Translating XUML Models to pycca

Andrew Mangogna

Revision: 1.2
7 May 2011

1. Introduction

This paper presents a set of techniques to aid in manually translating an Executable UML (XUML) domain model into a "C" based implementation using **pycca**. Pycca is a language processing program that supports a domain specific language where data structures, state machines, operations and an initial instance population may be defined. In pycca, algorithmic processing is specified in the "C" programming language. Pycca transforms the domain specific language into "C" source and header files that are the implementation of the domain and may then be passed to a "C" compiler for code generation. Pycca does not generate any code *per se*, rather it generates initialized data structures that interface to the software architecture and packages the "C" action code, ordering the resulting output to match the requirements of a "C" compiler. Pycca targets the Single Threaded Software Architecture (STSA) that was designed for small, embedded control systems.

The primary purpose of pycca in the translation workflow is to handle the detailed and tedious generation of data structures, numeric encoding of state and event numbers and to order the output code file properly to satisfy the "C" compiler. This is a significant portion of what an automated model compiler does. However, to translate from the implementation independent logic of an XUML model into pycca does require manual steps directing how the implementation technology is to be applied. The steps in the translation workflow can be generally described as:

(1) Inspect the XUML model for the specifics of how the action code operates on the class data.

(2) Select implementation representations for class attributes and for relationship traversal information.

(3) Specify any state models in the corresponding pycca syntax.

(4) Translate operation and state action language statements into the corresponding "C" code that matches the chosen data structures.

1.1. Scope

This document is limited to techniques of translating XUML models into a "C" based implementation using pycca. This document assumes that you have a complete XUML domain model as a starting point. There is no discussion here about XUML or how to create XUML models. Readers are referred to the large set of XUML books and documents that are available.

1.2. Background

An XUML model consists of three projections of the problem space:

(1) A class model expressed as a normalized relational data schema.

(2) A life cycle model expressed as a set of interacting Moore type state models.

(3) A processing model expressed as data flow diagrams or as action language[1].

There are two fundamental choices related to how data operations are performed when translating to a statically typed language such as "C".

(1) Build a run-time engine that can emulate the behavior of the relational operations inherent in the actions so that operations can be independent of class type.

(2) Build specific code sequences for the various relational operations that perform the operation on a data structure that is specific to the class being operated on.

[1] Examples in this document use the SMALL action language syntax.

When translating to pycca, we are doing the latter. Each relational operation implicit in the actions of the XUML model is transformed into data structure specific code sequences that account for static nature of the typing system of "C". For example, if we wish to do a blind select on a class (*i.e.* selecting a subset of instances of a class based on a boolean expression involving the class attribute values), in a statically typed language we must emit different code for each distinct class and attribute expression being used for selection. Fortunately, most models perform selections on only a small number of possible attribute expressions and since the scale of applications targeted here is small, the possibility of emitting a large amount of essentially logically equivalent code is not of great concern. If that were a possibility, it would argue for using a run-time engine so that the space costs could be amortized over the entire application code base.

Since we are choosing to emit specific code sequences that are tied to a data structure, the first decisions must be to determine exactly how to represent a class from the Class Model as a "C" data structure. That is accomplished by examining both the class model and the processing model to determine what the code must do. In relational algebra, the operations are independent of the relation heading. In statically type languages, the relational operations are translated in the data structure specific code sequences. It is roughly analogous to the translation done by language compilers where the programming language attributes type to variables but treats operations as polymorphic and assembly language where memory is typeless and it is the operations that have type.

1.3. Basic Translation Workflow

The basic workflow described here for translating an XUML model is:

1. Perform an introspection of the model to determine the execution characteristics.
2. Define the class data structures for the model.
3. Encode 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 below.

2. Model Introspection

Traditionally, a model compiler's operation is directed by coloring or, using the more modern term, marking. Marking is the process of providing particular hints to the model compiler about the way in which the domain operates. Model compilers can deduce much of the behavior on their own by analyzing the model actions. For example, a model compiler is able to determine if a given class attribute is updated by examining all the actions of a domain and noting which attributes are written.

When performing the domain translation manually, it is up to the human translator to perform a similar set of model examinations to determine exactly what operations are performed on the class data. The following sections describe what information needs to be accumulated. Once the introspection is accomplished, then the data structure that represents the class may be chosen.

2.1. Attribute Operations

In this section we examine the introspection necessary to identify basic class operations. It is convenient to categorize attributes as:

Base attributes
describe some abstracted property of the real world entity which the class represents.

Referential attributes
are those used to implement relationships and relationship traversal.

Orthogonal to this classification, an attribute may be used as an identifier.

2.2. Identifiers

The domain class model is fundamentally a relational data model where the classes represent relation variables. Relation variables have one or more non-empty attribute sets which constitute an identifier for the tuples of the relation. When translating to a software architecture that holds all the domain data in memory[2], it is convenient and conventional to use the address of the instance data area as its identifier. The address is certainly unique and is convenient and efficient from an implementation point of view. Given that we use the address of the instance in memory is its primary means of identification, then the goal is to eliminate the storage for identifying attributes and use the instance address as the sole means of identification of the instance. Using the instance address as an identifier also implies that the addresses will be the primary way in which relationship information is stored and that pointer value dereferencing will be the primary means of navigating a relationship.

However, there are a couple of considerations that must be taken into account.

(1) It is generally considered an analysis error for the value of an identifying attribute to be modified anytime after the instance has been related to another instance across some relationship. Conceptually, relationships are composed by placing the values of the identifying attributes of the referenced instance into the referential attributes of the referring instance. Changing an identifying attribute value in a referenced instance by directly updating the attribute would then destroy the relationship reference. In a database based architecture, this situation could be handled by having the database management system cascade the update of the identifying attribute to change the values of those attributes that reference it. In minimal software architectures, this facility is not usually available. Domain models that arbitrarily update the value of an identifying attribute must be rejected.

(2) It sometimes the case that an identifying attribute is read and its value is used as a parameter to an algorithm. In this case, the attribute may not be eliminated and its value must be maintained. Fortunately, this is not a frequently occurring situation.

2.3. Referential Attributes

Referential attributes have values that match the value of some identifying attribute in another instance. Since we are replacing identifying attributes with the address of the instance in memory, then referential attributes are also replaced with corresponding address values. We will discuss how instance addresses are used to implement relationships below.

It is also possible to find that model actions read a referential attribute and use that value in an algorithmic fashion. In this case, the preferred method of access to the attribute is to follow the relationship back to the base identifying attribute. It is possible that this means traversing multiple references until the base attribute value is located. Note also that this is an implied traversal of the relationship and must be accounted for as described below.

[2] As contrasted to a software architecture that uses a Database Management System (DBMS) as a data storage and management component.

2.3.1. Attribute Access

All actions must be examined and every attribute must be marked as to whether or not it is read and/or updated. Updating is the most important, but any attributes that are not read can be eliminated. If base attributes are not used in a some fashion, then this indicates an analysis error.

2.3.2. Relationship Traversal

On the class model, relationships are considered bidirectional, *i.e.* it is possible to traverse the relationship along a path in either direction. In practice, most relationships in a class model are not traversed by the action language in both directions. This fact has a significant influence on the way in which the relationship information is stored. So part of the required model introspection is to determine which relationships are traversed and in what direction the traversal takes place. It is not necessarily an analysis error that a given relationship is not traversed by the action language. It is possible, *e.g.* for generalization relationships, for the relationship to exist to distinguish differing behavior of classes.

