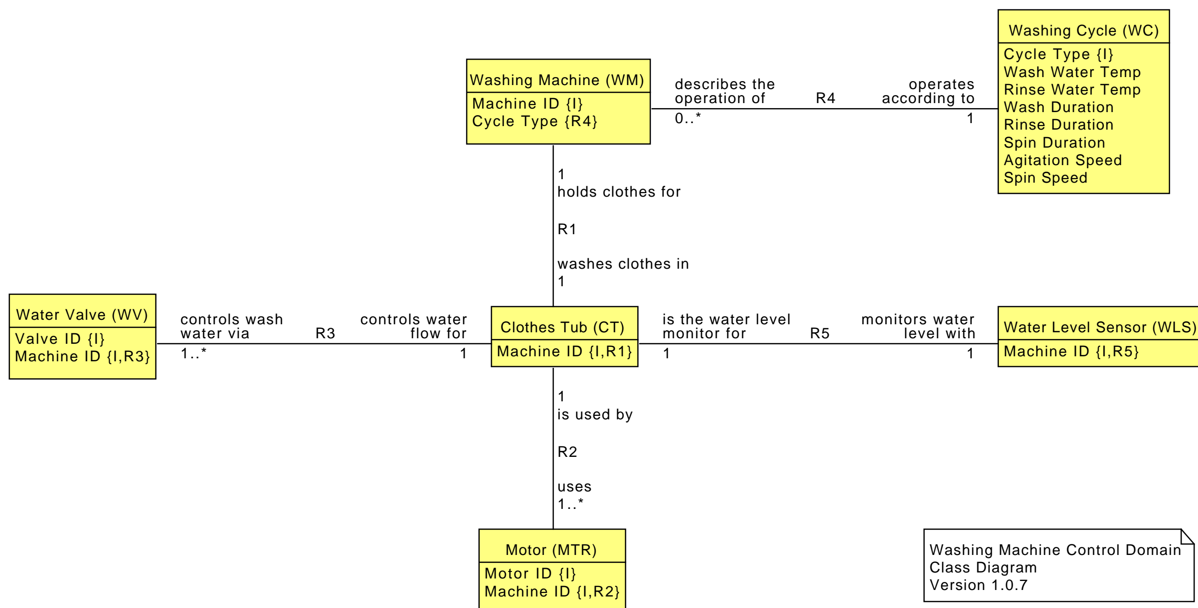
Example model

This paper uses a running example to illustrate how Tcl is used by `micca`. The example is fully worked out in the `micca` literate program document. The subject matter of the example is a simplified automatic clothes washer. The intent is to select a subject from ordinary experience to avoid needing detailed explanations of the problem.

In our washing machine world, a *Washing Machine* operates according to a *Washing Cycle*. The *Washing Cycle* is a set of parameters that specifies the control values that will turn dirty clothes into clean ones. The *Washing Machine* has a *Clothes Tub* into which the dirty laundry is placed.

There are also *Water Valves* to control the flow of water into and out of the *Clothes Tub* and *Motors* to run a water pump, agitate the *Clothes Tub* and rotate the *Clothes Tub* to spin excess water out of the clean laundry. Rounding out the machinery, there is a *Water Level Sensor* that will tell us when the *Clothes Tub* is filled with water or empty of water.

The following is a class diagram for the example. We use UML class diagram notation, but note that the semantics of Executable UML differ substantially from those of conventional UML and only a small subset of the many different UML diagram types are used in Executable UML. In particular, Executable UML does not use many of the the object oriented programming language concepts of UML. Experience has shown that it is much easier to adjust the meaning a person applies to a graphical symbol than it is to introduce a new, unfamiliar notation.



