The Shlaer-Mellor Metamodel

Lee W. Riemenschneider

July 5, 2022

Abstract

A metamodel is a model of a model. The metamodel provides the rules and constraints of modeling using a particular paradigm. This metamodel concerns the rules and constraints of the Shlaer-Mellor method, so it is a Shlaer-Mellor metamodel. Ideally, there should only be one Shlaer-Mellor Metamodel, but consensus hasn't been reached by Shlaer-Mellor modeling experts, so this report tries to justify the reasons for this metamodel to be the best candidate for the title, *The Shlaer-Mellor Metamodel*.

Part I

The Shlaer-Mellor Metamodel Domain

Chapter 1

Aspects of a Metamodel Model

A metamodel is a model of a model. The metamodel provides the rules and constraints of modeling using a particular paradigm.

The metamodel provides no guidance outside of its subject matter. e.g., the Shlaer-Mellor metamodel provides rules and constraints for modeling using the Shlaer-Mellor method, but it doesn't provides rules and constraints concerning the representation of the modeling elements. This allows the modeling to be done using any notation (graphical or text) that can unambiguously represent the elements used in the modeling. It also means that the metamodel also doesn't say how the model is to be used. i.e., how it is transformed to machine language, or how it might be run in simulation.

A metamodel is used to describe the rules for constructing a model. It specifies the construction elements and the constraints on the construction elements. It is the modeled depiction of the modeling method, often done using the modeling method.

The metamodel for Shlaer-Mellor modeling has the perspective of a single domain model. All other domains are viewed only as outgoing bridges requiring outside servicing. All accesses to the domain are viewed as incoming bridges that evoke actions within the domain model. Therefore, an instance of metamodel should only be used for verifying a single domain model at a time.

The Shlaer-Mellor Metamodel domain is partitioned into subsystems of closely related objects. This partitioning makes the domain easier to manage, but all the objects belong to the subject matter of Shlaer-Mellor domain modeling.

Chapter 2

The Data Subsystem

The subsystem of the domain model concerned with data usage.

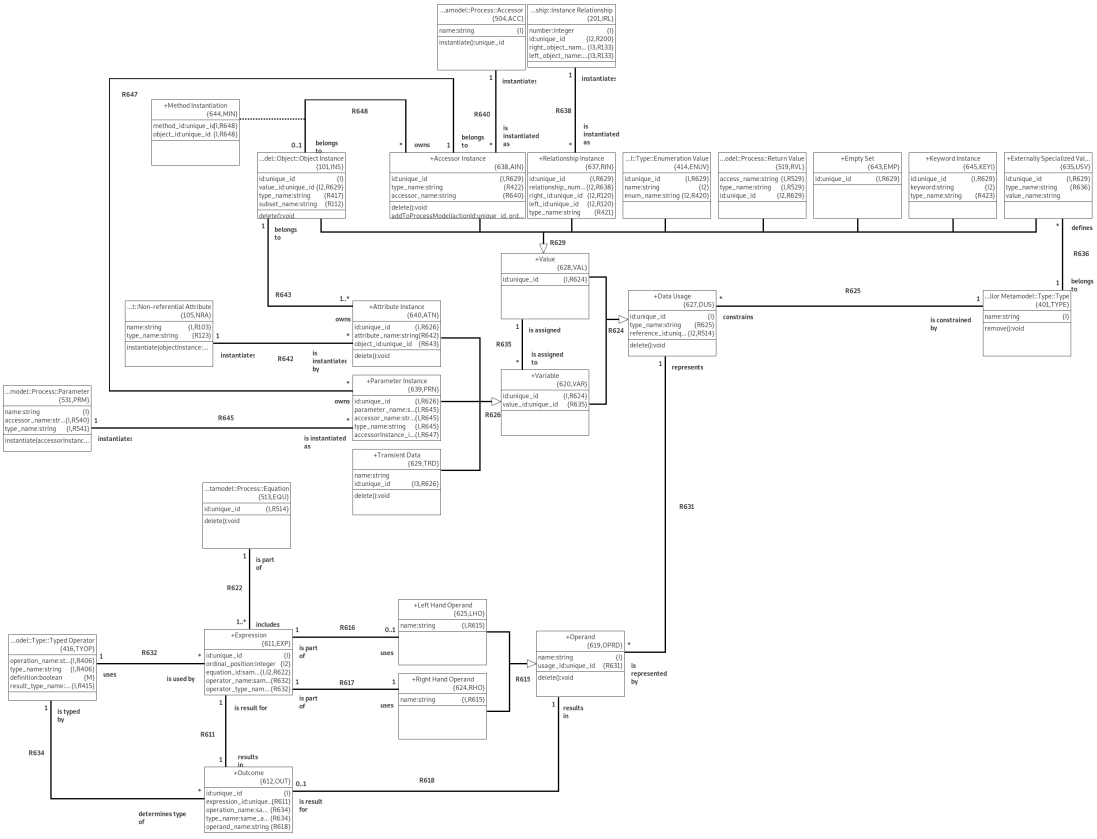

Figure 2.1: Data Subsystem Diagram

2.1 Object and Attribute Descriptions

2.1.1 Accessor Instance

An accessor instance is instantiated in a process model to represent the accessor at that point of access.

An instantiation is needed so the accessor can be represented with different parameter data sets at different places in the process model or within different process models in the domain.

2.1.1.1 Relational Attributes

accessor_name:R640

type_name:R422

***id:R629**

2.1.1.2 Operations

Algorithm 2.1 void Accessor Instance:delete()

```
select many prns related by self->PRN[R647];
for each prn in prns
        prn.delete();
        delete object instance prn;
end for;
select one ins related by self->INS[R648];
if (not empty ins)
        select one min related by self->MIN[R648];
        unrelate self from ins across R648 using min;
end if;
select one val related by self->VAL[R629];
unrelate self from val across R629;
select one dus related by val->DUS[R624];
unrelate val from dus across R624;
delete object instance val;
dus.delete();
delete object instance dus;
```

Algorithm 2.2 void Accessor Instance:addToProcessModel()

```
select one re related by self->VAL[R629]->DUS[R624]->RE[R514];
select any pro from instances of PRO where (selected.action_id == param.actionId
create object instance ela of ELA;
relate re to pro across R502 using ela;
select many prns related by self->PRN[R647];
for each prn in prns
        prn.addToProcessModel( actionId:param.actionId, ordinalPosition:param.or
end for;
```

2.1.2 Attribute Instance

An attribute instance models the usage of a non-referential attribute in the process models.

2.1.2.1 Relational Attributes

object_id:R643

attribute_name:R642

***id:R626**

2.1.2.2 Operations

Algorithm 2.3 void Attribute Instance:delete()

```
select one nra related by self->NRA[R642];
unrelate self from nra across R642;
select one ins related by self->INS[R643];
if (not empty ins)
        unrelate self from ins across R643;
end if;
select one var related by self->VAR[R626];
unrelate self from var across R626;
select one val related by var->VAL[R635];
unrelate var from val across R635;
select one dus related by var->DUS[R624];
unrelate var from dus across R624;
delete object instance var;
dus.delete();
delete object instance dus;
```

2.1.3 Data Usage

Data is used in Shlaer-Mellor via variable or value accesses. Data is always typed.

2.1.3.1 Attributes

***id:unique_id** A unique identifier for the data usage.

2.1.3.2 Relational Attributes

***^{2}reference_id:R514**

type_name:R625

2.1.3.3 Operations

Algorithm 2.4 void Data Usage:delete()

```
select one type related by self->TYPE[R625];
unrelate self from type across R625;
select one re related by self->RE[R514];
unrelate self from re across R514;
re.delete();
delete object instance re;
select many oprds related by self->OPRD[R631];
for each oprd in oprds
        oprd.delete();
        delete object instance oprd;
end for;
```

2.1.4 Empty Set

The empty set value is defined for all types, as all types define a set of values. e.g., 6 is a member of the set defined by the numeric type. The only legal operations performed on an empty set are population and comparison to the empty set.

2.1.4.1 Relational Attributes

***id:R629**

2.1.5 Expression

An expression is a statement containing two operands and an operator.

Expression evaluation in the metamodel is done using Reverse Polish Notation (RPN). The order of construction involves adding the left-hand operand, the the right-hand operand, and then the operator. This allows the metamodel to evaluate the expression and produce an outcome.

2.1.5.1 Attributes

***[2]ordinal_position:integer** Indicates the execution order of the expression within the equation.

***id:unique_id**

2.1.5.2 Relational Attributes

result_type_name:R632

rh_operand_name:R617

lh_operand_name:R616

operator_type_name:R632

operator_name:R632

equation_id:R622

2.1.6 Externally Specialized Value

The externally specialized value is the standard representation of a value that belongs to a set defined by commonly known types (e.g., the set of numerics).

As the actual value isn't that interesting to the metamodel, the attribute, value_name, is used to allow process model construction within the constraints the metamodel. This symbolic representation of the value is akin to the use of defines in C to represent "magic numbers".

2.1.6.1 Attributes

value_name:string A symbolic representative for the value. In the case of a symbolic value type, it could be the actual value.

2.1.6.2 Relational Attributes

type_name:R423

***id:R629**

2.1.7 Keyword Instance

A keyword is a special directive to the architecture. Keywords are non-mathematical operands in equations in the process models.

2.1.7.1 Attributes

***[2]keyword:string** The name of the action the keyword invokes.

2.1.7.2 Relational Attributes

type_name:R423

***id:R629**

2.1.8 Left Hand Operand

This is the operand on the left-hand side of the operator.

2.1.8.1 Relational Attributes

***name:R615**

2.1.9 Method Instantiation

A method instantiation occurs whenever a process model is using an accessor.

2.1.9.1 Relational Attributes

***object_id:R648**

***method_id:R648**

2.1.10 Operand

An Operand is a participant in an expression or sub-expression of an equation. All expressions are evaluated with only two operands, the one on the left-hand side of the operator and the one on the right-hand side of the operator.

2.1.10.1 Attributes

***name:string** A unique identifier for the type of operand.

2.1.10.2 Relational Attributes

***usage_id:R631**

Operations

Algorithm 2.5 void Operand:delete()

```
select one out related by self->OUT[R618];
if (not empty out)
        unrelate self from out across R618;
end if;
select one lho related by self->LHO[R615];
if (not empty lho)
        unrelate self from lho across R615;
        select one exp related by lho->EXP[R616];
        unrelate lho from exp across R616;
        delete object instance lho;
        select one equ related by exp->EQU[R622];
        equ.delete();
else
select one rho related by self->RHO[R615];
        if (not empty rho)
                unrelate self from rho across R615;
                select one exp related by rho->EXP[R617];
                unrelate rho from exp across R617;
                delete object instance rho;
                select one equ related by exp->EQU[R622];
                equ.delete();
        end if;
end if;
```

2.1.11 Outcome

An outcome is the result of an expression or the result of a call to a synchronous accessor (function).

2.1.11.1 Attributes

***id:unique_id**

2.1.11.2 Relational Attributes

result_type_name:R634

operand_name:R618

type_name:R634

operation_name:R634

expression_id:R611

****[2]equation_id:R622**

2.1.12 Parameter Instance

A parameter instance models the usage of a parameter in the process models. If the parameter is being assigned a value in the process model, then it is considered an activated parameter instance. If the parameter is being referenced in the process model, it is considered a placeholder parameter instance. A placeholder parameter instance will be uninitialized, so its associated value will be the empty set.

2.1.12.1 Relational Attributes

***id:R626**

***parameter_name:R645**

***accessor_name:R645**

***type_name:R645**

***accessorInstance_id:R647**

2.1.12.2 Operations

Algorithm 2.6 void Parameter Instance:delete()

```
select one prm related by self->PRM[R645];
unrelate self from prm across R645;
select one ain related by self->AIN[R647];
if (not empty ain)
        unrelate self from ain across R647;
end if;
select one var related by self->VAR[R626];
unrelate self from var across R626;
select one val related by var->VAL[R635];
unrelate var from val across R635;
select one dus related by var->DUS[R624];
unrelate var from dus across R624;
delete object instance var;
dus.delete();
delete object instance dus;
```

Algorithm 2.7 void Parameter Instance:addToProcessModel()

```
select any pro from instances of PRO where ( selected.action_id == param.actionI
create object instance ela of ELA;
select one re related by self->VAR[R626]->DUS[R624]->RE[R514];
relate re to pro across R502 using ela;
```

2.1.13 Relationship Instance

A relationship instance is the instantiation of an instance relationship. Just as object instantiations need to be tracked and handled, so do relationships. This is limited to instance relationships, as associative relationships are handled as object instances.

The relationship represents a table containing a row for every instance of the relationship, and a column for each object participating in the relationship. When looking at the object model, a relationship represents the empty table. In the process model, the rows of the table are populated. The rows of the table, then represent relationship instance values.

2.1.13.1 Relational Attributes

type_name:R421

***[2]left_id:R120**

***[2]right_id:R120**

***[2]relationship_number:R638**

***id:R629**

2.1.13.2 Operations

Algorithm 2.8 void Relationship Instance:delete()

```
select one rref related by self->RREF[R421];
unrelate self from rref across R421;
select one irl related by self->IRL[R638];
unrelate self from irl across R638; select one rins related by self->INS[R120.''
select one lins related by self->INS[R120.''is related to''];
if (not empty rins and not empty lins)
        unrelate lins from rins across R120.''is related to'' using self;
end if;
select one val related by self->VAL[R629];
unrelate self from val across R629;
select one dus related by val->DUS[R624];
unrelate val from dus across R624;
delete object instance val;
dus.delete();
delete object instance dus;
```

2.1.14 Right Hand Operand

This is the operand on the right-hand side of the operator.

