How to Translate XUML Models using **`micca`**

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	February 17, 2019	Start of writing.	GAM
1.0a1	March 23, 2019	First draft release.	GAM
1.0a2	March 30, 2019	Second draft release.	GAM
1.0a3	May 27, 2019	Third draft release.	GAM

Contents

List of Figures

List of Tables

List of Examples

Introduction

This paper presents a set of techniques to demonstrate translating an Executable UML (XUML) domain model into a "C" based program using `micca`.

`Micca` is a model translation tool. It works by accepting a platform specific description of an XUML model and generates code that may be compiled into an executable program. `Micca` targets small to medium sized embedded systems[1] and small to medium sized POSIX systems.[2] The implementation language is “C”. `Micca` translations are intended for a class of applications that are specific to the software platform for which it was created. Architecturally, `micca` translated domains are single threaded and use a callback scheme to implement state machine transitions. The class of applications is intended to be event driven and reactive, waking up as a result of external stimulus, performing a short lived computation to produce a response and then going quiescent.

In the next section, the overall workflow of accomplishing a model translation using `micca` is discussed. Subsequent chapters describe the various steps required to finish the translation.

In addition to this document, `micca` has a complete set of reference manual pages. It is recommended that you have those readily available before starting any translation.

Workflow overview

The following figure shows the overall workflow for creating a system using modeling and translating with `micca`.

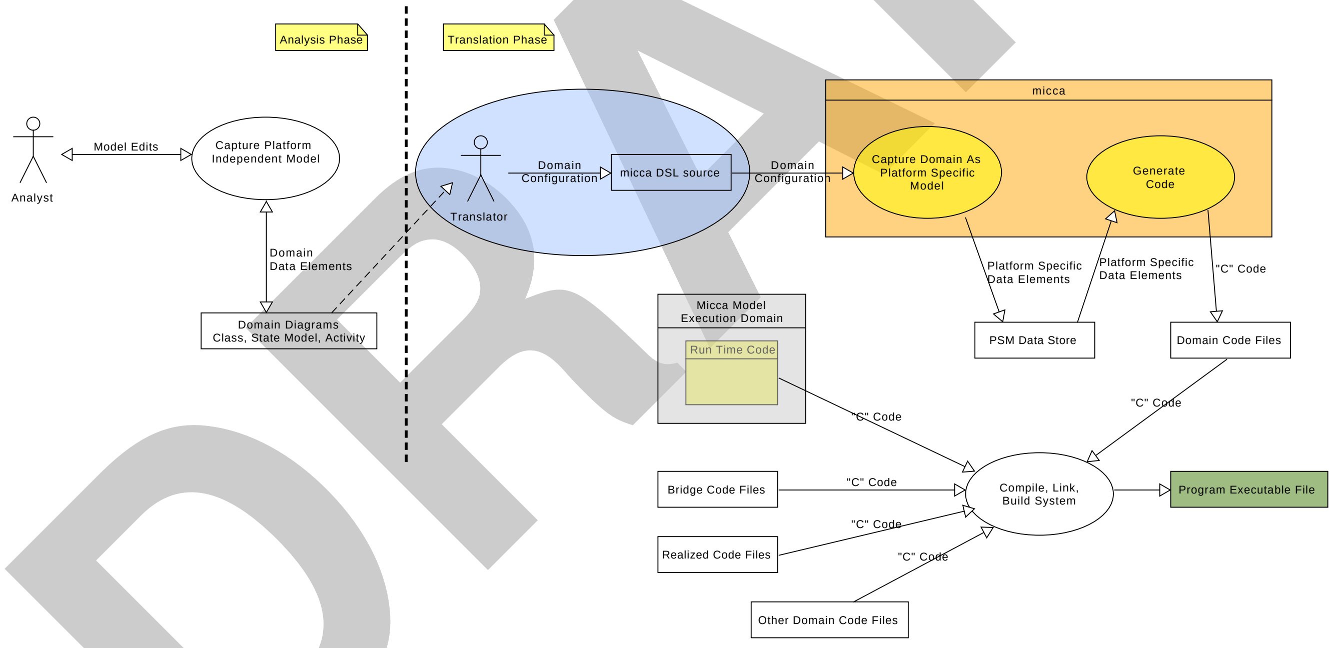


Figure 1: Workflow Overview

An **Analyst** creates one or more domain models. Most systems consist of several domains. During the analysis phase, the analysts captures the domain requirements in a platform independent model. We do not explain how to perform analysis here. Many books have been written on that subject and are listed in the bibliography.

[1]Exact sizes are difficult to estimate. Normally, `micca` generated systems run on bare-metal microcontrollers that contain at least 16 Kibytes of program memory and at least 8 Kibytes of RAM. The top end is not really fixed, although most microcontrollers top out at about 1 Mibytes of program memory and 256 Kibytes of RAM.

[2]The primary use case for running on a POSIX system is to perform testing on the translated domain before it is moved onto the target hardware. However, it is not unreasonable to field an application directly intended to run in a POSIX environment. In that environment, since the `micca` run-time is single threaded, the upper end of scaling will limit the size of any POSIX application where `micca` might be useful.

When the analysis is complete, it is transferred to a **Translator**. The Translator is a human being. The material transferred (as indicated by the dashed line in the figure) is a set of diagrams and text which we discuss below. Note that no machine readable content is transferred directly to the translation phase.

The Translator's role in completing the process is to examine the contents of the domain model materials and formulate a micca DSL source file that contains a platform specific configuration of the domain model which is *derived* from the domain model. We emphasize that the implementation is derived from the model. To do otherwise, lessens the value of the model and calls into question why the model was created at all. This document is written to explain that one particular step in the overall workflow (shown in the light blue oval in the diagram).

Once the domain configuration is complete, `micca` reads the file and captures the platform specific domain elements it contains. Using this data, code is generated for the domain. The code consists of a "C" source file and a "C" header file.

The source files for all the domains in the system, along with any bridge code or other conventionally realized code files are then compiled and linked together to form an executable for the program. This is the conventional way that multiple "C" source code files are built into an executable. Also supplied to the build process is the run-time code required by `micca`. Much of the code generated by `micca` depends upon or drives the operation of its run-time code. For example, the state machine dispatch that sequences the lifecycles of class instances is implemented in the run-time code and driven by data supplied in the "C" source code for a domain. We do **not** describe here how `micca` or its run-time code operate in detail. `Micca` and the run-time are fully described elsewhere. In this document, only an overview of the run-time operation is given to provide the context required for translating the domain.

There are several points of interest in this workflow.

- The tooling for the analysis is completely separated from that of the translation. Ideally, it would be more convenient to automate the translation process by eliminating the human translator. In practice, the complexity of UML tooling and specifically the narrow use of UML concepts in Executable UML makes that automation difficult. The solution chosen here is to apply human intelligence to bridge the tooling gap. We do not consider this any more unusual than applying human intelligence to write any other computer program.
- Because a human is involved, there are substantial possiblities for tailoring the solution to the specific needs of the target platform. This can be quite difficult when attempting the translation with generic tooling intented to support a wide array or the lowest common denominator of platforms.
- There is substantial amount of the system that is conventionally coded. Bridge code is supplied manually. Low level operating system or device specific code is usually required to interface to the external world. Claims of translation schemes that purport to yield 100% code generation should be viewed with skepticism.
- `Micca` is able to externally save its platform specific model population. This is used by companion tools, such as bosal, to aid in testing and analyzing the implementation.
- The person who performs the role of Analyst does not have to be the same person who performs the role of Translator. It is recommended that these two roles be done by different people. Having different persons perform the roles adds two important project checks:
  - More than one person becomes familiar with the content of the domain model.
  - The skill sets required to be a good Translator are quite different than those required of a good Analyst. Specialization can increase the efficiency of accomplishing the work.

  Experience has shown that all models have small logical inconsistencies and errors in them before they are translated. Such errors are not an indictment of the Analyst's talents. It is important that the Translator and the Analyst communicate and correct the inevitable errors that will be found in the first attempts to translate the model. The separation of analysis from translation is intended to separate the problem essentials from the implemenatation artifacts. It is **not** intended to erect a virtual wall across which political battles can be waged with one side contending the models can't be made to run and the other side contending that they are perfectly clear and consistent.

Complete model

Before translation can be begin, it is necessary to have a complete model. That seems like an obvious statement, but in our usual rush to finish systems there are many pressures to start translation before the model is complete. In the end, starting with a half-baked model wastes time, especially if the person performing the translation (as is our recommendation) is not the same person who created the model.

The need for a complete model does **not** imply that the model has to satisfy every possible requirement that could be envisioned for the domain. The necessity for the domain model be complete and self-consistent does **not** necessarily imply that it is complete in terms of the requirements allocated by the system or the services it provides to the system. Experience shows that models, like conventional code bases, benefit from incremental, iterative development. Call it agile or waterfall or whatever you like, skilled and effective practitioners of software development do not build systems in one strictly sequential undertaking. Development is iterative and the results of one iteration help direct subsequent efforts. The usual recommendation is to model core concepts that are central to the subject matter of the domain and work out from there to more peripheral concepts. Of course such recommendations have little to offer in terms of how to determine what the core concepts of a domain are. Experience shows that domain models, like a conventional code base, tend to have areas where it is less clear what the *right* thing is to do. Usually, some doubts enter into the process near the domain boundary and its interactions with other domains. This seems to be a property of any collection of logic when exceeding a certain size[3].

Domain models are precise and detailed and therefore don't have *fuzzy* parts that are stubbed out or left open for interpretation or dangling for future elaboration. We reject the entire concept of elaboration as corrosive to the separation of problem logic and implementation technology. The conflation of incremental model development with incremental elaboration of model details is especially destructive to obtaining full value for the modeling effort. Translation replaces the vague concepts of *high level* or *low level* with a more precise notion of platform independent and platform dependent. We view translation as a discontinuous step of taking a logical statement of a problem and applying computing technology to achieve an executable solution. An incomplete or vague model is simply not sufficient to perform a translation which itself is supposed to be *derived* from the model. We consider the following a *minimal* set of deliverables that are required before translation can begin.

- A class model of the domain data. The class model must contain:
  - A UML class diagram (in accordance with XUML usage of class diagram constructs).
  - Written descriptions of each class, relationship and attribute. Writing descriptions of the class model elements is an essential aspect of creating the class model and the presence of the descriptions indicates that the analyst has thoroughly considered the implications of the model.
  - A description of each model-level data type employed in the class model. The description must define the set of values (either by formula or enumeration) that constitute the data type. Data types should also define the set of operations allowed upon them if it is not clear from the context.
- A state model for each class that has non-trivial lifecycle behavior. The state models must contain:
  - A UML state diagram (in accordance with XUML usage). The diagram must show the initial state and any final states.
  - A state transition matrix in which every possible transition is defined. Every transition not shown on the state model diagram must be marked as **CH** (can't happen) or **IG** (ignored). It is suggested that ignored events be justified as to the reasoning for ignoring them.
  - A description of each state and the role it plays in the lifecycle of the class.
  - A state activity description of the processing for each state. The processing should be described in an unambiguous action language or in action data flow diagrams.
- A state model for each relationship that requires an assigner to serialize the creation of an instance of the relationship. The contents of for an assigner state model are the same as for a state model of a class lifecycle.
- A set of instance operation descriptions for each class that defines them, to include:
  - A description of the function and interface of the operation.
  - The processing performed by the operation in terms of action language or a data flow diagram.
- A set of domain operation descriptions that include:
  - A written description of the operation function and interface.
  - The processing of the domain operation presented as action language or a data flow diagram.

[3] which size is almost never exceeded by the simple examples usually presented

- For any mathematically dependent attribute, the action language formula required to compute the value of the attribute.
- A set of external entity descriptions that include:
  - A description of the logical dependencies allocated to the external entity by the domain.
  - A description of the function and interface for each operation performed by the external entity. This must include whether the operation is considered as synchronous or asynchronous in its behavior.
- An initial instance population of the domain including:
  - Values for every attribute of every instance that is to exist when the domain starts. The preferred manner of specification is via tabluar layout in a purely data oriented fashion. Values of referential attributes indicate the relationship instances that are part of the initial instance population. Using action language to define initial data populations is not acceptible.

Model facets

The construction of an XUML model procedes by considering three facets of the model:

a. the data facet where real world entities and associations are abstracted in to a set of logical predicates represented by data,

b. the dynamics facet where lifecycles of classes are represented by finite state machines, and

c. the processing facet where the computations of the domain are represented by an action language or data flow diagram.

Translation also procedes according to the three facets and in same order. Data structures must be determined, state machines are then specified and finally the code required for the activities is written. This preferred order yields the fastest and most consistent results.

Translation workflow summary

When a language compiler, say a “C” compiler, compiles a code file, it does not produce the code according to the way the program should have been written if you were a good programmer. A badly structured piece of source code produces an equally badly structured output. The compiler can work only with the what it is given and it must faithfully produce code that, when executed, accomplishes the logic contained in the original source. Despite the amazing abilities of modern compilers at code optimization, the compiler is unaware of any intent in the code or any higher order design in the system as a whole.

The same is true of translating XUML domain models. A Translator does not produce code for the model as it would have come from a good analysis. The output can only be *derived* from the model as it was given by the Analyst. If the model is overly complex and poorly conceived, the resulting translation will reflect that.

It is most important that the translation does not add or subtract anything from the logic of the domain model. The intent of translation is only to add the necessary software technology to make the domain run on the intended platform. That job alone is difficult enough. Consequently, a Translator tends to pay little attention to the details of what a domain actually accomplishes. It is a somewhat rote undertaking. Rather, a Translator’s focus is on formulating clear and concise program statements to reflect the actions of the model. Translation concerns are those of the computing platform and not those of the subject matter of the domain.

As described previously, the purpose of this document is to describe a single step in the overall workflow, namely, formulating a platform specific domain configuration from the platform independent modeling material transferred from the analysis phase The workflow for formulating the platform specific domain configuration can be summarized as:

1. Perform an introspection of the model, marking it to show the execution characteristics.
2. Translate the class model.
3. Translate the state models.
4. Translate the data flow diagrams or action language to "C" code.
5. Construct an initial instance population.
6. Construct the bridges between domains.

Each of these areas is discussed in a separate section below.

Model marking

When a language compiler transforms source code into something that is executable, it typically is structured to analyze the code in a series of phases on the path to producing something executable by a computer. The techniques used are many and varied. Some compilers produce machine specific assembly code or even skip the assembly step and produce machine code directly. Other compilers are targeted at virtual machines and produce code that is then interpreted (or further compiled) at run time to execute the program. The many different approaches all have trade-offs, yielding different characteristics, that make them more or less suitable for a particular problem.

Most compilers use a semantic checking and analysis phase. During this phase the compiler insures that language statements, which are syntactically correct, are actually meaningful within the rules of the way the language works. In all language systems, it is possible to have correct statements in the language that have no meaning. For example, if the language requires that variables be declared before they are used, an expression that refers to an undeclared variable has no meaning even if that expression was correctly composed of valid language tokens.

Language compilers also typically examine the execution characteristics of the program being compiled. Internally, a flow graph of execution is often generated and many of the optimizations performed by a compiler are operations that transform a flow graph into a semantically equivalent one that minimizes some aspect of the resulting code. For example, it is possible to examine the flow graph of a function that contains a loop construct and determine that some code piece may be *lifted* outside of the loop and still achieve the same result.

It should come as no surprise that translating an XUML model requires analogous examination. Fortunately, the situation is much simpler for a model than a compiled language. For the case of a model, a single pass over the processing description of the activities is sufficient to generate the introspection we need. We call this analysis of the model processing with the goal of characterizing the operation of the model as *marking the model.*

Semantic considerations

It is important to understand the purpose of the model marking. The first step of model translation is to cast the platform independent model into a platform dependent model. `Micca` operates off of a platform dependent specification of the domain properties and supplies a specific set of mechanisms by which model semantics may be realized as an actual implementation targeted at a specific type of computing platform. In the process of transforming the model into a platform specific description, logical aspects of the model must be made concrete in terms supported by the platform specific model.

For example, models specify the data types of attributes as a set of values. In platform specific terms, we must decide how that value set can be represented as a “C” data type, since our platform programming language is “C”. Usually the decision is easy and obvious, and we choose a data type that can represent the platform independent value set.