Note also that relationship traversal may be explicit or implicit. It is explicit if the traversal if found directly in an action. It is implicit if, as mentioned above, that a relationship must be traversed in order to obtain the value of an attribute that also serves as a referential attribute. In the case of dispatching a polymorphic event, there is an implicit traversal of the relationship by the software architecture to map the polymorphic event into an event of the currently referenced subtype.

2.3.3. Instance Selection

It is necessary to determine if a class instances are selected by means of their attribute values rather than by means of relationship traversal. This occurs when an action contains statements of the form

```
<class name>(<attr expr>) [3]
```

where:

<class name>
: is the name of a domain class

<attr expr>
: is an expression involving the attributes of the class, *e.g.*

```
Dog(Name = "Fido")
```

Several cases arise:

(1) If there are no such statements of this form for a class, then the class is accessed only by relationship traversal. This may provide options for how the class instances may be stored.

(2) If such selections are performed by domain operations, then it may be possible to use direct indices into the class instance storage array as an external identifier and thereby simplify the search to being an array indexing operation.

(3) If the class selection is performed frequently and/or the class contains a large number of instances, then it may be necessary to keep some form of an index to speed the access, e.g. a hash of the attributes upon which the selection depends.

[3] Or in BridgePoint Action Language, select one <var name> from instances of <class name> where ...

2.4. Model Dynamics

It is important to also mark the domain model according to the dynamic behavior of the classes. Classes instance counts and relationships that do not vary in time are candidates for much simpler data structures than their dynamic counterparts. In this section we discuss the introspection required to account for the domain dynamics.

2.4.1. Instance Dynamics

A class should be marked to indicate if any action creates an instance of the class. Instance may be created either synchronously by direct request or asynchronously be sending a creation event. Either method results in a class that has a dynamic number of instances of the lifetime of the domain's execution.

For classes that have static instance populations, *i.e.* there are no instances created by an action, there are two other ways that the class may be classified:

(1) If there is only a single instance of the class, then many aspects of dealing with the class are simplified. For example, any relationship traversal that passes through or terminates at a singleton class need not be actually traversed in code since there is only one possible result.

(2) Static class populations for which all the attributes of the class are only read, i.e. there are no attributes of the class that are updated, and which do not have an associated state machine are candidates for placing in constant variables. Linkers usually place constants in read-only memory and such memory is usually in greater supply than read-write memory for the types of systems under consideration here.

2.4.2. Relationship Dynamics

Relationships where **link** and **unlink** statements appear in some action are dynamic. It is usually the case that a class whose instance population is dynamic also has dynamic relationships. Any relationship that is dynamic must be so marked as this can have significant impact on the type of storage used for the relationship.

2.4.3. Current State Variable Considerations

Any class that has an associated state model, has an implied instance variable to hold the current state. This means that stateful classes cannot be stored in read-only memory.

3. Data Structures

After examining the domain actions for all the operations they perform, pycca classes can be defined that represent the classes of the domain model. Pycca arranges for each attribute to become a member of a "C" structure. The instances of the class are held in an array. Pycca uses the Single Threaded Software Architecture, STSA, and in that architecture there is no use of a system heap. Thus all class instances are stored in an array type variable that is sized for the worst case number of class instances. In this section, we determine how to use the model introspection to determine the pycca class definition.

3.1. Attribute Storage

After examining how the attributes of the classes are accessed, the following steps are used to define the attributes to pycca.

(1) Eliminate attributes that are not accessed.

(2) Eliminate referential attributes.

(3) Eliminate identifying attributes where possible.

(4) Map base attributes to an appropriate implementation type. The domain will have defined a set of data types for the attributes. These data types should be aliased to appropriate types in "C".

(5) Eliminate any dependent attributes as these will be obtained at run-time. The action associated with a dependent attribute is packaged into an instance operation.

3.2. Relationship Storage

Designing storage structures for relationships is generally more complicated than for base attributes. Each bidirectional XUML relationships can be factored into two or more unidirectional paths. A path represents a single relationship traversal in a given direction as it is found in an action. In this section we show how this decomposition works and how to represent it in pycca. There are several cases that must be considered:

(1) Singular paths for one-to-one relationships.

(2) One-to-many static paths.

(3) One-to-many dynamic paths.

(4) Associative relationships.

(5) Generalization relationships.

The strategy is to have structure members of a class that hold one or more pointers to the related instances. Generally, we wish to allocate pointers to support relationship navigation in all directions that are used by the action code. Note however this involves a space/speed trade off. By supplying pointers in both directions we are consuming memory (most probably RAM memory) for the pointer storage rather than performing a search on the instance array for the related instances.

For example, consider the case of a one-to-one relationship that is traversed in both directions. Allocating pointer memory in both structures appears as follows.

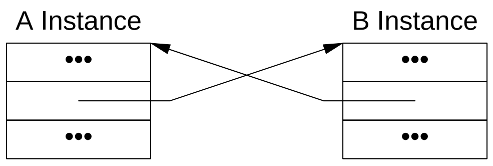

Figure 1. Relationship Traversal in Both Directions

In this case, we could have chosen to store only one pointer, say from A to B. With that choice, the traversal from A to B involves the usual pointer indirection. To traverse the relationship from B to A requires that all instances of A be examined to find the one whose stored relationship pointer value matches that of the B instance from which the traversal originates. This arrangement saves the storage of a pointer value, but makes the traversal of the relationship from B to A more costly in terms of execution speed since we must search the instances of A for a pointer value match. Usually, the trade off is made in favor of speed, *i.e.* if the relationship is traversed in both directions, then pointers are stored in both class instances. However, there can be cases where memory is dear enough and the frequency of the traversal low enough that a search of the instances is an attractive option. This option is available for all the possible relationship cardinalities, but in the discussion below we always assume that memory will be used to store the relationship information required for any traversal direction found in the action code.

3.2.1. Singular Path

The figure below shows two classes, A and B, that participate in a one-to-one relationship, R1. In the class model, one class is chosen to contain the referential attribute. That choice is arbitrary except for the case where one side is conditional and the other is unconditional and in that case, the class on the conditional side of the relationship is given the referential attribute.

When considering how to store the relationship information for this situation, the relationship can be decomposed into two paths, one from A to B and the other from B to A. If any action code traverses relationship R1 from A to B, then the "C" structure representing A needs to contain a member that is a pointer to the structure representing B. Similarly, if actions traverse the relationship in the direction of B to A, then the "C" structure for B will need a member that points to an A. In either case, if there is no action that traverses a relationship path in a particular direction, then no structure member corresponding to that direction need be defined. This is one reason that relationship traversal is carefully tracked.

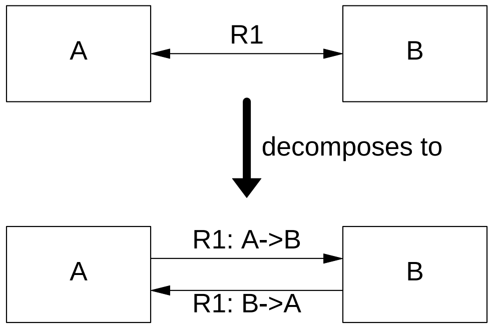

Figure 2. Decomposition of One-to-one Relationship

In pycca, the singular reference to another class is accomplished via the **reference** statement. So the definition of class A would include a statement:

```
class A
    •••
    reference R1 -> B
    •••
end
```

This would cause pycca to emit a "C" structure definition of the form:

```
struct A {
    •••
    struct B *R1 ;
    •••
} ;
```

The conditionality of such a relationship path can be indicated with the `NULL` value. If the relationship is unconditional, then an appropriate assertion can be included in the code before the value of the R1 member is used.

