Implicit bridging

In this section, we present an implicit bridging scenario from the perspective of the MX domain. The intent is to examine the additional requirements that implicit bridging places on the MX domain and the conceptual flow of execution needed to realize implicit bridging. To make the discussion more concrete, we replicate the example of bridging the LUBE and SIO domains from *Models to Code*.

Many of the implicit bridging ideas are found in *Aspect Oriented Programming* (AOP) approaches. The focus of AOP is to define places in the source code where it is possible to intervene in the flow of control to perform additional processing. AOP has its own vocabulary, which we do not use. Implicit bridging differs from AOP in two key respects:

1. We do not seek a means to specify an arbitrary place in the program source code. Rather, implicit bridging is limited to only a handful of distinct model execution situations. That limitation caters to the desire to discuss implicit bridging in a platform independent manner using the concepts and vocabulary already established for xUML development.
2. We wish to specify the behavior of the implicit bridging rather than allowing for inserting arbitrary code to execute. Our view is that bridging is a means to “bridge the semantic gap” which arises as a result of decomposing a system based on subject matter.

At system assembly time, it is assumed that the domain modelers confer to determine how the bridges between domains will be specified to meet the requirements flow indicated by the system domain chart. We make no statement as to how that happens, only that the specification results in the necessary information required by the MX domain to realize the implicit bridging. The expectation is that the specification information would consist of an implicit bridging meta-model population which the translation tooling would then use to produce the data that would drive the MX domain execution of the bridging. Note that the specification data used by an MX domain is usually a condensation and rearrangement of data that would appear in a meta-model population of bridging (*e.g.* the transition function of a state model takes on a distinctly different shape in an MX domain than in a meta-model population) and that the MX domain assumes the specification data has been validated during the model translation process. The intent of the using validated specification is to eliminate the need for the MX domain to have to consult the platform-specific model during run time and carries with it the assumption that the MX domain is not a direct interpreter of meta-model populations. Much of this section presents the bridging information and how it is used from the perspective of an MX domain.

Although the concepts of client domain and service domain are useful when applied to requirements flow, control flow is a more fluid situation. For example, most client domains will initiate some bridging operation to obtain a required service. However, it is also the case that a service domain may initiate some bridging operation as a means of providing a service required by its client domain. We see an example of this later. In both cases, the client / server relationship of the domains with respect to requirements flow remains the same.

Rather than use the terms *client* and *server* domains, we use the terms *initiator* and *terminal* domains. An initiator domain is one whose flow of execution matches a *bridgeable condition* that was specified at system assembly time. This condition is recognized by the MX domain and results in the execution of a *model operation* in the terminal domain. Since a bridgeable condition is executed by some instance of a class in the initiator domain and since it results in the execution of a model operation on some class instance in the terminal domain, we consider the class from the initiator domain to be a counterpart of the class from the terminal domain. Semantically, we expect the two classes to have disjoint abstractions of the same underlying real-world entity. The bridge operation is a function between these disjoint entities which provides the semantic underpinning between the domains and classes. This leads to the notion that a relation value can show the counterpart relationship. Given the counterpart relationship, we can talk about an *initiator class* and a *terminal class*

We define a *class counterpart map* as a relation value with the following heading:

Table 1: Class Counterpart Map Heading

Initiator Domain	Initiator Class	Terminal Domain	Terminal Class
String	**String**	**String**	**String**

For the LUBE / SIO example, the class counterpart relation value appears in tabular form as:

Table 2: LUBE / SIO Class Counterparts

Initiator Domain	Initiator Class	Terminal Domain	Terminal Class
String	**String**	**String**	**String**
LUBE	Injector	SIO	I/O Point
LUBE	Injector	SIO	Conversion Group
SIO	Range Limitation	LUBE	Injector
SIO	Continuous Input Point	LUBE	Injector

Relational and tuple algebra

We use relational and tuple algebra in these notes to describe the required data and operations for implicit bridging. The algebra is well defined and known to be as powerful as predicate logic. Experience also shows that relation-oriented data can be translated to a wide range of implementations. For our purposes, it serves as a universal way to describe data and operations that is software platform independent. We follow the definitions of tuple and relation given by Date and Darwen[a].

[a]C.J.Date and Hugh Darwen, *Databases, Types, and the Relational Model: The Third Manifesto*, 3rd Edition, Pearson Educational, Inc., 2007, ISBN: 0-321-39942-0, p. 19

As stated previously, implicit bridging is restricted to a fixed set of bridgeable conditions. Bridgeable conditions represent execution the Initiator Domain that has already completed. Model operations represent execution in the Terminal Domain which is invoked as a side-effect of having executed a bridgeable condition. The following list gives the bridgeable conditions possible for an Initiator Domain.

1. Transitioning into a state.
2. Transitioning out of a state[1].
3. An attribute value was read (which includes computing the formula for a dependent attribute).
4. An attribute value was updated.
5. An imminent event was signaled.
6. A delayed event was signaled.
7. A delayed event was canceled.
8. A method on a class instance was invoked.
9. An association reference was updated.
10. A generalization subclass instance was migrated to a different subclass.

Model operations take the form of actions to be performed in the Terminal Domain. Many model operations have similarities to bridgeable conditions. For example, when an Initiator Domain executes an action which signals an imminent event, the MX domain detects that action as a bridgeable condition. The bridgeable condition could be mapped to a model operation which signals an event in the Terminal Domain. The following list gives the model operations possible for a Terminal Domain.

1. Update an attribute value.
2. Signal an imminent event.
3. Signal a delayed event.

[1]Transitioning out of a state is not intended to confuse the Moore orientation of the state model. Much like an *ignored* event, transitioning out of a state can be viewed as a short hand for the set of transitions into the states accessible from all the outbound transitions for the given state.

4. Cancel a delayed event.
5. Invoke a method on a class instance.
6. Update an association reference.
7. Migrate a generalization subclass instance to a new subclass.

Bridgeable conditions interface

To conveniently discuss a bridgeable condition and its context, we assume that when the MX domain detects one of the above bridgeable conditions it invokes a function corresponding to the condition. A bridgeable condition function is responsible for determining if the specific bridgeable condition causes an interaction with a terminal domain. The details of that processing are discussed below.

This conceptual interface is *not* intended to place a requirement on any MX domain implementation to provide such an interface explicitly. Rather, the interface definitions are intended to specify the data types of values available at the time the bridgeable condition is detected and *not* the syntactic form necessary to marshal the data as might be required by some programming language implementation. MX domain implementations are free to gather the data specified as arguments to the conceptual bridgeable condition functions in any convenient manner. All bridgeable condition functions are defined as functions taking a single argument which is of *Tuple* type.

The tuple heading for the bridgeable condition interface is as follows:

Table 3: Bridgeable condition argument tuple heading

Domain	Class	InstID	Qualifier	Parameters
String	String	Tuple	String	Tuple

where:

Domain
The name of the Initiator Domain where the condition was executed.