2.1.14.1 Relational Attributes

***name:R615**

2.1.15 Transient Data

Transient data is data that holds it's value only for the span of execution of a process model.

2.1.15.1 Attributes

name A string identifier for the data store that is unique for the process.

2.1.15.2 Relational Attributes

***id:R626**

2.1.15.3 Operations

Algorithm 2.9 void Transient Data:delete()

```
select one var related by self->VAR[R626];
select one val related by var->VAL[R635];
unrelate var from val across R635;
unrelate self from var across R626;
select one dus related by var->DUS[R624];
unrelate var from dus across R624;
delete object instance var;
dus.delete();
delete object instance dus;
```

2.1.16 Value

Values are typed, read-only data, often held in variables.

2.1.16.1 Relational Attributes

***id:R624**

2.1.17 Variable

Variables are typed, modifiable instances of data used to hold values for further processing.

A variable is a subset of a type set, whether the variable contains one value or many. When a variable is created without value assignment, it is an empty subset, and it can be thought of as uninitialized, containing no value, or null/none in common software language terms.

Variables are the only allowed targets of assignment operations.

2.1.17.1 Relational Attributes

value_id:R635

***id:R624**

2.2 Relationship Descriptions

R611 An expression results in an outcome. The outcome is the result of an expression.

R615 An operand will be considered the left-hand or right-hand operand participating in the expression.

R616 An expression can use a left-hand operand, and a left-hand operand is always used in an expression. The boolean operator, not, is an example of an expression without a left hand operand.

R617 An expression uses a right-hand operand, and a right-hand operand is used in an expression.

R618 An outcome results in an operand. This is always true, because assignment is an operator, and there is no reason to have an expression without an assignment. An operand isn't always the result of an outcome.

Note: in complex equations, there are multiple outcomes and the production of temporary operands. e.g., x = (a + 5) * (a - 2), and a = 3: LHO1 = x; LHO2 = a; LHO3 = a; TYOP1 = '+'; RHO1 = 5; RHO2 = 2; TYOP2 = '+'. LHO1 = x; OUT1 = 8; LHO4 = 8; OUT2 = 1; RHO3 = 1. TYOP3 = '*'; LHO1 = x; OUT3 = 8; RHO4 = 8. TYOP4 = '='. OUT4 = 8; LHO5 = 8.

R622 An equation includes one or more expressions. The expressions compose the equation. Equation evaluation in the metamodel is done using Reverse Polish Notation (RPN). The order of construction involves evaluating the left-most expression, then evaluating the right-most expression, and then the operator. The metamodel uses the ordinal position of the expression to decide evaluation order.

R624 Data usage is either as a variable or a value.

R625 Data usage is constrained by one type, and a type can constrain multiple data usages.

R626 The variable data is transient data used by Process Models, attributes of object instances, or parameters of accessor instances.

R629 The specialization of all values used in Shlaer-Mellor modeling.

R631 An operand represents one data usage, and data usage can be represented by many operands. Even though an equation can be further broken down in a way that's not visible to the analyst, the break down represents atomic pieces of data usage, so while there are many data usages through the break downs in an operand, each individual break down is what is modeled by this relationship. e.g., x = func(z) + (y * z) breaks down to: * value of y times the value of z is assigned to invisible transient variable r. * value of z is assigned to parameter of func(), p. * return value of func(p) is assigned to invisible transient variable q. * value of q plus the value of r is assigned to x. 10 data usages(6 values + 4 variables), 10 operands

R632 An expression uses a typed operator, but a typed operator can be used in many expressions, even in the same equation.

R634 The outcome of an expression is typed by the typed operator's result type.

R635 Values can be assigned to one or more variables at any given time, and a variable always has an assigned value. If no explicit assignment has been made, then the variable holds the default value for the data type. Values exist without variables. Consider the set of numeric values. It would be wrong to suggest the numeral one or the decimal one-half don't exist in the absence of a variable. Variables in action language aren't created or typed without assignment (explicit or implicit), so there's no reason to assume an untyped (no default) variable would exist per this metamodel.

R636 An unspecialized value always belongs to the set defined by a type. A type provides the definition for all of it's values. This relationship requires creation of a value for every type. While this seems onerous, the relationship between value and variable implies that a default value exists for every type, therefore the values required by this relationship compose the set of default values.

R638 A relationship instance instantiates one instance relationship, and an instance relationship can be instantiated by many relationship instances.

R640 A accessor instance instantiates one accessor, and an accessor can be instantiated by many accessor instances.

R642 A non-referential attribute can be instantiated as one or more attribute instances, and an attribute instance is always the instantiation of a non-referential attribute. Referential attributes are always instantiated as the referred to non-referential attributes.

R643 An object instance owns one or more attribute instances, and an attribute instance always belongs to one object instance.

R645 A parameter can be instantiated as one or more parameter instances, and a parameter instance is always an instantiation of one parameter.

R647 A parameter instance belongs to an accessor instance, and an accessor instance can have many parameter instances.

R648 An accessor instance only belongs to an object instance during a method instantiation, and an object instance can have a method instantiation for one or more accessor instances.

Chapter 3

The Domain Subsystem

The subsystem of the metamodel concerning domain objects.

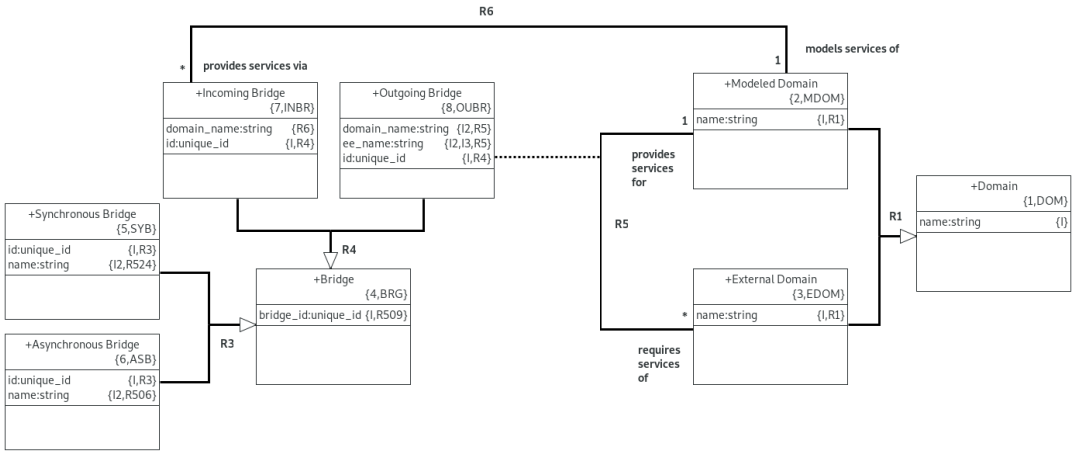

Figure 3.1: Domain Subsystem Diagram

3.1 Object and Attribute Descriptions

3.1.1 Asynchronous Bridge

An asynchronous bridge returns no data and only invokes action in the servicing domain.

Special considerations: A Transfer Vector provides an external domain with the information needed to invoke an event on an instance within the domain being modeled. This means that the transfer vector must include the instance reference and event

reference. It is up to the Architecture to determine how this information is handled, but the external domain should treat the transfer vector information as a composite. i.e., the external domain will not be able to access the instance or event references directly.

"The analyst can think of the transfer vector as a partial event (an event label and an instance identifier only) that will be filled out with supplemental data (if such is defined for the base event) and returned to Home as a complete event at some future time. ...

When Away receives a transfer vector from Home, Away regards the transfer vector as a data element of type 'transfer vector.' Away must save the transfer vector for later use. This is done by attributing the transfer vector to an object that acts as a surrogate for the thread of control in the sending domain. ...

When it is time for Away to provide the asynchronous notification to Home, Away invokes an asynchronous return wormhole, supplying as input data: • the previously saved transfer vector • any additional data elements to be returned to the calling domain (Home). These will be combined with the transfer vector as supplemental data items to form the event expected by Home. The asynchronous return wormhole acts as a way to "return via transfer vector" back to the Home domain."[7]

3.1.1.1 Relational Attributes

***^{2}name:R506**

***id:R3**

3.1.2 Bridge

"During analysis, a bridge between two domains represents a set of assumptions (from the client's perspective) and a set of requirements (from the server's).

- The client thinks of a bridge as a set of capabilities that it assumes will be provided by another domain. The client does not care which domain provides the capabilities.
- The server thinks of the bridge as a set of requirements. The server does not care which domain needs the service, and therefore makes no assumptions about the client."[2]

Bridges can contain both synchronous and asynchronous processes. The following is a list of assumptions, on bridging domains, taken from [2]:

- OOA Mechanisms: "The application and service domains assume that the mechanisms of OOA (data storage, event transmission, and the like) are provided in some form." This means the other domains have a way of persisting shared data, handling events directed at them, synchronizing time, etc. This is the same as client and server computing machines.
- Instant Data Attributes: "The application domain assumes that sensor-based attributes such as Cooking Tank.Actual Temperature have up-to-date values." The action of such a bridge will be considered part of the atomicity in the process model that calls it. Access will work as if it is from a data store.

- Counterparts: "Although an object in one domain doesn't require the existence of an object in another, an instance of object in one domain may have as a counterpart an instance of an object in another domain. For example, a train (in the Railroad Management domain) may have a counterpoint train icon in the User Interface domain."

3.1.2.1 Relational Attributes

***bridge_id:R509**

3.1.3 Domain

"A domain is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of the domain."[2]

3.1.3.1 Attributes

***name:string** The descriptive identifier for the domain.

3.1.4 External Domain

An external domain is a domain that requires or provides services to the domain being modeled. An external domain can be another Shlaer-Mellor modeled domain or a realized domain.

The realized domain can be modeled using some other modeling method, might be existing code, or might be an executable. The bridges from a modeled domain to a realized domain are defined in the model with respect to the modeled domain's requirements, and they can require some extra integration logic, external to the model, to meet the needs of the realized domain.

3.1.4.1 Relational Attributes

***name:R1**

3.1.5 Incoming Bridge

An incoming bridge represents a service provided by the domain being modeled. A synchronous, incoming bridge returns data to the external domain in a timely fashion. An asynchronous, incoming bridge doesn't return data the the external domain.

An incoming bridge might also be referred to as a "domain function".

Although incoming bridges are intended for external access, they can also be called internally by the domain being modeled.

3.1.5.1 Relational Attributes

***id:R4**

domain_name:R6

3.1.6 Modeled Domain

A modeled domain is a domain that has been modeled in accordance with the rules of the Shlaer-Mellor metamodel.

3.1.6.1 Relational Attributes

***name:R1**

3.1.7 Outgoing Bridge

An outgoing bridge is the requirement the domain being modeled places on an external domain for services. A synchronous, outgoing bridge is expected to return data from the external domain. An asynchronous, outgoing bridge is expected to invoke some action in the external domain.

3.1.7.1 Relational Attributes

***id:R4**

***[2]ee_name:R5**

***[2]domain_name:R5**

3.1.8 Synchronous Bridge

A synchronous bridge always returns data to the caller in a timely fashion. Timely should be defined by the domain providing the service.

A synchronous bridge provided by another domain can launch asynchronous actions to obtain the return value, but this domain will be blocked while waiting on the actions to complete. The analyst should take this into consideration. If the external domain definition of timely doesn't match the modeled domain definition of timely, then some mechanism must be put in place to allow the bridge to be treated as an asynchronous bridge.

Special considerations: A synchronous service is used to access a data value from an external domain.

NOTE: It is legal for the domain being modeled to call its own synchronous service as well.

3.1.8.1 Relational Attributes

***[2]name:R524**

***id:R3**

3.2 Relationship Descriptions

R1 A domain is either a modeled domain or a realized domain.

R3 A bridge is expected to synchronously return data, or asynchronously invoke action.

R4 A bridge is either incoming (handled internally) or outgoing (handled externally).

R5 In the context of the domain being modeled, the modeled domain might require services from external domains. An external domain specified in the model will provide services to only the modeled domain.
In reality, the external domain would provide services to other domains, modeled and non-modeled, but in the context of the domain being modeled, it is the only domain requiring services.
Even though a 1:M relationship doesn't require an associative object, making the outgoing bridge an associative object constrains outgoing bridges to be explicitly associated with the client-server relationship between two domains.

R6 An incoming bridge models the services provided by the modeled domain.

Chapter 4

The Dynamics Subsystem

The subsystem of the metamodel concerned with object and relationship dynamics.