3.2.2. Static Multiple Path

Relationships in the class model that are one-to-many are formalized by placing referential attributes in the class on the many side that refer to identifying attributes for the class on the single side. Such a relationship can be decomposed into two paths as shown in the figure below.

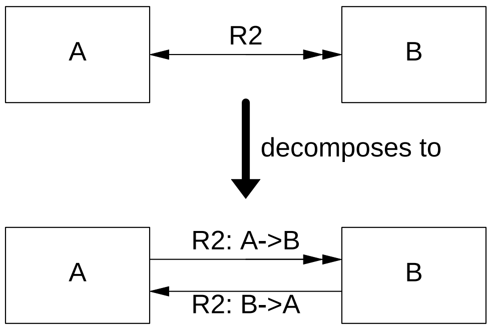

Figure 3. Decomposition of One-to-many Relationship

So we must consider the storage of each path separately. As usual, paths that are not actually traversed by action code need not have any storage structure allocated to them.

The path from the many side to the singular side (*i.e.* from B to A in the above example) is quite easy. It is the same type of singular reference as we saw in the last section and all the same description applies.

The more difficult path is when we must traverse from the one to the many side. Such traversals yield, at least conceptually, a set of instances. In this section, we discuss data structures that are useful when the relationship is static in nature. The next section discusses data structures that are more appropriate when the relationship is dynamic.

The figure below shows the structure for storing relationship information for a one-to-many path where the relationship is static over time.

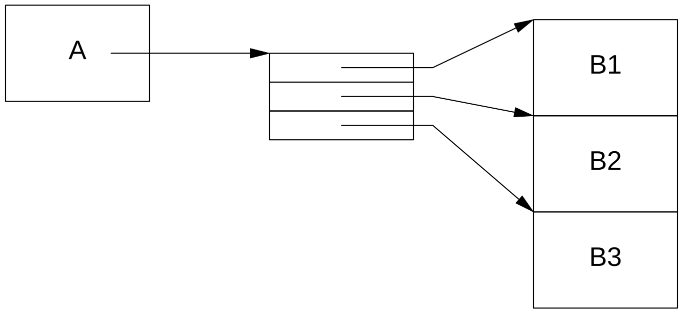

Figure 4. One-to-many Static Relationship Path Storage

The strategy is to define a structure member in A that is a pointer to an array of pointers each member of which in turn points to an instance of B. The array constitutes a representation of the set of B instances to which A is related. Since the relationship is static, the size of the pointer array is fixed by the initial instance population. The number of elements in the array of pointers will, in general, be different for each distinct instance of A. There are two ways in determine at run time how many elements are in the pointer array.

(1) The last element of the pointer array may be set to `NULL` to indicate that there are no more elements in the array.

(2) The count of elements may be stored as a structure member in A.

Pycca supports both types of one-to-many relationship storage using a variation on the **reference** syntax. Reference statements of the form:

```
class A
    •••
    reference R2 ->> B
    •••
end
```

or

```
class A
    •••
    reference R2 ->>n B
    •••
end
```

result in pointer arrays that are terminated by a `NULL` value. Reference statements of the form:

```
class A
    •••
    reference R2 ->>c B
    •••
end
```

cause an additional structure member to be defined in A that is initialized to contain the number of elements of the instance pointer array, as in:

```
struct A {
    •••
    struct B *const*const R2 ;
    unsigned R2__count ;
    •••
} ;
```

The choice of which manner to deal with the pointer array is determined by the operations on the set of related instances.

(1) A `NULL` terminated array is more convenient when the related instances set is primarily the target of some iteration. When the usual operation is to examine each related instance and then perform some operation on it, then `NULL` termination is somewhat easier to manage in the iteration loop.

(2) The counted array is more flexible. It can be used for iteration at the expense of an extra variable. In one particular case, the counted array is superior. This case arises when the related instances are identified by both a referential attribute referring to the one side of the relationships and an independent attribute that is unique only within the context provided by the referential attribute. In this arrangement, statements to select across the relationship can be resolved by an array indexing operations. For example, consider the following class model fragment.

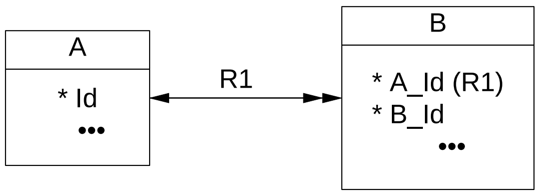

Figure 5. Class Model Fragment

If the values of `B_Id` can be encoded in small non-negative integers that are unique only in the context of a given `A_Id`, then a select operation across the relationship containing a where clause that restricts the selection to a particular `B_Id` can be coded as an array indexing operation on the counted pointer array. The count value can be used to insure that the array indexing operation is valid for the set of related instances. This affords instance selection that can be achieved in constant time rather than one that would be on the order of the number of related instances or worse the total number of instances of class B.

3.2.3. Dynamic Multiple Path

For the situation where there is a one-to-many relationship path that is dynamic, *i.e.* the relationship is the target of `link` and `unlink` operations, the storage alternatives discussed above for static relationship paths are not suitable. The set of instances related to a particular instance must be held in read/write memory since the set is modified during the running of the application.

Pycca provides two alternative storage strategies for dynamic one-to-many relationship paths:

(1) A fixed sized array of pointers to the referenced class instances is allocated as a structure member of the referring class. Any initial instances of the relationship are initialized and the remaining array slots are set to `NULL`. The action code can then manage the `NULL` valued slots by assigning instance pointer values on a `link` operation and assigning the `NULL` value on an `unlink` operation. This strategy is effective if an upper bound can be set on the number of related instances and if the set of referenced instances is approximately the same across all the instances of the referring class.

(2) A doubly-linked list of referenced instances is formed. The head of the list is allocated as a member of the referring class (A in this example) and a set of linked list pointers is allocated as a member of the referenced class. Pycca provides functions to add and remove instances from the list as a means of implementing `link` and `unlink` operations. This strategy is useful for tracking an arbitrary number of referenced instances at the expense of allocating the memory to hold the doubly-linked list pointer values.

To allocate a fixed size array for dynamic links, pycca uses the syntax:

```
class A
    •••
    reference R2 -ddd>> B
    •••
end
```

where **ddd** is a set of decimal digits. For example, a class definition of:

```
        class A
            •••
            reference R2 -20>> B
            •••
        end
```

will result in pycca defining a structure for A that contains:

```
        struct A {
            •••
            struct B *R2[20] ;
            •••
        } ;
```

Action code can then manage the R2 array using `NULL` values to indicate available array elements and non-`NULL` values that are instance pointers.

A fixed sized pointer array as given above does not work well when the upper bound of the related instance set size is either not easily determined or varies considerable between instances. For this case, pycca supports a doubly-linked list approach to storing a one-to-many relationship.