It is also the case, that most models do not make full use of all the capablities implied by the platform independent modeling language. This creates situations where optimizations are available to meet the specific use case of the model rather than necessarily providing a general capability that would support any model. For example, if the model activities never access the referential attributes of a class in a descriptive manner, then such attributes need not be stored since the underlying platform specifics supplied by `micca` handle implementing the referential implications of an XUML model.

The importance of model marking is then to gather the information about what the activities of the model actually do so as to make informed decisions about how to map those activities onto the platform specific mechanisms provide by `micca`.

Model metadata

The purpose of closely examining the processing performed by a model is to characterize the computations the model actually performs as opposed to the those computations that the modeling language might support. In this section, we specify the model metadata that needs to be collected during the model marking phase.

Class metadata

For classes, we are interested in determining if any activity in the model does the following:

- Create an instance, either synchronously or asynchronously via a creation event. Note that migrating a subtype in a generalization relationship is conceptually a synchronous delete / create / relate operation (although in the implementation is may happen quite differently).
- Delete an instance either synchronously or asynchronously as a result of entering a final state.
- Query the set of class instances based solely upon the values of identifying attributes.
- Query the set of class instances based upon the values of non-identifying attributes.

Attribute metadata

For class attributes, we wish to know:

- Any attribute whose value is read.
- Any attribute whose value is updated.
- Any mathematically dependent attribute that is updated (this is an analysis error).

Relationship metadata

For relationships, we are interested in the navigation of relationships by the activities in the model. Specifically:

- Creating, deleting or updating the instances of a relationship. This includes both creating instances of association classes and migrating subclass instances of a generalization to a different class. For associations with an associative class, the number of instances of the relationship equals the number of instances of the associative class. Creating or deleting an instance of an associative class creates or deletes an instance of the relationships the associative class realizes. For associations realized by referential attibutes in one of the participants, the number of instances of the relationship equals the number of instances of the class that contains the referential attributes. Creating or deleting an instance of a class which contains referential attributes also modifies the relationship instances. The number of instances of a generalization equals the number of instances of the class which serves as the superclass of the generalization. Migration of the type of a subclass to another type is conceptually a delete / create / re-relate operation.
- The direction of navigation of a relationship between the participating classes.
- Navigating generalization relationships from supertype to some subtype.
- Navigating generalization relationships from a subtype instance to its corresponding supertype instance.

State model metadata

For state models, we are interested in the signaling of events and the reachability of states. Foreach state model record:

- Each event signaled.
- Each polymorphic event signaled.
- Any states, other than final states, that have no outgoing transitions.
- Any states, other than an inital state, that have no incoming transitions.

Marking mechanics

The following is the suggested way to obtain and record the model metadata is:

- Start with a clean printout of the class diagram of the model on the largest size sheet of paper that can be conveniently obtained. Class diagrams for a cleanly presented model should be partitioned into smaller subsystems if the number of classes in the domain does not fit on readily available paper sizes. In practice, Letter size or A size paper is the most readily available and model drawings should take that into account.
- Also obtain clean printouts of the state model diagrams for each class which has a lifecycle state model.
- Read the action language or data flow diagrams of every processing activity in the model. This includes the state activites, domain operations and instance-based operations.
- While reading the model activities, annotate the class and state model diagrams with the information previously presented. For example, when action language statement reads an attribute, place an **R** next to the attribute's name. Similarly, attribute updates can be marked with a **U** or **W** character. The direction of relationship traversal can be annotated as an arrow pointing in the direction of the navigation and parallel to the relationship line on the class diagram. A tick mark next to an event name in a transition matrix can show that the event was signaled by some state activity.

The precise details of the notation used for the marking is not as important as insuring that the previously mentioned information is gathered and clearly indicated. A project may choose to standarize the notation. Do not hesitate to make notes on the diagrams regarding the behavior implied by the activity processing. The results of this model marking analysis directly feeds the next steps in the translation process.

Evaluating the marking

After the marking is completed, it should be evaluated for the following situations. The results of the evaluation should be reported back to the analyst for the domain.

Evaluating attributes

Class attributes play one or more roles in the abstraction of a class. The roles of an attribute may be broadly classified as:

Descriptive attributes
Descriptive attributes characterize class instances by defining some aspect of the instance that pertains to the subject matter of the domain. For example, a **Motor** class might have an attribute of **Torque** which describes the amount of torque the motor produces.

Identifying attributes
Identifying attributes are those attributes which are a part of a class identifier. Each class has one or more identifiers. Each identifier consists of one or more attributes. The set of attributes forming an identifier may *not* be a subset (proper or improper) of the attributes of another identifier of the class (.*i.e.* identifiers must be minimal identifiers). The values of the attributes of an identifier must be distinct among all the instances of a class. For example, a **Product Description** class might have two attributes, **Manufacturer Name** and **Model Name** that, together, are treated as an identifier. This implies that among all instances of **Product Description**, no two instances have the same values for both **Manufacturer Name** and **Model Name**.

Sometimes identifying attributes carry semantically meaningful information, such as the previous example. Some identifying attributes are system generated and have no intrinsic semantic meaning. The distinction is important in that the translation mechanism must create system generated identifying attribute values. Identifying attribute values that are semantically meaningful usually are supplied from outside of the domain. An important distinction between the two types of identifiers relates to where the responsibility for insuring uniqueness lies.

Referential attributes
Referential attributes are those attributes which characterize the relationships between class instances. The real-world associations between classes are formalized by a relationship, either an association or a generalization. Referential attributes have values that are the same as the values of identifying attributes the other participating class. In this way, we know which particular instances are related to each other.

Attributes often serve multiple roles. It is common for identifying attributes to also serve as referential attributes. Also, if the identifier to which a referential attribute refers is descriptive in nature, then the referential attribute is also descriptive, transitively.

Attributes are also classified as being *independent* or *dependent*. An independent attribute may take on any value defined for the data type of the attribute. It's value is not constrained by the value of any other attribute.

A value of a dependent attribute is mathematically related to other attributes. Dependent attributes have a *formula* which is calculated each time they are read and this formula must be supplied by the analyst. It is an analysis error to attempt to write to a dependent attribute. It is also an error to use a dependent attribute as an identifying or referential attribute.

Evaluating descriptive attributes

For attributes which have a descriptive role, the following situations should be noted.

Descriptive attributes neither read nor written
This is probably an analysis error. Attributes that are never accessed don't have much use.

Descriptive attributes read but not written
This situation could be analysis error or it might be that the domain expects the value to be bridged in and depends upon a service domain to keep the value up to date. This should be clear when the bridge specification for the domain is available.

Descriptive attributes written but not read
This situation, like the last, might be an analysis error or there might be a dependency on another domain which is expected to bridge the attribute value out. Again, this situation must be checked against the bridge specification for the domain.

Evaluating identifying attributes

For attributes which have an identifying role, the following situations should be noted.

Identifying attributes that are written
In XUML, updating the value of an identifying attribute is *not* allowed and this is an analysis error. Such an update would change the instance identification. This is an analysis error. The only way in XUML to change an instance is to delete it and then create a new one with different identifying attribute values.

Identifying attributes that are read
We make note of identifying attributes that are read and treated as descriptive attributes because it will have an impact on how the attribute is treated for translation.

Evaluating referential attributes

For attributes which have a referential role, the following situations should be noted.

Referential attributes that are written
In XUML, updating the values of referential attributes is *not* allowed and this is an analysis error. Such an update modifies the instances of the relationship which the referential attributes formalize. XUML action languages provide primitives to modify relationship instances and those must be used to change how instances are related.

Referential attributes that are read
If an attibute serves both referential and descriptive roles, domain activities may read the referential attribute. We make specific note this circumstance as it must be accounted for in the translation.

Evaluating instance creation

Attributes for instance creation
`Micca` insists that the value of all attributes and all related class instances be available at instance creation time. Each site of instance creation, either synchronous or asynchronous, must be evaluated to insure that a value is available for the attribute, the attribute has a default value or the attribute is a dependent attribute.

Evaluating relationships

Relationships that are never navigated
Relationships that are never navigated need not be translated and should be noted to the domain analyst as a potential error. It might be the case that the analyst has included a relationship in the model which will be used in a subsequent iteration. The relationship can be added easily when it is actually used by some activity.

Relationships with a constant number of instances
Relationships that do not change either in number or in the participants are deemed *static*. This property allows for some optimizations in the translation.

Associations that are partial functions
Associations with multiplicity and conditionality of:

a. 0..1 to 0..1
b. 0..* to 0..1
c. 1..* to 0..1

are partial functions and `micca` will insists that these relationships have an association class. If the analyst did not provide an association class for the association then one must be constructed. For relationships with many-to-many multiplicity, the analyst is expected to provide an association class and it is an analysis error if one has not been provided.

This particular discrepance arises from older formulations of XUML modeling that allowed using **NULL** value or a *special* value as a means of specifying partial functions. `Micca` does not use any construct that implies the existance of a **NULL** value.

Evaluating state models

Events that are never signaled
Events that are never signaled are an analysis error.

States with no outgoing transitions
States with no outgoing transitions are terminal states, *i.e.* there is no way to escape the state. This may or may not be an analysis error, but its existance should be noted.

States with no incoming transitions
States with no incoming transitions cannot be reached unless the class instances is created in that state. Any activity specified for a state with on incoming transitions is never executed. This may or may not be an analysis error, and it should be noted to the analyst.

Marking summary

Marking the model as has been describe here is the first preparatory step to translation. The goal of model marking is to enumerate those aspects of the model that are actually used by the domain activities when they execute. Marking also provides a valuable check on the correctness of the analytical models. With the model marking in hand, translation decisions about how to cast the model semantics into platform specifics are easier to make.

Syntax considerations

Before you can start composing the domain specification for `micca`, you need to understand the syntax of the DSL used. The `micca` DSL syntax is documented in the manual pages for `micca`. The easiest way to obtain the man pages is to request `micca` for the documentation. Executing the command:

```
micca -doc
```

causes micca to place a copy of the man pages in a sub-directory of the current directory called, miccadoc [4]. The man pages are in HTML which your favorite browser can render. In this section, we discuss some of the syntax of the DSL used to describe an XUML domain. This section is *not* a detailed enumeration of the commands, the man pages provide the ultimate reference for the DSL. Rather, our intention here is to give you a broad overview and a few tips and hints for the micca DSL.

Micca is written in the Tcl programming language. Tcl is a command-oriented language, similar in some ways to a command shell language that you might use to run programs from a command line terminal. The features of Tcl make it useful for creating domain specific languages, especially those that are intended to appear as declarative specifications. In fact, the DSL for micca is actually a Tcl script. Micca arranges for the script to be executed in an environment where, when the *commands* in the script are executed, the platform specific description of an XUML domain is generated. Little knowledge Tcl syntax is required to work with micca and we cover the points that you need below.

Like many other command-oriented languages, Tcl syntax is concerned with spliting up a command line into *words*. Tcl semantics are entirely tied up in the execution of the commands. Tcl syntax deals with the way a command is split up into words. Word splitting is done based on whitespace. The first word is taken as the command and the remaining words are passed as arguments to the command. This is similar to languages such as the Unix shell. Also similar to other command languages, quoting syntax is used to embed whitespace into an argument. When whitespace characters are required to be embedded in an command argument, the entire argument is quoted. There are only twelve rules of syntax for Tcl. For our purposes using micca, you only need a few and you can pick up what you need to know by reading the examples given here. There are many commands in the Tcl language, but you will only need the ones given in the micca man pages. Once you have assimulated a few syntax rules, you can effectively ignore that you are dealing with a larger programming language.

Quoting

Tcl uses braces ({ }) to quote a string and prevent it from being parsed into words and otherwise interpreted. Everything from the open brace to its *matching* close brace is considered a single argument and the brace characters themselves are discarded.

To describe a domain using the micca DSL, you use the domain command.

.Domain command example

```
domain atc {
    # A set of commands that are used to describe an XUML domain.
}
```

In this example, domain is a command and atc is the name of the domain we wish to specify. A domain is defined by a script of other commands and in this case that script is contained within braces. All the characters between the braces (minus the brace characters themselves) are considered as one argument to the domain command. So, the domain command takes three arguments.

Extending the above example, we can include the class command, which is used to define an XUML class.

.Class command example

```
domain atc {
    # A set of commands that are used to describe an XUML domain.

    # Define a class
    class Controller {
        # A set of commands that describe the characteristics of a class.
    }
}
```

[4]The micca executable contains a number of files that can be copied out. This insures the files are consistent with the version of micca you are running.

So, the `class` command can be used as part of the `domain` script. The `class` command takes an argument that is the name of the class and also takes a script of commands to define the characteristics of the class. Note, the braces used to define the boundary of the `class` command script do *not* affect the determination of the `domain` command argument. This is because the `class` script braces are *matched*, *i.e.* occur as a open brace (`{`) followed, at some point, by a closed brace (`}`).

Quoting hell

At some point you may find a `micca` script being rejected with an error that makes it appear as if nothing is working. Often the reason has to do with quoting. This is sometimes referred to as *quoting hell* since it can be bothersome to find the minor syntax error that sets things right. Matching braces is a good first place to look for finding such mistakes. Text editors usually support some notion of finding a matching brace.

Comments

Comments in Tcl have a quirk to them. The comment character is the hash (`#`) character. The hash and all the remaining characters on the line are discarded. The quirk is that the hash character must come at a point where the beginning of a command word is expected. In practice, it boils down to a few rules:

1. Comments on a line by themselves, with only whitespace preceding the hash character operate as you would expect, *i.e.* the entire line is discarded.
2. Comments after a command on the same line must be preceded by a semicolon (`;`). The semicolon is a command separator (as is a new line).
3. Comments that contain unmatched braces must precede the unmatched brace character with a backslash escape (`\`).

The following example shows the cases.

.Comment example

```
# This is a typical comment on a line by itself.

puts "my name is fred" ; # If there is a command on the same line, use a semicolon.

# This comment contains an umatched open brace, \{
```

Commenting out commands

Commenting out code using the usual comment characters is somewhat problematic because of the quirky nature of Tcl comments. The best approach, if you feel you must comment out some code, is to surround the section with an always false `if` command.

.Commenting out commands example

```
if 0 {
    # Any thing in the enclosing braces is not interpreted.
    # Make sure that all the braces inside the if are matched.

    # For example, here the Motor class definition is not interpreted.
    class Motor {
        # Motor definition commands.
    }
}
```

Line continuation

Commands in Tcl are separated by either newline characters or by semicolons. Usually, a single logical command is simply placed on a line by itself. If the command is longer than you want for a physical line or if you want a particular line layout for emphasis, then a command line can be continued onto the next physical line by ending the line with a backslash character (\). The backslash character must be immediately followed by the newline[5] character for the line continuation to have affect. Be careful if your text editor inserts arbitrary whitespace at the end of a line. Those editors usually have a setting to remove extraneous whitespace at the end of lines. The result is that the backslash character, newline character and any leading whitespace on the following line are replaced with a single space character.

In the following example, the `attribute` command is split over three lines, using two line continuations.

.Line continuation example

```
class Motor {
    attribute\
        Manufacturer int\
        Model int
}
```

Special characters in Tcl and C

Part of the specification of a domain includes “C” code for the activities of the domain. That “C” code is specified in the domain configuration and passed along to the output and eventually to the “C” compiler. Both Tcl and “C” treat some characters as special. The double quotes character (`"`) and brackets (`[]`) have meaning both for Tcl and for “C”. Since `micca` examines any text first, if you want to pass quotes or brackets on to the “C” compiler, they must be enclosed in braces. For example, if a class has an attribute that is a string, it might be defined as:

.Special characters example

```
domain pump {
    class Model {
        attribute Model_Name {char[25]}
        # ... other class commands
    }
}
```

The `attribute` command specifies a class attribute. Here, we define an attribute called, `Model_Name`, and give its “C” data type name as `char[25]` (a 25 byte character array). Notice that the data type specification is in braces to prevent `micca` from interpreting the bracket characters. Should you get an error to the effect of, “unknown command: 25”, you can look for bracket characters not surrounded by braces (in Tcl, brackets are used for command nesting).