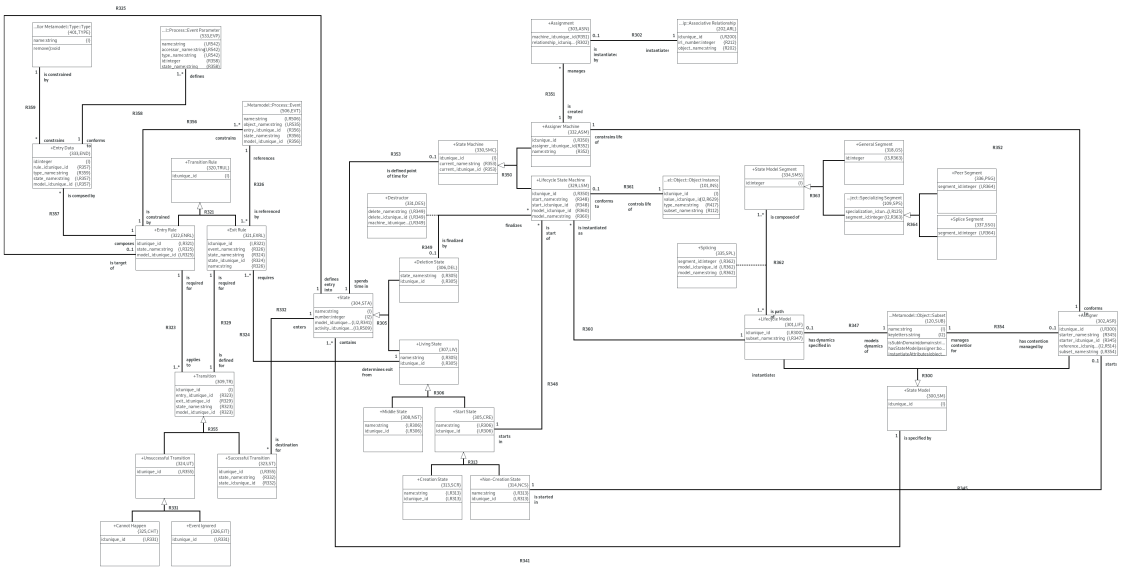

Figure 4.1: Dynamics Subsystem Diagram

4.1 Object and Attribute Descriptions

4.1.1 Assigner

The Assigner is a "state model that is responsible for creating instances of the relationship by associating instances of the participating objects with one another. ... the

purpose of the Assigner is to act as a single point of control through which competing requests are serialized".[2]

An Assigner is object-based, not instance-based. This means that state is maintained for the object, not for each instance of the object.

4.1.1.1 Relational Attributes

***subset_name:R354**

***[2]reference_id:R514**

starter_id:R345

starter_name:R345

***id:R300**

4.1.2 Assigner Machine

An assigner machine is the runtime instantiation of an assigner model. The assigner machine exists for the life of the system, so it is not dynamically created or destroyed.

4.1.2.1 Relational Attributes

name:R352

assigner_id:R352

***id:R350**

4.1.3 Assignment

An assignment is the result of an assigner state model creating a relationship between two object instances. The assignment represents the associative relationship instantiation and the associative object instantiation.

4.1.3.1 Relational Attributes

***relationship_id:R302**

***machine_id:R351**

4.1.4 Cannot Happen

"The "can’t happen" entry is reserved for occasions when the event simply cannot happen in the real world. For example, the event V3: Door opened cannot happen when the oven is in state 5, since in that state the door is already opened."[2]

4.1.4.1 Relational Attributes

***id:R331**

4.1.5 Creation State

A creation state is a start state that might not be depicted on a state model. The Shlaer-Mellor notation showed an arrow coming from nothing and entering a state. In that case, the nothing is the creation state. A creation state will never have a transition entering it.

4.1.5.1 Relational Attributes

***id:R313**

***name:R313**

4.1.6 Deletion State

A deletion state, upon exit, causes the lifecycle state machine and associated object instance to cease to exist.

4.1.6.1 Relational Attributes

***id:R305**

***state_name:R305**

4.1.7 Destructor

The destructor is responsible for removing all segments upon completion of the state machine processing.

4.1.7.1 Relational Attributes

delete_name:R349

delete_id:R349

machine_id:R349

4.1.8 Entry Data

An entry data is a member of the set of data defined for an entry rule. Any event transitioning into the entry rule must carry the exact same amount and type of parameter data as the entry data.

4.1.8.1 Attributes

***id:unique_id** A unique identifier for the entry data.

4.1.8.2 Relational Attributes

***model_id:R357**

***state_name:R357**

type_name:R359

rule_id:R357

4.1.9 Entry Rule

The entry transition rule enforces the "same data rule", which states, "All events that cause a transition into a particular state must carry exactly the same event data."[2]

OL:MWS[2] and OOA96[6] define the data as, identifier data and supplemental data, where the supplemental data is the external data supplied as event parameters, and the identifier data is the target state designator.

This means that an entry rule is composed of the parameter data to be carried by the event causing the transition, and the destination state designator.

4.1.9.1 Relational Attributes

***model_id:R325**

***state_name:R325**

***id:R321**

4.1.10 Event Ignored

"If an object refuses to respond to a particular event when it is in a certain state, enter "event ignored" in the appropriate cell. When an event is ignored, the instance stays in the same state it is in and does not re-execute the action. ... Note that although the event is ignored in the sense of not causing a transition, the event is used up by the state model."[2]

4.1.10.1 Relational Attributes

***id:R331**

4.1.11 Exit Rule

An exit transition rule defines what happens when an event occurs while in the current state. This means that an exit rule is defined for every event defined for the state model. The exit rule requires the event designator and the transition result, which is a destination state or a transition failure designator.

4.1.11.1 Relational Attributes

name:R326

state_id:R324

state_name:R324

event_name:R326

***id:R321**

4.1.12 General Segment

A general segment is included in all instantiations of an object specialization branch. It can be the whole lifecycle model for an object instance.

4.1.12.1 Relational Attributes

***id:R363**

4.1.13 Lifecycle Model

A lifecycle state model abstracts the common behavior of an object that applies to all instances. "Two forms of state models are commonly used in analysis; in OOA, we use the Moore form."[2]

Although there is no direct relationship between instance and lifecycle, the relationship is explicitly directed by the mandatory relationship between lifecycle and object and the mandatory relationship between instance and object. The empty set makes this very ugly to model explicitly, because the relationship between Instance and Lifecycle has to be 1c:Mc and be dependent on the already modeled path from Instance to Object to Lifecycle. The logical associative object would be "Active Instance", which would imply a "Passive Instance" object be created as part of a subtyping of Instance.

A Lifecycle is instance-based, not object-based. This means that state is maintained for each instance of the object, not for the object, but the model of behavior is defined the same for all instances of the object.

4.1.13.1 Relational Attributes

***subset_name:R347**

***id:R300**

4.1.14 Lifecycle State Machine

A lifecycle state machine is the runtime instantiation of a lifecycle state model. The lifecycle state machine can be composed of many state model segments, when the instantiation involves multiple subsets with lifecycle state models, or is composed from one segment when instantiating only one subset with a state model.

4.1.14.1 Relational Attributes

instance_id:R361

model_name:R360

model_id:R360

start_id:R348

start_name:R348

***id:R350**

4.1.14.2 Operations

Algorithm 4.1 void Lifecycle State Machine:delete()

```
select one smc related by self->SMC[R350];
select one sta related by smc->STA[R353];
unrelate smc from sta across R353;
unrelate smc from self across R350;
delete object instance smc;
select one lif related by self->LIF[R360];
unrelate self from lif across R360;
select one cre related by self->CRE[R348];
unrelate self from cre across R348;
select one del related by self->DEL[R349];
if (not empty del)
        select one des related by self->DES[R349];
        unrelate self from del across R349 using des;
        delete object instance des;
end if;
select one ins related by self->INS[R361];
unrelate self from ins across R361;
```

4.1.15 Living State

A living state is any state of the object in an assigner state model, or if in a lifecycle state model, any state that doesn't automatically delete the associated instance at the end of state action processing. All living states should have outgoing and incoming transitions.

4.1.15.1 Relational Attributes

***id:R305**

***name:R305**

4.1.16 Middle State

The middle state is a state that isn't a start or deletion state.

4.1.16.1 Relational Attributes

***id:R306**

***name:R306**

4.1.17 Non-Creation State

A non-creation state is a start state that has been designated as the starting state for the existance of the state model. A non-creation state can contain a process model and be a destination for state-to-state transition. The instance must be created outside of the state model.

4.1.17.1 Relational Attributes

***id:R313**

***name:R313**

4.1.18 Peer Segment

A peer segment is a specializing segment that specializes two lifecycles. It is instantiated when two overlapping sets share a common behavior for a portion of their lifecycle.

4.1.18.1 Relational Attributes

***segment_id:R364**

4.1.19 Splice Segment

A splice segment is a specializing segment that belongs to a single lifecycle.

4.1.19.1 Relational Attributes

***segment_id:R364**

4.1.20 Splicing

A splicing represents the association between a state model segment and the lifecycle model. When you have a single splicing instance, you don't really have a splicing as defined in OL:MWS[2], but the concept that a set is a subset allows this to be modeled as all lifecycle models are splicings.

4.1.20.1 Relational Attributes

***model_name:R362**

***model_id:R362**

***segment_id:R362**

4.1.21 Start State

A start state is any living state that has been designated to be the state of existence upon object realization or instance creation in the system.

4.1.21.1 Relational Attributes

***id:R306**

***name:R306**

4.1.22 State

"A state represents a condition of the object in which a defined set of rules, policies, regulations, and physical laws applies."[2]

4.1.22.1 Attributes

***[2]number:integer** A number for the state, that can uniquely identify it within its state model.

***name:string** A symbolic value describing the state, that can uniquely identify it within its state model.

4.1.22.2 Relational Attributes

***[3]activity_id:R509**

****[2]model_id:R341**

4.1.23 State Machine

A state machine is the runtime instantiation of a state model.

4.1.23.1 Attributes

***id:unique_id** A unique identifier for the state machine.

4.1.23.2 Relational Attributes

current_id:R353

current_name:R353

4.1.24 State Model

A state model formalizes the dynamic behavior of an object or of subsets of an object. The state models are either instance based, called lifecycle models, or assigners, which exist regardless of instantiation.

For lifecycles, Shlaer-Mellor allows the concept of a split state model. In a split state model, a segment of the state model is modeled in an object specialization. The whole lifecycle depends upon which specializations compose the instantiation. This allows object specialization on behavior.

4.1.24.1 Attributes

***id:unique_id** A unique identifier for the state model.

4.1.25 State Model Segment

A state model segment is any portion of the state model that exists only in one subset of the object. Since the subset can be a set, a state model segment in the general case is the whole state model. In the case where the subset doesn't represent the whole set, the state model segment is a behavior specialization of the object.

4.1.25.1 Attributes

***id:unique_id** A unique identifier for the state model segment.

4.1.26 Successful Transition

In a successful state-to-state transition, the result of the event was a state entry.

4.1.26.1 Relational Attributes

state_id:R332

state_name:R332

***id:R355**

4.1.27 Transition

A transition represents a change in state within the same object state model.

4.1.27.1 Attributes

***id:unique_id** A unique identifier for the transition.

4.1.27.2 Relational Attributes

model_id:R323

state_name:R323

exit_id:R329

entry_id:R323

4.1.28 Transition Rule

A transition rule defines the signature required for entry into or exit from a state.

4.1.28.1 Attributes

***id:unique_id** A unique identifier for the transition rule.

4.1.29 Unsuccessful Transition

An unsuccessful transition does not change the state of an instance, and can be used to trigger error handling mechanisms as directed by the architecture.

4.1.29.1 Relational Attributes

***id:R355**

4.2 Relationship Descriptions

R300 There are two types of state models in Shlaer-Mellor, Lifecycle of an instance and Assigner for relationships involving competition.

R302 An assignment instantiates an associative relationship and an instance of the associative object. As not all associative relationships involve contention, not every one is the result of an assignment.

R305 A state in the state model is either a living state, or in the case of a lifecycle model is a deletion state to remove the object instance upon completion of the state action.

R306 A living state can be designated as a creation state. A state model will only have one creation state.
The creation aspect could be made an attribute of living state, but that would imply that the creation aspect wasn't permanent. There should be no reason to dynamically change which state is designated for creation.

R313 A start state is either a creation state, or a non-creation, entry state.

R321 A transition rule defines either the rules for entry into or exit from a state.

R323 Every transition requires that an entry rule is defined for the destination state. An entry rule applies to all transitions into the state.

R324 A living state requires one or more exit rules to specify the conditions required to exit the state.

R325 An entry rule specifies the entry conditions required to be met before entry into the target state.
Non-creation, start states might only have exit transitions, so the requirement to have an entry rule is conditional.

R326 The exit rule must contain a reference to the event that causes exit from the current state. An event is always referenced by an exit rule.

R329 Every transition requires an exit rule, so an exit rule is defined for each transition.

R331 An unsuccessful transition is constrained by the rules of Shlaer-Mellor to be an "event ignored" or "can't happen" result.

R332 A successful transition always enters a state. A state can be specified as the destination for more than one transition. Only a start state might not be a destination for a successful transition.

R341 A state model contains states. The states are specified by the state model.

R345 An assigner is started in the designated starting state on application start.

R347 A subset, which can be a whole set, can have its dynamics modeled in a lifecycle state model. A lifecycle state model will always model the dynamics of one subset.

R348 All state machines start in a start state designated for the state model. There is only one start state for a state machine, but it is assigned for all the state machine instances.

R349 A deletion state finalizes a state machine. The deletion state process will destroy all the associated state machine instances.

R350 A state machine is either a lifecycle machine or an assigner machine.

R351 Assignments are created and managed by the assigner state machine.

R352 The assigner state model constrains the operation of the assigner state machine. The assigner state machine must conform to the assigner state model.
Unlike for lifecycle models and machines, the relationship is unconditional on both ends, because the assigner machine isn't tied to an instantiation of an object.

R353 A state machine is currently spending time in a state, and a state is a defined point of time for a state machine.