Domain specific languages in Tcl

The class diagram graphic of the example translates into the following `micca` DSL statements. In the interests of space, we give only part of the domain script.

```
domain wmctrl {
    class WashingMachine {
        attribute MachineID {char[32]}
        statemodel {
            transition Stopped - Start -> FillingToWash
            transition FillingToWash - Full -> Washing
            # ... and other transition commands
```

```
            state Stopped {} {
                // "C" code for the Stopped state
            }
            # ... and other state commands
        }
    }
    class WashingCycle {
        attribute CycleType {char[32]}
        attribute WashWaterTemp WaterTemp_t
        attribute RinseWaterTemp WaterTemp_t
        attribute WashDuration unsigned
        attribute RinseDuration unsigned
        attribute SpinDuration unsigned
        attribute AgitationSpeed WashSpeed_t
        attribute SpinSpeed WashSpeed_t
        # ... and other Washing Cycle class properties
    }
    association R4 WashingMachine 0..*--1 WashingCycle
    # ... and the specification of the other classes in the diagram
}
```

The above is actually a valid Tcl script despite not containing any readily identifiable core Tcl commands. The command oriented nature of Tcl along with its simple syntax can be used to construct a DSL that is declarative in its intent.

Use of namespaces

The `micca` DSL script is executed in a set of namespaces where the commands resolve locally without additional command name qualification. The `domain` command is used to specify the characteristics of an executable domain model. The `class` and `association` commands are defined in the namespace where the body of the `domain` command is evaluated. The `attribute` command is defined in the namespace where the body of the `class` command is evaluated. There are many other commands used to specify the relationships between classes and the dynamics of class lifecycles and all commands that take a script body as an argument have an associated namespace in which the body is executed.

`Micca` uses child namespaces to segregate the commands in the configuration DSL and to control command name resolution. The namespace where the configuration DSL runs is defined as:

```
namespace eval @Config@ {
    # The "domain" command is defined in this namespace.
```

```
    namespace eval DomainDef {
        # The "class" command is defined in this namespace.
        # ...
    }
    namespace eval ClassDef {
        # The "attribute" command is defined in this namespace.
        # The "statemodel" command is defined in this namespace.
        # ...
    }
    namespace eval StateModelDef {
        # The "transition" command is defined in this namespace.
        # The "state" command is defined in this namespace.
        # ...
    }

    # ...
}
```

Each DSL command taking a script argument invokes `ConfigEvaluate` to execute the script in a given namespace.

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

This command uses the ability of the `apply` command to execute a lambda function in a given namespace. For example, the `class` procedures uses `ConfigEvaluate` as follows:

```
ConfigEvaluate [namespace current]::ClassDef $body
```

Handling errors

Using the `apply` command is a convenient way to execute a script in a given namespace. However, if any error is encountered, execution terminates immediately. For a user, this is annoying since it implies that you discover errors in the script one at a time. We want the DSL processing to behave like a conventional language compiler and make a best effort to process the entire text to discover multiple errors in one pass.

A close look at `evalLambda` shows how the complete script can be evaluated.

```
1 variable evalLambda {{body} {
2     upvar #0 ::micca::@Config@::errcount errcount
```

```
3       upvar #0 ::micca::@Config@::configlineno configlineno
4       set lineno $configlineno
5       set command {}
6       foreach line [split $body \n] {
7           append command $line \n
8           incr lineno
9           if {[info complete $command]} {
10              try {
11                  eval $command
12              } on error {result} {
13                  set cleancmd [CleanUpCommand $command]
14                  log::error "line $configlineno: \"$cleancmd\":\n\"$result\""
15                  incr errcount
16              }
17              set command {}
18              set configlineno $lineno
19          }
20      }
21      if {$command ne {}} {
22          set cleancmd [CleanUpCommand $command]
23          log::error "line [expr {$lineno - 1}]: end of script reached in the\
24              middle of the command starting at line $configlineno: $cleancmd"
25          incr errcount
26      }
27      return $errcount
28  }
```

line 6 The strategy is to split the script `body` into lines.

line 7 Lines are accumulated into a command.