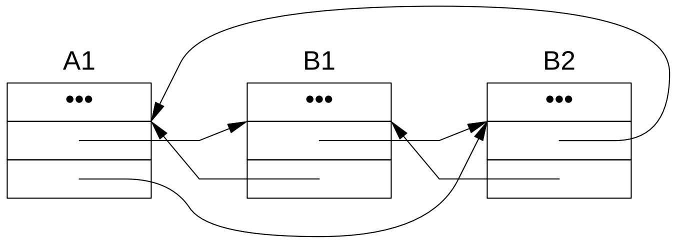

Figure 6. One-to-many Path as a Doubly Linked List

The figure above shows the pointer addressing when two instances of B are related to an instance of A. The use of linked lists is indicated to pycca by a variation of the **reference** statement syntax of the form:

```
        class A
            •••
            reference R2 ->>l B
            •••
        end
```

When pycca encounters this construct, it arranges for a list head to be placed in the structure for the referring class and a set of link pointers to be placed in the structure of the referenced class.

```
        struct A {
            •••
            rlink_t R2 ;
            •••
        } ;

        struct B {
            •••
            rlink_t R2__links ;
            •••
        } ;
```

Any instances of B referenced by an instance of A in the initial instance population are threaded together on the list that originates in A. The "C" data type, `rlink_t`, defines the two pointers of the doubly linked list. Pycca provides macros and underlying functions to add and remove items from the list, test if the list is empty and iterate across the list.

Note that it is possible for referenced class instances to be resident on multiple linked lists at the same time. To support this, the link pointer values are the address of the `rlink_t`-typed member within the class data structure of the referenced class. This means that the pointer values contained in `rlink_t` are not references to the beginning of the instance data. Pycca provides macros to convert the `rlink_t` pointer values to pointers to the beginning of the instance structure and this conversion is required before class attributes can be accessed.

3.2.4. Associative Relationships

Associative relationship are those relationships where a distinct class, known as the *associative class*, contains the referential attributes of the relationship. The associative class contains attributes to refer to the identifiers of two other participating classes (which are not necessarily distinct). Associative relationships may occur in all permutations of cardinality. The archetypal example is a many-to-many relationship since an associative class is required in this case to correlate all the relationship instances. The figure below shows a many-to-many relationship in Shlaer-Mellor graphical notation.

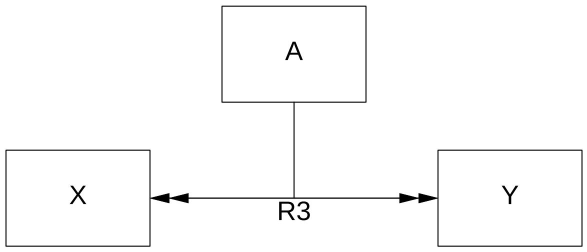

Figure 7. A Many-to-Many Relationship Graphic

In this figure, A is the associative class for R3 and holds attributes that refer to an identifier of both X and Y, the participating classes. An associative relationship can always be decomposed into two ordinary relationships of a fixed pattern. The figure below shows a drawing of this decomposition.

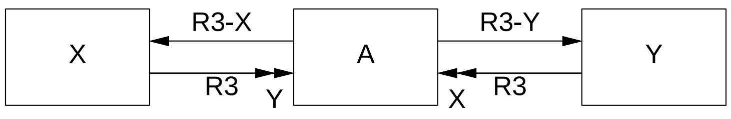

Figure 8. Decomposition of an Associative Relationship

In this figure, the associative class, A, has a singleton path to both participating classes, X and Y, *i.e.* R3-X and R3-Y, respectively. Note that the reference from A to both X and Y is unconditional, and this implies that if an instance of A exists it must refer to both an X and a Y and there are as many instances of A as there are instances of the relationship, R3. The relationship paths from the participants to A are derived from the model relationship. The cardinality of the path from X to A is the same cardinality as it appears on the Y side of the original model graphic. This is indicated by the Y annotation on the R3 labeled path from X to A in the above drawing. Similarly, the cardinality of the path from Y to A matches that of the X side in the model graphic. Note the swap that has happened in the decomposition as the role of A is expanded. This is a result of the unconditionality of the path from A to either participant. Thus in the decomposition, X has the same cardinality with respect to A as it did with Y in the modeled relationship because A always has a single, unconditional reference to an instance of Y. A corresponding statement can be made with respect to Y.

There are two considerations for defining the storage required to hold an associative relationship. Under certain conditions, all storage for A can be eliminated. If the associative class is used strictly for correlation purposes, *i.e.* if the associative class contains only attributes that reference the participating classes and does not have an associated state model, then storage for the associative class need not be defined and one of the one-to-many relationship path strategies discussed above can be used.

It is frequently the case that an associative class has other base attributes, participates in other relationships that require navigating to the associative class or it has a state model. In these cases the storage for the associative class must be defined and the above techniques for one-to-one and one-to-many relationship paths are then applied between the associative class and each of its participant classes. As an example, consider a many-to-many associative relationship where the associative class also holds some base attribute. The following pycca definitions of the classes could be used if the relationship is static.

```
class A
    •••
    reference R3_X -> X
    reference R3_Y -> Y
    •••
end
class X
    •••
    reference R3 ->> A
    •••
end
class Y
    •••
    reference R3 ->> A
    •••
end
```

3.2.5. Generalization Relationships

In XUML, a generalization relationship completely partitions a supertype class into a set of disjoint subtype classes. The referential attributes are contained in the subtype classes and unconditionally refer to an instance of the supertype. For each instance of the supertype, there must be exactly one related instance from among all the instances of all the subtype classes. The figure below show a simple generalization relationship diagrammatically.

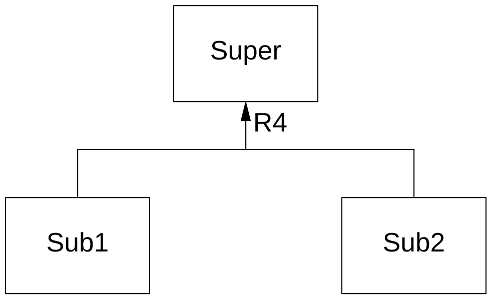

Figure 9. Generalization Relationship Diagram

Pycca supports two different techniques to define the relationship storage for generalization relationships. The techniques described here apply to generalization relationships where there is traversal of the relationship from the supertype to the subtype. In this case, we must be able to determine the type of the currently related subtype so that the traversal from a supertype instance to a subtype class to which it is *not* currently related can be determined to be the empty set. For the simple case where the generalization relationship is only traversed from the subtype to the supertype, a simple singular reference definition in the subtype is sufficient. However, when the relationship is traversed from the supertype to the subtype, we may choose the supertype to contain either a reference to its related subtype or that the subtype be included in the structure definition of the supertype class as a "C" union.

Generalizations Implemented by a Reference

Pycca uses constructs of the form,

```
class Super
    •••
    subtype R4 reference
        Sub1
        Sub2
    end
    •••
end
```

to define the storage in the Super class for the R4 traversal by reference pointers. This is translated into two structure members,

```
struct Super {
    •••
    SubtypeCode R9__code ;
    MechInstance R9 ;
    •••
} ;
```

The `R9__code` member holds an integer encoding of the type of the subtype. The `R9` member holds the actual subtype instance pointer value. Note that the value of the `R9` member should never be `NULL`.

Generalizations Implemented as a Union

Pycca uses constructs of the form,

```
class Super
    •••
    subtype R4 union
        Sub1
        Sub2
    end
    •••
end
```

to define a generalization relationship that is implemented by containing the subtypes as a union member of the supertype data structure. This structure translates into:

```
struct Super {
    •••
    SubtypeCode R9__code ;
    union {
        struct Sub1 R4_Sub1 ;
        struct Sub2 R4_Sub2 ;
    } R4 ;
    •••
} ;
```

The subtype code field remains the same as with references, however, now the memory associated with the subtype instances is contained in a union that is actually a member of the supertype data structure. The advantage of the union construct is that the storage required for a reference pointer is no longer necessary. The disadvantage is that the memory allocated to every instance of the supertype to accommodate the union is the largest of any union member. Thus subtypes that vary greatly in size have the potential to waste memory. Also note that if a given class is a subtype of two distinct generalization relationships (*a.k.a.* a multiple inheritance situation), then the union storage strategy may not be used for both generalization relationships. It is possible to have one generalization stored as a union and the other by reference. This is, fortunately, an uncommon situation.