Similarly, if you were defining an instance of the `Model` class for the initial instance population you might specify:

.Double quoted characters example

```
population pump {
    class Model {
        instance ar-model Model_Name {"ar-27-np"}
        # ... other population commands
    }
}
```

[5] We refer the the last character as the *newline* character. Different environments have different notions of how the end of an text record is encoded. Tcl automatically takes these differences into account and we are safe to think of text records as simply being newline terminated.

Note in this case, we want to define a Model instance with the name, "ar-27-np". It is important that the quote characters get to the “C” compiler since they have a specific meaning in declaring a string literal. However, double quotes are meaningful to micca and so it is necessary to enclose the value of the Model_Name attribute in braces. A good rule of thumb is to enclose in braces anything you want to insure is passed through untouched to the “C” compiler.

Other Tcl commands

Since a micca description of an XUML domain is actually a Tcl script, you do have the entire power of the Tcl language available when defining a domain. In practice, that power is seldom, if ever, used. The micca DSL was designed to appear as a declarative specification. There is really little use for variables or control structures in describing the semantics of an XUML domain, and using other Tcl language features is discouraged. The commands of the DSL are all that are really needed to describe a domain.

Translating data

The first facet of an XUML model to be translated is the data. It is recommended that all the classes and relationships be defined first and that you insure that micca reads the DSL contining just the data specifications without error. In subsequent additions to the domain configuration, state models and domain activities are added. Getting the foundations in data correct are essential to the remaining parts of the translation and so they must be done first.

A micca domain specification starts with the domain command. For brevity, the examples in this section assume they are placed in the domain script definition of a domain command. The following shows a sketch of how a domain script could be structured. The order of commands in a domain configuration script is arbitrary. Micca reads and parses the entire script before attempting to generate code. Consequently, you need not be concerned about forward declarations or insuring that things are some how *defined* before they are *used*. The micca DSL is a specification language and not an executable language.

Example 0.1 Domain command example

```
domain atc {
    # Domain commands define the characteristics of a domain.
    # These are:
    #       class
    #       association
    #       generalization
    #       domainop
    #       eentity
    #       typealias
    #       interface
    #       prologue
    #       epilogue

    class Controller {
        # Class commands define the platform classes.
        # These are:
        #       attribute
        #       classop
        #       instop
        #       statemodel
        #       polymorphic
        #       constructor
        #       destructor
    }

    association R1 Station 0..1--1 Controller

    # ...
    # and so on, giving the platform specific description of the domain
}
```

How memory is managed

When translating the data portion of an XUML model, it is helpful to understand the platform specifics on which the translated code is run. `Micca` assumes that all domain data is held in the primary memory of a processor and, consequently, the memory is directly addressable. This is a common situation, but it is important to emphasize that there is no assumption of secondary storage such as a disk drive and no presumed persistance as might be provided by a database management system. The execution architecture provide by `micca` is *not* targeted at the class of applications which would require secondary data storage.

`Micca` also insists that the maximum number of instances of each class be defined at compile time. The `micca` run-time does no dynamic memory allocation from a system heap, *i.e.* there are no calls to `malloc()`. Each class has its own pool of memory where instances are either part of the initial instance population or created at run-time. The memory pool is declared as a "C" variable of array type. Initial instances differ from dynamically created instances by providing initial values for each attribute. The memory pool for a class must be sized for the worst case number of instances the class will ever have and that size is known when the "C" code for the translation is compiled. Memory is handled in this way to make it more deterministic.

The most important concepts about how data is managed by `micca` are:

1. Each platform class definition is converted into a "C" structure definition.
2. The instances of a class are stored in an array of structures corresponding to the platform class definition and the size of the array is fixed at compile time.

For programmers who are more accustomed to dynamic memory allocation, worst case memory sizing may seem restrictive or sub-optimal. In practice, it is slightly more work to consider the number of instances a class may have, but rarely are an *arbitrary* number of class instances required. For example, one may consider wanting an arbitrary number of network connections. In truth, operating systems do not allow processes to consume extremely large numbers of resources and even if the resources were available, at some time the ability to service connections runs up against the compute power of the processor. It may seem like an arbitrary number are required, but what if that number were 7 million or 250 million? In practice, the number of instances of a class are typically small. Furthermore, if your application truly makes demands for very large numbers of class instances, them you should choose a different target translation platform than `micca` provides.

Because all data is held in primary memory and the size of that memory is known at compile time, the platform specific model that `micca` targets has two characteristics that are particularly helpful to the implementation.

1. All class instances have an architecturally generated identifier that consists of its address in primary memory. This identifier is used internal to the domain for all references to the class instances.
2. All class instances have an architecturally generated identifier that can be used outside of the domain in order to specify the identity of class instances requesting or receiving services. The external identifier is the index in the storage pool array of a class. It is integer valued and unique within a given class. To uniquely identify an instance outside of a domain, we must know both the class to which the instance belongs and its external identifying integer value. `Micca` automatically generates pre-processor symbols for this information.

Neither of the previous characteristics is particularly profound. Most programs that use data from primary memory use a *pointer* or *object id* as a reference handle. Accessing the class indirectly through a reference is a fundamental mechanism supported by the addressing modes of all modern processors. The mechanism is so common that we often forget that we are imposing an identification scheme on the underlying classes. However, it is important to note that the use of a memory address as a class instance identifier does *not* enforce any identity constraints imposed by identifying attributes of a class. This limitation is considered below when we discuss the processing associated with instance creation.

Translating data types

A class model defines data types for its attributes. The data type definitions must specify a set of values that attributes of the type may have and any non-obvious or uncommon operations on the values. It is rare to see unadorned data types such as, integer, in a well designed class model.

At translation time, “C” data types must be chosen that accomodate the properties of the model data types. `Micca` provides the `typealias` command to specify a name for a “C” data.

There is an inherent ambiguity of types in the “C” language. Because “C” has a `typedef` statement, new type names may be introduced. It is not possible for `micca` to resolve that ambiguity[6]. To break the ambiguity, `micca` requires that all type names end in `_t`, following the usual conventions for standard type names.

For example, if there is a model type for direction which has two values, up and down, then it can be translated as:

```
typealias Direction_t {enum {
    Up, Down}
}
```

Note the braces used to enclose the definition.

In this manner, translations can name the model level types and supply equivalent “C” data types for them.

Note that the type system for model level types is open ended. A model might, for example, define a 3-d vector and a set of operations on 3-d vectors that can be applied to attributes of that type. Such complex types require the translation to supply

Translating classes

In our usage here, we are overloading the term *class* to have different meanings. A model class and a platform-specific class are two distinct things. They are related. They are counterparts of each other in the model and platform realms, but they are logically different entities and that distinction must be clear in a Translator’s thinking. In this section, we discuss how to transform the logic of a model class into the specifications of an platform-specific class. To be clear, we will refer to *model class* when we are discussing the class characteristics as they are presented in the XUML data model and *platform class* when we are discussing how the `micca` implementation of the model level concepts is realized.

Purely descriptive attributes

Attributes that are descriptive in nature and play no other role in the model class are declared using `attribute` commands. For example, a model class might appear as:

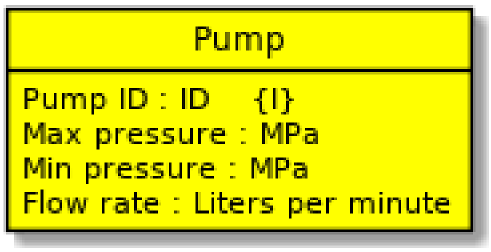


Figure 2: Pump model class

The three attributes, **Max pressure**, **Min pressure**, and **Flow rate** are descriptive of the characteristics of a **Pump** and serve no other role in the model class. They are specified in the platform class using the `attribute` commmand.

Pump platform class

[6]At least not without parsing all the “C” content.

```
class Pump {
    attribute Max_pressure MPa_t
    attribute Min_pressure MPa_t
    attribute Flow_rate LpM_t
}
```

Note
Class names and attribute names must be valid “C” identifiers[a]. These names are passed along to the “C” compiler to name structures and structure members. Analysts may name entities as they wish for the clarity of the analytical model. Translation must take the names and make them suitable for the target platform. In this case, it usually suffices to replace any characters that cannot be in a “C” identifier with the underscore (_) character. It is important **not** to obfuscate the correspondence between the attribute name in the model and the one used in the implementation. We also assume that the MPa_t and LpM_t type aliases have been defined as discussed previously.

[a]A valid “C” identifier must start with a letter or underscore and be followed by any number of letters, underscores or decimal digits. Note, some “C” compilers may limit the number of characters in an identifier that are considered significant. Most modern “C” compilers to not have such limitations.

Default attribute values

An attribute may be given a default value. If the analyst provides a default value specification in the model it can be used as a default value for the translation by given the attribute command the -default option.

For the previous example, if the class had appeared as:

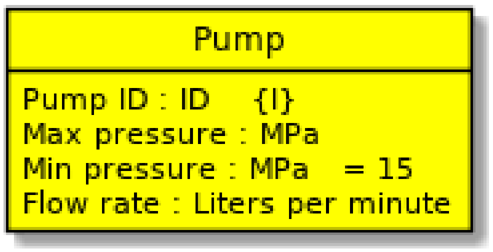


Figure 3: Pump model class with default attribute

We can specify the default value as:

Pump platform class

```
class Pump {
    attribute Max_pressure MPa_t
    attribute Min_pressure MPa_t -default 15
    attribute Flow_rate LpM_t
}
```

Attributes which have a default value can be omitted when creating an instance, either at run-time or as part of the initial instance population.

Zero initialized attributes

The attribute command also takes an optional -zeroinit option which gives the attribute a default value of all bits being zero. Like an attribute with a default value, an attribute declared as -zeroinit need not have a value provided when it is created.

Dependent attributes

An `attribute` may be modeled as *mathematically dependent*. For those attributes, the analysis model must provide a *formula* in action language that computes the value of the attribute. The formula must then be translated into “C”. The following is a brief example of a dependent attribute translation.

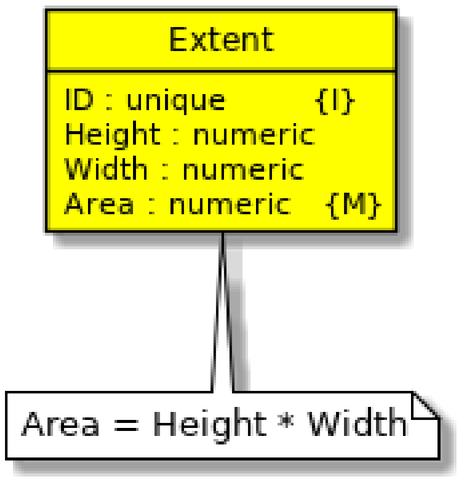


Figure 4: Model class with dependent attribute

We can specify the attribute as dependent and supply the code necessary to compute the formula.

Pump platform class

```
class Extent {
    attribute Height unsigned
    attribute Width unsigned
    attribute Area -dependent {
        *Area = self->Height * self->Width ;
    }
}
```

We consider how the “C” code is specified in a following section.

Identifying attributes

Since `micca` is providing an identifier for each instance, our goal is to eliminate identifying attributes whenever possible. Attributes that are system generated and serve only an identifying or referential role may simply be elided. In the previous examples, the model attributes, **Pump.Pump ID** and **Extent.ID** were not present in the corresponding specifications for the platform class.

In the following situations identifying attributes may **not** be eliminated:

- The attribute serves a descriptive role for the processing and it’s value is read and used in a computation.
- The attribute value is needed to insure an identity constraint.

In both these cases, the attribute, with an appropriate “C” data type, must be included in the platform class definition.

Referential attributes

Once again, the platform specific choice of `micca` to provide an identifier for a class instance, means that identifier can be used in place of referential attributes to manage relationships. As we see in the following section, `micca` is able to provide the necessary storage and operations on class instance references to implement relationships. So, again our intent is to eliminate referential attributes when possible.

There is a situation where this is *not* possible.

- If the model reads a referential attribute and treats it descriptively, *i.e.* uses the attribute value in some computation, then it is necessary to translate that attribute access into a relationship navigation that ultimately terminates at the identifier value to which the attribute refers.

Consider the following class model fragment.

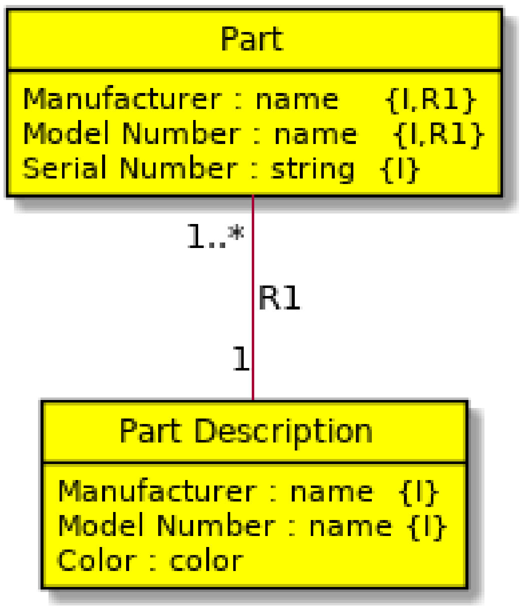


Figure 5: Model class with referential attribute

Assume that there is an activity in the model which determines the serial numbers of all the **Parts** manufactured by the Acme company. That activity is a search of the **Part** instances finding those whose **Manufacturer** attribute equals "Acme". In this situation, **Part.Manufacturer** can be eliminated from the platform class, but **Part Description.Manufacturer** must be retained. The search must then be coded to navigate the **R1** association to obtain the value of the **Manufacturer** attribute. We know that referential attributes refer to identifying attributes and by following the relationship (perhaps multiple relationships) we must eventually arrive at an identifying attribute value.

So the above situation might be translated as:

```
class Part_Description {
    attribute Manufacturer {char[25]}
    attribute Model_Number {char[25]}
    attribute Color unsigned
}

class Part {
    attribute Serial_Number {char[25]}
    classop MRT_InstSet findByManufacturer {manuf {char const *}} {
    <%  Part instset matchedParts                              %>
    <%  Part foreachInstance partRef                           %>
            // This statement navigates R1 from Part to Part Description
    <%      instance partRef findOneRelated descRef R1         %>
            if (strcmp(descRef->Manufacturer, manuf) == 0) {
    <%          instset matchedParts add partRef               %>
            }
    <%  end                                                    %>
        return matchedParts ;
    }
}

association R1 Part 1..*--1 Part_Description
```

For the moment, don't be startled by the contents of the `classop`. We will cover translating actions into "C" later. For now, we want to emphasize that the search for **Part** instances to match a given manufacture's name does not use a **Part** attribute. Rather, the value of the **Manufacturer** is obtained from the **Part Description** by navigating the **R1** association. So, no **Manufacturer** attribute is stored in a **Part** and any time the **Manufacture** attribute value is needed, it is obtained by navigating the relationship.

Translating relationships

Model relationships represent constraints on the number and conditionality of how class instances are associated with each other in the real world. Fundamentally, the multiplicity and conditionality of a relationship restrict membership in the underlying set that class instances represent. In a model, relationships are realized by referential attributes having values equal to those of identifying attributes. Since `micca` supplies a unique identifier for class instances, those identifiers are used to realize relationships for platform classes.

`Micca` supports translating the relationships by:

- Generating data structures and providing operations to handle the references for instances that participate in a relationship.
- Verifying at run-time that the constraints implied by the relationship are not violated by the execution of domain activities.

Direction of a relationship traversal

In a model, relationships do not have an inherent *direction* associated with them. It is possible to navigate the relationship in either direction starting with an appropriate instance of a participating class. However, for translation purposes it is convenient to give relationships a direction. The concept of a direction of a relationship shortens the amount of specification necessary to describe how navigating from one instance to other instances across a relationship. It is not generally necessary to specify which class the navigation arrives at since that information is known to `micca` by the specifications in the platform model. A direction concept also helps disambiguate some situations that arise in reflexive associations.

The direction of a relationship is determined by:

- For simple associations where referential attributes in one class are used to realize the association, the forward direction is from the referring class (*i.e.* the class that contains the referential attributes) to the referenced class.
- For generalization relationships, the forward direction is from subclass to superclass, which is also the direction of referring class to referenced class.
- For relationships realized by associative classes, there is no inherent direction and a direction is chosen by the Translator.