Class
The name of the class in *Domain* whose instance executed the condition.

InstID
A tuple giving the identifier of the instance of *Class* that executed the bridgeable condition. Note, the heading of the **InstID** tuple is a projection of the relation heading for *Class*, projected across a set of attribute names that form an identifier of *Class*.

Qualifier
A value which further describes the particulars of the bridgeable condition. Bridgeable conditions represent a semantic action taken by a class instance and the qualifier specifies some aspect of that action. For example, the bridgeable condition of having entered a state is qualified by a value representing the name of the entered state.

Parameters
A tuple holding values associated with the bridgeable condition. Note, the heading of the **Parameters** tuple is associated with the specific bridgeable condition. Some bridgeable conditions do not carry any parametric data and the Parameters tuple would then be the empty tuple.

The bridgeable condition functions do *not* return anything. There is no *call site* defined for a bridgeable condition to which a value could be returned and there is no defined interface between the conceptual bridgeable condition function and any executing action of the Initiator Domain. This choice has implications discussed later.

Closer examination of the argument tuple heading shows that the bridgeable condition uses *Domain*, *Class*, and *InstID* to specify the locale of where the bridgeable condition was executed.

The *Qualifier* attribute is a string and specifies the model subcomponent affected by the execution of a bridgeable condition. The set of allowed strings is determined by what the qualifier represents in the context of the bridgeable condition. The following table gives an abbreviated name for the bridgeable condition and defines the allowed set Strings for the qualifier of each bridgeable condition.

Table 4: Bridgeable condition qualifier value sets

Bridgeable Condition	**Qualifier value set**
State entered	State ∈ {*Class* state names}
State exited	State ∈ {*Class* state names}
Attribute read	Attribute ∈ {*Class* attribute names}
Attribute updated	Attribute ∈ {*Class* attribute names}
Event signaled	Event ∈ {*Class* event names}
Delayed event signaled	Event ∈ {*Class* event names}
Delayed event canceled	Event ∈ {*Class* event names}
Method invoked	Method ∈ {*Class* method names}
Association updated	Association ∈ {Association names for which *Class* is a referring participant}
Subclass migrated	Class ∈ {*Subclass* names of a generalization}

N.B. that the set of allowed values for the qualifier of a bridgeable condition, is a function of the Domain, Class, and Bridgeable Condition. For example, if we wish to have the MX domain detect the bridgeable condition of **State entered**, the allowed values for the State qualifier are only those state names defined for the Domain and Class of the transitioning instance. This is unusual. We assume that the means used to specify the bridgeable condition has access to a population of the meta-model for the domain and that there is some appropriate class in the domain's meta-model for which a restriction based on Domain and Class followed by a projection across the Domain, Class, and another attribute yields a relation value containing the specific names allowed as a qualifier value for the bridgeable condition of an instance of the given Domain and Class.

The bridgeable condition function argument tuple always contains the **Domain**, **Class**, **InstID**, and **Qualifier** attributes. The following table relates a bridgeable condition to the heading of the **Parameters** tuple for each of the conceptual bridgeable conditions functions.

Table 5: Bridgeable condition parameter definitions

Condition	**Parameters**
State entered	The arguments to *State* as a tuple. Note the heading of the **Parameters** tuple must match the defined arguments for *State*. If *State* does not take any arguments, then the tuple is the empty tuple.
State exited	The empty tuple. There are no parameters for this condition.
Attribute read	A tuple of degree one whose only attribute is the name of the *Class* attribute that was read and whose data type matches that of *Class* attribute named in the qualifier, and whose value is that obtained when the attribute was read.
Attribute updated	A tuple of degree one whose only attribute is the name of the *Class* attribute that was updated and whose data type matches that of the *Class* attribute, of the *Class* attribute, and whose value was assigned to the attribute.
Event signaled	The arguments to *Event*. Note the heading of the **Parameters** tuple must match the defined arguments for *Event*. If *Event* does not carry any event parameters, then the tuple is the empty tuple.
Delayed event signaled	A tuple containing two attributes. One attribute is named, *Delay*, and is the number of milliseconds of delay for the signaled event. The other attribute is named, *Arguments* and contains the arguments to *Event*. Note the heading of the **Arguments** tuple must match the defined arguments for *Event*. If *Event* does not carry any event parameters, then the tuple is the empty tuple.

Table 5: (continued)

Condition	Parameters
Delayed event canceled	There are no parameters for this condition.
Method invoked	The arguments with which *Method* was invoked as a tuple. The heading of the tuple must match the parameter names and data types defined for the parameters of *Method*.
Association updated	A tuple whose heading is the set of referential attributes for the association named in the qualifier in *Class* with their defined data types and corresponding values.
Subclass migrated	A tuple containing two attributes. A *Generalization* attribute gives the name of the generalization for which a subclass was migrated. An *Instance* attribute is a tuple type that gives the value of the newly migrated subclass.

Model operation interface

By analogy to the bridgeable condition functions, we define the model operations as conceptual functions which are executed in the Terminal Domain. Again, this description is *not* intended to constrain the manner in which an MX domain may implement executing a model operation. Rather, this is intended to provide a more convenient way to talk about the parameters which must be available to carry out the intent of the model operation.

A model operation function has the same interface as a bridgeable condition function, namely a single tuple with attributes of **Domain**, **Class**, **InstID**, **Qualifier** and **Parameters**.

Also like bridgeable conditions, the **Qualifier** for a model operation has string values that depend upon the operation. The following table shows the meaning of the qualifiers for each model operation.

Table 6: Model operation qualifier value sets

Operation	Qualifier
Update attribute	Attribute ∈ {*Class* attribute}
Signal event	Event ∈ {*Class* event}
Signal delayed event	Event ∈ {*Class* event}
Cancel delayed event	Event ∈ {*Class* event}
Invoke method	Method ∈ {*Class* method}
Update association	Association ∈ {association in which *Class* is a referring participant}
Migrate subclass	Class ∈ {*Subclass* of a generalization}

The following table shows the heading of the **Parameter** tuple for each model operation function.

Table 7: Model operation parameter definitions

Operation	Parameters
Update attribute	A tuple of degree one whose only attribute is the name of the *Class* attribute to update whose data type matches that of *Class* attribute named in the qualifier, and whose value to be the updated value of the attribute.
Signal event	The arguments to *Event*. Note the heading of the **Parameters** tuple must match the defined arguments for *Event*. If *Event* does not carry any event parameters, then the tuple is the empty tuple.