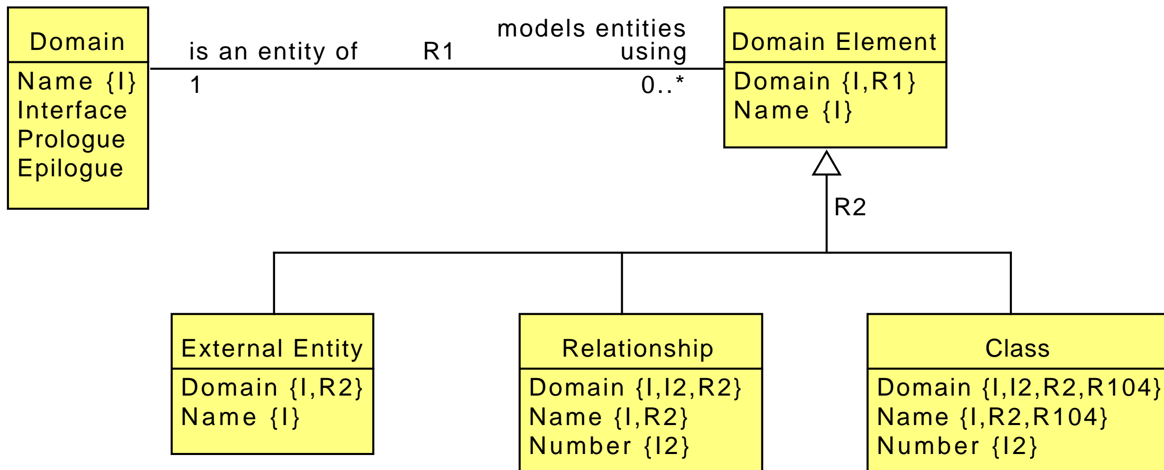
line 9 The `info complete` command is used to determine if the accumulated command is potentially complete. If so, then it is evaluated. If not, then the next line is appended and another try is made.

line 10 Any errors that occur in the command evaluation are caught by the `try` command. After some bookkeeping, evaluation continues on the next line of the `body`.

Relational structured data

The primary purpose of `micca` is to generate "C" code that implements the model specified by the DSL. The DSL is used as a text-based user interface to populate a platform specific model for the Executable UML execution rules as they are realized in the "C"-based target software platform.

The following figure shows a small fragment of the micca platform specific model.



There are 86 classes and 78 relationships in all in the micca platform specific model. Micca uses the rosea package to manage the platform model. Rosea is similar in intent as micca except the translation is targeted at Tcl as the implementation language rather than "C".

Rosea uses a DSL similar to that of micca. The differences arise from the use of Tcl as a target and that rosea is based on TclRAL which directly supports the identity and referential integrity implied by the constructs in the class diagram. These integrity checks, provided by formalizing the micca platform model using rosea, are an important part of the DSL processing. The DSL commands insert their data into the appropriate classes in the micca platform. The data is inserted during a transaction and any violations of the integrity of the data relationships are detected at the end of the transaction. This level of data checking, provided automatically by TclRAL, would otherwise have to be coded into the micca application itself. The burden of coding integrity checks into the application is particularly acute when faced with changes to the underlying platform model.

The class diagram fragment for the micca platform model can be transliterated into the following rosea script.

```
class Domain {
    attribute Name string -id 1\
        -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
    attribute Interface string -default {}
    attribute Prologue string -default {}
    attribute Epilogue string -default {}
}
class DomainElement {
    attribute Domain string -id 1
```

```
10      attribute Name string -id 1\
11          -check {[::micca::@Config@::Helpers::isIdentifier $Name] &&\
12              ![::regexp -- {__[A-Z]+\Z} $Name]}
13      reference R1 Domain -link {Domain Name}
14  }
15  association R1 DomainElement 0..*--1 Domain
16  class Class {
17      attribute Domain string -id 1 -id 2
18      attribute Name string -id 1
19      attribute Number int -id 2
20      reference R2 DomainElement -link Domain -link Name
21      reference R104 ValueElement -link Domain -link Name
22  }
23  class Relationship {
24      attribute Domain string -id 1 -id 2
25      attribute Name string -id 1
26      attribute Number int -id 2 -system 0
27      reference R2 DomainElement -link Domain -link Name
28  }
29  class ExternalEntity {
30      attribute Domain string -id 1
31      attribute Name string -id 1\
32          -check {[::micca::@Config@::Helpers::isIdentifier $Name]}
33      reference R2 DomainElement -link Domain -link Name
34  }
35  generalization R2 DomainElement Class Relationship ExternalEntity
```

line 13 Describes the {R1} annotation on the Domain attribute of the Domain Element class shown in the diagram. The `reference` command, occurring within the definition of the `DomainElement` class, states that `DomainElement` contains a referential attribute that realizes the `R1` association to `Domain` by "linking" the `Domain` attribute in `DomainElement` to the `Name` attribute in `Domain`. By linking, we mean that the value the `Domain` attribute of any instance of `DomainElement` must be equal to the value of the `Name` instance for some instance of the `Domain` class. `Rosea` uses `TclRAL` to enforce this constraint on the attribute values.