R354 An assigner manages contention for a subset, and a subset has contention managed by an assigner.

R355 A transition is a successful transition or an unsuccessful transition. Modeling both of these allows the state transition table to be fully specified.

R356 An event is always constrained by one entry rule, and an entry rule constrains all the events that are directed to the same state.

R357 An entry rule is composed of its entry data, even in the no entry data case. Entry data is used to compose a single entry rule.

R358 A single entry data is associated with one or more event parameters, but an event parameter conforms to only one entry data.

R359 Entry data is constrained by a type, and a type can constrain entry data.

R360 A lifecycle model is instantiated as a lifecycle state machine, and a lifecycle state machine instantiates a lifecycle model.

R361 A lifecycle state machine controls the life of one object instance, and an object instance can be controlled by a lifecycle state machine.
Every instance has it's own state machine.

R362 A lifecycle model is composed of one or more state model segments, and a state model is a path in one or more lifecycles. The multiplicity of the latter is driven by the possibility of a set intersection.

R363 A state model segment is a general segment in the case where it represents the whole state model, and a state model segment is a specializing segment in the case where it represents only a portion of the state model.

R364 A specializing segment can belong to more than one lifecycle when there are intersecting sets with common behavior; this type of segment is a peer segment. When the specializing segment belongs to only one lifecycle, it is a splice segment.

Chapter 5

The Object Subsystem

The subsystem of the metamodel concerning objects, instances, and attributes.

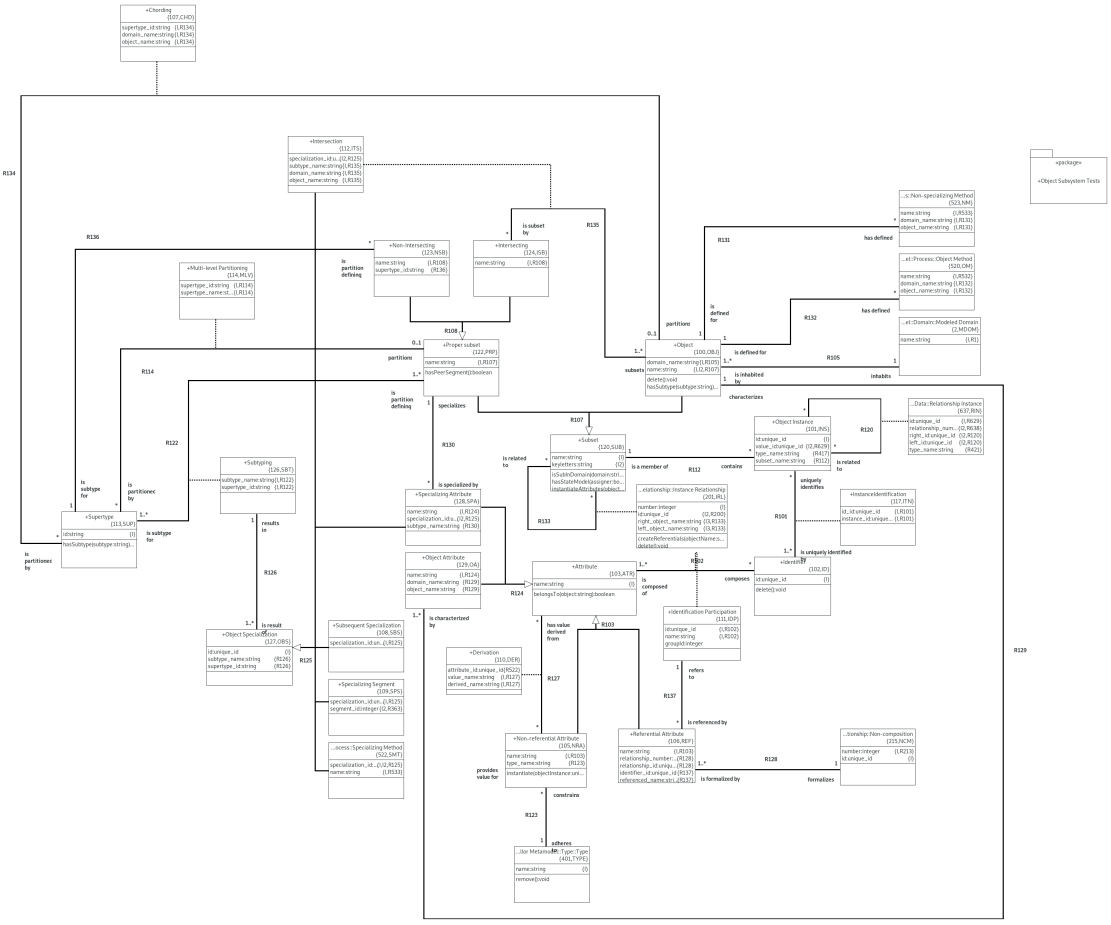

Figure 5.1: Object Subsystem Diagram

5.1 Object and Attribute Descriptions

5.1.1 Attribute

An attribute is used to characterize aspects of the objects within a domain.

5.1.1.1 Attributes

***name:string** A descriptive string concerning the characterization.

5.1.1.2 Operations

Algorithm 5.1 boolean Attribute:belongsTo()

```
belongsTo = true;
select one oa related by self->OA[R124] where selected.object_name == param.obje
if (empty oa)
        select one spa related by self->SPA[R124] where selected.subtype_name ==
        if (empty spa)
                belongsTo = false;
        end if;
end if;
return belongsTo;
```

5.1.2 Chording

A chording represents all the chords belonging to a supertype partitioning.

Proper subsets are never partitioned by a chord, so the supertype partitioning of a subset requires a separate relationship in the metamodel.

5.1.2.1 Relational Attributes

***object_name:same_as<Base_Attribute>**

***domain_name:same_as<Base_Attribute>**

***supertype_id:same_as<Base_Attribute>**

5.1.3 Derivation

A derivation is a dependency mapping between two attributes. The attributes can be from the same object or different objects.

5.1.3.1 Relational Attributes

***derived_name:same_as<Base_Attribute>**

***value_name:same_as<Base_Attribute>**

attribute_id:same_as<Base_Attribute>

5.1.4 Identification Participation

Formalizes all attribute participation in identification of an object's instances.

5.1.4.1 Attributes

groupId:integer A number used to identify a group of identifiers. Each group of identifiers are used to uniquely identify an object instance.
The preferred identifier will have a groupId value of 1. The preferred identifier is the only identifier used to formalize an instance relationship as referential attributes.
"An object may have several identifiers, each composed of one or more attributes. For example, an Airport object may have the attributes
Airport Code
Latitude
Longitude
City
Number of Passenger Gates
The Airport Code attribute is an identifier of the Airport object, and the combination of Latitude and Longitude is another identifier of Airport.
If an object has multiple identifiers, one such identifier is chosen as the preferred identifier."[2]
In tools, like BridgePoint, an indication is placed next to identifying attributes, and grouping of identifiers is done numerically. e.g., {I}, {I2}, {I3}, etc. This metamodel is just using a numeric indicator. The numeral, 1, is the preferred identifier, and like BridgePoint, the editor part of the toolset might choose to only display numerals greater than one.

5.1.4.2 Relational Attributes

***name:same_as<Base_Attribute>** The name of the attribute used for identification.

***id:same_as<Base_Attribute>** The unique identifier for the Identifier instance.

5.1.5 Identifier

An identifier is used to uniquely identify the members of a set. "An identifier is a set of one or more attributes whose values uniquely distinguish each instance of an object."[2]

The set always has one identifying attribute that applies to all set members, but some identifying might only apply to some subset members. e.g., a subset formed from the intersection of two sets will have at least two identifying attributes, one from each set.

5.1.5.1 Attributes

***id:unique_id** A unique identifier for the instance identifier in the domain.

5.1.5.2 Operations

Algorithm 5.2 void Identifier:delete()

```
select many idps related by self->IDP[R102];
for each idp in idps
        select one atr related by idp->ATR[R102];
        unrelate self from atr across R102 using idp;
        delete object instance idp;
end for;
select many itns related by self->ITN[R101];
for each itn in itns
        select one ins related by itn->INS[R101];
        unrelate self from ins across R101 using itn;
        delete object instance itn;
        ins.delete();
        delete object instance ins;
end for;
```

5.1.6 Instance Identification

An instantiation is the creation of an identifier for an instance that is a member of a subset.

5.1.6.1 Relational Attributes

***instance_id:same_as<Base_Attribute>**

***id_id:same_as<Base_Attribute>**

5.1.7 Intersecting

An intersecting proper subset is formed from the intersection of two or more sets.

5.1.7.1 Relational Attributes

***name:same_as<Base_Attribute>**

5.1.8 Intersection

The intersection associative object is used to track the participants in the intersecting proper subset formation brought about by multiple objects.

5.1.8.1 Relational Attributes

***object_name:same_as<Base_Attribute>**

***domain_name:same_as<Base_Attribute>**

***subtype_name:same_as<Base_Attribute>**

***[2]specialization_id:same_as<Base_Attribute>**

5.1.9 Multi-level Partitioning

Multi-level partitioning abstracts the relationship formed when a proper subset is further partitioned (subset) by a supertype partitioning.

A multi-level partitioning is instantiated whenever a subtype is subtyped in the object model.

5.1.9.1 Relational Attributes

***supertype_name:same_as<Base_Attribute>**

***supertype_id:same_as<Base_Attribute>**

5.1.10 Non-Intersecting

A non-intersecting proper subset is a subset of only one set.

5.1.10.1 Relational Attributes

supertype_id:same_as<Base_Attribute>

***name:same_as<Base_Attribute>**

5.1.11 Non-referential Attribute

A non-referential attribute "is an abstraction of a single characteristic possessed by all entities that were themselves abstracted as an object."[2]

It should be noted that at least one of the non-referential attributes related to an object will participate in identification of the instances. While precedence has been set in some tools to not abstract arbitrary identifiers, a primary key must exist, and it would be specious to say we can assume it's existence.

5.1.11.1 Relational Attributes

type_name:same_as<Base_Attribute>

***name:same_as<Base_Attribute>**

5.1.11.2 Operations

Algorithm 5.3 void Non-referential Attribute:instantiate()

```
select any ins from instances of INS where selected.id == param.objectInstance;
create object instance atn of ATN;
relate atn to self across R642;
relate atn to ins across R643;
create object instance var of VAR;
relate var to atn across R626;
create object instance dus of DUS;
relate dus to var across R624;
select one type related by self->TYPE[R123];
relate type to dus across R625;
```

5.1.12 Object

"An object is an abstraction of a set of real-world things such that:

- all the things in the set, the instances, have the same characteristics, and
- all instances are subject to and conform to the same set of rules and policies."[2]

5.1.12.1 Relational Attributes

name:same_as<Base_Attribute>

domain_name:same_as<Base_Attribute>

5.1.12.2 Operations

Algorithm 5.4 bool Object:hasSubtype()

Determines if the named subtype is this object or part of this object's subtyping hierarchy.

```
rc = false;
if (self.name == param.subtype)
        rc = true;
else
        select many sups related by self->SUP[R134];
        for each sup in sups
                rc = sup.hasSubtype( subtype:param.subtype );
                if (rc)
                        break;
                end if;
        end for;
end if;
return rc;
```

5.1.13 Object Attribute

An object attribute applies to all instances of the object across any subset boundaries. It is a non-specializer.

5.1.13.1 Relational Attributes

****[2]object_name:same_as<Base_Attribute>**

***domain_name:same_as<Base_Attribute>**

****[2]name:same_as<Base_Attribute>**

5.1.14 Object Instance

A set member. e.g., an instantiation of an object as a specified value.

NOTE: The term, "specified value", indicates that all the attributes of the subset have been assigned values.

5.1.14.1 Attributes

***id:unique_id** A unique identifier for the instance within the domain.

5.1.14.2 Relational Attributes

subset_name:same_as<Base_Attribute>

type_name:same_as<Base_Attribute>

***[2]value_id:same_as<Base_Attribute>**

5.1.14.3 Operations

Algorithm 5.5 void Object Instance:delete()

Satisfies deleting all relationships to this instance and all related instance data, such as attribute instances, method instances, and state machines. Doesn't delete this instance. That must be handled by the caller. The architecture domain should call this at the end of processing a deletion state.

```
select one sub related by self->SUB[R112];
// Cleanup relationships
select many rins related by self->RIN[R120.''is related to''];
for each rin in rins
        select one ins related by rin->INS[R120.''is related to''];
        unrelate self from ins across R120.''is related to'' using rin;
        rin.delete();
        delete object instance rin;
end for;
// Cleanup attributes
select many atns related by self->ATN[R643];
for each atn in atns
        unrelate self from atn across R643;
        atn.delete();
        delete object instance atn;
end for;
// Cleanup methods
select many ains related by self->AIN[R648];
for each ain in ains
        ain.delete();
        delete object instance ain;
end for;
// Cleanup state machines
select one lsm related by self->LSM[R361];
if (not empty lsm)
        lsm.delete();
        delete object instance lsm;
end if;
// Cleanup self unrelate self from sub across R112;
select many itns related by self->ITN[R101];
for each itn in itns
        select one id related by itn->ID[R101];
        unrelate self from id across R101 using itn;
        delete object instance itn;
end for;
```

5.1.15 Object Specialization

An object specialization is any means of subsetting a set defined by an object into specialized instances. The specialization can occur via data or behavior.