Table 7: (continued)

Operation	Parameters
Signal delayed event	A tuple containing two attributes. One attribute is named, *Delay*, and is the number of milliseconds of delay for the signaled event. The other attribute is named, *Arguments*, and is of Tuple type which contains the arguments to *Event*. Note the heading of the **Arguments** tuple must match the defined arguments for *Event*. If *Event* does not carry any event parameters, then the tuple is the empty tuple.
Cancel delayed event	The empty tuple. There are no parameters for this condition.
Invoke method	The arguments to be given to *Method* when it is invoked. The heading of the tuple must match the parameter names and data types defined for the parameters of *Method*.
Update association	A tuple whose heading is the set of referential attributes for the association named in the qualifier in *Class* with their defined data types and corresponding values.
Migrate subclass	A tuple containing two attributes. A *Generalization* attribute gives the name of the generalization for which a subclass was migrated. An *Instance* attribute is a tuple type that gives the new value to migrated subclass.

Wiring specification

We define a *wiring specification* as a relation value containing the correspondence between an initiator domain, class, bridgeable condition, and condition qualifier value and a terminal domain, class, model operation, and operation qualifier of a terminal domain. The initiator and terminal domains in a wiring specification must be distinct, *i.e.* one may *not* wire a bridge reflexively.

The heading of the wiring specification relation is:

Table 8: Wiring specification relation heading

Initiator Domain	**Initiator Class**	**Bridgeable Condition**	**Condition Qualifier**	**Terminal Domain**	**Terminal Class**	**Model Operation**	**Operation Qualifier**
String	String	String	String	String	String	String	String

The following is the wiring specification, in tabular form, for the LUBE / SIO example.

Table 9: LUBE / SIO Wiring Specification

Initiator Domain	**Initiator Class**	**Bridgeable Condition**	**Condition Qualifier**	**Terminal Domain**	**Terminal Class**	**Model Operation**	**Operation Qualifier**
LUBE	Injector	State entered	Start injection	SIO	I/O Point	Update attribute	Value
LUBE	Injector	State entered	Quit low pressure injection	SIO	I/O Point	Update attribute	Value
LUBE	Injector	State entered	Complete good injection	SIO	I/O Point	Update attribute	Value
LUBE	Injector	State entered	MONITORING	SIO	Conversion Group	Signal event	Sample
LUBE	Injector	State entered	SLEEPING	SIO	Conversion Group	Signal event	Stop

Table 9: (continued)

Initiator Domain	Initiator Class	Bridgeable Condition	Condition Qualifier	Terminal Domain	Terminal Class	Model Operation	Operation Qualifier
SIO	Range Limitation	State entered	OUT OF RANGE	LUBE	Injector	Signal event	Above inject pressure
SIO	Range Limitation	State entered	IN RANGE	LUBE	Injector	Signal event	Below inject pressure
SIO	Range Limitation	State entered	OUT OF RANGE	LUBE	Injector	Signal event	Above dissipation pressure
SIO	Continuous Input Point	Attribute updated	Value	LUBE	Injector	Update attribute	Pressure

Again, we make no statement as to how the MX domain obtains the wiring specification. We assume that during translation the data is produced in a suitable form for the MX domain from some population of a bridging meta-model. Note that the wiring specification is *global* in the sense that the MX domain must be able to detect the occurrence of bridgeable conditions in any initiator domain and direct the execution of model operations in any terminal domain.

Note also that by treating the wiring specification as a relation value, each tuple must be unique. This implies that the same qualified bridgeable condition of a class in an initiator domain may be associated with multiple model operations in different terminal domains. Consider the first row in the above wiring specification table. When the LUBE domain enters the **Start injection** state there is a wiring specification entry to update the value of an I/O Point in the SIO domain. This is an I/O Point used to actuate the injection machinery. Although there is no user interface (UI) domain in the *Models to Code* example, it is easy to envision that the system could have a user interface and another wiring specification entry with the same Initiator Domain, Initiator Class, Bridgeable Condition, and Condition Qualifier could specify a model operation in the UI domain to update the value of some UI element that would give a visual indication to the user when injection starts.

The wiring specification defines how the semantics of the initiator domain classes are reflected into the classes of a terminal domain. Bridgeable conditions and model operations are instance based and it is necessary to map the instance correspondence between the initiator domain and the terminal domain. We define an *instance correspondence map* to be a relation value with the following heading:

Table 10: Instance Correspondence Map Heading

Initiator Domain	Initiator Class	Initiator InstID	Terminal Domain	Terminal Class	Terminal InstID
String	**String**	**Tuple**	**String**	**String**	**Tuple**

For our example, the instance correspondence map is shown below in tabular form.

Table 11: LUBE / SIO Instance Counterparts

Initiator Domain	Initiator Class	Initiator InstID	Terminal Domain	Terminal Class	Terminal InstID
LUBE	Injector	IN1	SIO	I/O Point	IOP4
LUBE	Injector	IN2	SIO	I/O Point	IOP5
LUBE	Injector	IN3	SIO	I/O Point	IOP6
LUBE	Injector	IN1	SIO	Conversion Group	CG1

Table 11: (continued)

Initiator Domain	Initiator Class	Initiator InstID	Terminal Domain	Terminal Class	Terminal InstID
LUBE	Injector	IN2	SIO	Conversion Group	CG2
LUBE	Injector	IN3	SIO	Conversion Group	CG3
SIO	Range Limitation	RL1	LUBE	Injector	IN1
SIO	Range Limitation	RL2	LUBE	Injector	IN2
SIO	Range Limitation	RL3	LUBE	Injector	IN3
SIO	Range Limitation	RL4	LUBE	Injector	IN1
SIO	Range Limitation	RL5	LUBE	Injector	IN2
SIO	Range Limitation	RL6	LUBE	Injector	IN3
SIO	Range Limitation	RL7	LUBE	Injector	IN1
SIO	Range Limitation	RL8	LUBE	Injector	IN2
SIO	Range Limitation	RL9	LUBE	Injector	IN3
SIO	Continuous Input Point	IOP1	LUBE	Injector	IN1
SIO	Continuous Input Point	IOP2	LUBE	Injector	IN1
SIO	Continuous Input Point	IOP3	LUBE	Injector	IN1

Note we are not being precise with the values of the Initiator InstID and Terminal InstID attributes. The Initiator InstID and Terminal InstID attributes are typed as tuples whose heading matches the attributes of an identifier and whose value matches that for a particular instance. For example, the identifier for the Injector class consists of a single attribute named, *ID*, which is a system generated value whose type is also named, *ID*. The Initiator Instance attribute tuple properly should be {ID : ID : IN1}. For this simple case, we'll skip the full tuple specification for the sake of clarity, but it is important when there are multiple attributes in the identifier. When a class has multiple identifiers, sanity seems to indicate that it is necessary to agree upon one particular identifier for the instance correspondence relation.

For counterpart classes whose instance populations are constant, the instance correspondence relation can be built at system assembly time. For counterpart classes whose instance population varies during the system run time, it is necessary for the MX domain to maintain the instance correspondence table at run time. This is discussed in a separate section below.