Parsing "C" type names

It is necessary for `micca` to understand some aspects of "C" data types. It must be able to generate "C" code that declares variables and so must be able to parse a "C" data type name.

For example, the `attribute` command we saw in the washing machine control section requires a "C" type name as an argument. `Micca` validates the argument as a proper "C" type name and uses the type name later in code generation. To accomplish parsing type names, `micca` uses the *Parser Tools* from `tcllib`. The parser tools use a *Parsing Expression Grammar* (PEG) to specify the generation of a parser. We do not describe the background of PEGs here, but note that PEGs are not able to represent left-recursive grammars and this will entail some transformation of the usual formulation of "C" as a left-recursive grammar (typically LR(1) that is used to generate a parser from `yacc` or `bison`).

The PEG used by `micca` was derived from a full C99 PEG written by Ian Piumarta. Since `micca` only parses type names, the grammar was condensed to those portions used by `micca` and translated to the PEG syntax used by the Parser Tools. The grammar file is too long to present here in its entirety, so we discuss only selected parts.

The following is the first set of grammar productions for the type name parser.

```
PEG datatype (type_name)
type_name <-
    specifier_qualifier_list abstract_declarator? EOF ;

abstract_declarator <-
    pointer? direct_abstract_declarator /
    pointer ;

direct_abstract_declarator <-
    direct_abstract_declarator_head direct_abstract_declarator_tail* ;

direct_abstract_declarator_head <-
    LPAREN abstract_declarator RPAREN /
    direct_abstract_declarator_tail ;

direct_abstract_declarator_tail <-
    array_declarator /
    LPAREN parameter_type_list? RPAREN ;

array_declarator <-
    LBRACKET assignment_expression? RBRACKET /
    LBRACKET STAR RBRACKET ;

pointer <-
    (STAR type_qualifier_list?)+ ;

type_qualifier_list <-
    type_qualifier+ ;
```

```
declaration_specifiers <-
    storage_class_specifier declaration_specifiers? /
    type_specifier declaration_specifiers? /
    type_qualifier declaration_specifiers? /
    function_specifier declaration_specifiers? ;

specifier_qualifier_list <-
    (type_specifier / type_qualifier)+ ;

# ... and many more grammar statements
#
#

END ;
```

Because of the peculiarities of "C" type names there is a fundamental ambiguity in "C" related to type names. We can show the ambiguity as follows.

```c
unsigned long var1 ;
typedef unsigned long ulong ;
ulong var2 ;
```

The `typedef` statement in "C" can introduce a new name (or alias) for a type. In this example, `var1` and `var2` have the same data type, but syntactically we must parse `ulong` as a type name. "C" uses syntax for variable declarations that is intended to remind you of the type of the value held by the variable when it is used in an expression. In this example, when using `var1` in an expression you know its type is `unsigned long`. A "C" compiler usually deals with this ambiguity by augmenting the set of valid type names for the lexical analyzer when the `typedef` statement is recognized.

Since `micca` does not parse the passed through "C" code for activities and since the internals of the parser generated from the PEG are not generally accessible, the usual compiler solution is not available to us. `Micca` resolves this by directly recognizing all the standardized type names (*e.g.* `int8_t`, `uint8_t`, etc.) and by using a naming convention.

```
type_specifier <-
    int8_t / int16_t / int32_t / int64_t / int_least8_t / int_least16_t /
    int_least32_t / int_least64_t / int_fast8_t / int_fast16_t / int_fast32_t /
    int_fast64_t / void / char / short / int / long / float / double /
    signed / unsigned / _Bool / bool / _Complex / complex / _Imaginary /
    imaginary /
    uint8_t / uint16_t / uint32_t / uint64_t / uint_least8_t / uint_least16_t /
    uint_least32_t / uint_least64_t / uint_fast8_t / uint_fast16_t /
```

```
    uint_fast32_t / uint_fast64_t / intptr_t / uintptr_t / intmax_t /
    uintmax_t / size_t / ptrdiff_t /
    struct_or_union_specifier / enum_specifier / typedef_name ;

typedef_name <-
    <upper> <alnum>* '_t' WHITESPACE /
    'MRT_' <alnum>+ WHITESPACE ;
```

The naming convention recognizes the type names used by the `micca` run-time (those type names starting with `MRT_`) as well as any type names that start with an upper case character followed by alphanumeric characters and finally ending in an `_t` suffix.