5.1.15.1 Attributes

***id:unique_id** A unique identifier for the specialization.

5.1.15.2 Relational Attributes

supertype_id:same_as<Base_Attribute>

subtype_name:same_as<Base_Attribute>

5.1.16 Proper subset

A proper subset is not equal to the containing set. If the boundaries of the set aren't considered to be a partition, then a proper subset exists whenever a set is partitioned.

It should be noted that a proper subset can have all the attributes and behavior of the containing set, but still be a proper subset if there exists another proper subset of the same set with different attributes and/or behavior.

5.1.16.1 Relational Attributes

***name:same_as<Base_Attribute>**

5.1.16.2 Operations

Algorithm 5.6 boolean Proper subset:hasPeerSegment()

Determines if this subset's state model segment is a peer state model segment. This routine checks for a supertype partitioning across R122, then gets the supertype object(s) across R114 and determines if there is an associated state model segment. If there is no associated state model segment, then this method calls itself to recurse through the supertype object's hierarchy.

```
// Determine if this is a peer segment. i.e., not a splice
peer = true;
// Get supertyping
select one sub related by self->SUB[R107];
select any sup related by sub->PRP[R107]->SUP[R122];
if (not empty sup)
        select many sups related by sub->PRP[R107]->SUP[R122];
        for each sup in sups
                // Get subertype object
                select one prp related by sup->PRP[R114];
                if (not empty prp)
                        // Does this subtype have a state model segment?
                        select any sps related by prp->SBT[R122]->OBS[R126]->SPS
                        if (not empty sps)
                                peer = false;
                        else
                                peer = prp.hasPeerSegment();
                        end if;
                        if (not peer)
                                break;
                        end if;
                end if;
        end for;
end if;
return peer;
```

5.1.17 Referential Attribute

"Referential attributes are used to tie an instance of one object to the instance of another."[2]

5.1.17.1 Relational Attributes

referenced_name:same_as<Base_Attribute>

identifier_id:same_as<Base_Attribute>

relationship_id:same_as<Base_Attribute>

relationship_number:same_as<Base_Attribute>

***name:same_as<Base_Attribute>**

5.1.18 Specializing Attribute

A specializing attribute is one that applies only to some instances of an object.

5.1.18.1 Relational Attributes

subtype_name:same_as<Base_Attribute>

***[2]specialization_id:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

5.1.19 Specializing Segment

A specializing segment applies to only a subset of instances of the object.

5.1.19.1 Relational Attributes

***[2]segment_id:same_as<Base_Attribute>**

***specialization_id:same_as<Base_Attribute>**

5.1.20 Subsequent Specialization

A subsequent specialization occurs due to other specializations. It shows that the subset exists without being linked to a specific metamodel element.

The subsequent specialization can be due to specialization of the subtype leaf with no other differentiator at its own level.

5.1.20.1 Relational Attributes

***specialization_id:same_as<Base_Attribute>**

5.1.21 Subset

A proper subset represents a partitioning of a set or, in the case of an intersection, multiple sets. An improper subset represents the entire set, so an object can also be classified as subset.

In OOA terms, a subtype always implies a subset, so set theory terms are used to provide better paths to common relationships in the metamodel.

5.1.21.1 Attributes

***[2]keyletters:string** Keyletters allow one to be lazy about referring to a subset.

***name:string** A name of the subset, that is unique in the domain. For an improper subset, this will be the name of the object, but for a proper subset, it will be a name, that is a sub-classification of, the object name.

5.1.21.2 Operations

Algorithm 5.7 boolean Subset:isSubInDomain()

```
rc =false;
select one obj related by self->OBJ[R107];
if (not empty obj)
        if (obj.domain_name == param.domain)
                rc = true;
        end if;
else
        select one prp related by self->PRP[R107];
        // Check for root supertype.
        select one obj related by prp->NSB[R108]->SUP[R136]->OBJ[R134];
        while (empty obj)
                select many itss related by prp->ISB[R108]->ITS[R135];
                for each its in itss
                        if (its.domain_name == param.domain)
                                rc = true;
                                break;
                        end if;
                end for;
                if (not rc)
                        // Move up to next subtyping
                        select one prp related by prp->NSB[R108]->SUP[R136]->PRP
                        // Check for root supertype.
                        select one obj related by prp->NSB[R108]->SUP[R136]->OBJ
                end if;
        end while;
        if (not rc and (obj.domain_name == param.domain))
                rc = true;
        end if;
end if;
return rc;
```

Algorithm 5.8 bul Subset:hasStateModel()

Returns true if the type of state model specified already exists for the subset.

```
rc = true;
if (param.assigner)
        select one asr related by self->ASR[R354];
        if (not empty asr)
                rc = false;
        end if;
else
        select one lif related by self->LIF[R347];
        if (not empty lif)
                rc = false;
        end if;
end if;
return rc;
```

Algorithm 5.9 void Subset:instantiateAttributes()

Instantiate attributes associated directly with this subset, and then get the next level subsets and tell them to instantiate their attributes.

```
select one obj related by self->OBJ[R107];
if (not empty obj)
        select many oas related by obj->OA[R129];
        for each oa in oas
                select one nra related by oa->ATR[R124]->NRA[R103];
                select one type related by nra->TYPE[R123];
                nra.instantiate( objectInstance:param.objectInstance );
        end for;
else
        select one prp related by self->PRP[R107];
        select many spas related by prp->SPA[R130];
        for each spa in spas
                select one nra related by spa->ATR[R124]->NRA[R103];
                select one type related by nra->TYPE[R123];
                nra.instantiate( objectInstance:param.objectInstance );
        end for;
        select many sups related by prp->SUP[R122];
        for each sup in sups
                select one sub related by sup->OBJ[R134]->SUB[R107];
                if (empty sub)
                        select one sub related by sup->PRP[R114]->SUB[R107];
                end if;
                sub.instantiateAttributes( objectInstance:param.objectInstance )
        end for;
end if;
```

Algorithm 5.10 void Subset:associateReferentials()

Finds identifiers in this subtype and creates referential identifiers, then finds any higher-level sub/supertypes and repeats the operation.

```
select any irl from instances of IRL where selected.number == param.relationship
// First collect any identifiers for this subtype
irl.createReferentials( objectName:self.name );
// Then search up the hierarchy until the root supertype is found.
select one obj related by self->OBJ[R107];
if (empty obj)
        select one prp related by self->PRP[R107];
        select many sups related by prp->SUP[R122];
        for each sup in sups
                select one sub related by sup->OBJ[R134]->SUB[R107];
                if (empty sub)
                        select one sub related by sup->PRP[R114]->SUB[R107];
                end if;
                sub.associateReferentials( relationship:param.relationship );
        end for;
end if;
```

Algorithm 5.11 boolean Subset:isSubsetOf()

Checks to see if this subset is a member of the named set (object).

```
rc = false;
select one obj related by self->OBJ[R107] where selected.name == param.object;
if (not empty obj)
        rc = true;
else
        select one nsb related by self->PRP[R107]->NSB[R108];
        if (not empty nsb)
                select one obj related by nsb->SUP[R136]->OBJ[R134] where selecte
                if (not empty obj)
                        rc = true;
                else
                        select one sub related by nsb->SUP[R136]->PRP[R114]->SUB
                        rc = sub.isSubsetOf( object:param.object );
                end if;
        else
                select one isb related by self->PRP[R107]->ISB[R108];
                select any obj related by isb->OBJ[R135] where selected.name ==
                if (not empty obj)
                        rc = true;
                end if;
        end if;
end if;
return rc;
```

Algorithm 5.12 void Subset:instantiateMethods()

```
select one obj related by self->OBJ[R107];
if (not empty obj)
        select many nms related by obj->NM[R131];
        for each nm in nms
                select one im related by nm->IM[R533];
                im.instantiate( objectInstance:param.objectInstance );
        end for;
else
        select one prp related by self->PRP[R107];
        select many smts related by prp->SBT[R122]->OBS[R126]->SMT[R125];
        for each smt in smts
                select one im related by smt->IM[R533];
                im.instantiate( objectInstance:param.objectInstance );
        end for;
        select many sups related by prp->SUP[R122];
        for each sup in sups
                select one sub related by sup->OBJ[R134]->SUB[R107];
                if (empty sub)
                        select one sub related by sup->PRP[R114]->SUB[R107];
                end if;
                sub.instantiateMethods( objectInstance:param.objectInstance );
        end for;
end if;
```

5.1.22 Subtyping

Subtyping is the defining of subsets via a supertype partitioning. The relationship is abstracted to capture the many to many condition that arises from an intersection of two objects.

On the object model, the subtyping is the leg from the supertype part of the relationship to the subtype object.

5.1.22.1 Relational Attributes

***supertype_id:same_as<Base_Attribute>**

***subtype_name:same_as<Base_Attribute>**

5.1.23 Supertype

A supertype represents a partitioning of the set associated with a specialized object. The supertype isn't a subset of the object, but can collect a subset of attributes that are shared by the subsets formed by the supertype's partitioning.

The representation of the metamodel supertype on the object model is the part of the supertype/subtype relationship attached to the supertype object, not the supertype object itself.

The identifier for the supertype is the name of the graphical relationship, that denotes the subtyping. i.e., the relationship with the bar across it in the Shlaer-Mellor notation, or the relationship with the triangle on the end in the UML notation.

5.1.23.1 Attributes

***id:string**

5.1.23.2 Operations

Algorithm 5.13 boolean Supertype:hasSubtype()

```
rc = false;
select any prp related by self->PRP[R122] where selected.name == param.subtype;
if (not empty prp)
        rc = true;
else
        select many sups related by self->PRP[R122]->SUP[R114];
        for each sup in sups
                rc = sup.hasSubtype( subtype:param.subtype );
                if (rc)
                        break;
                end if;
        end for;
end if;
return rc;
```

5.2 Relationship Descriptions

R101 An identifier uniquely identifies instances when an object is instantiated. The instance might have multiple identifiers, each of which provides a separate path to identification of the instance.

R102 An identifier is composed of one or more attributes. Since the identifier applies to only one instance, the attributes used can compose many identifiers, so each identification is abstracted as an object.

R103 Attributes can be non-referential, which means they are used to give value or identification to an instance of an object, or referential, which means they are used to identify the instances related to the instance of an object.

R105 An object inhabits only one modeled domain. The modeled domain can contain many objects, but always has at least one.

R107 Subsets are classified as proper or improper subsets. Improper subsets are an entire set, which is known in Shlaer-Mellor terms as an object. This partitioning allows an association between object instance and subset, which allows the object instance object to represent an instance member of an unspecialized object or an instance member of a subset of a specialized object.

R108 Proper subsets are formed by partitioning of a single set or partitioning by intersection of more than one set.
Making this distinction is important to establish less ambiguous relationships in the model with regard to multiplicities between objects, subsets, attributes and instances.

R112 An instance is a member of one subset. The subset can contain many instances, or no instances in the case where an object hasn't been instantiated for that subset. This supports the concept of the empty set.

R114 A proper subset can be partitioned by supertype partitioning. As the subset is already formed by a partition, this is referred to as multi-level partitioning. In the case where the subset is partitioned by more than one supertype, it is called multi-way partitioning.

R120 Instances are related to other instances when a relationship is instantiated.

R122 Subsets are formed by supertype partitioning, and a subset is a subtype for one or more supertypes. The partitioning relationship is thus abstracted as a subtyping. Non-intersecting subtypes are further constrained by R136 to belong to only one supertype.
"OOA does not permit creating an instance of the supertype without creating an instance of one subtype, and vice versa."[2]

R123 Every non-referential attribute adheres to a known type. The type imposes type constraints on the attribute.
NOTE: Referential attributes will also have a known type, but those are set where they are non-referential attributes.

R124 An attribute can provide unique data about a set, or can provide unique data about a subset of a set. The latter case is considered a specialization.

R125 An object specialization is a specializing attribute, an intersection of objects or a specializing lifecycle segment.

R126 An object specialization results in one subtyping of an object, and the subtyping can be the result of multiple defined object specialization types.
While a single specialization actually results in two subtypings, additional subtypings in the same generalization will not result in two subtypings. e.g., An object, A, exists, and it is decided an attribute only applies to some instances of

A and not others. A subtype, B, is created to hold the attribute, and a subtype, C, is created to represent the instances without the attribute. Another attribute is defined to only apply to some instances of A, so subtype, D, is created. D requires no subsequent specialization, because the remaining instances are already defined by B and C.

R127 A non-referential attribute value can depend upon the value of other attributes. An attribute can provide value for many dependent attributes.[5] has a discussion on the types of dependencies possible.

R129 An object attribute characterizes an object. The object and all subsets of an object are characterized by object attributes, as all of the object attributes apply to all instances of the object.
As there is always an identifier attribute, an object is always characterized by at least one object attribute.

R130 A specializing attribute always specializes one proper subset. A proper subset can be specialized by many specializing attributes.

R128 A referential attribute always formalizes a non-composition, instance relationship. All non-composition, instance relationships are formalized by one or more referential attributes.

R131 An object can have defined non-specializing methods, and a non-specializing method is defined for one object.

R132 An object can have defined object methods, and an object method is defined for one object.

R133 Subset can be related to other subsets.

R134 A supertype can partition an object, or it can a partition a proper subset. An object is partitioned by one or more supertypes.
A partitioning of an object by a single supertype should be pictured as one or more parallel chords, each crossing the arc of the set at two distinct points.
When the partitioning of the same object is done by more than one supertype, the chords of the supertypes intersect each other.