4. State Model

There is no significant transformation required to specify state models to pycca. The Moore type state machines used in XUML are directly supported. All the event types of XUML are also directly supported by STSA. The pycca manual describes the syntax that must be used. The process of translating a state model into pycca is quite mechanical.

5. Code Translation

After all the class data structures have been determined, it is then possible to translate actions into "C" code sequences. In this section, we consider "C" constructs that will be used. Fundamentally, we show mappings from action language to "C" statements.

Pycca provides a large number of "C" preprocessor macros to help interface to the software architecture and to hide the naming conventions pycca uses for symbols that it defines. It is important to use the preprocessor macros where possible. Coding to the underlying symbol names and architecture interfaces will make the resulting domain code fragile with respect to changes in both pycca or the software architecture.

5.1. Instance Variables

Instance variables are declared by using the `ClassRefVar()` macro.

```
ClassRefVar(A, anA) ;
```

If the class has a constant population, then the reference must be declared as constant,

```
ClassConstRefVar(A, anA) ;
```

There are other variations of macros for sets of references, constant sets, *etc*.

5.2. Attribute Access

Instance operations and state actions always have an implicit variable named, `self`. `Self` is a constant pointer to the instance for which the operation or state action is intended. Note `self` is not defined as a parameter to instance operations or states in the pycca source. The software architecture arranges for the value of `self` to be passed correctly to state actions. However, when an action invokes an instance-based operation, it must supply the first argument that will be treated as the value of `self`. This is a requirement of the "C" language. It is possible, of course, to have instance references that are obtained by navigating relationships or by direct selection from classes.

Conventional base attributes can be accessed as structure members of `self`. For example, the pycca class declaration:

```
class A
    •••
    attribute (int color)
    •••
end
```

means that in an instance-based context that the color attribute could be updated by,

```
self->color = 27 ;
```

5.3. Event Generation

Generating events is a common action activity. Pycca provide a set of macros to make it convenient. It is necessary to specify when the event is generated to whether or not it is self or non-self directed.

5.3.1. Delayed Events

Pycca also supplies macros that are directly useful for generating delayed events. One use of delayed events is noteworthy. Sometimes it is necessary to divide a long running state action into smaller parts to reduce the response latency of the system. This involves arbitrarily limiting the computation of a state action and then generating an event to continue it. It is desired to generate a non-self directed event so that other events in the queue can be dispatched before the long running action is resumed. The preferred means of doing this is to generate a delayed event to `self` that has a zero time. This event will be immediately dispatched as a non-self directed event.

5.3.2. Event Parameters

The macros pycca provides for event generation are most convenient when there are no parameters to the event. If an event has parameters, then a three step process is necessary to generate the event.

(1) Obtain an event control block.

(2) Set the values of the event parameters.

(3) Post the event control block.

The pycca reference manual has detailed instructions.

5.4. Relationship Navigation

For each of the means of storing relationship information discussed above, there is a corresponding manner in which the relationship can be navigated. In this section we discuss the "C" code that is used to navigate relationships. The details of the code will depend upon the exact manner in which the relationship reference pointers have been stored.

5.4.1. Singular Navigation

Pycca class definitions of the form,

```
class A
    •••
    reference R1 -> B
    •••
end
```

yield "C" structure definitions that have members named the same as the reference. This allows easy access to instance reference for the relationship.

```
ClassRefVar(B, b) = self->R1 ;
```

Note that pycca does not know about relationship conditionality. If the relationship between A and B is conditional, then it is possible that the value of the R1 member is `NULL`. In this case, the standard `assert()` macro is useful.

```
assert(self->R1 != NULL) ;
ClassRefVar(B, b) = self->R1 ;
```

That conditionality can also be tested as in:

```
if (self->R1) {
    ClassRefVar(B, b) = self->R1 ;
    // Do something with b
}
```

5.4.2. Multiple Static Navigation

Classes that have one-to-many relationship storage most frequently use the references for iteration on the set of related instances. Here we consider such relationships that are static. The next section considers the dynamic one-to-many paths.

For those classes with `NULL` terminated multiple reference storage, the iteration construct is:

```
for (ClassRefConstSetVar(B, bSet) = a->R2 ; *bSet ; ++bSet) {
    ClassRefVar(B, thisB) = *bSet ;
    // Do some something to this related B instance.
}
```

For those classes with a counted set of related instances, the iteration construct is:

```
ClassRefConstSetVar(B, bSet) = a->R2 ;
ClassRefConstSetVar(B, bEnd) = bSet + a->RefCountMember(R2) ;
for ( ; bSet != bEnd ; ++bSet) {
    ClassRefVar(B, thisB) = *bSet ;
    // Do some something to this related B instance.
}
```

Here the loop termination is based not on the value pointed to by the iterator but rather the value of the iterator relative to the end of the reference pointer array.

5.4.3. Multiple Dynamic Navigation

The corresponding iteration constructs for dynamic multiple relationship storage must also take into account the two ways that the storage is managed.

For counted dynamic multiple relationship storage, the iteration construction is:

```
ClassRefSetVar(B, bSet) = a->R2 ;
ClassRefSetVar(B, bEnd) = bSet + COUNTOF(a->R2) ;
for ( ; bSet != bEnd ; ++bSet) {
    if (*bSet) {
        ClassRefVar(B, thisB) = *bSet ;
        // Do some something to this related B instance.
    }
}
```

There are two noteworthy points here. Because the number of elements in the reference storage array is a constant, we can determine the number of elements using the `COUNTOF()` macro. Also since any given element of the reference storage array may be `NULL`, we must test to make sure that we can dereference the pointer to the instance.

When the multiple dynamic relationship path is stored as a linked list, pycca provides a macro for the iteration pattern.

```
rlink_t *link ;
PYCCA_forAllLinkedInst(a, R2, link) {
    ClassRefVar(B, thisB) = PYCCA_linkToInstRef(link, B, R2) ;
    // Do some something to this related B instance.
}
```

In this case, a pointer to the links is ranged over all the instances on the relationship chain. Note that the link variable is *not* an instance reference, but rather is a link reference. This is necessary so that a given instance can be on several linked list chains at the same time using the same linked list code for access. So it is necessary to *up cast* the link value to the instance reference pointer before we can operate on the instance directly.

5.4.4. Associative Navigation

Since associative relationships decompose into two ordinary relationships, the iteration construct for associative navigation must combine the the two. Here we show iteration across a many-to-many relationship using static counted multiple reference storage. Similar constructs can be used when the one-to-many aspect is stored in other ways. So traversing from an instance of X to a set of instances of Y via an association class, A, is coded as:

```
ClassRefConstSetVar(A, assocSet) = x->R3 ;
ClassRefConstSetVar(A, assocEnd) = assocSet + x->RefCountMember(R3) ;
for ( ; assocSet != assocEnd ; ++assocSet) {
    ClassRefVar(A, thisA) = *assocSet ;
    ClassRefVar(Y, thisY) = thisA->R3_Y ;
    // Do some something to this related Y instance.
}
```

This code shows the combination of the traversal to the associative instances which then have a singular reference to the related Y instance.