With this resolution of the ambiguity, the parser generated by the parser tools from the PEG can recognize all "C" type specifications. For example, `int (*)(void)` specifies a type which is a pointer to a function accepting no arguments and returning an `int`. Parsing this type specification results in the following abstract syntax tree (AST).

```
<type_name> :: 0 12
    <specifier_qualifier_list> :: 0 3
        <type_specifier> :: 0 3
            <int> :: 0 3
    <abstract_declarator> :: 4 12
        <direct_abstract_declarator> :: 4 12
            <direct_abstract_declarator_head> :: 4 6
                <LPAREN> :: 4 4
                <abstract_declarator> :: 5 5
                    <pointer> :: 5 5
                        <STAR> :: 5 5
                <RPAREN> :: 6 6
            <direct_abstract_declarator_tail> :: 7 12
                <LPAREN> :: 7 7
                <parameter_type_list> :: 8 11
                    <parameter_list> :: 8 11
                        <parameter_declaration> :: 8 11
                            <declaration_specifiers> :: 8 11
                                <type_specifier> :: 8 11
                                    <void> :: 8 11
                <RPAREN> :: 12 12
```

In this representation of the AST, the token name is given followed by the starting and ending offsets into the parsed string. With the parsed AST, a variable of this type can be declared by walking the tree and finding the correct offset where the variable name must be inserted. In this example, inserting a variable name after the <STAR> token of the `<direct_abstract_declarator_head>` (*i.e.* after the character at offset 5) results in a valid variable declaration, *e.g.* `int (*funcptr)(void)`.

Code generation

The code generated by `micca` falls into two broad categories:

1. Declaration of data structures and initialized variables such as classes and state model transition tables. The code generated for this has no executable component, providing the run-time code with the data it uses to implement domain specific behavior.
2. Activity code that represents the processing performed by the model. This is typically code supplied by a user that is packaged into "C" functions.

To accomplish the code generation, `micca` uses the `textutil::expander` package from `tcllib`. The expander package provides conventional template expansion functionality where ordinary text is passed to the output and embedded commands are executed, passing the result of the command to the output. A separate template is used for each of the categories of generated code described above.

Generating the domain header file

For each domain, `micca` generates two files: a "C" header file containing interfacing information and a "C" code file containing the executable portion of the model translation. In this example, we show part of the header file generation to illustrate one use of template expansion.

The following is the expansion template used to the generate "C" header file for a domain.

```
set headerTemplate {
    <%banner%>
    #ifndef <%headerFileGuard%>
    #define <%headerFileGuard%>
    #include "micca_rt.h"
    #include <assert.h>
    <%interface%>
    <%interfaceTypeAliases%>
    <%domainOpDeclarations%>
    <%externalOpDeclarations%>
    <%eventParamDeclarations%>
    <%portalIds%>
    <%portalDeclaration%>
    #endif /* <%headerFileGuard%> */
}
```

The strings used to mark the beginning and end of a command have been set to <% and %> for this expansion. A procedure is defined for each command in the template. "C" requires substantial type annotation and declarations usually must appear before the definitions of the code and

data. The ordering of the expansion commands in the header template is contrived to match the requirements of the "C" compiler to insure that components are declared before they are defined.

The general pattern of the template expansion procedures is to query the underlying `micca` platform model and construct a string containing the required "C" language statements. `Rosea` provides the necessary commands to query the platform model and `TclRAL` provides a complementary set of commands to operate on the relation values obtained from the query. The generated language statements are returned as a string by the procedure and the template expansion code replaces the command in the template with the procedure result.