R135 An intersecting subset is formed from the intersection of two or more objects, and an object can be partitioned by more than one intersecting subset. The intersection associative object tracks each 'subset partitions object' association. e.g., in the simplest example, a subset C is formed from the intersection of object A and object B; two intersections are needed to represent this: CA and CB. This case has one intersecting subset, two objects, and two intersections.

R136 A non-intersecting subset is a subtype for one supertype, and the supertype defines the partitioning for all of its subtypes.
This relationship adds further constraint, for non-intersecting subsets, to R122, which defines all of the subtyping relationships.

R137 A referential identifier refers to an attribute participating in identification of an object instance. An identification participant can be referenced by a referential attribute.

Chapter 6

The Process Subsystem

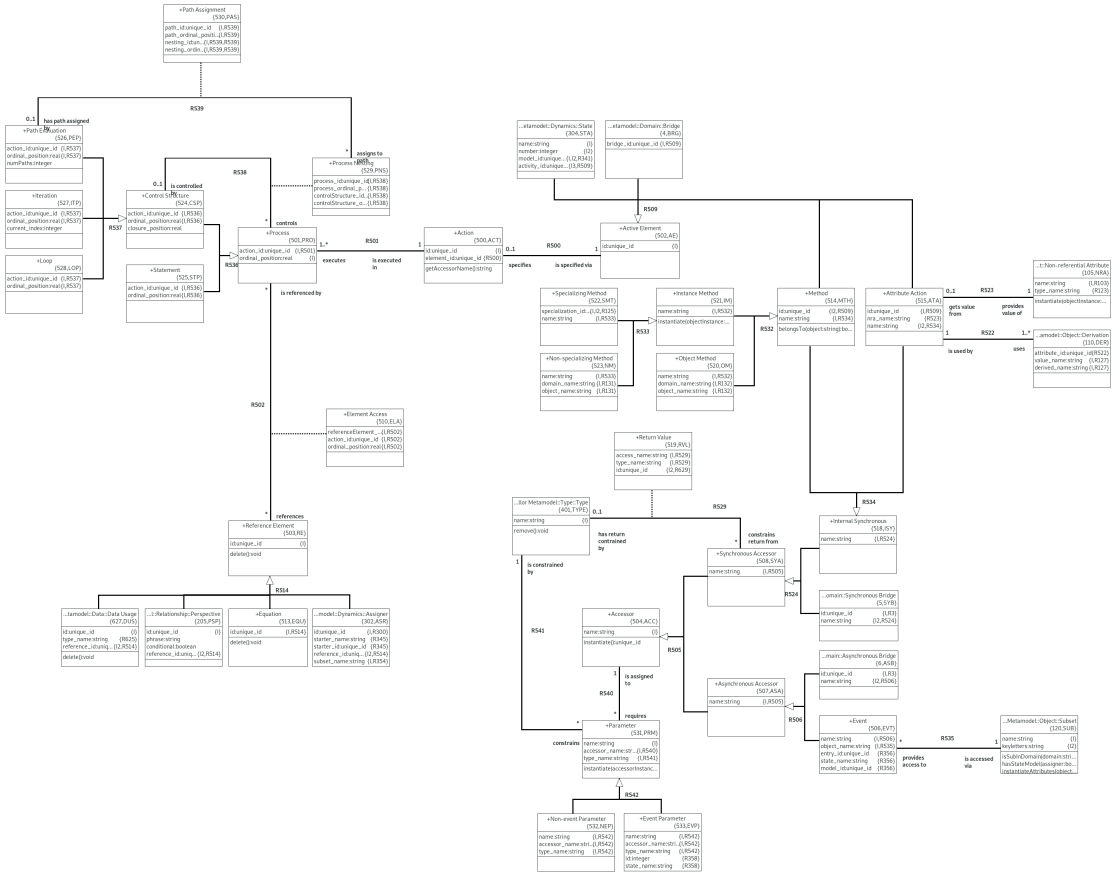

Figure 6.1: Process Subsystem Diagram

6.1 Object and Attribute Descriptions

6.1.1 Accessor

An accessor provides access to reference elements and defines the signature of the access. The accessor represents bridges, events, and operations. In other words, things that would be known as functions, procedures, or methods in third generation programming languages like C or Java.

An accessor is a value once instantiated in the metamodel, that can be referenced in process models.

6.1.1.1 Attributes

***name:string**

6.1.1.2 Operations

Algorithm 6.1 unique_id Accessor:instantiate()

Create an instance of this accessor, and it's parameters.

```
create object instance ain of AIN;
select any aref from instances of AREF;
relate ain to aref across R422;
relate ain to self across R640;
create object instance val of VAL;
relate ain to val across R629;
create object instance dus of DUS;
relate val to dus across R624;
select one type related by aref->CORE[R403]->TYPE[R401];
relate type to dus across R625;
create object instance re of RE;
relate dus to re across R514;
select many prms related by self->PRM[R540];
for each prm in prms
        prm.instantiate( accessorInstance:ain.id );
end for;
return ain.id;
```

6.1.2 Action

"All the processing that goes on in the system is stated in the actions."[2]

An Action is a single, atomic, unit of processing that can read data, write data (implied modify), make conditional choices, and launch the execution of other actions.

6.1.2.1 Attributes

***id:unique_id**

6.1.2.2 Relational Attributes

element_id:same_as<Base_Attribute>

6.1.2.3 Operations

Algorithm 6.2 string Action:getAccessorName()

Returns the name of the accessor or state that contains this action. In the case of states, different event (accessor) names can be used to cause entry into the state, so the state name must be used.

```
name = "";
select one acc related by self->AE[R500]->BRG[R509]->ASB[R3]->ASA[R506]->ACC[R50
if (empty acc)
        select one acc related by self->AE[R500]->BRG[R509]->SYB[R3]->SYA[R524]-
        if (empty acc)
                select one acc related by self->AE[R500]->ATA[R509]->ISY[R534]->
                if (empty acc)
                        select one acc related by self->AE[R500]->MTH[R509]->ISY
                        if (empty acc)
                                select one sta related by self->AE[R500]->STA[R5
                                name = sta.name;
                        end if;
                end if;
        end if;
end if;
if (not empty acc)
        name = acc.name;
end if;
return name;
```

6.1.3 Active Element

An active element is any OOA construct that contains a process model. The active element must be accessible from other elements and can access other elements. The other elements can be internal or external to the domain being modeled.

6.1.3.1 Attributes

***id:unique_id**

6.1.4 Asynchronous Accessor

An asynchronous accessor executes some time after the action in which it was generated completes. (see [2] page 131)

6.1.4.1 Relational Attributes

***name:same_as<Base_Attribute>**

6.1.5 Attribute Action

An attribute action is used to calculate the value of a derived attribute.

6.1.5.1 Relational Attributes

***[2]name:same_as<Base_Attribute>**

nra_name:same_as<Base_Attribute>

***id:same_as<Base_Attribute>**

6.1.6 Control Structure

A control structure process is used to encapsulate other process execution. Typical control structures are if-then-else and while loops.

6.1.6.1 Attributes

closure_position:real The ordinal position of the last process in the control structure. A closure position of zero means the control structure is unclosed.

6.1.6.2 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

6.1.7 Element Access

Element Access represents usage of a modeled element by a Process.

6.1.7.1 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

***referenceElement_id:same_as<Base_Attribute>**

6.1.8 Equation

An equation is statement of equality using other values or variables. In OOA, an equation results in assignment to a variable.

6.1.8.1 Relational Attributes

***id:same_as<Base_Attribute>**

6.1.8.2 Operations

Algorithm 6.3 void Equation:delete()

```
select one re related by self->RE[R514];
unrelate self from re across R514;
re.delete();
delete object instance re;
select many exps related by self->EXP[R622];
for each exp in exps
        select one tyop related by exp->TYOP[R632];
        unrelate exp from tyop across R632;
        select one out related by exp->OUT[R611];
        unrelate out from exp across R611;
        select one tyop related by out->TYOP[R634];
        unrelate out from tyop across R634;
        select one oprd related by out->OPRD[R618];
        unrelate out from oprd across R618;
        delete object instance out;
        select one rho related by exp->RHO[R617];
        unrelate exp from rho across R617;
        select one oprd related by rho->OPRD[R615];
        unrelate rho from oprd across R615;
        delete object instance rho;
        select one dus related by oprd->DUS[R631];
        unrelate oprd from dus across R631;
        delete object instance oprd;
        select one lho related by exp->LHO[R616];
        if (not empty lho)
                unrelate exp from lho across R616;
                select one oprd related by lho->OPRD[R615];
                unrelate lho from oprd across R615;
                delete object instance lho;
                select one dus related by oprd->DUS[R631];
                unrelate oprd from dus across R631;
                delete object instance oprd;
        end if;
        unrelate self from exp across R622;
        delete object instance exp;
end for;
```

6.1.9 Event

An event causes a transition to occur in a state machine. The event therefore provides access to execute any action caused by the transition.

6.1.9.1 Relational Attributes

model_id:same_as<Base_Attribute>

state_name:same_as<Base_Attribute>

entry_id:same_as<Base_Attribute>

***object_name:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

6.1.10 Event Parameter

An event parameter is a parameter assigned to an event. The subtype is required for enforcing the "same data rule" for entry into a state.

6.1.10.1 Relational Attributes

model_id:same_as<Base_Attribute>

state_name:same_as<Base_Attribute>

id:same_as<Base_Attribute>

***type_name:same_as<Base_Attribute>**

***accessor_name:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

6.1.11 Instance Method

An instance method requires an instantiation, before it can be used. The dynamics of the instance method only apply to a single object instance.

Instance methods are favored over events when actions need to occur within the atomicity of a process.

6.1.11.1 Relational Attributes

***name:same_as<Base_Attribute>**

6.1.11.2 Operations

Algorithm 6.4 void Instance Method:instantiate()

```
select one acc related by self->MTH[R532]->ISY[R534]->SYA[R524]->ACC[R505];
instId = acc.instantiate();
select any ain from instances of AIN where selected.id == instId;
select any ins from instances of INS where selected.id == param.objectInstance;
create object instance min of MIN;
relate ain to ins across R648 using min;
```

6.1.12 Internal Synchronous

Internal synchronous accesses actions inside the domain being modeled.

NOTE: It is legal for the domain being modeled to call its own synchronous service as well.

6.1.12.1 Relational Attributes

***name:same_as<Base_Attribute>**

6.1.13 Iteration

An iteration process steps through an ordinal value from a specified starting point to a specified ending point. Typical iteration processes in third generation programming languages are for loops.

This analysis only supports a single step index into a fixed length array, therefore the iteration always starts at the first array element (one-based) and indexes through the end of the array. External editors will have to adjust their value arrays accordingly or use a loop.

6.1.13.1 Attributes

current_index:integer This attribute tracks the current position in the array whose value is contained in the variable.

6.1.13.2 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

6.1.14 Loop

A loop process performs any contained processes until a specified condition is met. Typical loop processes in third generation programming languages are while and do-while statements.

6.1.14.1 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

6.1.15 Method

Objects can have methods associated with them to handle synchronous processing tasks.

In Object Lifecycles[2], methods weren't explicitly called as such, but the processes used in the process models were explicitly tied to an object and available for reuse in many process models.

6.1.15.1 Relational Attributes

***name:same_as<Base_Attribute>**

***[2]id:same_as<Base_Attribute>**

6.1.15.2 Operations

Algorithm 6.5 boolean Method:belongsTo()

```
rc = true;
select one om related by self->OM[R532] where selected.name == param.object;
if (empty om)
        select one nm related by self->IM[R532]->NM[R533] where selected.name ==
        if (empty nm)
                select one obs related by self->IM[R532]->SMT[R533]->OBS[R125] w
                if (empty obs)
                        rc = false;
                end if;
        end if;
end if;
return rc;
```

6.1.16 Non-event Parameter

A non-event parameter is assigned to any accessor that isn't an event.

6.1.16.1 Relational Attributes

***type_name:same_as<Base_Attribute>**

***accessor_name:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

6.1.17 Non-specializing Method

A non-specializing instance method applies to all instantiations of an object, regardless of subtyping.

In the object model, these appear in unspecialized objects and root supertypes of specialized objects only.

6.1.17.1 Relational Attributes

***object_name:same_as<Base_Attribute>**

***domain_name:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

6.1.18 Object Method

An object method doesn't require an instantiation to be invoked. It is the method analog to the assigner state machine. It is used for encapsulating synchronous object dynamics that aren't specific to an object instance.

Typical usage of an object method is to initialize an instance unconditionally related to another instance in cases where actions must be performed upon creation of the instance. Such a condition can't use state machine creation, as the asynchronous nature violates the unconditional aspect of the relationship.

6.1.18.1 Relational Attributes

***object_name:same_as<Base_Attribute>**

***domain_name:same_as<Base_Attribute>**

***name:same_as<Base_Attribute>**

6.1.19 Parameter

A parameter specifies a typed data instance whose value is accessible by the action associated with the accessor.