5.4.5. Supertype Generalization Navigation

The "C" code required to navigate a generalization relationship depends, naturally enough, upon whether or not the subtypes are stored in a union or the supertype contains a pointer reference to a subtype. However, in both cases it is necessary to be able to determine if a navigation from the supertype class to a subtype class is in fact the empty set. This translates into requiring a test to determine if the currently related instance is of the proper class type. So action language of the form,

```
        self->R4[Sub1] > ˜s1 | None? !isEmpty, !isNotEmpty
        !isNotEmpty:    27 > ˜s1.Temp
```

must include a test of the form,

```
if (PYCCA_isSubtypeRelated(self, Super, R4, Sub1)) {
    // Here we are currently related to a Sub1.
}
```

to insure that we are currently related to the subtype that is the target of the relationship navigation.

Continuing with this example, if the subtype is stored in a union in the supertype, the above action language expression would be translated as:

```
if (PYCCA_isSubtypeRelated(self, Super, R4, Sub1)) {
    ClassRefVar(Sub1, s1) = PYCCA_unionSubtype(self, R4, Sub1) ;
    s1->Temp = 27 ;
}
```

If the generalization relationship storage was chosen to be by reference, then the appropriate translation is:

```
if (PYCCA_isSubtypeRelated(self, Super, R4, Sub1)) {
    ClassRefVar(Sub1, s1) = PYCCA_referenceSubtype(self, R4, Sub1) ;
    s1->Temp = 27 ;
}
```

5.4.6. Subtype Generalization Navigation

Navigation from a subtype instance to a supertype is much simpler since, by definition, the relationship path from a subtype instance to a supertype instance is unconditional. Of course, the "C" code sequence depends upon whether or not the subtype is stored in a union in the supertype. The simple case is when the subtype instances are stored separately and contain a reference to the supertype. In this case,

```
        42 > sub1 -> R4[Super].Volume
```

translates to

```
sub1->R4->Volume = 42 ;
```

When a union is involved, then some pycca macros are useful.

```
ClassRefVar(Super, s) = PYCCA_unionSupertype(sub1, Super, R4) ;
s->Volume = 42 ;
```

5.5. Class Instances Iteration

STSA requires that all class instances[4] be stored in a dedicated, fixed-sized, memory pool that is specific to the class. That memory pool is declared as an array variable. Consequently, it is relatively easy to iterate across the instance array for a particular class. Usually, it is sufficient to simply iterate over the instances of the class in the order in which they are stored in the class array.

[4] Except subtype instances that are stored in a union member of a supertype.

For example, a class based operation that increments the Count attribute of all instance of a class might be translated to:

```
class operation
incrCount(
    int cnt)
{
    ThisClassRefVar(iter) ;
    PYCCA_forAllInstOfThisClass(iter) {
        iter->Count += cnt ;
    }
}
```

5.5.1. Selecting an Arbitrary Instance

Because instances are allocated in simple arrays, that fact can be used for selecting instances based on the value of attributes. Action language statements of the form:

```
Dog(Age > 5)
```

select the set of Dog instances where the Age attribute is greater than 5. This is accomplished by sequentially iterating over the array elements for a class and checking if there is a match on one or more attribute values. Pycca provides macros to support the iteration. It is useful to distinguish between an static population and a dynamic one. For a static class population, there is no need to test if the instance slot is in use, since by definition all elements in the class storage array for a static population are used.

```
ClassRefVar(Dog, d) ;
PYCCA_forAllInst(d, Dog) {
    if (d->Age > 5) {
        // Do something with old dogs.
    }
}
```

For a dynamic population, another test is required to insure that the values held in the instance structure members are indeed valid.

```
ClassRefVar(Dog, d) ;
PYCCA_forAllInst(d, Dog) {
    if (IsInstInUse(d) && d->Age > 5) {
        // Do something with old dogs.
    }
}
```

5.5.2. Selecting an Instance by Identifier

When selecting an instance by the values of attributes that constitute a class identifier, at most one instance will be found. Action language of the form,

```
Dog(Name = "Fido") | generate Run
```

will generate at most one event to the instance of Dog where Dog.Name equals Fido. Pycca provides macros to do the iteration across the instance storage pool. Again, the distinction between static and dynamic populations is made. For static populations the translation is:

```
ClassRefVar(Dog, d) ;
PYCCA_selectOneStaticInstWhere(d, Dog, strcmp(d->Name, "Fido") == 0) ;
if (d != EndStorage(Dog)) {
    PYCCA_generate(d, Dog, Start, self) ;
}
```

There is a corresponding macro for dynamic instances.

```
ClassRefVar(Dog, d) ;
PYCCA_selectOneInstWhere(d, Dog, strcmp(d->Name, "Fido") == 0) ;
if (d != EndStorage(Dog)) {
    PYCCA_generate(d, Dog, Start, self) ;
}
```

In both cases note the test of the iteration variable against the value of the end of the class storage array. This insures that a Dog of the given name is actually found. Otherwise, the iteration variable is left at the end of the storage array, which, in usual "C" iterator style, is actually one past the end of the last element of the class array. Thus, EndStorage(Dog) is a valid address, but may *not* be dereferenced.

5.5.3. Array Indices as Identifiers

As we have already discussed, our primary means of identifying instances is by the address of the instance in memory. This is convenient and generates simple code since the pointer can be dereferenced to obtain the values of attributes. However, pointer values should not be passed outside of a domain. To do so seriously breaks the encapsulation of the domain and allows code external to the domain to perform dangerous operations. Yet it is still sometimes necessary to arrange for external code to refer to some type of identifier of an instance. If the choice of how the external code identifies an instance is arbitrary, then the array index of the instance in its storage array is a convenient external identifier. The pointer to the instance value is then easily and inexpensively computed inside of the domain where the location of the class storage array is known.

5.5.4. Instance Sets from Relationships

Many times action language statements navigate one or more relationships to obtain a set of instances which are then subjected to some further processing. For example the action language statement,

```
        self -> R1 -> R3 | generate Start
```

specifies that the set of instances obtained by navigating across R1 and R3 all have the Start event generated to them. Assuming that R1 and R3 are one-to-many relationship paths, one way to translate this would be to accumulate, in some manner, the instance set and then iterate across the set generating the Start event to each instance in the set. However, it is not necessary actually to accumulate the set of instance reference pointer values. The iteration can be *nested* downward so that we consider each referenced instance one at at time. Assuming that R1 and R3 are stored as a `NULL` terminated static relationship path, the the following code will accomplish the intent of the action language without accumulating the related instances as a set.

```
for (ClassRefSetVar(B, bSet) = self->R1 ; *bSet ; ++bSet) {
    ClassRefVar(B, aB) = *bSet ;
    for (ClassRefSetVar(C, cSet) = aB->R3 ; *cSet ; ++cSet) {
        PYCCA_generate(Start, C, *cSet, self) ;
    }
}
```

It is often the case that relationship navigation can be nested in this or a similar fashion so that the intended operation can be performed on each instance as it is obtained rather than trying to accumulate the instance set implied by the relationship navigation then then iterating through the accumulated set.

5.6. Relationship Dynamics

In this section we consider how to translate action code that uses the **link** and **unlink** operations to form and dissolve relationships between instances. This is a run time operation. For binary relationships, relationship dynamics are specified by statements of the form,

```
        (instA, instB) | link R1
```

and of the form

```
        (instA, instB) | unlink R1
```

5.6.1. Singular Relationships Dynamics

For singular relationships, the linking operation is accomplished by simply storing a pointer value into the appropriate structure member. Unlinking is accomplished by storing a `NULL` value. Thus, linking would appear as,

```
        instA->R1 = instB ;
        instB->R1 = instA ;
```

and unlinking is simply,

```
        instA->R1 = NULL ;
        instB->R1 = NULL ;
```

Note that formally in the action language both participating instances must be supplied. In keeping with our strategy of not allocating storage for relationships that are not actually traversed by the action code, under some circumstances one of the instances may not be required. In this case, the action code that determined the unused instance may not have to be included in the translation. If the unused instance is not referenced elsewhere in the action, it is possible to eliminate all the code associated with finding it.