As an example of this pattern, we examine the declaration of domain operations by the `domainOpDeclarations` procedure from line 9 of the header template. A *Domain Operation* is a "C" function provided by the domain model as the means to access some aspect of the domain. The set of domain operations forms the primary programming interface to the domain.

```
proc domainOpDeclarations {} {
    variable domain
    set result [comment "Domain Operations External Declarations"]

    set opRefs [DomainOperation findWhere {$Domain eq $domain}]
    set params [deRef [findRelated $opRefs ~R6]]
    set ops [pipe {
        deRef $opRefs |
        relation project ~ Domain Name ReturnDataType Comment |
        relation rename ~ Name Operation |
        ralutil::rvajoin ~ $params Parameters
    }]

    relation foreach op $ops {
        relation assign $op
        if {$Comment ne {}} {
            append result [comment $Comment]
        }
        set plist [relation list $Parameters DataType -ascending Number]
        set pdecl [expr {[llength $plist] == 0 ?\
                "void" : [join $plist {, }]}]
        append result "extern $ReturnDataType\
                ${Domain}_${Operation}\($pdecl\) ;\n"
    }

    return $result
}
```

line 5 Find the *Domain Operations* for the domain whose code is currently begin generated

(stored in the `domain` variable).

line6 Find the parameters of the Domain Operation by traversing the `R6` relationship. In the `micca` platform model, `R6` associates a *Domain Operation* to zero or more formal *Domain Operation Parameters*.

lines 7-12 This series of commands creates a relation having an attribute that is also a relation value. If the *Domain Operation* has parameters, then the Parameters attribute is a non-empty relation value with a heading containing the Name, Number, and Data Type of the parameter. Otherwise, the Parameters attribute is the empty relation value with the same heading. This is the relational equivalent of an outer join and does not depend upon `NULL` values. Note, the `pipe` command rewrites the sequence of commands in such a way that the result of one command is used directly as an argument to the next command.

line 19 Since "C" is a language that supplies function arguments by position, the parameters are ordered by the number assigned to them when they were specified in the DSL.

line 22 Note that we prepend the domain name to the operation name in order to avoid naming conflicts in the global namespace. Naming conventions are necessary for a a language like "C" which does not have support for sophisticated module or name segregation.

For our example washing machine control domain, the following is the portion of the header file generated by the `domainOpDeclarations` procedure.

```
/*
 * Domain Operations External Declarations
 */
extern int wmctrl_createWasher(char const *) ;
extern bool wmctrl_deleteWasher(char const *) ;
extern bool wmctrl_startWasher(char const *) ;
extern void wmctrl_selectCycle(char const *, char const *) ;
extern void wmctrl_init(void) ;
```

Generating state activity code

In Executable UML, most of the computing for the domain happens as part of the actions associated with a state model. In our example model, part of the lifecycle of a washing machine is to fill the clothes tub with water before starting the washing agitation. This happens in the *Filling To Wash* state.

There are many ways to express, in an implementation independent fashion, the computations that are required to fill the clothes tub. Here we use a pseudocode that expresses the model level processing.

Filling To Wash action language

```
# Fill the tub with wash water.
```

```
wc .= /R4
ct .= /R1
Fill(temp : wc.WashWaterTemp) -> ct
```

This pseudocode states:

1. Traverse the `R4` association to find the related instance of *Washing Cycle* and call that instance `wc`.
2. Traverse the `R1` association to find the related instance of *Clothes Tub* and call that instance `ct`.
3. Signal the *Fill* event to the related *Clothes Tub* instance and use the value of the *WashWaterTemp* attribute in the related *Washing Cycle* as the argument to the *Fill* event which determines the temperature of the water that should be placed in the *Clothes Tub*.

The platform independent action from above is translated into a platform specific action before it is passed to `micca`. State activities usually consist of model level actions, such as traversing relationship and signaling events, and conventional expression evaluation and control of the execution flow. `Micca` supplies an embedded command language to handle model level actions and relies on a programmer to transcribe action pseudocode into the embedded command language combined with "C" expressions and flow of control. For the *Filling To Wash* state, this transcription is as follows.

Filling To Wash `micca` source

```
1 state FillingToWash {} {
2     <%my findOneRelated wc R4%>
3     <%my findOneRelated ct ~R1%>
4     <%instance wc assign {WashWaterTemp washtemp}%>
5     <%instance ct signal Fill temp washtemp%>
6 }
```

The embedded commands, again marked by <% and %> are expanded using the `textutil::expander` package to generate the code that is placed in the domain "C" code file. In this case, a different instance of an expander is used than the one for the header and code expansions. In fact, the two expansions are proceeding at the same time since generating the code file for a domain involves generating the state activity functions which are placed in the code file.