`Micca` uses some additional syntax to specify the direction of relationship traversal. The syntax to navigate a relationship in the *forward* direction, is simply the name of the relationship, *e.g.* `R42`. The syntax to navigate in the *backward* direction prepends a tilde (~) character to the name of the relationship, *e.g.* `~R42`.

The following cases arise:

- For simple non-reflexive associations, forward and backward traversal are unambiguous and results in selecting instances of the participating classes.
- For generalizations, forward navigation always yields *exactly one* instance of the superclass. Backward navigation must include the name of a participating subclass. Consider a generalization, called **R19**, which has a superclass named **Lamp** and a subclass named **Table_Lamp**. Navigation from an instance of **Lamp** to an instance of **Table_Lamp** is specified as `{~R19 Table_Lamp}`. The result of such navigation yields *at most one* instance of **Table_Lamp**.
- For associative classes, four situation arise.
  - Navigating in the forward direction from one class to its corresponding class uses the usual relationship name, *e.g.* `R13`.
  - Navigating in the backward direction between the two participating classes uses the tilde prefixed name, *e.g.* `~R13`.
  - Navigating forward to the associative class is specified by including the associative class name, *e.g.* `{R13 Product_Selection}`.
  - Navigating backward to the associative class is specified by both a tilde prefixed relationship name and the name of the associative class, *e.g.* `{~R13 Product_Selection}`.

Translating simple associations

Consider the following model fragment:

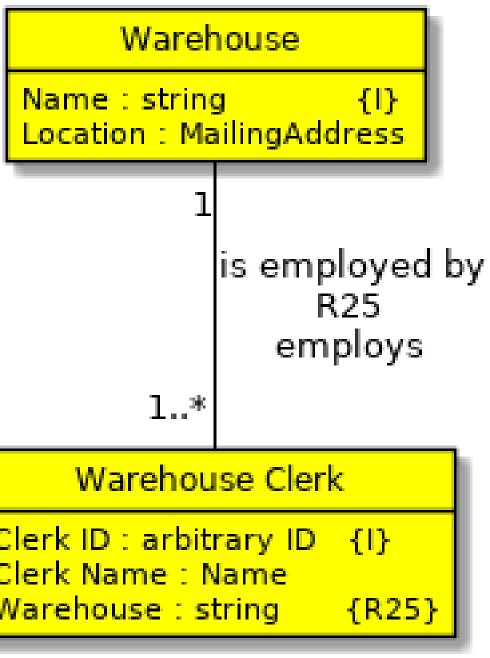


Figure 6: Simple association

This fragment shows an *at least one* to *one* association between a **Warehouse Clerk** and a **Warehouse**. This situation would be translated as:

```
class Warehouse {
    attribute Name {char[50]}
    attribute Location {char[50]}
}

class Warehouse_Clerk {
    attribute Clerk_ID unsigned
    attribute Clerk_Name {char[50]}

    # Referring class for R25
}

association R25 Warehouse_Clerk 1..*--1 Warehouse
```

The `association` command defines the characteristics of the association between classes. The syntax of the multiplicity and conditionality, *i.e.* **1..*--1**, is intended to be mnemonic of the notation used in the XUML class diagram. Note this argument has no embedded whitespace. The order of the classes in the command is in the forward direction of traversal. The class which contains the referential attributes (**Warehouse_Clerk**) appears first in the command arguments. In this example, navigating from an instance of **Warehouse_Clerk** across **R25** yields *exactly one* instance of **Warehouse** and navigating from an instance of **Warehouse** across **~R25** yields *at least one* instance of **Warehouse**.

Translating associative classes

Some associations require an associative class to realize the mapping between participating instances. An association class is required for:

- Associations that are *many* to *many* in multiplicity.
- Associations that represent partial functions, namely, **0..1—0..1**, **0..*--0..1**, and **1..*--0..1**.
- Associations that have descriptive attributes describing properties of the association itself, *i.e.* associations with attributes that are not strictly referential.

Instances of the association class correspond directly to instances of the relationship. Association class instances directly enumerate the mapping between the instances sets of the participating classes To translate an association class, the `association` command takes an `-associator` option. Consider the following model fragment:

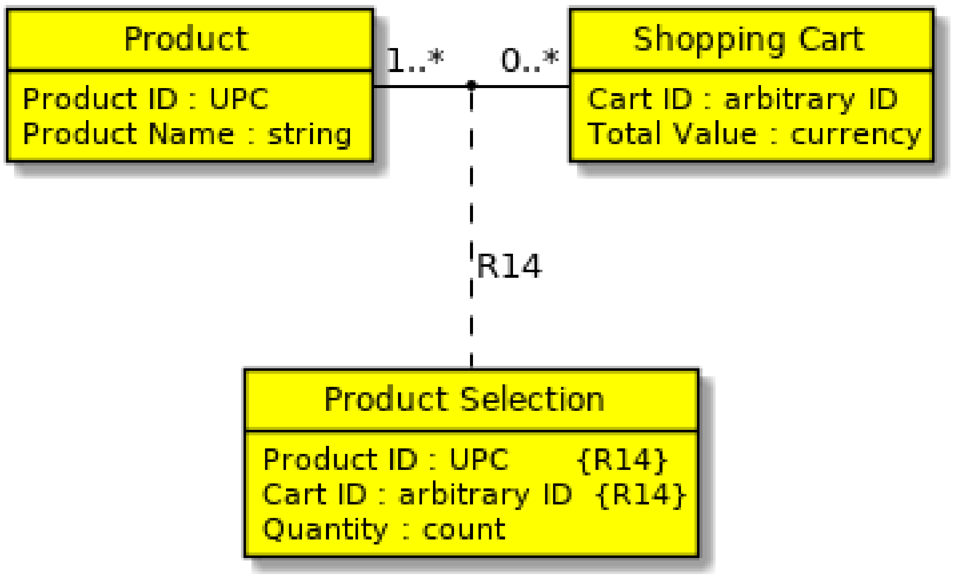


Figure 7: Association class

This association would be translated as:

```
class Product_Selection {
    attribute Quantity unsigned
}

association R14 -associator Product_Selection Product 1..*--0..* Shopping_Cart
```

Translating generalizations

Consider the following model fragment:

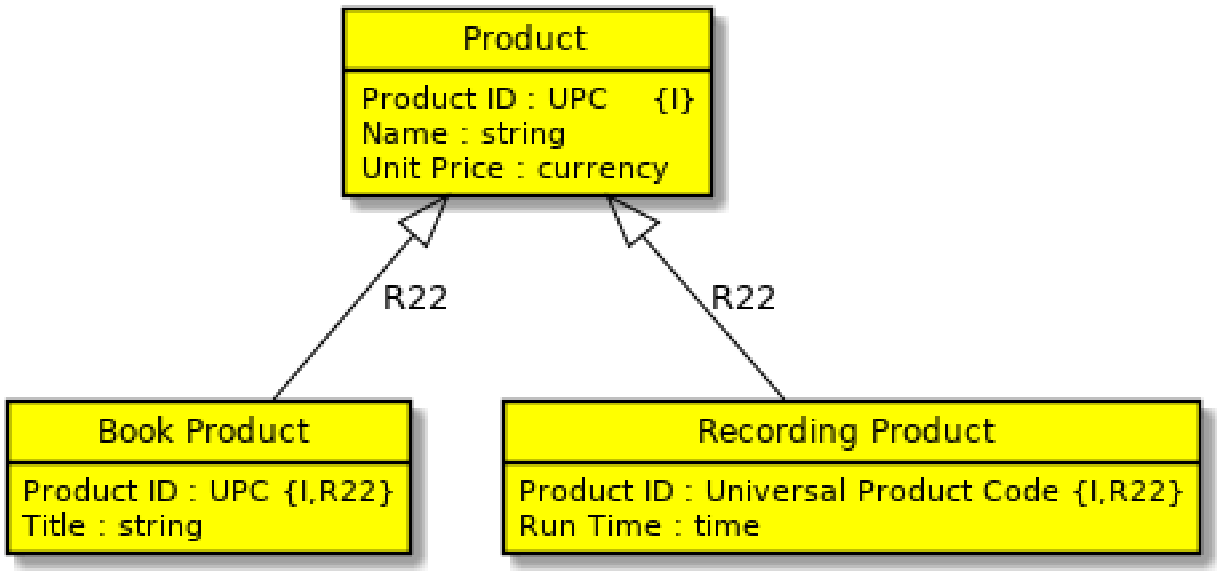


Figure 8: Product generalization

This is a generalization relationship. In XUML, a generalization does *not* represent inheritance. Rather, it represents a disjoint set partitioning. Strictly speaking, the relationship graphic should be annotated with `{disjoint, complete}`, but since this is the only type of generalization used in XUML, the annotation is usually dropped as graphical clutter.

This situation would be translated as:

```
class Product {
    attribute Name {char[25]}
    attribute Unit_Price Money_t

    # superclass of R22
}

class Book_Product {
    attribute Title {char[25]}

    # subclass of R22
}

class Recording_Product {
    attribute Run_Time Duration_t

    # subclass of R22
}

generalization R22 Product Book_Product Recording_Product
```

i.e. we use the `generalization` command giving the name of the relationship, the name of the superclass and the names of the subclasses which participate in the generalization.

`Micca` provides two ways to store instances that participate in a generalization:

Reference
: In this technique, subclass instances have their own storage pools and a reference to the superclass is embedded in the subclass data structure. Navigating from subclass to superclass is accomplished by de-referencing the superclass reference.

Union
: In this technique, the superclass is defined to contain, as part of its data structure, a union of the data structures of the subclass of the generalization. There is no separate instance storage pool for the subclasses. Navigating from the subclass to the superclass is accomplished by pointer arithmetic.

In the previous example, if we had chosen to implement **R22** in a union, we would have written:

```
generalization R22 -union Product Book_Product Recording_Product
```

The trade-offs for the two types of storage are:

- Generalizations implemented by reference are applicable to all situations. The cost is the memory in the subclass to store the reference to the superclass and the memory in the superclass to store a reference to the subclass. There is a separate memory pool for each subclass and these must be allocated. If there are no instances of a particular subclass, then its memory pool is not used.
- Generalizations implemented by union save the storage of pointer values. The memory pool of the superclass serves as memory for subclass instances. However, if there is a large discrepancy in the size of subclass instances, then memory may not be as well utilized as in the case of references.

The computation to navigate the relationship is negligible in both cases.

For most generalizations, the `-union` implementation works well. There is one case where it cannot be used. If a subclass is subject to a compound generalization, *i.e.* a class serves a subclass role in multiple independent generalizations, then only one of the generalizations may use the `-union` option and the others must use `-reference` (which is the default if neither option is specified). This situation is rare.

Static associations

For associations which are *static*, *i.e.* the population of relation instances does not change over the course of running the domain, the `association` command takes a `-static` option. This option changes the relationship storage mechanism for associations with multiplicity greater than one to be an array of pointers of fixed length. Normally, association of multiplicity greater than one use linked lists to store the references of the relationship. The `-static` option can reduce memory usage and simplify the code generated to navigate the relationship toward the *many* side.

Populating the data

The final step in translating the data model for a domain is to specify its population. Populating the data model consists of specifying *initial instances* and the amount of storage to allocate for each platform class. Initial instances are those class instances that exist before the domain runs. As discussed previously, each platform class has its own storage pool for instances. In addition to initial instances, you can specify the number of additional slots to allocate to the storage pool. The total number of instances is then the sum of the initial instances and the number of additional slots allocated for run-time usage. Note, if an initial instance is deleted at run-time, then its storage slot becomes available for subsequent use by an instance created at run-time.

It is advisable to keep the population of a domain in a file separate from its configuration. As translated domains are tested, it is convenient to use a variety of different populations to drive the code execution paths for testing. `Micca` will read multiple files for a domain, so keeping test populations separate from production populations is convenient.

The analytical materials supplied to the translation must include a population of the data model. The recommended practice is to supply the population in tabular notation, one set of class instances per page, similar to a page in a spreadsheet. Values for all attributes are supplied. It is possible to specify the model level population purely by data values.

The `population` command is used to define the storage requirements and any initial instances for the domain. One of the few command ordering requirements for `micca` is that the `population` command must follow the domain definition given by the `domain` command. The following shows a brief sketch of a population.

Example 0.2 Domain population example

```
population atc {
    # Population commands define the data values in a domain
    # These are:
    #       class
    #       assigner

    class Controller {
        # Class commands the instances of a single class.
        # These are:
        #       instance
        #       table
        #       allocate
    }

    # ...
    # and so on, giving the population of the domain
}
```

The `population` command takes the name of the domain to be populated and a script of population commands to specify the characteristics of the population. For brevity, in the following examples we assume the commands are contained in a `population` script.

Initial instance names

Part of the process of defining a population involves not only specifying values for the attributes but also insuring the relationship references are specified. As we have previous discussed, referential attributes are usually elided from the model class when formulating the platform class. We can deduce the relationship references from the model population by examining the values of the referential attributes and finding the corresponding values in the other participating class of the relationship. But for platform classes, we are taking a different view of relationship references and using referential attribute values is not an option open to us. So, each initial instance is given a *name*. Each instance name must be unique within the class of the instance. The instance name serves as a means of specifying to which instances relationships refer. The instance name serves as a surrogate for the referential attribute values. So when we specify instances, the relationship number, *e.g.* **R27**, and the name of an instance are used to specify which instance are related.

Initial instances

`Micca` provides two commands to specify initial instances. The `instance` command specifies values for a single class instance. The `table` command specfies multiple instances in a tabular layout.

Referring to a previous example, assume the analysis model has specified the following initial instance population. Here the data is presented in tabullar form and the population is entirely specified by the data values of the model attributes.

Table 1: Part Description Population

Manufacturer {I}	Model Number {I}	Color
"Acme"	"S27"	22
"Sunshine"	"B42"	47

Table 2: Part Population

Manufacturer {I,R1}	Model Number {I,R1}	Serial Number
"Acme"	"S27"	"A001"
"Acme"	"S27"	"A002"
"Sunshine"	"B42"	"B034"
"Sunshine"	"B42"	"B037"

For the platform classes, the referential attributes in **Part** were elided. Remaining are the **Serial_Number** attribute and the fact that **Part** is the referring class in **R1**. This situation may be translated as follows.

```
class Part_Description {
    instance screw Manufacturer {"Acme"} Model_Number {"A27"} Color 22
    instance bolt Manufacturer {"Sunshine"} Model_Number {"S42"} Color 47
}

class Part {
    table       {Serial_Number  R1}\
    s1          {"A001"         screw}\
    s2          {"A002"         screw}\
    b1          {"B034"         bolt}\
    b30         {"B037"       bolt}
}
```

Here we have chosen to use the `instance` command for populating the Part_Description class and the `table` command for the Part class. Note that the referential attributes in the model class (*i.e.* **Manufacturer** and **Model Number**) have been replaced by specifying that **R1** is satisfied by referencing the **screw** or **bolt** named instances from **Part_Description**.

Class storage allocation

The `allocate` command is use to specify an additional number of class instance storage slots. The total number of instances for a given class is the number of initial instances plus the number given in the `allocate` command.

For the previous example, we can allocate 10 additional instances memory slots for the **Part_Description** class as follows:

```
class Part_Description {
    instance screw Manufacturer Acme Model_Number A27 Color red
    instance bolt Manufacturer Sunshine Model_Number S42 Color blue

    allocate 10
}
```

The maximum number of **Part_Description** instances that can exist simultaneously is 12, two defined as initial instances and 10 allocated for run-time creation. Note, that at run time, there is no distinction between initial instances and unallocated instances. So, deleting an instance that was specified as an initial instance makes its memory available to be used to create an instance at run-time.