MX Bridging Operations

We are now prepared to discuss conceptually how the MX domain accomplishes detecting bridgeable conditions and executing the corresponding model operations. We assume the wiring specification and instance correspondence relations have been prepared by the translation tools in a form suitable for the platform specifics of the MX domain. The wiring specification is constant for the lifetime of the system (*i.e.*, we specifically exclude any notions of dynamic, run-time bridge specifications). It is assumed that the wiring specification is encoded in the appropriate programming language and in a data structure assumed by the MX domain. The same can be said for the instance correspondence relation. Whether a domain's instance population is constant or not, any corresponding instances in the initial instance population must be present in the instance correspondence relation value.

We use the term, *bridge operation*, to encompass the following conceptual execution:

1. Detecting the execution of a wired bridgeable condition in the Initiator Domain.
2. Finding the corresponding set of instances in the Terminal Domain for which a model operation is specified by the wiring specification.
3. Formulating the model operation parameters from the parameters of the bridgeable condition.
4. Executing the model operation in the context of the Terminal Domain.

Detecting wired bridgeable conditions

Whenever a domain executes a bridgeable condition, the MX domain must determine if the condition has a wiring specification to a terminal domain. The information available to the MX domain to make this determination is given by the context of the execution and the value of the conceptual bridgeable condition function **Argument** parameter. This information was detailed above. It is then necessary to restrict the wiring specification relation to those tuples where the conjunction of the following Boolean expressions is true.

```
wiring spec.Initiator Domain = Argument.Domain
wiring spec.Initiator Class = Argument.Class
wiring spec.Bridgeable Condition = invoked bridgeable condition name
wiring spec.Condition Qualifier = Argument.Qualifier
```

Consider when the IN1 instance of the LUBE::Injector class has entered the **Start Injection** state. Conceptually this happens when the event dispatch mechanism of the MX domain causes a transition in the state machine of the IN1 instance and invokes the bridgeable condition function:

```
State entered(
    Argument :
    Tuple {
        Domain : String,
        Class : String,
        InstID : Tuple {ID : ID},
        Qualifier : String,
        Parameters : Tuple {}
    } :
    {
        Domain : LUBE,
        Class : Injector,
        InstID : IN1,
        Qualifier : Start injection
        Parameters : {}
    }
```

The MX domain then computes the restriction of the wiring specification where:

```
wiring spec.Initiator Domain = Argument.Domain AND
wiring spec.Initiator Class = Argument.Class AND
wiring spec.Bridgeable Condition = State entered AND
wiring spec.Condition Qualifier = Argument.Qualifier
```

This restriction on the wiring specification yields the following relation value:

Table 12: Wiring Specification Restriction for Start Injection example

Initiator Domain	Initiator Class	Bridgeable Condition	Condition Qualifier	Terminal Domain	Terminal Class	Model Operation	Operation Qualifier
LUBE	Injector	State entered	Start injection	SIO	I/O Point	Update attribute	Value

Consider also a counter example. When an instance of LUBE::Injector enters the **Cancel injection** state, there are no tuples in the wiring specification which match and the resulting restriction is empty. An empty restriction of the wiring specification implies that the condition has no wiring specification. We expect that most of the execution in a domain does not yield bridgeable conditions which appear in a wiring specification and thus yield an empty restriction on the wiring specification. MX domain implementations will be pressed to find a computationally efficient way to detect the empty case to limit the intrusiveness of implicit bridging on the normal execution of domains.

Consider one further example of this step in the bridge operation. When an instance of SIO::Range Limitation enters the **OUT OF RANGE** state, the restriction of the wiring specification returns a relation value of cardinality two.

Table 13: Wiring Specification Restriction for Range Limitation example

Initiator Domain	Initiator Class	Bridgeable Condition	Condition Qualifier	Terminal Domain	Terminal Class	Model Operation	Operation Qualifier
SIO	Range Limitation	Enter state	OUT OF RANGE	LUBE	Injector	Signal event	Above inject pressure
SIO	Range Limitation	Enter state	OUT OF RANGE	LUBE	Injector	Signal event	Above dissipation pressure

In this case, there are potentially two different model operations to perform. Which operations are actually performed is determined by the next step in the bridge operation which finds the instance correspondence bound to the wiring specification.

Instance correspondence

Restricting the instance correspondence relation to those tuples where the Initiator Instance matches the instance executing the bridgeable condition yields the potential instance participation in the bridge operation, *i.e.* a restriction where the conjunction of the following Boolean expressions is true:

```
Initiator Domain = Argument.Domain AND
Initiator Class = Argument.Class AND
Initiator Instance = Argument.InstID
```

Continuing with our Injector example, restricting the counterpart instance relation by:

```
Initiator Domain = LUBE AND
Initiator Class = Injector AND
Initiator Instance = Tuple {ID : ID : IN1}
```

gives the following result relation:

Table 14: Instance Correspondences for Start Injection example

Initiator Domain	Initiator Class	Initiator Instance	Terminal Domain	Terminal Class	Terminal Instance
LUBE	Injector	IN1	SIO	I/O Point	IOP4
LUBE	Injector	IN1	SIO	Conversion Group	CG1

Performing a join[2] of the restricted wiring specification and the restricted instance correspondence relations across:

```
wirespec.Initiator Domain = instcp.Initiator Domain AND
wirespec.Initiator Class = instcp.Initiator Class AND
wirespec.Terminal Domain = instcp.Terminal Domain AND
wirespec.Terminal Class = instcp.Terminal Class
```

yields the required relation to direct the execution of a model operation to a particular instance in the Terminal Domain. For the LUBE::Injector example, the resulting relation value is:

[2] A “natural” join if you must.

Table 15: Bridge Operation Specification for Start Injection example

Initiator Domain	**Initiator Class**	**Initiator Instance**	**Bridgeable Condition**	**Condition Qualifier**	**Terminal Domain**	**Terminal Class**	**Terminal Instance**	**Model Operation**	**Operation Qualifier**
LUBE	Injector	IN1	State entered	Start injection	SIO	I/O Point	IOP4	Update attribute	Value

This relation value informs us that when the IN1 instance of LUBE::Injector enters the **Start injection** state that we want to execute an **Update attribute** operation on the **Value** attribute of the IOP4 instance of SIO::I/O Point.