The result of the expansion is the following "C" function.

Filling To Wash generated "C" code

```
1 static void
2 WashingMachine_FillingToWash__ACTIVITY(
3     void *const s__SELF,
4     void const *const p__PARAMS)
5 // <%my findOneRelated wc R4%>
```

```
6  // <%my findOneRelated ct ~R1%>
7  // <%instance wc assign {WashWaterTemp washtemp}%>
8  // <%instance ct signal Fill temp washtemp%>
9  {
10         #define MRT_STATENAME "FillingToWash"
11     MRT_INSTRUMENT_ENTRY
12     struct WashingMachine *const self = s__SELF ;
13 #line 72 "wmctrl.micca"
14     // instance self findOneRelated wc R4
15     struct WashingCycle *wc ;
16     struct WashingCycle *t__T1 = self->R4 ; // R4
17     wc = t__T1 ;
18     // instance self findOneRelated ct ~R1
19     struct ClothesTub *ct ;
20     struct ClothesTub *t__T2 = self->R1__BACK ; // ~R1
21     ct = t__T2 ;
22     // instance wc assign {{WashWaterTemp washtemp}}
23     WaterTemp_t washtemp ;
24     washtemp = wc->WashWaterTemp ;
25     // instance ct signal Fill {temp washtemp}
26     MRT_ecb *t__T3 = mrt_NewEvent(2, ct, self) ; // Fill
27     struct wmctrl_ClothesTub_Fill__EPARAMS *const t__T4 =
28         (struct wmctrl_ClothesTub_Fill__EPARAMS *)t__T3->eventParameters ;
29     t__T4->temp = washtemp ;
30     mrt_PostEvent(t__T3) ;
31         #undef MRT_STATENAME
32 }
```

line 11 Macros are inserted to facilitate instrumenting the entry into a state action function. A preprocessor define is used to control what, if any, code is inserted for instrumentation purposes.

line 13 Optionally, `micca` will insert `#line` directives to reference any compiler messages back to the `micca` source.

line 16, 20 In the `micca` generated "C" based implementation, traversing a singular association (`R4` and `R1` in this case), is implemented using a pointer stored in the class structure.

lines 26-30 Signaling an event which passes an argument is implemented by requesting an event data structure from the run-time code, assigning the argument value into the data structure, and requesting the run-time to post the event.

Note that there may seem to be a number of variable declarations and assignments that are not strictly necessary. They arise because relationship traversal may be *chained* together in a single `findOneRelated` command and the last traversal in the chain is the desired related instance. In

this case, the chain is only a single relationship. Any reasonable optimization levels for the "C" compiler will remove the superfluous variables and assignments.

Space does not allow discussing the template expansion procedures that generate code. They are considerably more complicated as they perform validation of the request against what is allowed by the `micca` platform model and they must track a symbol table of "C" variables. However, the interested reader can find the complete source code and descriptions of all the code generation procedures in the `micca` documentation.

Summary

`Micca` is an abstract application, taking a specification of an executable software model and generating the code and data needed to build a program that behaves in accordance with the model. It uses several Tcl technologies to accomplish this.

1. Tcl is an interpreted language that can hold a representation of source code in a variable and execute it at run time. This combined with namespaces makes building declarative styled DSLs much easier. The simplicity of Tcl syntax means that it is not necessary to build a conventional grammar for a DSL having a more complex syntax.
2. Complicated data structures can be held as relation values using `rosea`. This allows building a large set of declarative data constraints that need not be added as data validation code in the application. `Rosea` also provides operations to search and navigate the classes of the platform model and this capability is used in the code generation for `micca`.
3. Available parser generation tools allow specifying complex grammars by means of a PEG and generating Tcl code that will parse the language. In `micca`, a PEG was used to parse "C" type names.
4. Template expansion provides a report generation capability that can be used to generate code. The internal `micca` model was queried by commands embedded in the template and the query results are formulated as "C" language statements. This allows easy ordering of the generated code to match the requirements of a "C" compiler.

Resources

`Micca` is freely available from either Model Realization or chiselapp.