Populating associative classes

Specifying the population of an associative class requires a different format. Recall that an associative class has references to the two other classes that participate in the association. So when we give an initial value for references, we must give a list of values. Consider the previous example of the **Product_Selection** class. It population might appear as:

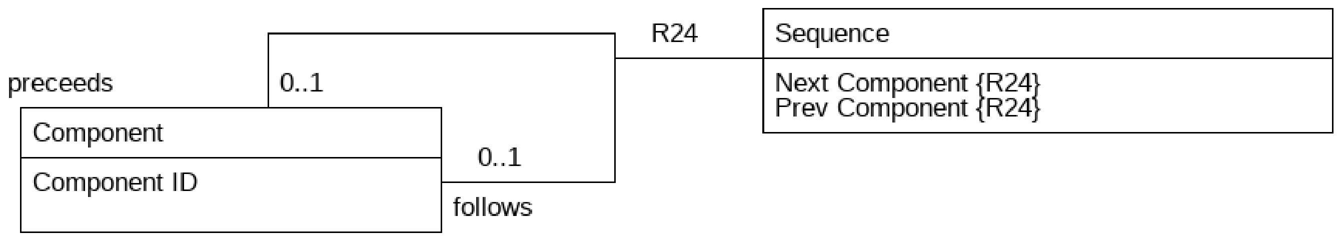
```
class Product_Selection {
    instance ps1 R14 {Product prod1 Shopping_Cart sc1} Quantity 1
    instance ps2 R14 {Product prod2 Shopping_Cart sc2} Quantity 2

    # ...
}
```

So for an associative class, specifing the references needed for a relationship involves specifying a set of pairs giving the class name and the instance name of each participating class instance.

Populating associative classes in reflexive relationships

The previous scheme for populating associative classes does not work when the association is reflexive. In a reflexive association, both instances participating in the association are of the same class. We cannot use class names alone to distinguish the reference. In a model, the verb phrases that describe the association are used to determine the direction of navigation. In the case of reflexive associative classes, we use `forward` and `backward` to specify how the references are stored. Thus it is necessary to associate the verb phrases of the model to the words `forward` and `backward`. It does not matter which direction is deemed forward or backward as long as the association of which verb phrase of the model is deemed forward and which is deemed backward is used consistently.

Consider the following simple reflexive association.



If we adopt the convention that *preceeds* is associated to `backward` and "follows" is associated to `forward`, then we could generate a population for the **Sequence** class as:

```
class Component {
    instance c1 Component_ID 1
    instance c2 Component_ID 2
    instance c3 Component_ID 3
    instance c4 Component_ID 4
}

class Sequence {
    instance s1 R24 {backward c1 forward c2}
    instance s2 R24 {backward c2 forward c3}
    instance s3 R24 {backward c3 forward c4}
}
```

This population creates a chain of **Component** instances. Navigating from **c2** in the forward direction will select the **c3** instance. Navigating from **c1** in the backward direction selects no instances.

Translating dynamics

The second facet of the model to translate is that of the dynamics. Translating the dynamics of the model requires the least amount of transformation of the three model facets. The semantics of the Moore state models used in XUML are directly mapped to `micca` commands.

How execution sequencing works

Just as in translating data, it is helpful to understand the platform specifics of execution sequencing when constructing the translation of the state models. In this section, we give a overview of the major concepts in how `micca` sequences execution. This section is not a complete description of the execution run-time. That may be found in the `micca` literate program document. Here we only present the concepts that help in understanding the intent of the translation commands.

Broadly speaking, there are three forms of execution sequencing that happen when a `micca` translated model runs.

Synchronous execution
A domain activity may invoke an ordinary "C" function. Control is transferred to the function and when it returns execution continues where it left off. Synchronous execution is the usual, familiar, sequential flow of control. We don't discuss this any further since it is fully supported by "C" language primitives. When invoking a function in "C", the compiler arranges to pass control to the function and arranges for the function to return back to the subsequent code. The `micca` run-time is not involved.

Asynchronous execution
A domain activity executes until it needs to wait on other actions in the domain. Other actions use *events* to signal synchronization points and these signals can cause an activity to resume executing in a particular location. Asynchronous execution is accomplished by a state machine dispatch mechanism. We explain the details of how the `micca` run-time performs state machine dispatch in the following section.

Preemptive execution
During the execution of any domain activity, it is possible to suspend the execution of the activity, preemptively, run a body of code, and return to the preempted activity to continue. Preemptive execution is non-deterministic in that you don't know when it may happen and have only limited means to control periods of time when it is *not* allowed to happen. Preemptive execution corresponds to *interrupts* when running natively on a computer or *signals* [7] in a POSIX environment. The `micca` run-time provides a means to synchronize preemptive execution with the other forms. We discuss this aspect when we take up bridging of domains.

The `micca` run-time supports a single thread of execution on which both synchronous and asynchronous domain activities execute. The nature of the execution architecture is completely event-driven. Any domain activity that blocks waiting for some condition, prevents all but preemptive execution from running. This approach is well suited to applications which react to external conditions by performing a time-bound computation. Clearly, there are certain types of applications for which this execution model is not well suited. The `micca` run-time is not *universal* in the sense of applicable to every conceivable application, but an event-driven, reactive execution model accomodates a broad class of applications well.

How state machine dispatch works

The execution rules of XUML require that domain activities exhibit *run-to-completion* semantics. This means the execution of an activity *appears* as if it runs to its natural end without being preempted by other activities that may also be running. When state models interact, they *signal* an *event* to indicate the need to synchronize with another condition. Consequently when an activity signals an event, it continues to run and the act of signaling has no perceived effect on current execution of the activity. Since `micca` only has a single execution context on which anything runs, there is no real parallel execution that must accounted for. The `micca` run-time uses the conventional mechanism of *queuing* signaled events to allow the signaling activity to continue to its completion. Only after an activity which signals an event completes is an event considered for dispatch.

Event dispatch follows Moore semantics. A transition table is used as a transfer function. The current state of the instance receiving the event and the event itself determine the new state into which the receiving instance is placed. Upon entry into a

[7] The POSIX version of the `micca` run-time treats the change of state of a file descriptor in the same manner as signals.

state, any activity associated with the state is executed. The transition table is a matrix containing one row for each state of the state model and one column for each event to which the state model responds. The same transition table is used for all instances of a given class since all instances of a class exhibit the same behavior. The entry at the intersection of state rows and event columns is the new state into which the transition is made. XUML semantics also support the notion that an event can be ignored or is an error condition and these situations are also encoded in the transition matrix cell.

The `micca` run-time distinguishes between two contexts in which an event may be signaled.

1. The event is signaled from within a state activity, *i.e.* one state machine is signaling an event to another state machine (possibly signaling an event to itself).
2. The event is signaled outside a state activity either as a signal from outside the domain or as a result of requesting the system to signal the event at some future time (*i.e.* a *delayed* signal).

This distinction defines the concept of a *thread of control* as follows:

- An event signaled from outside the domain starts a thread of control.
- The thread of control continues until all the events (if any) signaled by state activities initiated by the thread of control are delivered.
- After all the events precipitated by the thread of control event are dispatched, the next thread of control event is considered.

So, the `micca` run-time actually as two event queues. One queue is used to store events that come from outside of a state machine context and start a thread of control The other queue is used to store events that arise during the execution of a single thread of control.

All the effects of an event which starts a thread of control are realized before starting another thread of control. This is important because at the end of each thread of control, the `micca` run-time will check the referential integrity of the data model. The associations and generalizations defined by the class model and encoded in the data translation define a set of constraints on whether and how many class instances are related to each other. Any activity that either creates or deletes instances or otherwise rearranges the related instances must insure that the data model is consistent with the constraints implied by the associations and generalizations of the class model. This means that any activity that modifies relationships must:

a. perform any compensitory action required to make the data model consistent within in the same activity that modified the relationship instances, or
b. signal an event that when dispatched will eventually cause some state activity to run which makes the data model consistent before the thread of control ends.

The archetypical example of data consistency is two classes which participate in a one-to-one unconditional association. Creating an instance of one class necessarily implies that an instance of the other class must also be created. The creation of the second class instance is the compensitory action required to make the data model consistent. That instance creation may happen in the same state activity or an event may be signaled which results in another state activity eventually creating the required class instance.

Translating state models

Note at this stage we are *not* translating the action language of the activities associated with any state model. State model activities should be left empty at this time so we can focus on getting the correct structure of states and transitions. We discuss the translation of the processing in a subsequent section.

For classes with a state model, add a `statemodel` command to its class definition:

```
class LightBulb {
    # light bulb attributes, etc.

    statemodel {
        # Statements that define the states and transitions
        # of the class state model.
    }
}
```

For brevity in the following examples, we assume the `statemodel` command is contained within the appropriate `class` definition script.

Two things must be defined for a state model.

1. The states must be defined. The code to execute when the state is entered is part of that definition, however we defer that until the skeleton of the state model is constructed.
2. The transitions between states must be associated to the events which cause those transitions. When an event is received, it causes:
   a. A transition to a state (possibly the same state receiving the event).
   b. Is ignored. This is given the symbol, **IG**.
   c. Is an error conditions because it is logically impossible for the event to be received in the state, *i.e.* it can't happen. This is given the symbol **CH**.

Defining states

To specify a state, `micca` uses the `state` command.

Consider a simple light bulb.

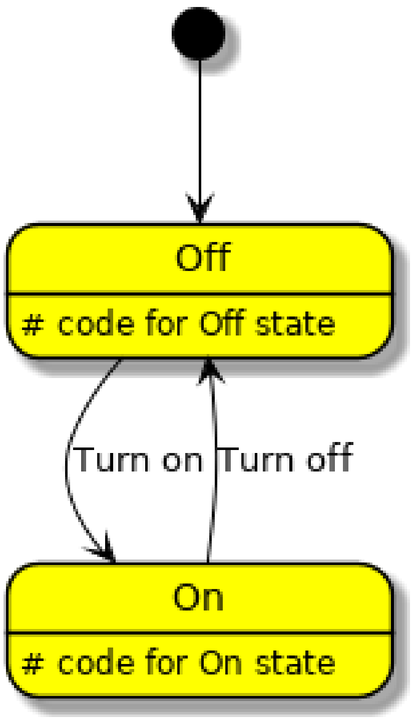


Figure 9: Light Bulb State Model

The micca commands for the states in this diagram are as follows.

```
statemodel {
    state Off {} {
        # code to execute when the Off state is entered.
    }

    state On {} {
        # code to execute when the On state is entered.
    }
}
```

Here we have defined two states, `Off` and `On`. The `state` command takes three arguments:

1. The name of the state.
2. The parameters of the state.
3. The body of “C” code to execute when the state is entered.

State activity parameters are carried along in the event that causes the transition *into* the state. We discuss states with parameters more below. Here there are no parameters to the state activity, so we specify them as empty ({}).

Defining transitions

The other aspect of a state model translation is to specify all the state transitions. The starting point for translating the state transitions is, naturally enough, the transition matrix of each state model. The analytical model must provide a complete state transition matrix in addition to the state model diagram for each state model. Consider the transition matrix for our simple light bulb model. Each state has a row in the matrix. Each event has a column in the matrix.

Table 3: Light Bulb State Transitions

State / Event	Turn off	Turn on
Off	IG	On
On	Off	IG

The micca command to specify transitions is `transition`. The `transition` command specifies one cell of the transition matrix. We can complete the translation of this state model as follows.

```
statemodel {
    state Off {} {
        # code to execute when the Off state is entered.
    }

    state On {} {
        # code to execute when the On state is entered.
    }

    transition Off - Turn_off -> IG
    transition Off - Turn_on -> On
    transition On - Turn_off -> Off
    transition On - Turn_on -> IG
}
```

Note

Note that micca does not have any ordering requirements on the state or transition commands. You can put all the transition command first or mix them around in any way you wish. One common organization is to list the *outgoing* transitions for a state immediately after the state command defining the state. Project should decide their own conventions for specifying the states and transitions.

Default transitions

A state model diagram typically only shows transitions from one state to another. It does not show ignored (IG) or can't happen (CH) pseudo-transitions. In our translation of the example, we explicitly stated all the IG transitions. This can be tedious if there are a lot of these types of transitions to encode. So, micca allows you to set a default transition. Any transition *not* mentioned in a transition command is assigned the default transition.

Normally, the default transition is **CH** (for some very good reasons a default of **CH** catches many errors and generally the default transition should be kept as **CH**). However, some state models, assigners in particular, ignore most events that do not cause an explicit transition to a state. Our light bulb example also ignores events. To change the default transition, use the defaulttrans command. Our light bulb example might have been written as:

```
statemodel {
    defaulttrans IG

    state Off {} {
        # code to execute when the Off state is entered.
    }
    transition Off - Turn_on -> On ; # ❶

    state On {} {
        # code to execute when the On state is entered.
    }
    transition On - Turn_off -> Off
}
```

❶ Note the spaces around the - and -> characters. This is *syntactic sugar* intended to be mnemonic of the transition and its presence is required.

Some find this manner of specifying the state model clearer. Note also the defaulttrans command may appear anywhere and need not be the first command in the state model definition.

Initial state

All state models must specify an initial state in which to start. For our light bulb example, the initial state is **Off**. This is indicated by the state model graphic having a black circle (the pseudo-initial state) connected to the **Off** state by an unlabeled arrow.

The initialstate command is used to specify the initial starting state for a state model. If no initialstate command is specified in the state model definition, then micca chooses the first state defined as the initial state. It is usually wise to specify the initial state, if only for documentation purposes and to make sure things don't go wrong if the order of state definitions is changed. Our light bulb example might have been written as:

```
statemodel {
    initialstate Off
    defaulttrans IG

    state Off {} {
        # code to execute when the Off state is entered.
    }
    transition Off - Turn_on -> On

    state On {} {
        # code to execute when the On state is entered.
    }
    transition On - Turn_off -> Off
}
```

Initial state semantics

How initial states get handled is fundamentally an analysis concern. However, it is worth repeating here how activities associated with initial states are handled. Sometimes, problems with initial state activity execution are assumed to be translation problems.

In XUML, there are two ways to create class instances:

1. Synchronously, where a state activity requests a class instance be created and that instance exists when the invocation of the create request completes. If the class has a state model, the newly created instance is placed in an initial state according to:

    a. the default initial state defined in the `statemodel` script, or

    b. the *initial state requested by the state activity* performing the synchronous creation. It is an option when the instance creation operation is invoked, to request a specific state into which the newly created instance is to be placed. This feature is *not* often used, but it does exist and it has useful cases.

    Regardless of how the initial state of a synchronously created instance is determined, the state activity associated with the initial state **is not executed**.

2. Asynchronously, where a state activity sends a creation event. After signaling a creation event, the state activity that performed the signaling continues executing and the new class instance is created at some later time. When the creation event is eventually dispatched, the class instance is created in the *pseudo-initial state*, and the creation event causes a transition out of that state. This would be indicated on a state model diagram by outgoing arrows from the pseudo-initial state that are labeled with an event name. There is only one pseudo-initial state, but there may be many events that cause a transition out of it (and any event which does not cause a transition out of the pseudo-initial state will cause a **CH** transition if it is used as a creation event). When the transition out of the pseudo-initial state happens a new state is entered, and the **state activity of the new state is executed**.

The fundamental rule is that a state activity is only executed when a transition causes the state to be entered. Creation events cause a transition from the pseudo-initial state into a new state and therefore the activity of the new state is executed. Synchronous instance creation does not cause any transition, but only sets the value of the initial state and thereby determines the manner in which *subsequent* events are handled.

We discuss how to specify creation events below.

Final states

The lifecycle of some classes is over when they enter a certain state. Consider the state model below.