For the SIO::Range Limitation example, the restricted counterpart instance relation is:

Table 16: Instance Correspondences for Range Limitation example

Initiator Domain	**Initiator Class**	**Initiator Instance**	**Terminal Domain**	**Terminal Class**	**Terminal Instance**
SIO	Range Limitation	RL1	LUBE	Injector	IN1
SIO	Range Limitation	RL2	LUBE	Injector	IN2
SIO	Range Limitation	RL3	LUBE	Injector	IN3
SIO	Range Limitation	RL4	LUBE	Injector	IN1
SIO	Range Limitation	RL5	LUBE	Injector	IN2
SIO	Range Limitation	RL6	LUBE	Injector	IN3
SIO	Range Limitation	RL7	LUBE	Injector	IN1
SIO	Range Limitation	RL8	LUBE	Injector	IN2
SIO	Range Limitation	RL9	LUBE	Injector	IN3

Assuming the bridgeable condition is detected when the RL1 instance of SIO::Range Limitation has entered the **OUT OF RANGE** state, then the resulting bridge operation relation value is:

Table 17: Bridge Operation Specification for Range Limitation example

Initiator Domain	**Initiator Class**	**Initiator Instance**	**Bridgeable Condition**	**Condition Qualifier**	**Terminal Domain**	**Terminal Class**	**Terminal Instance**	**Model Operation**	**Operation Qualifier**
SIO	Range Limitation	RL1	Enter state	OUT OF RANGE	LUBE	Injector	IN1	Signal event	Above inject pressure

Note in this case how the instance correspondence relation associates three different instances of Range Limitation (RL1, RL4, and RL7) to the same Injector instance (IN1). These correspond to the three different events sent by Range Limitation Instances to an Injector instance. In the case of the example, it is the instance of Range Limitation which is monitoring the inject pressure (RL1) for Injector, IN1, which triggers signaling the event to the IN1 instance of LUBE::Injector.

Formulating the Model Operation

After computing the bridge operation specification, the next step in is to formulate the model operation. Although our examples have only yielded a bridge operation specification of cardinality one, in general several bridge operations may be specified for a given bridgeable condition and a given initiator domain instance.

One simple way to solve this situation is to associate with each tuple of the wiring specification an *ad hoc* function which would compute the model operation argument tuple from the information present in the bridge condition argument tuple. We are trying to avoid *ad hoc* processing just to perform simple mappings that can be specified in data. The remainder of this section explores some ideas in this area. Note that we specifically do not allow for any user supplied “computation” or “hook” in the formulation of the model operation argument tuple. The idea is to have the system build the argument tuple by a mapping function specified at system assembly time. It may be the case that the reliance on specified data is *not* sufficient to handle some reasonable use cases. Since we are always dealing with finite sets, specifying functions as a formula are never strictly necessary. But practicalities do apply to large sets. For example, converting an integer range into one of three enumerated values can be done with a surjective mapping from the integer range to the enumerated values. But if the integer range is large, say 10000, then using an enumerated mapping is so clumsy that clearly a combination of integer relation operations such as greater than or less than is a better use of computational resources. This is an area for further consideration. It may also be the case that the specification mechanism presented here is so unwieldy even in its most optimistic form that it is not practical to use the technique. Again, this is another area for continuing study.

The interface to a model operation function is the same as for the bridgeable conditions functions, namely a single tuple whose heading is:

Table 18: Model operation argument tuple heading

Domain	**Class**	**InstID**	**Qualifier**	**Parameters**
String	String	Tuple	String	Tuple

A tuple of this type can be constructed by the concatenation of two tuples, one with a heading consisting of **Domain**, **Class**, **InstID**, and **Qualifier** and the other with the heading of **Parameters**.

The first tuple in the concatenation is easily constructed from the bridge operation specification as:

Domain	**Class**	**InstID**	**Qualifier**
String	String	Tuple	String
bopspec.Terminal Domain	bopspec.Terminal Class	bopspec.Terminal Instance	bopspec.Operation Qualifier

It remains to construct the **Parameters** tuple for the model operation. With a **Parameters** tuple, then the concatenation of it with the previous tuple derived from the bridge operation spec yields the model operation argument tuple.

There are three sources of data that can be used to construct the **Parameters** tuple for a model operation:

1. A literal constant tuple value.
2. The data contained in the **Parameters** tuple of the bridgeable condition function.
3. A bridgeable condition qualifier mapping which maps values of the bridgeable condition qualifier to the value of a single attribute in the **Parameters** tuple.

Both constant tuple values and qualifier maps are values which can be specified at system assembly time. The values in the bridgeable condition **Parameters** attribute represents quantities that can only be known at run time.

We seek a function which takes the bridgeable condition **Parameters** and **Qualifier** attributes plus some, as yet unspecified, bridge mapping data to produce the **Parameters** tuple for the model operation **Argument** tuple.

We envision the function operating as follows:

- The model operation **Parameters** attribute is a tuple concatenation of of tuples obtained from
  - zero or more literal tuple constants
  - a series of transformations on the bridgeable condition **Parameters** tuple
  - a tuple obtained by mapping the bridgeable condition **Qualifier** value

The following figure shows a schematic view of the processing to obtain the model operation **Parameters** tuple.

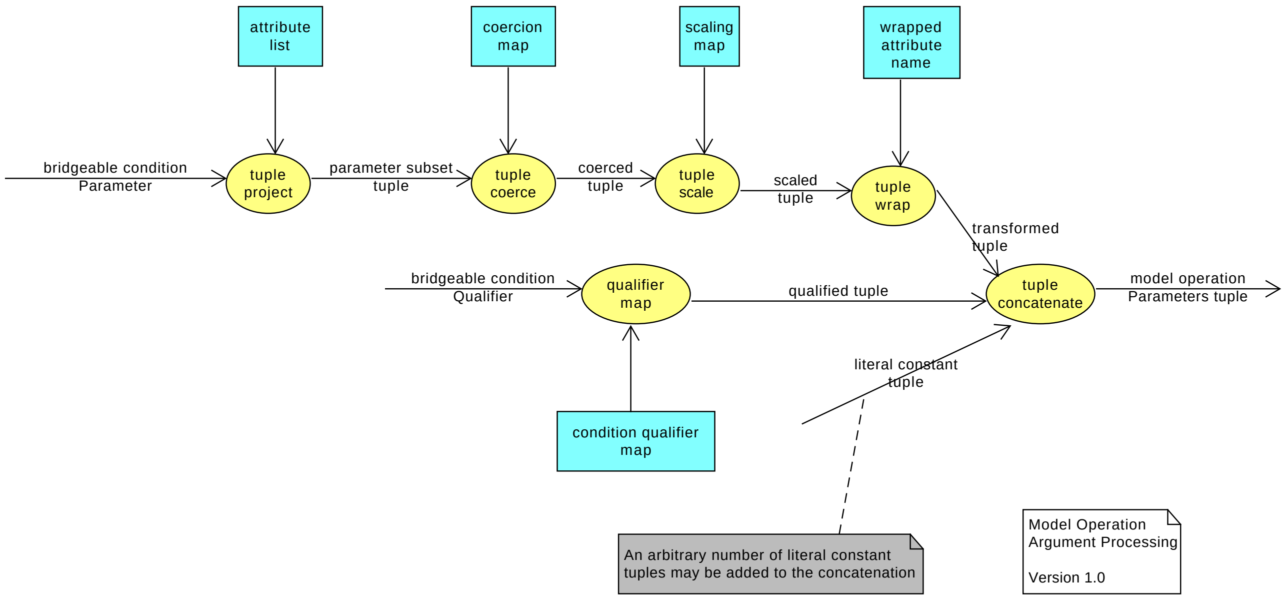

Figure 7: Model Operation Parameter Processing

Transforming the bridgeable condition Parameter

The following transformation steps take values from the bridgeable condition **Parameters** tuple and transform them into some component of the model operation **Parameters** tuple.