6.1.19.1 Attributes

***name:string**

6.1.19.2 Relational Attributes

***type_name:same_as<Base_Attribute>**

***accessor_name:same_as<Base_Attribute>**

6.1.19.3 Operations

Algorithm 6.6 void Parameter:instantiate()

Create an instance of this parameter related to the specified accessor instance.

```
select any ain from instances of AIN where selected.id == param.accessorInstance
create object instance prn of PRN;
relate prn to self across R645;
relate prn to ain across R647;
create object instance var of VAR;
relate prn to var across R626;
select any emp from instances of EMP;
select one val related by emp->VAL[R629];
relate val to var across R635;
create object instance dus of DUS;
relate var to dus across R624;
select one type related by self->TYPE[R541];
relate dus to type across R625;
create object instance re of RE;
relate dus to re across R514;
```

6.1.20 Path Assignment

A path assignment is done by the path evaluation control structure to assign the nested process to a specific path.

6.1.20.1 Relational Attributes

***nesting_ordinal_position:same_as<Base_Attribute>**

***nesting_id:same_as<Base_Attribute>**

***path_ordinal_position:same_as<Base_Attribute>**

***path_id:same_as<Base_Attribute>**

6.1.21 Path Evaluation

A path evaluation process evaluates one or more expressions and chooses the processes to execute. Typical path evaluation processes in third generation programming languages are if-then-else and switch-case statements.

6.1.21.1 Attributes

numPaths:integer The number of unique paths in this control structure. Examples:
an if statement has one path
an if-else statement has two paths
an if-elif-else statement has three paths
a switch-case statement has paths equal to the number of cases

6.1.21.2 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

6.1.22 Process

A Process is a single execution statement in a textual model, or a single flow in a graphical model. It can read data, write data (implied modify), make a conditional path choice, and execute another process model.

6.1.22.1 Attributes

***ordinal_position:real** The position of the process in the action. The ordinal positions of processes determine sequence of execution when the action is invoked.

6.1.22.2 Relational Attributes

***action_id:same_as<Base_Attribute>**

6.1.23 Process Nesting

A process nesting is a process contained within a control structure. The control structure determines when the process is executed.

6.1.23.1 Relational Attributes

***controlStructure_ordinal_position:same_as<Base_Attribute>**

***controlStructure_id:same_as<Base_Attribute>**

***process_ordinal_position:same_as<Base_Attribute>**

***process_id:same_as<Base_Attribute>**

6.1.24 Reference Element

A reference element is any Shlaer-Mellor element that can be accessed as part of a procedural model. A reference element invokes an action and/or assigns a value to a variable.

The reference element constrains which elements in the metamodel can participate in a process.

6.1.24.1 Attributes

***id:unique_id**

6.1.25 Return Value

A return value specifies a type defined for an access that returns a data value.

6.1.25.1 Relational Attributes

***[2]id:same_as<Base_Attribute>**

***type_name:same_as<Base_Attribute>**

***access_name:same_as<Base_Attribute>**

6.1.26 Specializing Method

A specializing instance method exists in only a subset of object instantiations.

6.1.26.1 Relational Attributes

***name:same_as<Base_Attribute>**

****[2]specialization_id:same_as<Base_Attribute>**

6.1.27 Statement

A statement process executes a single expression. It doesn't contain other processes. The typical executions are assignment, event generation, or external action calls.

6.1.27.1 Relational Attributes

***ordinal_position:same_as<Base_Attribute>**

***action_id:same_as<Base_Attribute>**

6.1.28 Synchronous Accessor

A synchronous accessor executes during the time that the action is running. (see [2] page 131)

Only synchronous accessors can return a value. Action is suspended until the value is returned. This means that a synchronous service provided by another domain can launch asynchronous actions to obtain the return value, but this domain will be blocked while waiting on the actions to complete. The analyst should take this into consideration.

6.1.28.1 Relational Attributes

***name:same_as<Base_Attribute>**

6.2 Relationship Descriptions

R500 An Active Element is the container for the Action. The Active Element provides access, either internal or external, for launching the Action.

R501 The Action executes Processes upon launch. The Processes are unique for the Action. Reuse of Processes is achieved by Actions launching other Actions via their Active Element.

R502 Processes can use one or more Referenced Elements for data access and Action execution.

R505 An accessor is an asynchronous accessor or a synchronous accessor.

R506 A synchronous accessor requires asynchronous action from an internal event or an external domain.

R509 Only certain elements of OOA modeling can contain process models and therefore are active elements.

R514 Only certain elements of OOA can be referenced by process models, so those elements are constrained by this relationship.

R522 An attribute action uses the derivations defined for an attribute to calculate the attribute's value. The derivations are only used by one attribute action.

R523 An attribute action is defined for providing a value for only one non-referential attribute. Not all non-referential attributes have their value derived.

R524 A synchronous accessor requires synchronous action from internal processing or an external domain.

R529 A type can be used to constrain returns from synchronous accesses, and a synchronous access can return a specified data type.

R532 Methods can be applied to instantiated object instances or uninstantiated objects.

R533 An instance method can exist for all objects instances or only a subset of object instances.

R534 An internal, synchronous accessor is either a method or an attribute action.

R535 An event is accessed via a subset, and a subset can provide access for many events.

R536 Processes are single statements or control structures that contain other processes.

R537 The control structures supported in this analysis are path evaluation, iteration, and loops.

R538 A control structure controls zero to many processes, and a process can have its execution controlled by a control structure. Another way to look at this relationship is as containment where a control structure can contain other processes. Scoping is avoided in this analysis as variables are scoped by the action, not by any processes.

R539 A path evaluation assigns its nested processes to specific paths.

R540 An accessor can have parameters assigned to it. A parameter is always assigned to an accessor.

R541 A parameter is constrained by a type, and a type can constrain many parameters.

R542 A parameter is either an event parameter or a non-event parameter. The event parameter subtype is required to enforce the "same data rule" on entry into a state.

Chapter 7

The Relationship Subsystem

The subsystem of the metamodel concerning relationships.

This diagram borrows heavily from the miUML[8] metamodel, because why reinvent the open-sourced wheel? The key differences are the extra level of abstraction due to instance representation in this metamodel, and the lack of a generalization relationship in this metamodel; generalizations aren't relationships in this metamodel.

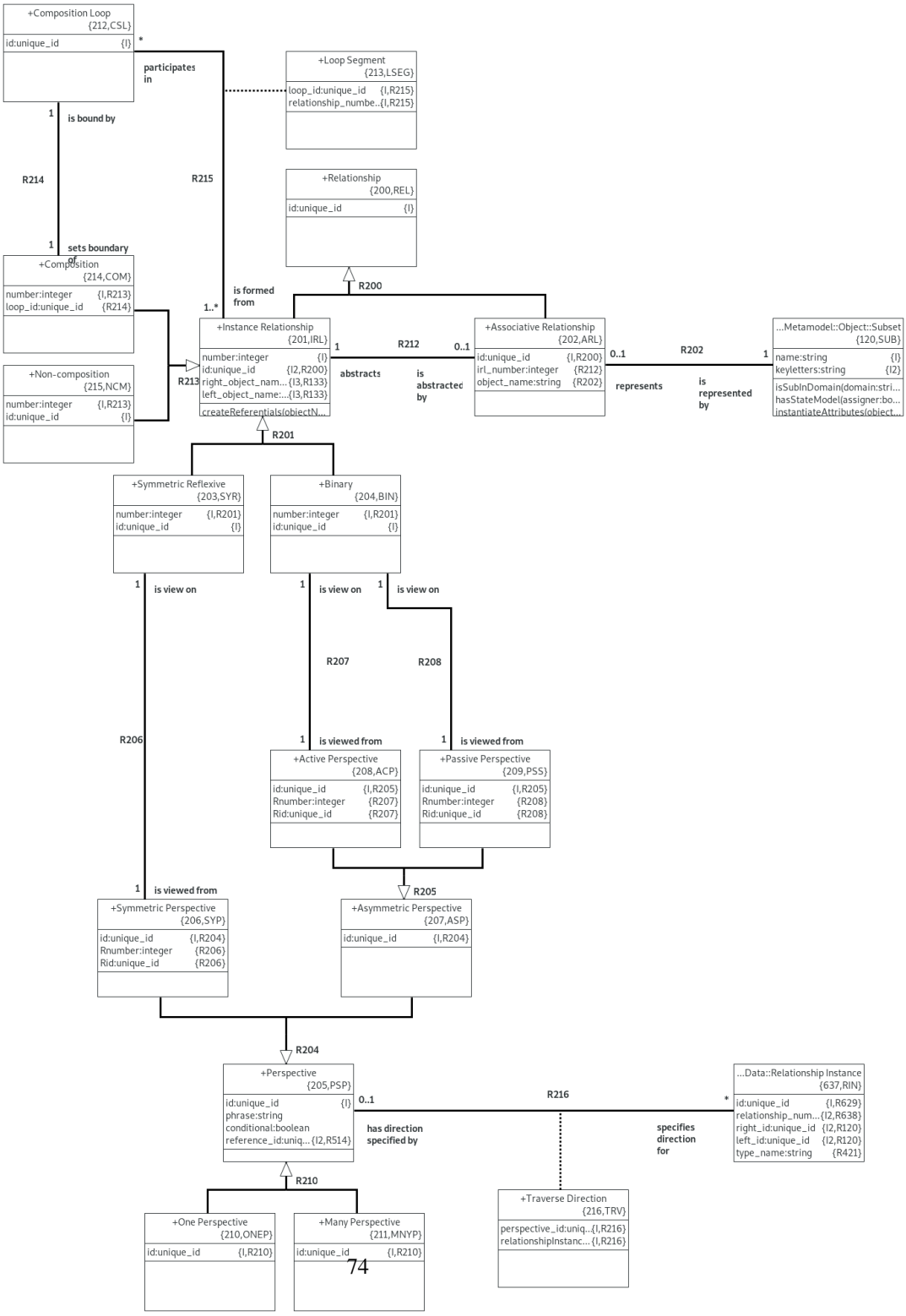

Figure 7.1: Relationship Subsystem Diagram

7.1 Object and Attribute Descriptions

7.1.1 Active Perspective

"A Binary Association has two Perspectives, one Active and one Passive Perspective. In fact, the two sides of an Association could have just as easily been designated as the A side and the B side. Using the terms Active / Passive offers the modeler a systematic way to choose the phrase to apply to each side. For example, the phrase pair configures / is configured by readily establishes the Perspective sides.

If it's not clear from the phrase names which side should be active or passive, then arbitrarily assign each role and be done with it. You can always query the metamodel later to find out which is which. Any miUML class diagram editor should provide easy UI access to this query (highlight the A/P sides)."[8]

7.1.1.1 Relational Attributes

Rid:same_as<Base_Attribute>

Rnumber:same_as<Base_Attribute>

***id:same_as<Base_Attribute>**

7.1.2 Associative Relationship

An associative relationship requires further abstraction by an object. The set defined by the object allows instances of relationships to have further relationships, data properties, and dynamic processing.

An associative relationship relates the associative object to a relationship between object instances.

7.1.2.1 Relational Attributes

object_name:same_as<Base_Attribute>

irl_number:same_as<Base_Attribute>

***id:same_as<Base_Attribute>**

7.1.3 Asymmetric Perspective

"Each side of a Binary Association has a distinct Perspective, either Active or Passive. Since each side is from a different point of view, it establishes an Asymmetric Perspective."[8]

7.1.3.1 Relational Attributes

***id:same_as<Base_Attribute>**

7.1.4 Binary

"The term 'binary' means that there are exactly two perspectives on this type of Association. It does NOT mean that there are two Classes. A reflexive Binary Association may be created on a single Class such that each of the two Perspectives is viewed from the same Class."[8]

7.1.4.1 Attributes

***id:unique_id**

7.1.4.2 Relational Attributes

***number:same_as<Base_Attribute>**

7.1.5 Composition

An instance of composition is the loop segment containing the relationship instance that is the result of the composition equation.

"[When a relationship is the logical consequence of other relationships,] Such a relationship is said to be formed by composition (as in composition of functions in mathematics. [...] A relationship formed by composition cannot be formalized in referential attributes, since the connections between the instances is already given by the connections between the [composing relationships].

A relationship formed by composition is annotated on the model as [composed relationship = 1st composing relationship + 2nd composing relationship [+ nth composing relationship ...]]."[2]

"Composed Relationships. Another special case occurs when the constraint on the referential attribute is such that it identifies a single instance of the associated object. [...] Composition of relationships captures the constraint directly in data [...] However the use of composition is limited in that it requires that the constraint always identify a single associated instance."[5]

Both definitions identify compositions as relationship combinations that loop back to the originating instance. miUML[8] calls compositions, "Constrained Loops", because the composition equation forms a constraint on the object instances allowed to participate.

OOA '96[5] also discusses collapsed referential identifiers, but the example shows them to be an alternative way to draw compositions on the object model. This metamodel will only model a composition and leave the way to display it up to the model editor.

7.1.5.1 Relational Attributes

loop_id:same_as<Base_Attribute>

***number:same_as<Base_Attribute>**

7.1.6 Composition Loop

A composition loop is the set of relationships participating in the composition.

7.1.6.1 Attributes

***id:unique_id**

7.1.7 Instance Relationship

An instance relationship describes the constraints and associations imposed on instantiation of related objects.