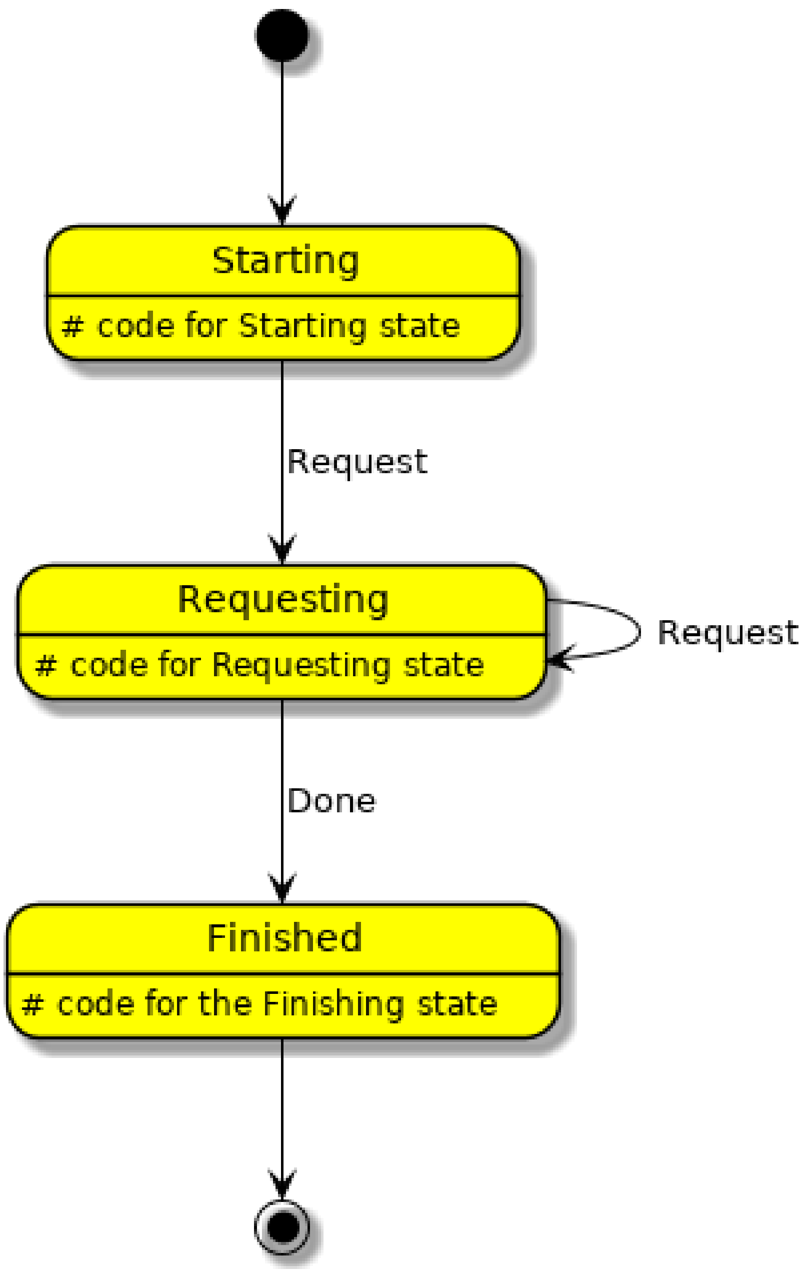

Starting
# code for Starting state
Request
Requesting
# code for Requesting state
Request
Done
Finished
# code for the Finishing state

Figure 10: State Model with Final State

When the **Finished** state is entered and after its state activity has been executed, it transitions (with no event label on the arrow) to a final state which is shown by a black circle with a white halo. Any instance of a class entering a final state is automatically deleted by the run-time code. To indicate a final state, `micca` uses the `final` command.

```
statemodel {
    # state and transition definitions
    # ...

    final Finished
}
```

It is possible to have multiple final states. It is also possible to use the `final` command several times or to list several states in the invocation of the `final` command.

States with parameters

So far, none of the states we have shown has had any parameters. Events may carry argument values that are available to state activity. The parameter signature of any event which causes a transition *into* a state must match the parameter signature of the state activity. This is a corollary to the way Moore state machines operate since the activity is executed upon entry into the state and any parameters must be immediately available to the activity. Normally, this is an analysis concern and up to the analyst to get the two signatures to match correctly. However, `micca` does detect such errors.

Consider our previous state model with the addition that a **Request** event carries with it a count of the number of items to request and a size of items to request.

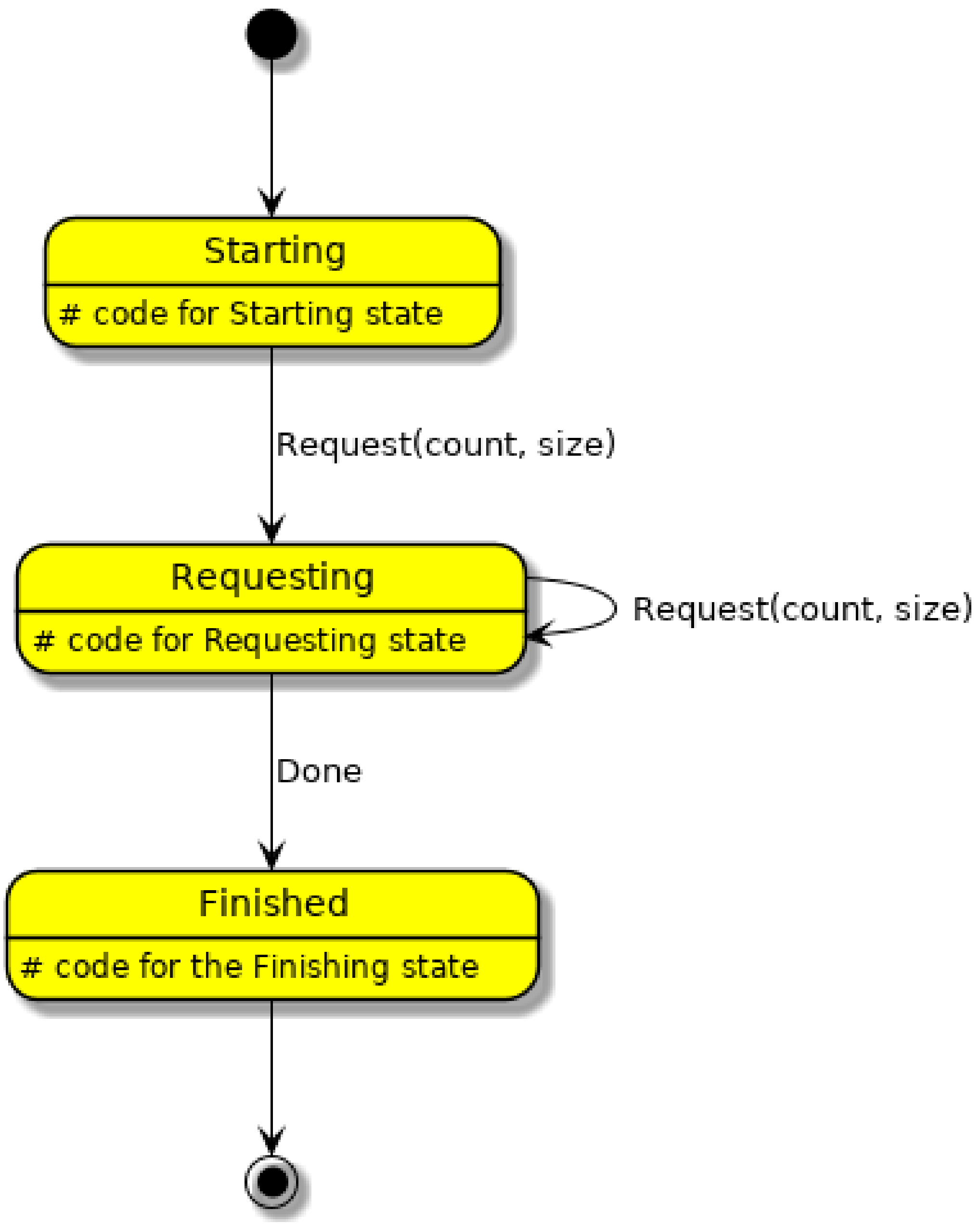

Starting
# code for Starting state
Request(count, size)
Requesting
# code for Requesting state
Request(count, size)
Done
Finished
# code for the Finishing state

Figure 11: State Model with State Parameters

We must specify the signature of the **Requesting** state to show the arguments it receives when entered.

```
statemodel {
    initialstate Starting
    defaulttrans CH
    final Finished

    state Starting {} {
        # code for the Starting state
    }
    transition Starting - Request -> Requesting
    state Requesting {count int size int} {
        # code for the Requesting state
        # two variables, count and size are defined as int's
        # and are available to the code in the state activity
    }
    transition Requesting - Done -> Finished

    state Finished {} {
        # code for the Finished state
    }
}
```

So parameters to a state are given in the second argument of the `state` command. That argument is a list of alternating argument name / argument type pairs. The argument name will show up as a variable in the generated “C” code. The argument type is a “C” type name, in this case, both arguments were `int` types. The signature of the **Requesting** state is (in “C” terms) `(int, int)`.

Event definitions

You might have noticed that there have not been any definitions of the events. Usually, there is no need to define the events of the state model. `Micca` will find all the event names mentioned in the `transition` statements and just use them.

However, sometimes it is useful to be able to specify an event. This is usually the case when an event carries arguments. So, in the previous example, we could have specified:

```
statemodel {
    # commands to specify the state model
    # ...

    event Request count int size int

    # other commands to specify the state model
    # ...
}
```

Event specifications are usually not necessary because `micca` can recognize that a state signature with two integer arguments is the same signature regardless of what the argument names are. It is the data types of the arguments that determine the signature of an event, *not* the names given to the arguments.

One reason for defining an event signature has to do with bridging, which we discuss later. When you are constructing an event to signal via a bridge and that event has arguments, explicitly defining the event in the state model makes the naming of the arguments explicit in the bridge code. It’s an advanced topic and we discuss it later when we talk about bridging.

Polymorphic events

In XUML, polymorphism arises when classes participate in a generalization relationship and the subclasses in the relationship have lifecycles modeled by state models. We do not discuss all the aspects of polymorphic events here since our concern is translation.

The usual convention in model diagrams marks polymorphic events by pre-pending an asterisk, *i.e.* ***Run**. This is done as an aid in keeping track of things in the model. `Micca` does *not* use such conventions.

`Micca` requires that a polymorphic event be defined as part of the class specification for the superclass. The event can be given any name (as long as it can be turned into a “C” identifier). When an event by the same name is consumed (*i.e.* appears in a transition statement) in a leaf state model, `micca` knows, because the name is the same, that the event was polymorphic.

So if a model contains a class, **Torpedo**, which defines a polymorphic event, **Fire** which takes a parameter of *speed*, it would be specified as:

```
class Torpedo {
    # commands to specify the attributes, etc. of a Torpedo
    # ...

    polymorphic Fire speed float

    # ...
    # other commands to specify a Torpedo
}
```

Defining a polymorphic event is similar in syntax to defining an ordinary event. The main difference is that the polymorphic event is defined as part of the `class` command script and *not* as part of the `statemodel` [8]. A subclass consumes the polymorphic event, when an event of the same name as the polymorphic event is used in a `transition` command within a `statemodel` script for the subclass.

`Micca` understands that in multi-level generalizations, polymorphic events not consumed by intermediate level subclasses are deferred to lower level subclasses. It also understands all the other rules about polymorphic events and insists the specification be correct.

Creation events

Previously, we briefly discussed creation events in the context of the semantics of initial states. A creation event is one which causes an outgoing transition from the pseudo-initial state. A creation event appears in a `transition` command just as any other event. The difference is that the pseudo-initial state has the special name of, `@`. For example:

```
statemodel {
    # commands to define the state model of the class
    # ...

    transition @ - Go -> Starting

    # ...
    # other commands to define the state model of the class
}
```

In this example, **Go** is a creation event that causes a transition from the pseudo-initial state (`@`) to the **Starting** state. There is nothing particularly special about the **Go** event in this example. It is signaled to a newly created instance when an asynchronous instance creation is requested in an activity. The name of the creation event must be supplied as part of the process of asynchronously creating an instance. The event may have parameters or not depending upon what the analytical model has specified. It is also possible that the **Go** event is used in transitions other than the one from the pseudo-initial state.

Assigner state models

The nature of some associations between classes is competitive. The concurrency rules of XUML imply that instances of a competing relationship must be created by serializing the actions involved. This is accomplished by associating a state model to the relationship itself and those state models are called *assigner* state models. The archetypal example for assigner state models is the assignment of store clerks to service customers:

Consider the following model fragment.

[8]Polymorphic commands have no effect on any state behavior of the superclass. The superclass may have its own state model which is unaffected by any polymorphic events defined in the superclass. Polymorphic events only affect subclass behavior.

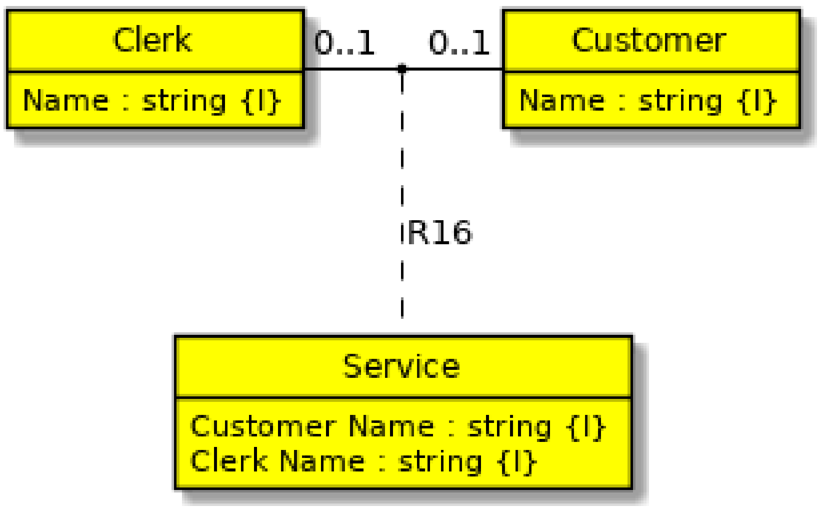


Figure 12: Clerk / Customer Association

Since the state model is bound to the association, it is specified as part of the association definition. For this example:

```
domain operations {
    # commands to specify domain characteristics
    # ...

    association R16 -associator Service Clerk 0..1-0..1 Customer {
        statemodel {
            # state model definition for the R16 assigner
        }
    }

    # ...
    # other commands to specify domain characteristics
}
```

The commands used to specify an assigner state model are the same as for a class state model, so we do not repeat them here. The difference is that the `statemodel` command and its definition script are part of the definition of the association itself and *not* to any particular class (indeed in this case, the **Service** class may have its own state model).

`Micca` will arrange for the assigner to exist before the domain starts to run and will place the assigner in the inital state as specified by the `statemodel` script.

Multi-assigner models

Usually, there is only a single instance of an assigner. This was the case in the previous example. However, some competitive relationships are more complicated and are partitioned by the instances of another class. Extending our previous example, it may be the case that a **Clerk** may only service **Customers** when the **Customer** is in the same department where the **Clerk** works, *i.e.* a **Clerk** is not allowed to service **Customers** outside of his/her department. In that case, we would need an instance of the assigner for each instance of **Department**. This is a multi-assigner situation. The `identifyby` command, used in the script of an association, specifies that the assigner is a multi-assigner and gives the class which partitions the instances.

Extending our previous example, we would have:

```
domain operations {
    # commands to specify domain characteristics
    # ...

    association R16 -associator Service Clerk 0..1-0..1 Customer {
        identifyby Department ; # >>> Here, we indicate a multi-assigner
        statemodel {
            # state model definition for the R16 assigner
        }
    }

    # ...
    # other commands to specify domain characteristics
}
```

Multi-assigner population

Since there are multiple instances of a multi-assigner, we must define a population for those instances. This is accomplished with the `assigner` command within a population. Again expanding upon our previous example:

```
population operations {
    # commands to specify populations of classes
    # ...

    assigner R16 {
        instance sports_service sports_department
        instance apparel_service apparel_department
        instance garden_service garden_department
    }

    # ...
    # other population commands
}
```

The `assigner` command and the script containing the `instance` commands define three assigners for service associated with three departments where **Clerks** work. Here, `sport_department` *et.al* are names of instances of the **Department** class which is the partitioning class for the multiple assigners.

Note that for single assigners, no population specification is required since `micca` already knows how many assigner instances to create (*i.e.* one).

Translating processing

The processing facet of a model is the third and most difficult to translate. Translating domain actions requires the Translator to reformulate the intent of an action language or data flow diagram into “C” language statements. The semantic gap between these representations is larger than that of either the dynamics facet (where it is virtually zero) or the data facet (where there are a relatively small set of rules and guidelines).

Action language — what action language?

The first quandry we encounter is to determine how domain activities are described to the Translator. Since we don’t transfer any machine readable content from the analysis effort, there is no requirement for a formal, parsable description of how activities are expressed. Many action languages have been defined. It is possible the analyst decided to express the processing facet as a data flow diagram with associated descriptions of the processes. It could be that the domain actions are written in a more informal pseudo-code. Regardless of how the actions are expressed, projects are strongly encouraged to adopt one particular way to express domain actions for all the domains in the project. Multiple types of action specifications only confuses the matter.