It is also common for unlink operations to be eventually followed by a corresponding link operation. In these cases, it is not necessary to set a reference pointer to `NULL` if in subsequent statements it will be set to a new value. If the setting to `NULL` is to be optimized away, some care must be taken that there are *no* code paths where it will not be set to a new value. Note that most modern compilers can also diagnose this situation. If a second assignment is made to a variable and between the two assignments the value of the variable was not used, most compiler optimizers will eliminate the first assignment. So the safe coding practice is to include the assignment to `NULL` even if it appears redundant and let the compiler handle the optimization. If the "C" compiler is not that capable, the you can choose to make the optimization manually.

5.6.2. Multiple Relationships Dynamics

The code sequences required for link and unlink operations on multiple dynamic relationship legs are dependent upon the choices made for the relationship storage. In the example below, we assume A and B participate in a one-to-many relationship from A (1) to B (M) and that relationship storage is allocated for both paths.

For counted storage, pycca provides the `PYCCA_relateToMany()` and `PYCCA_unrelateFromMany()` macros. Then, action code such as,

```
        (instA, instB) | link R2
```

is translated as,

```
        instB->R2 = instA ;
        ClassRefSetVar(B, bSet) ;
        PYCCA_relateToMany(bSet, instA, R2, instB) ;
        assert(bSet < instA->R2 + COUNTOF(instA->R2)) ;
```

The assertion in the last statement checks that a slot in the relationship storage array was actually found. The corresponding unlink code is,

```
        instB->R2 = NULL ;
        ClassRefSetVar(B, bSet) ;
        PYCCA_unrelateFromMany(bSet, instA, R2, instB) ;
        assert(bSet < instA->R2 + COUNTOF(instA->R2)) ;
```

The assertion in this case checks that we indeed found the reference to instB in the relationship storage array.

For linked list storage, pycca provides the `PYCCA_linkToMany()` and `PYCCA_unlinkFromMany()` macros. In this case, the link operation is similar,

```
        instB->R2 = instA ;
        PYCCA_linkToMany(instA, R2, instB) ;
```

as is the corresponding unlink operation.

```
        instB->R2 = NULL ;
        PYCCA_unlinkFromMany(instB, R2) ;
```

Note that in the case of the above unlink operation it is not necessary to supply the instance reference from the one side of the relationship. This is because of the way that doubly linked, circular lists operate. To remove an item from such a linked list requires only the reference to the item being removed.

5.6.3. Associative Relationships Dynamics

Dynamic associative relationships must deal not only with the participating instances but also with the associative class. In general, linking an associative relationship implies allocating an instance of the associative class and, correspondingly, unlinking will free the associative class instance. Often actions involve unlinking one relationship instance and linking a new one to a different instance. In those cases, the associative class instance may be reused. In this section we will consider several variations on the linking and unlinking of an associative relationship. In the examples, we will assume that A is the associative class of a many-to-many relationship, R3, with participating classes X and Y. We also assume that the X and Y classes have one-to-many linked list type of relationship storage for the associative class. So we assume the class definitions are:

```
class A
    •••
    reference R3_X -> X
    reference R3_Y -> Y
    •••
end
class X
    •••
    reference R3_X ->>l A
    •••
end
class Y
    •••
    reference R3_Y ->>l A
    •••
end
```

Associative Relationship Linking

For actions that create an instance of an associative relationship, it is necessary to allocate an instance of the associative class. Consider the action code,

```
(instX, instY) | link R3 >> A
```

which would be translated into,

```
ClassRefVar(A, a) = PYCCA_newInstance(A) ;
a->R3_X = instX ;
a->R3_Y = instY ;
PYCCA_linkToMany(instX, R3_X, a) ;
PYCCA_linkToMany(instY, R3_Y, a) ;
```

Here, a new instance of the associative class is allocated. The unconditional singular references from the associative class to the participating classes are assigned. Finally, the associative instance is placed on the linked lists of the participating classes.

Associative Relationship Unlinking

Unlinking is is just the inverse of the linking processing. The action language,

```
(instX, instY) | unlink R3 << A(a)
```

would be translated into,

```
PYCCA_unlinkFromMany(a, R3_X) ;
PYCCA_unlinkFromMany(a, R3_Y) ;
PYCCA_destroyInstance(a) ;
```

Associative Class Reuse

One other common case is to unlink one instance and immediately link to another instance. For this case, the action code,

```
(instX1, instY) | unlink R3 << A(a)
(instX2, instY) | link R3 >> A
```

can be translated as,

```
        PYCCA_unlinkFromMany(a, R3_X) ;
        a->R3_X = instX2 ;
        PYCCA_linkToMany(instX2, R3_X, a) ;
```

This code sequence optimizes away the superfluous unlink operation of instY from the R3_Y linkage and as well as the assignment of instY into the R3_Y member of the associative class. Note that this optimization is correct only in the case were there are *no* branches in the flow of control between the link and unlink actions. There are two other things to be aware of:

(1) If the associative class has a state model, then the current state of the associative class must be considered. An unlink followed by a link operation will need to reset the current state to some initial state, presumable a default initial state for the state machine. This is in keeping with the idea that the unlink operation deallocates the associative class instance and the link operation allocates a new one.

(2) The associative class may have other descriptive attributes whose values need to be set. These new values would be given in the action language statement and code must be generated to install the new values into the newly linked associative class instance.

5.6.4. Subtype Migration

To migrate a subtype to a different type is a two step process. First, the subtype code must be changed to the value for the new type. Second, the reference pointer or union member must be adjusted for the new type.

When using references pointers for the generalization, typically, the current instance is deleted and a instance of the new type is allocated. The reference to the new instance is then stored into the subtype member of the supertype data structure.

```
// Migrate Sub1 to Sub2 by reference.
PYCCA_destroyInstance(self->R4) ;
ClassRefVar(Sub2, s2) = PYCCA_newInstance(Sub2) ;
PYCCA_relateSubtypeByRef(self, Super, R4, s2, Sub2) ;
```

When a union is used to hold the generalization, pycca provides a macro to help with the initialization of the union subtype member.

```
// Migrate Sub1 to Sub2 by union.
PYCCA_migrateSubtype(self, Super, R4, Sub2) ;
PYCCA_initUnionInstance(self, R4, Sub2) ;
```

5.7. Class and Instance Based Operations

Class and instance based operations pose no special translation difficulties as these constructs have a direct representation in pycca syntax. Macros are provided to hide the naming conventions. Class operations should be invoked using `ClassOp()` or `ThisClassOp()`. Instance operations use the corresponding macros, `InstOp()` or `ThisClassInstOp()`. Note that because of limitation in "C", it is necessary to supply explicitly the `self` instance reference when invoking an instance based operation.

5.8. External Operations

External operations also present no special difficulties in translation. Pycca syntax explicitly supports defining external operations. Note that you may include code in the external operation definition, but that code is not part of the output of the translation. It may be useful to include code to define precisely the expected behavior of the external operation. Also, some additional tooling may be able to make use of any code found in the external operation definition.