1. The **Parameter** tuple is projected to obtain the subset (possibly improper) that contribute to the result. The data which drives this operation is a list of names of attributes in the **Parameter** tuple. An empty list of names implies the null-ary projection implying that no attributes of **Parameters** contribute to the result.
2. The tuple formed by the subset of attributes from the previous project step is *coerced*. Coercion can change the name of an attribute and it's data type. The result value is a type conversion from the original data type to the coerced attribute data type. The processing is governed by a *coercion map* which is a relation value with the following heading:

Table 19: Coercion map heading

BC_Attribute	**MO_Attribute**	**MO_Datatype**
String	String	Datatype

The cardinality of the coercion map must equal the degree of the **Parameter** tuple subset. The values of the **BC_Attribute** must match the attribute names of the input tuple. The values of the **MO_Attribute** must match the attribute names for the model operation **Parameter** tuple and the values are typed according to the **MO_Datatype** attribute. It is assumed that there is a system supplied type conversion function that is used to transform the bridgeable condition value to the value type of the model operation. This is discussed below. The value type for the attribute is known because the heading of the model operation **Parameter** tuple is known at system assembly time.

3. The coerced tuple is then scaled. Only linear scaling is supported and is supplied to handle any change of units that might be required. The scaling is controlled by a relation value with the following heading:

Table 20: Scaling map heading

Attribute	**Gain**	**Bias**
String	@TypeOf(Attribute)	@TypeOf(Attribute)

where “@TypeOf(Attribute)” is the data type of the attribute given by the value of **Attribute**.

4. The final transformation step handles situations where the model operation **Parameter** tuple has an attribute which is also of tuple type. In that case the scaled tuple must be *wrapped*. Wrapping is controlled by the name given to the wrapped attribute. The *tuple wrap* operation creates a tuple of degree one whose name is given by the *wrapped attribute name*, whose data type is tuple, and whose value is the scaled tuple input. Only a few model operation have a **Parameter** tuple which has a tuple valued attribute and an empty wrapped attribute name is interpreted to perform no wrapping operation.

Basic type conversions

We assume the system supplies a set of basic data types and a function which can convert the bit representation of a value of a given type into a value whose bit representation is given by another type. At a minimum, the system should supply basic types that are practical approximations to the following sets:

- Unsigned integers ($\mathbb{N}$)
- Signed integers ($\mathbb{Z}$)
- Real numbers ($\mathbb{R}$)
- Text strings

Systems may choose to provide types which approximate other sets as might be necessary to meet system requirements. System supplied data types for date/time and complex numbers may be useful. Note that xUML models assume that the system can generate arbitrary identifiers. Such values are only suitable for comparison for equality. There is no conversion available between system generated identifiers and any other basic type.

We envision a function:

```
convert(source type, source value, result type) => result value
```

where *source type* is some encoding of the basic system supplied types. Note it is possible for the data type conversions to fail. For example, converting the string, "foo", to an unsigned integer is not defined. However, we do assume that every non-string data type can be constructed from an appropriately formatted text string, *e.g.* there are well established mechanisms to convert the string, "42", into a bit encoding that represents the number 42.

Bridgeable condition qualifier mapping

The bridgeable condition qualifier usually carries a major portion of the semantics associated with a bridge specification. As such, mapping the qualifier value to a tuple of degree one is supported.

An *condition qualifier map* associates the *value* of a bridgeable condition qualifier to an attribute name and attribute value to be used as part of the model operation **Parameter** value. The action qualifier map is a relation value having the following heading:

Table 21: Condition qualifier map heading

Qualifier	**Result**
String	Tuple

The condition qualifier map may be used for multiple bridge operations. For that case, the bridge operations must all have the same wiring specification and the values of the **Qualifier** attribute are the projection of the **Condition Qualifier** from the wiring specification.

The *qualifier map* operation constructs finds the tuple in the condition qualifier map which matches the value of the bridgeable condition **Condition Qualifier** value and extracts the corresponding tuple. If the condition qualifier map is the empty relation the result of the qualifier map operation is the empty tuple.

Model operation argument examples

In this section, we continue with the LUBE / SIO examples to show how the parameters to the model operation are formed.

Lube injection

First, we consider the Injector states which actuate or deactuate the injection mechanism. The wiring specification wires the *State entered* condition to an *Update attribute* operation. The **Parameter** tuple of the model operation argument is a tuple of degree one giving the attribute of the I/O point to update and its value. The entered states in the Injector class model that are the Bridgeable Condition in the wiring specification do not have any arguments and so the **Parameter** tuple of the bridgeable condition cannot contribute any data to the model operation. This leaves us with a simple condition qualifier map as shown here:

Table 22: Model operation Injector state to I/O Point qualifier map

Qualifier	**Result**
Start injection	Tuple {Value : Numeric : 1}
Quit low pressure injection	Tuple {Value : Numeric : 0}
Complete good injection	Tuple {Value : Numeric : 0}

The following figure shows the processing to obtain the model operation **Parameter** tuple. For simplicity, the processing steps which do not directly contribute to the result are not shown.