Practice shows, counter to what you might think, the exact manner of expressing the model processing actions is not as important is might seem. In well designed models, most activities are small and highly focused in what they must accomplish. Complex mathematical calculations are usually delegated to specialized libraries. For example, there is little reason to model and write action language for a Fourier transform. Fourier transforms are a well researched and implemented world unto themselves and would yield little insight into the larger problem. Model code is usually more concerned with when and on what data would a transform be applied. What is important is that the actions be expressed in a form that is clear and unambiguous.

In our discussion and examples here, we will used a variety of ways to show model actions and how they are translated using `micca`. Sometimes a pseudo-code narrative is sufficient. However, we will tend to use an action language called, *Scrall*, the most. Scrall is an open source action language designed to leverage the distinct semantics of XUML.[9]

Categories of processing

Broadly speaking, domain actions fall into two categories:

Expression evaluation and flow of control
: Actions need to be able to compute things, such as adding or multiplying values. Actions need to be able to test values and change the flow of execution. These are basic operations required to perform any reasonable computation.

Model level actions
: Action languages have primitives that affect model level components. For example, signaling an event to an instance is a model level action that is usually expressed as a basic construct in the action language.

Examined from a different perspective, these two categories of actions can also be seen as:

- Constructs that are directly supported by the implementation language ("C" in our case) and for which there is a direct mapping from the action onto the implementation language.
- Constructs that are built using using data and code constructs of the target language to implement operations on model level elements.

Consider the following action snippet:

```
(Altitude < Minimum) ? Pull up -> me
```

The intent of the action is to test whether the altitude value is less than some minimum and if so, then to signal the **Pull up** event to the instance running the action, *i.e.* **me**. A translation of these ideas to "C" might look like:

```
unsigned Altitude ;
unsigned Minimum ;

// Some code to assign values to both Altitude and Miniumum.
// Perhaps something like:
// Altitude = self->Altitude ;
// Minimum = self->Minimum ;

if (Altitude < Minumum) {
    // OK, now what? How do I signal an event?
}
```

[9] Need a reference here!

Signaling an event to an instance is not a base concept in “C” and there is no direct, handy statement in the language that will do it. Consequently, micca has devised data structures and supplies code as part of the run-time library to perform that operation. There is nothing mysterious about that code. It is ordinary “C” code and can be viewed like any other “C” implementation. The function to signal events is called mrt_SignalEvent() and it accepts arguments required to accomplish the model level action of sending an event signal to a class instance. It is possible to translate the activity by writing the “C” code that invokes mrt_SignalEvent() directly, and that would cause the signal to be generated.

The difficulty of coding directly against the run-time library interfaces is:

- There is a large cognitive burden to understand the details of the run-time API and supply the correct argument values.
- Direct coding against the run-time API strongly couples the model translation to the run-time interface. Future changes to the run-time that might be required to correct errors or extend its functionality could break a translation.

The micca solution to this is to provide an *embeddable* command language that generates the code needed for model level actions. One significant advantage of the embedded commands is that micca has access to the platform specific model when the code generation happens. Consequently, error diagnosis is much improved. If you attempt to signal an event to a class instance and that class does not accept such an event, micca is able to diagnose that condition and issue an appropriate error message. This helps detect errors earlier in the process rather than relying on the “C” compiler.

Micca embedded command language

An activity in a micca translation, *e.g.* a state activity, appears as ordinary “C” code with other commands embedded into the code. The commands are embedded by surrounding them with the <% and %> characters. Our previous example would appear as:

```
unsigned Altitude ;
unsigned Minimum ;

// some code to assign values to both Altitude and Miniumum

if (Altitude < Minumum) {
<%  my signal Pull_up       %>
}
```

There are a few rules about the embedded commands:

- The command syntax is the same as for micca itself, *i.e.* it is Tcl syntax.
- Leading or trailing whitespace around the embedded command does not matter. So you may format the embedded commands in whatever way you find clearest. The above formatting tries to keep the <% and %> markers out of the way to make indentation more apparent.
- Embedded commands **do not nest**.
- Some embedded commands generate multiple “C” code statements and include all the necessary semicolon punctuation. Other embedded commands are intended to be used in expressions and so do not place semicolons in the output. The reference documentation for the embedded command language states which behavior each command has, but you will quickly know the difference by the intent of each embedded command.
- During the code generation, the embedded command is removed from the input and the semantically equivalent “C” code is placed in the output at the same location.
- Since it is unknown how many “C” statements a command may generate, all statements should be treated as *compound* statements and surrounded by curly braces (as we did in the if test of the previous example).

Generated code

Most people who perform a model translation using micca initially are curious about the contents of the generated “C” code. It usually takes some experience to trust the tooling and examining the code can garner that trust. In the end, everything must end up as “C” code and we should be able to read that code. What follows is a sample showing a state activity and its generated code.

First, we show the micca source. This is a state activity, so it would be part of a state model.

```
statemodel {
    # Code for states, events, et.al.

    state Autocycle_Session_Deactivate {} {
    <%  my update Deactivate true                %>
    <%  my signal Deactivate                     %>
    }
}
```

And the following is the generated code snippet.

```
static void
Autocycle_Session_Deactivate(
    struct Autocycle_Session *const self)
// <%  my update Deactivate true              %>
// <%  my signal Deactivate                   %>
{
    MRT_INSTRUMENT_ENTRY
    // instance self update Deactivate true
    self->Deactivate = true ;
    // instance self signal Deactivate {}
    mrt_SignalEvent(6, self, self, NULL, 0) ; // Deactivate
}
```

There are several points of interest:

- State activities are turned into ordinary “C” functions. Most names are made file static to avoid cluttering the global space of names and to lessen the chance of a name conflict. State activities to not return values. All of their side effects are recorded in the class data.
- Each state activity function has an argument called, self, which is a reference to the class instance for which the action is being run.
- The contents of the state action is placed as a comment at the beginning of the function body.
- MRT_INSTRUMENT_ENTRY is a macro that can be used for logging and instrumentation output or it can be defined to be empty and no additional code is generated.
- As each embedded command is expanded, a comment containing the command is placed in the “C” code. In this example, my signal is the same as instance self. The my prefix is just a shorthand.
- Updating an attribute is done via pointer indirection on the instance using structure members defined as part of the class’s attributes.
- Signaling an event happens by invoking a function in the run-time with the proper argument values. In this case the **Deactivate** event has been numbered 6, and the event is self-directed with no parameters. Micca generates all the numbers used to encode things such as states and events.
- The output code is neatly formatted. Although the code is not examined often, during debugging you may be setting breakpoints in the generated code and it is helpful if it is readable and there are clues about how the generated code relates back to the model.

We do not show any other generated code in this document. How the code is generated is discussed in the literate source document for micca. Code generation test cases are available for those interested in the precise details of how the generated output appears.

As we stated previously, model actions are a mix of ordinary “C” code, used for expression evaluation and flow of control, plus embedded commands to perform model level operations. In the end, the result looks like “C” but with what can be imagined as unusual looking function invocations to perform the model level semantics.

Categories of domain activities

The processing accomplished by a model occurs in very specific places. In general, model activities do not sequence the overall flow of execution. That responsibility is given to the Model Execution domain, which in this case is the run-time code supplied by micca. The following sections describe where the processing in a domain is found.

Regardless of category of an activity, the translation process is the same. The same embedded commands are available.

State activities

The primary processing of a domain occurs in activities associated with states. State activities cannot be directly invoked. Rather, they are executed when an event is received which causes a transition into a state. Micca converts the “C” code specified for a state activity into an ordinary “C” function of file static scope. State activities have an automatically declared self variable that references the class instance receiving the event.

Domain operations

Domain operations are actvities which provide an exterally callable interface to the domain. Micca converts domain operations into “C” function of external scope.

Instance operations

Instance operations are defined within a class and represent common code factored into a “C” function. Instance operations have an automatically declared self variable that references the class instance upon which the operation is invoked.

Class operations

Micca allows the definition of an operation associated with a particular class. Class operations do not arise from the XUML model. They represent code that the Translator has added to the implementation for some particular purpose. For example, identity constraints are usually enforced by a class operation and this topic is covered in a subsequent section.

Referring to instances

Since many operations are directed at instances, *e.g.* access an attribute, we must have some way to refer to an instance. In this section, we describe the ways provided by micca to refer to instances of platform classes.

Instance references

Operations on class instances are accomplished by being given or by finding an *instance reference*. To access the attributes, navigate a relationship or signal event, you must have some way to identify the class instance which is affected. This is accomplished with an *instance reference variable*. An instance reference variable is simply a variable whose value refers to a particular class instance. There is no notion of an instance reference variable that refers to an instance of an arbitrary class. Instance references are typed according to the class to which the instance belongs.

Micca provides an embedded command to declare an instance reference variable. Each platform class defined in the domain has a command defined for it. For example, if there is a Part class in the domain, then the following declares an instance reference variable for a Part.

```
<%  Part refvar screw  %>
```

This command will declare a “C” variable named, screw, which can hold a value that refers to an instance of the platform class, Part. This construct is not frequently used since most embedded commands will automatically declare any necessary “C” variables that hold the command result. However, sometimes you will want to control the scope of the variable by using this command.

In practice, an instance reference variable is actually a “C” pointer and so carries with it all the usual admonitions about pointers. In particular, micca does use an instance reference variable value of NULL to indicate that the variable does not refer to any instance. For example, searching for a particular instance or navigating a conditional relationship can yield an instance reference variable whose value is NULL.

The **self** instance reference variable

For state activities and instance operations, micca automatically declares and initializes an instance reference variable called, self. The self instance reference variable refers to the instance receiving the event or to which the instance operation applies.

The micca embedded command language uses the instance command for all operations that use an instance reference variable to perform a model level action on an instance.

For example:

```
<%  instance red_part attr Weight   %>
```

accesses the **Weight** attribute of the instance referenced by the red_part variable.

As a shorthand, embedded commands that refer to the instance self can simply use the my command. For example, rather than write:

```
<%  instance self update Color red  %>
```

one can write:

```
<%  my update Color red  %>
```

Instance reference sets

Micca also supports the concept of an *instance reference set*. As its name implies, an instance reference set can refer to multiple instances (and possibly zero). Like instance reference variables, instance reference sets are typed to the class to which the instances belong. An example of declaring an instance reference set is:

```
<%  Part instset red_parts   %>
```

Like instance reference variables, instance reference sets are declared using the embedded command which is named the same as a platform class. Instance reference sets are implemented using bit fields. This is a compact representation, but the maximum number of elements in an instance reference set must be known at compile time. The default value is 128, *i.e.* an instance reference set can only describe the set of instances for a class that has at most 128 instances. This value can be changed by defining the MRT_INSTANCE_SET_SIZE pre-processor macro to a different number. We cover all the operations on instance reference sets in a subsequent section.

Notice that we have defined two different data types to hold instance references:

1. An instance reference variable can hold a reference to only a single class instance.
2. An instance reference set can hold references to multiple class instances.

In theory, an instance reference variable is redundant since an instance reference set is capable of holding a single instance reference. In practice, operations and storage for an instance reference variable are significantly less costly, computationally. Since dealing with a single instance is also a frequent situation, it is a benefit to the implementation to provide both a single reference and a set of references and allow the translator to choose which works best for a given situation.

Accessing instance attributes

Because accessing attributes is so common, the `micca` run-time provides several ways to access them. The choice of which access technique to use can be made based on the intent of the action language.

The `attr` command

The most basic form of attribute access is to use the `attr` command. This embedded command generates code to access one named attribute of the instance. For example, to change the `Speed` attibute of an instance referenced by a variable called `motor`, we could write:

```
<% instance motor attr Speed %> = 25 ;
```

The expansion of the `attr` embedded command yields a "C" expression that is suitable for using as either an `lvalue` (as above) or an `rvalue`. As an `rvalue` it might appear as:

```
unsigned new_speed = <% instance motor attr Speed %> + 10 ;
```

Assigning attributes to variables

Sometimes an activity need access to several attributes or the computation on an attribute is most clearly shown when it assigned to a local variable. The `assign` embedded command will place attribute values into ordinary "C" variables of the appropriate type.

Let's say we interested in computing an area from the width and height of a rectangle:

```
<%  my assign Width Height  %>
unsigned area = Width * Height ;
```

In this example, the **Width** and **Height** attribute values have been placed into ordinary "C" variable also named `Width` and `Height`. The types of the variable are the same types as the attributes.

Sometimes it is inconvenient to name the variable holding attribute values the same as the attribute itself. This might lead to a naming conflict. In those cases, the attribute arguments to the `assign` command can be two elements lists: the first item is the attribute name and the second is the name of the "C" variable into which the attribute value is assigned. The previous example could be written as:

```
<%  my assign {Width myWidth} {Height myHeight}  %>
unsigned area = myWidth * myHeight ;
```

In both cases, the generated code handles the details of declaring the "C" variables and assigning the attributes to them.

Another important use for the `assign` command is to handle dependent attributes. Recall that a dependent attribute is a read-only attribute whose value is determined by a formula. The `assign` command is the **only** way to obtain the value of a dependent attribute (using other embedded commands on a dependent attribute will result in an error). The `assign` command is invoked on a dependent attribute in the same was as for any other attribute. The `micca` code generator will recognize that the attribute is of the dependent variety and arrange to evaluate the formula, placing the result into a "C" variable.

Updating attributes

The counterpart to the assign command is the update command. The update command generates code to update the value of one or more attributes. For example:

```
<%  my assign Width Height  %>
unsigned area = Width * Height ;
Width += 10 ;
Height /= 10 ;
<%  my update Width Width Height Height  %>
```

The update command takes attribute name / value pairs and generates code to assign the attribute value into the referenced instance. In the above case, the name of the attribute and the name of a “C” variable are the same. The attribute value need not be a variable. Any “C” *rvalue* is good. For example:

```
<%  instance pump update Can_run_status false  %>
```

In this case, **Can_run_status** is a boolean attribute whose value was set to false.

You can even use more complex *rvalue* expressions as long as you enclose them in braces.

```
<%  instance color update Red {20 * 3}  %>
```

Direct reference

Finally, it is no secret that class instances are structures which have members named the same as the attribute names of the class. Given an instance reference variable, you may read or write an attribute by directly accessing the structure member. For example:

```
if (pump->Max_pressure > 3000) {
<%  instance pump signal Overpressure  %>
}
```

This approach has the disadvantage that if the instance to which pump refers does not have an attribute named Max_pressure, then the error is not found until compile time. For simple cases such as this example, using the attr embedded command is preferable.

However, since embedded commands do not nest, there will be times when direct attribute access is the only alternative. We will see these cases below when discussing relationship navigation.

Signaling events

Signaling events is a common domain activity. It appears in action languages in a variety of syntactic forms such as:

```
Run -> motor
```

or

```
signal Run to motor
```

where **Run** is the name of an event and `motor` is an instance reference variable or an instance reference set.

Simple event signaling is translated into `micca` as:

```
<%  instance motor signal Run       %>
```

There is no embedded `micca` command to signal a set of instances. You must construct a loop to accomplish that. In the following example, we construct an instance reference set by navigating a relationship looking for a particular status and then signal all the set members.

```
<%  Motor instset idle_motors                                            %>
<%  my selectRelatedWhere idle_motors mtr {mtr->Status == Idle_Status} R1   %>
<%  instset idle_motors foreachInstance motor                            %>
<%      instance motor signal Run                                        %>
<%  end                                                                  %>
```

It is often the case that action language statements define the set of instances by some set of conditions and then operate on the accumulated set as a separate statement. This is what we saw in the previous example. The `selectRelatedWhere` command was used to accumulate a set of instances and then each accumulated instance is visited in a loop and signaled.

If there is no other use for the instance reference set, (*i.e.* `idle_motors` in this example), then it is not necessary to go the to trouble of accumulating it. The above example could be written to iterate across the instances selecting the ones that match some criteria and then operating on each selected instance as it is found. For example, it could have been rewritten as:

```
<%  my foreachRelatedWhere motor {motor->Status == Idle_Status} R1          %>
<%      instance motor signal Run                                        %>
<%  end                                                                  %>
```

Signaling events with parameters

Sometimes events have parameters and to signal them implies that you must supply arguments at run time. Suppose from the previous example that the **Run** command requires a `speed` parameter. In that case, you can signal the event as:

```
<%  instance motor signal Run speed 3.7             %>
```

Event parameters are given as parameter name / value pairs. Parameter names must match the names of the event signature. All parameters must be supplied with a value, but the order of the name / value pairs is not significant. `Micca` will arrange to deliver the argument values correctly.