6. Populating a Domain

Once a domain is translated, the next step is to supply an initial instance population. A domain is much like an automaton and requires that the initial conditions be specified before it can function. Pycca supplies two way to define instances. One is useful for singular instances and the other is useful for specifying multiple instances in a tabular arrangement. It is best to keep instance populations in a separate file. You may wish to have several different populations for testing and final deployment. Keeping the populations in separate files makes it easier to deal with those parts that can change, *e.g.* the population itself.

7. Bridging Domains

The last step in assembling a set of domains into an application is to construct the bridges between the domains. Bridges are small pieces of *glue code* whose primary responsibility is to translate from the semantics of one domain into the semantics of another domain.

In XUML, the domain chart show the dependencies of one domain on another. This is sometimes stated as the *flow of requirements* between domains. The lines on the domain chart show the assumptions that one domain makes on services supplied to it by another domain. It is easy to confuse the lines in a domain chart to be some type of flow of control. During the course of analysis, those requirement dependencies are realized by actual transfer of control and data. From the point of view of a domain and in **pycca** terms, it supplies services via its domain operations and makes demands on other domains via its external operations[5]. It is important to understand that the semantics of the domain operations and external operations of a domain are in the semantic terms of the domain, without reference to whatever other domain may supply a needed service. At some point in time, the external operations required by one domain must be matched against the services supplied by some other domain and there must be some translation between the semantic terms of the two different subject matters. When analysis of multiple domains is proceeding simultaneously, some care must be taken to insure that the serviced needed by one domain can indeed be supplied. This is where the usual maxim of,

analyze from the top down and build from the bottom up

derives. By analyzing from the top down, the service requirements and interface needs of a domain can be established prior to the analysis of the domain providing those services. By building from the bottom up, needed services are put in place and more easily tested without having to construct elaborate service stubs that simulate complicated behavior.

An example is in order. Consider two domains, one controlling a chemical reaction process and another responsible for gathering all the data from sensors. Controlling a chemical reaction in a vessel will require knowledge of the temperature. The reaction management domain delegates acquiring the temperature info to the process sensor domain. When the temperature is needed, the reaction management domain will invoke an external operation of the form:

```
external operation
get_vessel_temp(
    VesselIdType vessel_id)
: (VesselTempType)
```

Thus the service request from the reaction management domain is in terms that are consistent with its subject matter, *i.e.* given a reaction vessel, tell me the temperature of it.

[5] In some quarters of XUML, there is discussion of so called *implicit bridging*. This is a concept similar in many ways to aspect oriented programming wherein the interaction with external domains would be specified outside of the domain model itself. While an interesting concept, it is not one that **pycca** attempts to implement. Hence in this discussion, all external interactions appear explicitly in the actions of the domain.

The process sensor domain provides services to obtain all the data in the plant and convert it to meaningful engineering units. It provides a service,

```
domain operation
get_sensor_input(
    SensorIdType sensor_id)
: (InputValueType)
```

Again the semantics of the service are in the semantic terms of the provider of the service.

Clearly, the external operation signature of the reaction management domain does not exactly match the domain operation signature of the process sensor domain. The bridge between the domains must provide the semantic mapping. The process sensor domain does not know anything about reactor vessels and the reaction management domain does not know anything about sensors. The bridge is required to translate a `vessel_id` into a `sensor_id`, invoke the `get_sensor_input()` domain operations and convert the returned value from `InputValueType` to `VesselTempType`.

Before we can figure out how to map a `vessel_id` to a `sensor_id`, we need to determine how the vessel and sensors are to be identified outside of the domain boundary. Inside the domain, we use a pointer to the instance as its identifier. This is as we discussed above and is the basis for storage strategies that make relationship traversal convenient and efficient. However, passing pointers outside of the domain is definitely a *bad idea*. Passing a pointer to a piece of data that was purposely declared static to prevent any access to it is a serious breach of encapsulation. Fortunately, there is a convenient external identifier for a class. Since all instances of a class are held in a storage pool and that storage pool is in fact a "C" array, the index of an instance into its storage array is a convenient identifier of the instance. An array index is easily converted back into an instance pointer inside the domain and easily computed so it can be handed off outside of the domain. So when a domain gets an instance identifier on input it is a simple matter to compute the instance pointer by the following C / pycca idiom:

```
assert(vessel_id < COUNTOF(BeginStorage(VESSEL))) ;
ClassRefVar(VESSEL, vessel) = BeginStorage(VESSEL) + vessel_id ;
```

The `assert()` makes sure we don't index outside of the array boundaries and under many circumstances a run time test of the value of `vessel_id` is warranted. Since pointer arithmetic in "C" is scaled by the size of the object pointed to, the simple addition of the index value to the base address of the class storage computes the instance pointer value. Analogously, the difference between an instance pointer and the base address computes an array index that is useful as an identifier of the instance outside of the class.

```
VesselIdType vessel_id = vessel - BeginStorage(VESSEL) ;
```

Again, the implicit scaling of address arithmetic works its magic and does exactly what we want.

So, having established the index of an instance into its storage array as an appropriate and convenient way to identify the instance outside of the domain boundaries, we now need to know what identifier values make sense for a domain. Since **pycca** *knows* all the information we need, it can supply the necessary values. When **pycca** is invoked with the **-ids** or **-dataportal** option, it places in the generated header file a set of preprocessor defines that encode the classes, instances, attribute and events of the domain into the integer numbers that we will need for the bridging between domains. This leads us to bridge code that might appear as follows.

```
#include "rvm.h"
#include "psm.h"

VesselTempType
eop_rvm_get_vessel_temp(
    VesselIdType vessel_id)
{
    static SensorIdType const tempSensorMap[RVM_VESSEL_INST_COUNT] = {
        PSM_SENSOR_VESSEL_A_TEMP_INST_ID,
        PSM_SENSOR_VESSEL_B_TEMP_INST_ID,
        PSM_SENSOR_VESSEL_C_TEMP_INST_ID,
    } ;

    assert(vessel_id < COUNTOF(vesselSensorMap)) ;
    return psm_get_sensor_input(tempSensorMap[vessel_id]) ;
}
```

Here we assume that **rvm.h** is the generated header file for the Reactor Vessel Management domain and that **psm.h** is the generated header file for the Process Sensor Management domain. We further assume that the instance population of **rvm** consists of three vessel: A, B, and C. Correspondingly, **psm** defines temperature sensors for vessels A, B and C. The preprocessor symbols, **RVM_VESSEL_INST_COUNT**, **PSM_SENSOR_VESSEL_A_TEMP_INST_ID**, **PSM_SENSOR_VESSEL_B_TEMP_INST_ID** and **PSM_SENSOR_VESSEL_C_TEMP_INST_ID** are supplied by **pycca**. The bridge is then a simple mapping of `vessel_id` to `sensor_id` by indexing into an array and supplying the resulting value to `psm_get_sensor_input()`. Here we assume that the return values are the same type or at least types that can be implicitly converted by "C".

Not all bridging is a simple as this example. Sometimes the bridge will map identifiers to function pointers and it may be necessary to invoke domain operations by pointer rather than by name. Other bridges may be dynamic, depending upon the dynamic nature of the counter part instances. In these cases it is necessary to manage the mapping array at run time. Despite these complications, the basic concept of mapping some form of instance identifier to its counter part in another domain solves most of the bridging problems. This simple example also illustrates the importance of developing the initial instance population of a domain *before* constructing the bridges as the bridge contents usually depend heavily on the instance identifiers are associated with the initial instance population.

Table of Contents

Table of Figures