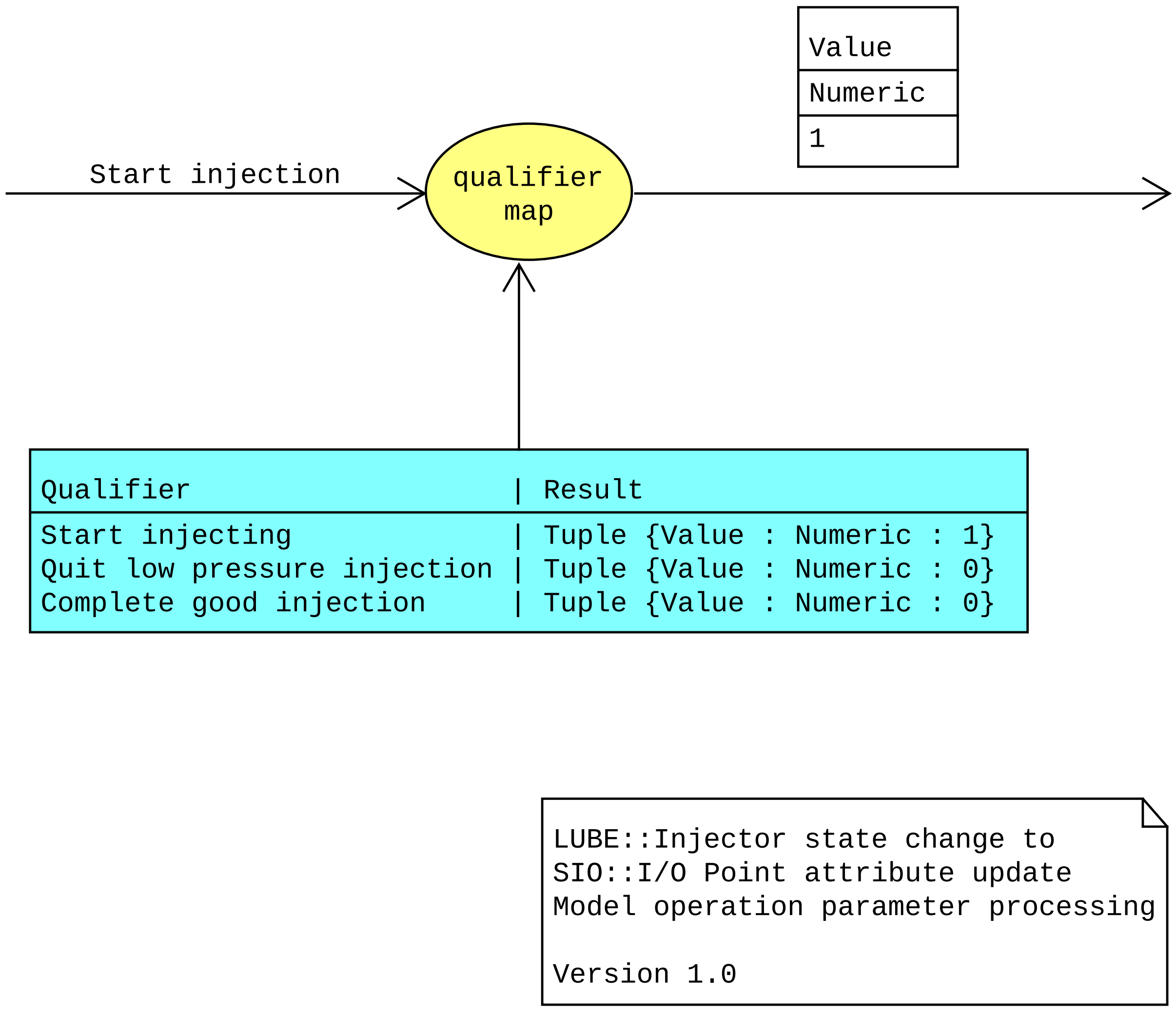
Value
Numeric
1
Start injection
qualifier map
Qualifier | Result
Start injecting | Tuple {Value : Numeric : 1}
Quit low pressure injection | Tuple {Value : Numeric : 0}
Complete good injection | Tuple {Value : Numeric : 0}
LUBE::Injector state change to
SIO::I/O Point attribute update
Model operation parameter processing
Version 1.0

Figure 8: Lube::Injector / SIO::I/O Point Model Operation Parameter

Monitoring injection pressure

Referring to the LUBE / SIO wiring specification, in the LUBE domain the Injector states **MONITORING** and **SLEEPING** are wired to the Conversion Group class in the SIO domain. The wiring accomplishes starting and stopping of monitoring the injection pressure. The monitoring requirement in LUBE is reflected into SIO by starting and stopping the periodic conversion of injector pressure transducer signals. This is an example of a model operation that is completely specified by the model operation qualifier and the **Parameters** argument to the model operation function is simply a constant, namely the empty tuple.

Injection pressure limits

The LUBE::Injector state model reacts to pressure values that are out of specified ranges. The LUBE domain depends upon SIO to notify it when those ranges are exceeded. The SIO::Range Limitation class detects if input values lie outside of the specified bounds. This entry in the wiring specification shows the SIO domain as the Initiator Domain and is an example of a service domain, SIO in this case, initiates a bridge operation to fulfill the requirements delegated to it. This is another example of the model operation being completely described by the wiring specification because the LUBE::Injector events do not have any arguments. So, the empty tuple serves as an appropriate value for the model operation **Parameters** argument.

Measured pressure values

This last example is associated with updating the value of the LUBE::Injector **Pressure** attribute when a new measurement is made. The SIO::Continuous Input Point instances hold measured values from transducers. The wiring specification associates updates to the **Value** attribute of a Continuous Input Point to a model operation to update the **Pressure** attribute of a LUBE::Injector instance. The **Parameters** tuple for the model operation is composed from the **Value** attribute of a Continuous Input Point by tuple coercion. The coercion map in this case is:

Table 23: Coercion map for Value to Pressure

BC_Attribute	**MO_Attribute**	**MO_Datatype**
Value	Pressure	Numeric

The following figure shows the transformation processing necessary to take a Continuous I/O Point **Value** into an Injector **Pressure**. Again, we do not show those parts of the general transformation chain which do not contribute to the result.