Delayed signals

A delayed signal is a request from some domain activity to the model execution environment to signal an event on its behalf at some time in the future. The embedded `micca` command to accomplish a delayed signal is similar to that for an immediately dispatched signal. Suppose we wanted to signal a motor to run at one second in the future:

```
<%  instance motor delaysignal 1000 Run speed 3.7   %.
```

There is a different command word, `delaysignal`, which is followed by the delay time in milliseconds (`1000` in this example). Otherwise, the remaining command arguments are the same as for an ordinary signal.

How delayed signals work

The micca run-time handles the delayed dispatch, so there are some details of delaying a signal that you need to understand to insure a faithful translation.

- There may be only one outstanding delayed signal for any given event between any sending / receiving pair of instances (which might be the same instance). This is the same as saying that delayed events are identified by the name of the event, the sending instance and the receiving instance. This is a rule of the XUML execution semantics.
- The unit of time is milliseconds, so there is no way to have really high frequency signal delays.
- The delay time is the **minimum** time before the signal will be delivered. Software execution can introduce small additional delays or jitter.
- Delivery of any delayed signal starts a new thread of control.
- It is acceptable to have a delay time of zero. This means the event is queued immediately and, when it is dispatched, will start a new thread of control.
- The run-time interprets any attempt to have a duplicate delayed signal, *i.e.* an attempt to create a new delayed signal for the same event between the same sending / receiving pair as a request to cancel the current delayed signal and post a new one at the new delay interval. This is usually the most convenient outcome.

The micca run-time consumes a single timing resource on the target platform to maintain a delayed event queue to manage the outstanding delayed signals. For the POSIX version of the run-time, SIGALRM is used to time the delayed event queue. For other platforms, a physical timer peripheral is used. As you can see from the previous rules of how delayed signals work, their use is best suited for timeouts and relatively course grained timing. Applications which require precise, repeatable, periodic or high frequency time notifications should use actual hardware timers or other system resources to generate such events.

Canceling delayed signals

Delayed signals may be cancelled. In action language you might find:

```
Run !-> motor
```

or something like

```
cancel Run to motor
```

These types of action language constructs are translated as:

```
<%  instance motor canceldelay Run              %>
```

Here, motor is an instance reference variable name holding a reference to the receiver of the event. By default, the sender of the event is assumed to be the activity invoking the canceldelay command, *i.e.* self. If that is not the case, and an activity is canceling a delayed signal sent by different instance, the an instance variable name holding an instance reference to the sender of the delayed signal can be supplied. In this example, if the original Run event was signaled by an instance other than self, then the cancelation might be written as:

```
<%  instance motor canceldelay Run controller   %>
```

where controller is an instance reference variable refering to the instance orignally sending the signal.

Self-directed delayed signals may be canceled using the my command, as in:

```
<%  my canceldelay Stop              %>
```

There are a few other rules about canceling delayed events:

- Only events that were signaled as delayed signals may be canceled.
- It is guaranteed that a canceled signal is not delivered.
- It is not an error to cancel a signal that has already been delivered or was never sent.

Remaining time for delayed signals

The other allowed operation on delayed signals is to request the amount of time remaining before the signal is delivered. It is sometimes necessary to adjust the timing of a delayed signal. Continuing with the above examples, we can determine the amount of time before a delayed signal is dispatched by:

```
MRT_DelayTime t = <% instance motor delayremaining Run %> ;
```

Note that the `delayremaining` embedded command creates an *rvalue* of type `MRT_DelayTime`. The following code sequence adds 100 ms to the delay for the **Run** event:

```
MRT_DelayTime runTime = <% instance motor delayremaining Run %> ;
if (runTime != 0) {
    runTime += 100 ;
<%  instance motor delaysignal runTime Run        %>
}
```

Note the test of `runTime` for zero. A zero time returned from `delayremaining` indicates that the delayed event has already been dispatched or never existed. In this case, notice that we are taking advantage of the semantics of `delaysignal` by simply requesting a new one at a new delay time, knowing that the current one is canceled automatically.

Navigating relationships

Domain activities frequently *navigate* relationships to obtain references to related instances. This situation is common enough that `micca` supplies a variety of specialized embedded commands to handle specific circumstances.

`findOneRelated`
Finds a single instance along a relationship navigation chain.

`findRelatedWhere`
Finds a single instance along a relationship navigation chain where a boolean expression evaluates to `true`.

`foreachRelated`
Visits each instance along a relationship navigation chain.

`foreachRelatedWhere`
Visits each instance along a relationship navigation chain where a boolean expression evaluates to `true`.

`selectRelated`
Creates an instance reference set of instances along a relationship navigation chain.

`selectRelatedWhere`
Creates an instance reference set of instances along a relationship navigation chain where a boolean expression evaluates to `true`.

Navigating simple associations

Navigating simple associations

Navigating associative relationships

Navigating associative relationships

Navigating reflexive relationships

Navigating reflexive relationships

Navigating generalizations

Navigating a generalization from the subtype class to the superclass type is the easiest situation to understand. There is always an unconditional reference from a subtype class instance to its related supertype class instances. For example if **R22** is a generalization defined as:

```
generalization R22 Product Book_Product Recording_Product
```

Then we can navigate from a **Book_Product** instance (call it `book`) to a **Product** instance using something like:

```
<%  instance book findOneRelated product R22    %>
```

When the generated code executes, an instance reference value is placed into a "C" variable named, `product`, that is the currently related superclass instance of the subclass instance contained in the `book` instance reference variable. References from subtype instances to superclass instances are always singular and unconditional. This means that `product` is always non-NULL and there is always only one related superclass instance.

Navigating from a superclass instance to a subclass instance is more complicated. Recall, that in XUML, we interpret a generalization as always being *disjoint and complete*. In practical terms, this means that the superclass instance is always related to exactly one subclass instance, but we do not know to which subclass the related instance belongs. This is a property of the set partitioning that the generalization represents in modeling terms.

Conceptually, we can always find the related subclass instance by exhaustively trying each subtype. Navigating from the superclass to a subclass is navigating in the *reverse* direction and the desired subclass name must be specified. This might appear as something like:

```
<%instance product findOneRelate book {~R22 Book_Product} %>
if (book != NULL) {
    // We are related to a Book_Product
    // do something with the book instance
} else {
<%  instance product findOneRelated record {~R22 Recording_Product} %>
    if (record != NULL) {
        // We are related to a Recording_Product
        // do something with the record instance
    }
}
```

This approach can be tedious if there are even a modest number of subtype classes and we insist upon covering all the possible subclasses. To help this situation, `micca` provides an embeddeded command called, `classify`, that outwardly appears similar to a `switch` statement on the subclass type.

```
<%R22 classify product prodtype     %>
<%  subclass Book_Product           %>
    // Here, prodtype is a reference to a Book_Product instance
    // Do something with protype appropriate for a Book_Product
<%  end                             %>
<%  subclass Recording_Product      %>
    // Here, prodtype is a reference to a Recording_Product instance
    // Do something with protype appropriate for a Recording_Product
<%  end                             %>
<%end                               %>
```

There are several things to note here.

- Each generalization relationships has an embedded command defined to be the same as its name, *i.e.* **R22** in this case, and the `classify` command applies to that generalization.
- A `subclass` command is used to define the actions for a particular subclass of the generalization. The corresponding `end` command defines the boundary of the activity for that subtype.
- The superclass instance, here `product`, is checked to determine if it is related to an instance of the subclasses listed in the `subclass` commands.
- The related instance reference is placed in the `prodtype` variable. **N.B.** `prodtype` is a differently typed variable in each of the `subclass` bodies that matches the type of the recognized subtype. So, in the `subclass Book_Product` branch, `prodtype` is an instance reference to a `Book_Product` instance.
- The scope of the variable holding the subclass instance reference is local to each `subclass` branch. In this example, the scope of `prodtype` is limited to the `subclass` branches and is not defined outside of them.

In the previous example, all possible subclasses are listed in the `subclass` commands. This will not always be the case and so it is possible to use a `default` command as part of a `classify` sequence. Consider a generalization defined as:

```
generalization R33 Lamp Table_Lamp Floor_Lamp Ceiling_Lamp
```

If we have an activity which does something specific for a Table_Lamp, but otherwise does some common processing for the other subtypes, we can translate that situation as:

```
<%R33 classify lamp sublamp             %>
<%  subclass Table_Lamp                 %>
        // Here, sublamp is a Table_Lamp reference
<%  end                                 %>
<%  default                             %>
        // Here, sublamp is not defined
<%  end                                 %>
<%end                                   %>
```

The `default` situation has some additional rules:

- No subtype instance reference variable is in scope in the `default` case (*i.e.* `sublamp` is not defined in the previous example for the body of the `default` case). Since an instance reference variable is typed according to the class of the instance to which it refers, no instance reference variable can be properly typed for the `default` case.
- If the `classify` command has a `subclass` section for each possible subclass of the generalization, then the `default` section is neither needed nor desirable. `Micca` detects the situation where *not all* subclasses are given and there is *no* `default` section. In that case it issues a warning. This helps detect the situation where a previously written `classify` command exhaustively covered all the subclass cases and then a new one was added to the data translation without updating the processing translation. So, `default` sections should only be used where you are deliberately handling only a subset of the possible subclass cases.

Searching for class instances

Micca provides two embedded commands to locate class instances, findByName and findWhere.

The first command, findByName is rather specialized. When the initial instance population is defined, each initial instance is given a *name*. That name is used to handle relationship associations in the inital instances and stands for an instance reference. It can do the same at run time. For example, if there is a single instance of some class named, singleton, the you can directly reference using findByName.

Micca translation

```
<%  Schedule findByName singleton theSchedule       %>
```

This will result in an instance reference to the singleton instance of the Schedule class being place in the "C" variable called, theSchedule. So if you know by context that you want a particular instance that was defined in the initial instance population, the findByName command will obtain is by direct means without any searching.

A more general search is obtained using the findWhere command.

Scrall action

```
airports in my country .= Airport(1, Country : my country)
```

Micca translation

```
<%  Airport findWhere airports_in_my_country\
        {strcmp(airports_in_my_country->Country, my_country) == 0}       %>
```

This translation assumes that the Country attribute is a string and the my_country variable points to a string. So strcmp() is used to make the comparison. The findWhere command sequentially searches the instances of the class assigning a reference to the instance variable and evaluating the condition. It stops when it finds the first match and the instance reference will be NULL if no match is found.

Interating over class instances

Interating over class instances

Handling sets of instances

Handling sets of instances

Creating class instances

Creating class instances

Synchronous creation of class instances

To create an instance, the values of **all** attributes must be supplied. Attribute values may be supplied explicitly at the point of creation or by default values or by declaring the attribute to be zero initialized. (Note that dependent attributes are not initalized since they are computed by a formula at each access). All attributes also means that all relationship associations must be defined at the point of creation. This insistence insures that there is never any *partially* initialized instances floating round in the program.

In Scrall, a synchronous instance creation would appear as:

Scrall action

```
ac .= *Aircraft(Tail: t, Altitude: a, Heading: h, Location: l, Airspeed:s)
```

This assumes `t`, `a`, `h`, `l`, and `s` are variables with appropriate values.

The corresponding embedded command is:

Micca translation

```
<%  Aircraft create ac Tail t Altitude a Heading h Location l Airspeed s %>
```

Note it is not necessary to declare a variable named, `ac`, to hold the instance reference value. The `micca` code generator will automatically declare the variable if needed.

When relationship references must be specified, this is done using the relationship number and an instance reference variable. Note the specification of relationships is *not* done using the values of referential attributes. Rather, the notion is abstracted away from the individual attributes and the association is expressed in terms of the number of the relationship and a reference to the related instance. In this case the action language might appear as:

Scrall action

```
v .= *Owned Vehicle( License number:l ) &R1 me
```

Micca translation

```
<%  Owned_Vehicle create License_number l R1 self   %>
```

Note that the requirement to supply instance references at creation time to satisfy the referential attribute values will impose an order to instance creation in certain cases. For example, classes related across a generalization relationship must be created in supertype to subtype order since the subtype instance must have an instance reference to its related supertype instance to satisfy the referential integrity. So, if there were class definitions such as:

Micca class definition

```
class Lamp {
    attribute Manufacturer {char[32]}
    attribute Model ModelType_t
}
class Table_Lamp {
    attribute Height unsigned
}
class Desk_Lamp {
    attribute Wattage unsigned
}

generalization R22 Lamp Table_Lamp Desk_Lamp
```

Then creating an instance of a **Desk_Lamp** requires creating an instance of **Lamp** and that must be done first.

Micca translation

```
<%  Lamp create lamp Manufacturer {"Edison"} Model L100     %>
<%  Desk_Lamp create dlamp Wattage 100 R22 lamp             %>
```

Also note in the `micca create` command, the order of attributes and relationship names does not matter.

Asynchronous creation of class instances

Scrall action

```
New folder(Parent: parent id) -> *Folder
```

Micca translation

```
<%  Folder createasync New_folder {Parent parent_id}        %>
```

Deleting class instances

Deleting class instances

External entities

External entities

External identification of instances

External identification of instances

Lesser used translation features

In this section we describe features of `micca` that are used less frequently during the translation process. Most of these features handle situation where you need some measure of control over the final contents of the generated code or header files.

Prologue

`Micca` supports a `prologue` command that can be used when defining a domain. The `prologue` is arbitrary text that is included at the beginning of the generated code file. Often, this is used to include additional header files or for forward declarations.

For example, if a domain were to need a two-dimensional vector as an attribute type it could be introduced as:

Example 0.3 Prologue command example

```
prologue {
    typedef struct point {
        float x ;
        float y ;
    } Point_t ;

    static void pointSum(Point_t *p1, Point_t *p2, Point_t *result) ;
    static void pointDiff(Point_t *p1, Point_t *p2, Point_t *result) ;
}
```

This statement would place the declarations near the beginning of the generated file. You can use several `prologue` command. The text is simply concatenated in the order they appear and included as a whole at the tope of the generated code file.

Epilogue

Similarly, the `epilogue` command gathers text which is placed at the end of the generated code file. Continuing the previous example, we could place the implementation of the declared functions in the `epilogue`.

Example 0.4 Epilogue command example

```
epilogue {
    static void
    pointSum(
        Point_t *p1,
        Point_t *p2,
        Point_t *result)
    {
        result->x = p1->x + p2->x ;
        result->y = p1->y + p2->y ;
    }

    // And similarly for pointDiff.
}
```

Interface

Just like the generated code file, there are times when you need to control some of the content of the generated header file. The `interface` command places arbitrary text at the beginning of the header file generated by `micca`. It behaves similarly to `prologue` and `epilogue` in that multiple `interface` commands simply concatenate their contents. The `interface` command is useful for including other header files or defining data types used in domain operations.

Example 0.5 Interface command example

```
interface {
    #include "mylibrary.h"
}
```

- constructor
- destructor

Bridge code

Bridge code

- instance identification
- semantic mapping
- use of the micca portal
- use of domain operations
- synchronization with interrupts

Code organization

Code organization

Bibliography

Books

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

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

[3] [mtoc] Leon Starr, Andrew Mangogna and Stephen Mellor, Models to Code: With No Mysterious Gaps, Apress (2017), ISBN 978-1-4842-2216-4

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

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

[6] [sm-states] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in States, Prentice-Hall (1992), ISBN 0-13-629940-7.

Articles

[7] [micca-litprog] G. Andrew Mangogna, Micca: Translating XUML Models, 2018, http://repos.modelrealization.com/cgi-bin/fossil/mrtools/doc/trunk/micca/doc/micca.pdf

[8] [micca-manual] G. Andrew Mangogna, micca manual pages, 2018, http://repos.modelrealization.com/cgi-bin/-fossil/mrtools/doc/trunk/micca/doc/HTML/toc.html

[9] [ls-articulate] Leon Starr, How to Build Articulate UML Class Models, 2008, http://www.modelint.com/how-to-build-articulate-uml-class-models/

[10] [ls-time] Leon Starr, Time and Synchronization in Executable UML, 2008, http://www.modelint.com/time-and-synchronization-in-executable-uml/

Index