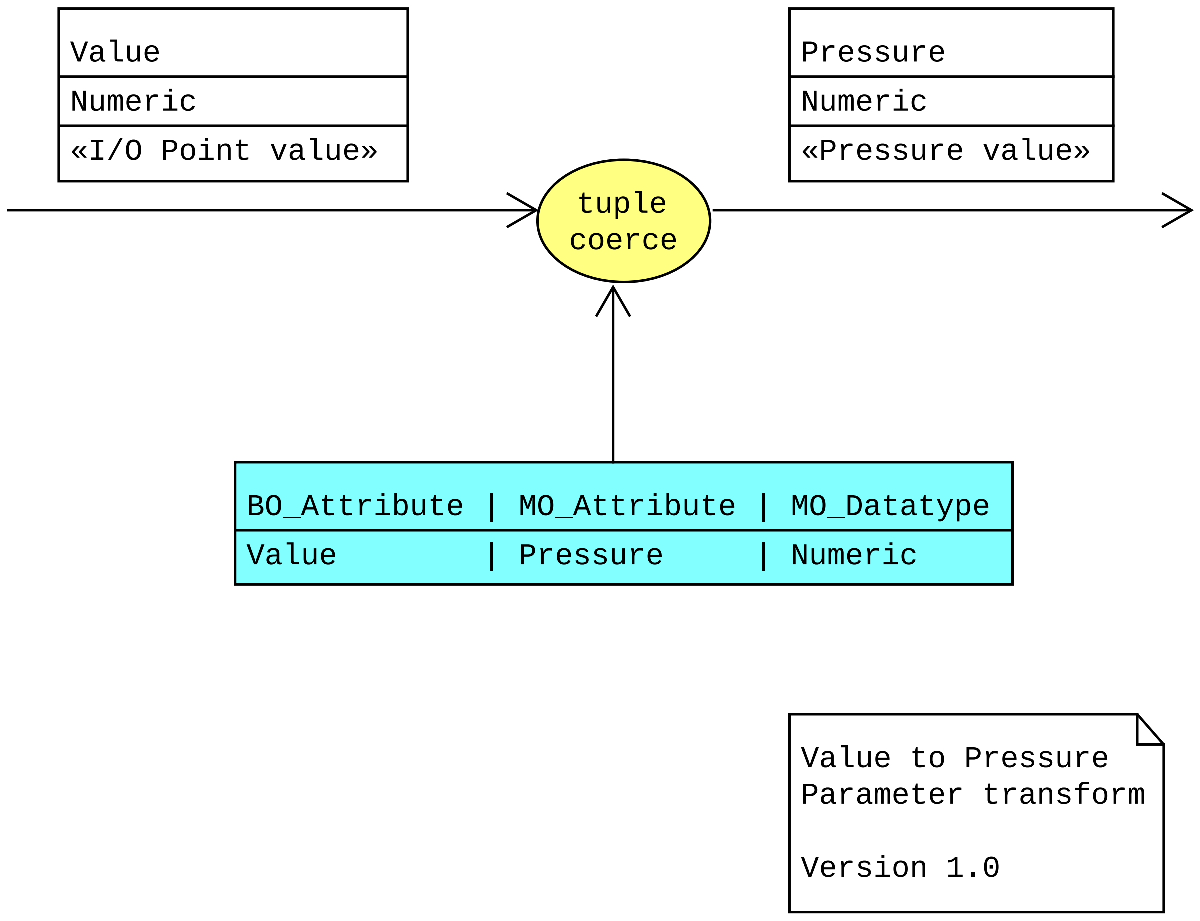
Value
Numeric
«I/O Point value»
tuple coerce
Pressure
Numeric
«Pressure value»
BO_Attribute | MO_Attribute | MO_Datatype
Value | Pressure | Numeric
Value to Pressure Parameter transform
Version 1.0

Figure 9: SIO::Continuous Input Point to LUBE::Pressure Model Operation Parameter

Executing the Model Operation

Once the model operation argument tuple has been assembled, the model operation must be invoked with its argument tuple in the context of the Terminal Domain. We assume that executing the model operation is synchronous with respect to the Initiator Domain, *i.e.* the effects of the model operation are made before execution resumes in the Initiator Domain. This is another aspect of “run to completion” rules. Synchronous model operation execution needs more consideration of its implications upon system determinism.

We see three cases here:

1. For systems that run in a single process, obtaining access to the Terminal Domain context is within the bounds of the MX domain. Execution of the model operation should present no substantial problems.
2. For systems that have domains running in separate processes, there will be at lease one MX domain in each process. Executing a model operation in the context of a different process implies some form of interprocess communications (IPC) between the MX domain in each process. This carries with it the usual complications associated with serializing the model operation onto the IPC mechanism.
3. For system that have domains running in separate computers, the execution of the model operation comes with the usual difficulties of distributed systems. The implication is there is a network communications mechanism which is fraught with all the usual difficulties. This type of system has many different failure modes which might disrupt the notion of running model operations to completion.

Bridging Thread of Control

Just as we recognize a *domain thread of control* when a domain runs, there is an analogous concept of the *bridging thread of control*. The bridging thread of control is the trace of domain activations stemming from a given bridgeable condition. It would seem clear that the path through the implied graph of domain activations should be acyclic.

It is an open question as to whether executing a model operation in a domain that is activated by a bridgeable condition in another domain implies a transaction in the domain which executes the model operation. First impressions indicate that model operations should be considered as starting a domain thread of control in the Terminal Domain. More study is needed here to determine the implications.

Bridging Dynamic Instance Populations

Important
This part of the notes on implicit bridging is not complete. I have been working a scheme that was analogous to what has been described for bridge operations. In an attempt to work up an example, I discovered that either what I had defined would not work or if it did work would not be worth much.
Right now I’m looking into the concept that Date describes as *compensatory actions*. But for now, I have no solution to propose for handling dynamic populations of counterpart classes.

Comments on implicit bridging

I will impose upon you to make some comments about the implicit bridging scheme presented here.

First, it seems complex, perhaps too much so. Of course, most new things appear complex at first encounter. In this case, I deliberately avoided introducing any more notation than necessary in an attempt to avoid having to learn both bridging concepts and notation. That makes the language of the descriptions more dense and stilted which does not contribute any measure of simplicity.

Even if the ideas about implicit bridging hold some merit, there is much work to explore and refine them. That will truly come when there is a functioning implementation that can be used to judge the engineering practicalities. To my knowledge, there are no functioning xUML MX domains that support implicit bridging.

I can almost hear the ringing in my ears of comments to the effect of, “do we really neeeeeeeed this?” The obvious answer is, no. We have been building systems with xUML for many years without any well defined or accepted implicit bridging. But I believe there are use cases where implicit bridging is beneficial.

Consider building a User Interface domain. This is one case where relatively intrusive access to the application domain data is necessary. Another is a Communications Interface. Many systems have both but the manner and data accessed by UI domain and a COMM domain are quite different. Having to bring out explicit interfaces for both would impact the application domain significantly. One does not expect substantial domain reuse for an application domain. But we know that application domains evolve over time as new requirements arise. The question is would having implicit bridging to service domains, particularly those service domains which need significant access to application domain data ease the problem of building and maintaining an application domain. This needs a suitable case study to determine.

Domain reuse is a goal of using xUML to build systems. I personally have a set of service domains that I carry around. In my case I am resigned to reworking the explicit bridging as necessary for any reuse. It’s not a pleasant task, but is workable and results in better productivity.

There are a few consequences that occur as a result of some of the decisions made.

- Since model operations return no value, it is not possible to implicitly bridge reading an attribute to obtain the attribute value from another domain. This decision was an attempt to avoid having to define an interface from the MX domain into the executing action of a domain that could be used to pass a value. The only way attribute values from one domain are passed to another is via writing to the attribute and implicitly bridging the update in an initiator domain to an update in a terminal domain.
- We assume the action semantics of xUML include subclass migration as a functional primitive. Conceptually, subclass migration involves deleting one subclass instance and creating a different subclass in the same generalization. It is not clear that this situation needs to be accounted for in the dynamic instance population effort. If it is *not* included, then there is no way to have generalization subclasses be counterpart classes. It is not clear whether that is a *good thing* or a *bad thing*. Right now it is simply *not a thing*.

Finally, we all have experienced to need for adequate tooling to undertake system building by modeling. Tooling always lags the continuing improvements in the software methodology concepts. At first appearance, implicit bridging sharply points out our collective lack of tooling to help the situation. It can be handled manually with spreadsheets to tally the correspondences, but only programmatic access to meta-model populations can insure the consistency required and enforce the rules of coherency.