7.1.7.1 Attributes

***number:integer**

7.1.7.2 Relational Attributes

***[3]left_object_name:same_as<Base_Attribute>**

***[3]right_object_name:same_as<Base_Attribute>**

***[2]id:same_as<Base_Attribute>**

7.1.7.3 Operations

Algorithm 7.1 void Instance Relationship:createReferentials()

```
select any sub from instances of SUB where selected.name == param.objectName;
select many idps related by sub->OBJ[R107]->OA[R129]->ATR[R124]->IDP[R102];
if (empty idps)
        select many idps related by sub->PRP[R107]->SPA[R130]->ATR[R124]->IDP[R1
end if;
for each idp in idps
        select one atr related by idp->ATR[R102];
        create object instance refatr of ATR;
        refatr.name = atr.name + "(R" + TC::intToString( i:self.number ) + ")";
        create object instance ref of REF;
        relate ref to refatr across R103;
        relate idp to ref across R137;
        select one ncm related by self->NCM[R213];
        relate ref to ncm across R128;
end for;
```

7.1.8 Loop Segment

A loop segment is a relationship instance participating in a composition loop.

7.1.8.1 Relational Attributes

***relationship_number:same_as<Base_Attribute>**

***loop_id:same_as<Base_Attribute>**

7.1.9 Many Perspective

This is a Perspective with a multiplicity of many.[8]

7.1.9.1 Relational Attributes

***id:same_as<Base_Attribute>**

7.1.10 Non-composition

Per Object Lifecycles[2], the basis of the composition definition is the composition of functions in mathematics, so a function in this usage is a relationship instance that is used to compose the composition relationship instance.

7.1.10.1 Attributes

***id:unique_id**

7.1.10.2 Relational Attributes

***number:same_as<Base_Attribute>**

7.1.11 One Perspective

"This is a Perspective with a multiplicity of one."[8]

7.1.11.1 Relational Attributes

***id:same_as<Base_Attribute>**

7.1.12 Passive Perspective

"A Binary Association has two Perspectives, one Active and one Passive Perspective. In fact, the two sides of an Association could have just as easily been designated as the A side and the B side. Using the terms Active / Passive offers the modeler a systematic way to choose the phrase to apply to each side. For example, the phrase pair configures / is configured by readily establishes the Perspective sides.

If it's not clear from the phrase names which side should be active or passive, then arbitrarily assign each role and be done with it. You can always query the metamodel later to find out which is which. Any miUML class diagram editor should provide easy UI access to this query (highlight the A/P sides)."[8]

7.1.12.1 Relational Attributes

Rid:same_as<Base_Attribute>

Rnumber:same_as<Base_Attribute>

***id:same_as<Base_Attribute>**

7.1.13 Perspective

"A Perspective is a point of view from a hypothetical Instance on an Association."[8]

Perspective was taken from the miUML metamodel, which doesn't have an instance object, so in this metamodel, instance isn't hypothetical and always has a perspective when involved in a relationship.

Further discussion on perspective can be found in the chapter 4, "Reflexive Relationships", in OOA '96[5].

7.1.13.1 Attributes

conditional:boolean

phrase:string

***id:unique_id** The unique identifier for the perspective.

7.1.13.2 Relational Attributes

***[2]reference_id:same_as<Base_Attribute>**

7.1.14 Relationship

"A relationship is an abstraction of a set of associations that hold systematically between different kinds of things in the real world."[2]

The relationship represents a table containing a row for every instance of the relationship, and a column for each object participating in the relationship. When looking at the object model, the relationship represent the empty table. In the process model, the rows of the table are populated. The rows of the table, then represent relationship instance values.

7.1.14.1 Attributes

***id:unique_id**

7.1.15 Symmetric Perspective

"A Unary Association has only one Perspective. Given two Objects (or the same Object linked to itself) on a Unary Association, the role played by either side of the Link is identical. There is, consequently, just one Symmetric Perspective.

Therefore, only one phrase name, one multiplicity and one conditionality need be specified for a Unary Association."[8]

A Unary Association in this metamodel always involves just one Object. Unary Association is the same as Symmetric Reflexive Relationship in OOA '96[5].

7.1.15.1 Relational Attributes

Rid:same_as<Base_Attribute>

Rnumber:same_as<Base_Attribute>

***id:same_as<Base_Attribute>**

7.1.16 Symmetric Reflexive

The symmetric reflexive relationship is described in OOA '96[5]. A symmetric reflexive relationship has the same multiplicity,conditionality, and verb phrase at both ends of the relationship, so those specifiers are squished into one perspective.

7.1.16.1 Attributes

***id:unique_id**

7.1.16.2 Relational Attributes

***number:same_as<Base_Attribute>**

7.1.17 Traversal Direction

The direction set by the perspective on a relationship to use to traverse the relationship to the linked object instance.

7.1.17.1 Relational Attributes

***relationshipInstance_id:same_as<Base_Attribute>**

***perspective_id:same_as<Base_Attribute>**

7.2 Relationship Descriptions

R200 Two types of relationships exist in a Shlaer-Mellor model: instance-based and associative.

R201 The two types of instance-based relationships are unary and binary. The designation of unary and binary refer to the number of perspectives of the instances involved, and not the number of instances.

R204 A perspective is an asymmetric perspective or symmetric perspective.

R205 An asymmetric perspective can be viewed in active or passive tense.

R206 The symmetric perspective is the view both from and on the unary relationship.

R207 The active perspective is viewed from one side of the binary relationship.

R208 The passive perspective is viewed from one side of the binary relationship.

R210 "The direction of reference of a Referential Attribute is determined by the availability of a One Perspective. Conditionality is less significant in this regard. So to capture the basic reference rule that 'it is always possible to refer to a One Perspective' it is necessary to abstract the One / Many specialization. See the Formalization Subsystem to see how it is used.
Regardless of conditionality, every Perspective is either One or Many (1, M) or, in UML terminology, (0..1, 1 or 0..*, 1..*)."[8]

R212 An instance relationship can abstract a single associative relationship, but an associative relationship is always abstracted by an instance relationship.

R213 An instance relationship can be a non-composition (formalized by referential attributes) or a composition (formalized by a relationship loop) instance relationship.

R214 A composition is bounded by a composition loop. The composition loop bounds the composition.

R215 An instance relationship can belong to one or more composition loops. The composition loop requires a contiguous, closed path from many instance relationships.

R202 An associative relationship is always abstracted by an object or subset of an object. When an object is used to abstract an associative relationship, it's life span is the same as the relationship's life span.

R216 A relationship instance can have traversal direction specified by a relationship perspective, and a relationship perspective can specify traversal direction for many relationship instances.

Chapter 8

The Type Subsystem

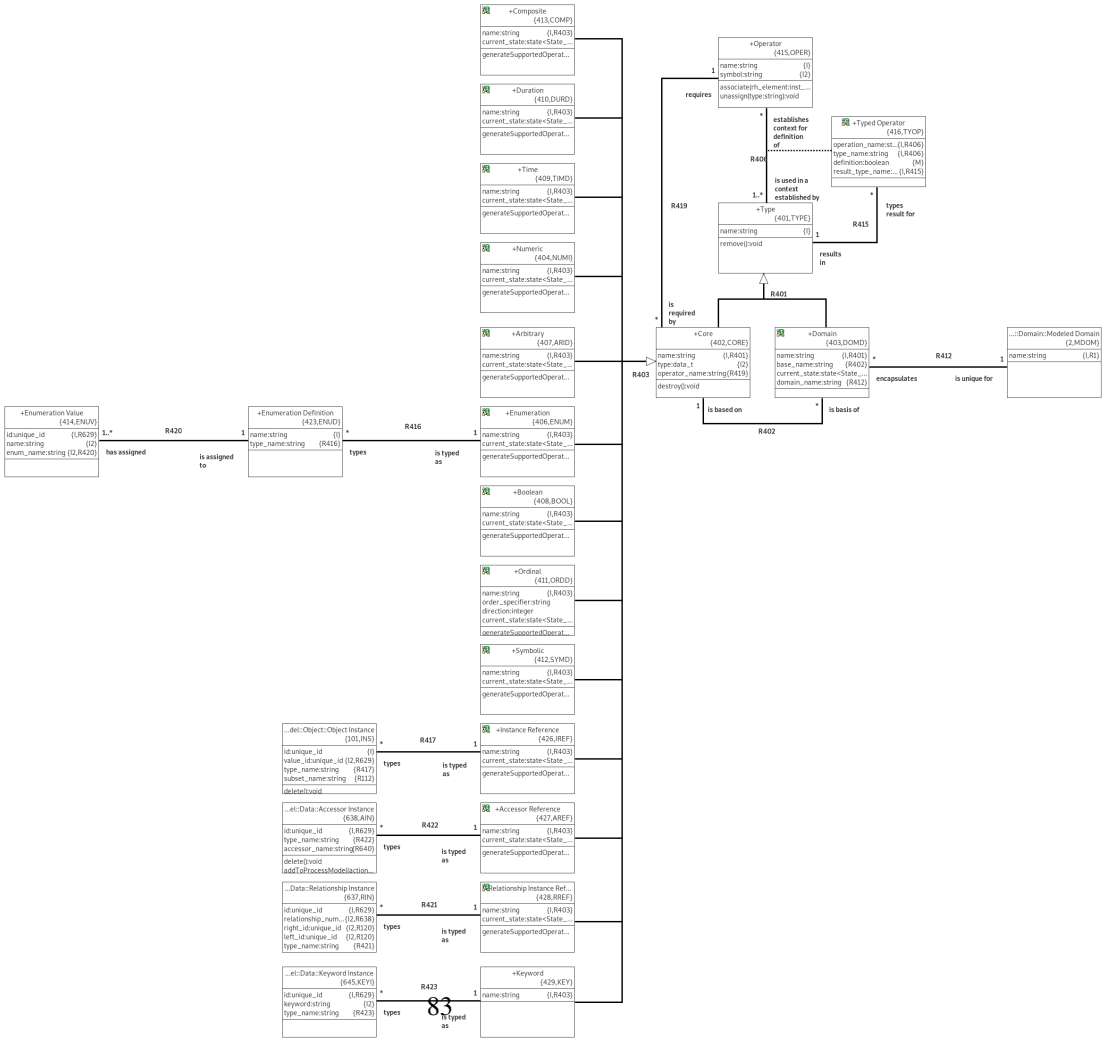

Figure 8.1: Type Subsystem Diagram

8.1 Object and Attribute Descriptions

8.1.1 Accessor Reference

An accessor reference is the type to which a variable with an accessor as a value must conform. In OAL this is an event instance; in C it would be like a function pointer.

8.1.1.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.1.2 Operations

Algorithm 8.1 void Accessor Reference:generateSupportedOperators()

The operations permitted for instance reference data types are:

- the comparisons = and != (identical and not identical in value)
- the set existence checks of empty and not empty.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"accessor reference
```

8.1.1.3 Instance State Model

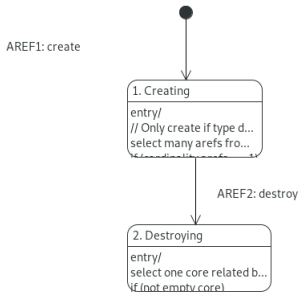

Figure 8.2: Accessor Reference State Model

Algorithm 8.2 Creating

```
// Only create if type doesn''t exist
select many arefs from instances of AREF;
if (cardinality arefs == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "accessor reference";
   core.type = data_t::ACCESSOR_REFERENCE;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of type AREF attempted!");
   generate AREF2:destroy to self;
end if;
```

Algorithm 8.3 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.2 Arbitrary

"To define a data type for data elements that represent arbitrary identifiers: data type <data type name> is arbitrary

The implementation of an arbitrary type like all the base data types is determined by the architecture domain. Hence the analyst should make no assumptions as to how this is done: the arbitrary type may be implemented as a handle, an integer, a character string, or by any other scheme the architects devise. For this reason, the analyst cannot specify a default value for the base data type arbitrary."[6]

8.1.2.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.2.2 Operations

Algorithm 8.4 void Arbitrary:generateSupportedOperators()

The OOA of Data[6] prescribes no operators for arbitrary. Arbitrary is treated like a composite, where the internals are unknown, so the only valid operations permitted using data types based on arbitrary are limited to equality comparison and assignment to another arbitrary type.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"arbitrary", result
generate TYOP_A1:create(name:"equal", symbol:"=", type:"arbitrary", result:"bool
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"arbitrary", result:
```

8.1.2.3 Instance State Model

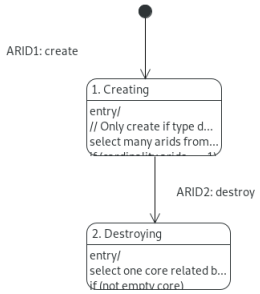

Figure 8.3: Arbitrary State Model

Algorithm 8.5 Creating

```
// Only create if type doesn''t exist
select many arids from instances of ARID;
if (cardinality arids == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "arbitrary";
   core.type = data_t::ARBITRARY;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of ARID attempted!");
   generate ARID2:destroy to self;
end if;
```

Algorithm 8.6 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.3 Boolean

"The boolean base data type is exactly what you expect: a pre-defined enumerated data type with values True and False. To define a domain-specific data type based on a boolean base type, write: data type <name> is boolean (default value is <value>)

The operations permitted for data elements based on these base types include the comparison operations, represented as = (identical in value) and != (not identical in value). The result of either comparison yields a data element of base type boolean. The logical operations, not, and, & or, are defined in the standard way."[6]

8.1.3.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.3.2 Operations

Algorithm 8.7 void Boolean:generateSupportedOperators()

"The operations permitted for data elements based on these base types include the comparison operations, represented as = (identical in value) and != (not identical in value). The result of either comparison yields a data element of base type boolean. The logical operations, not, and, & or, are defined in the standard way."[6]

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"boolean", result:"
generate TYOP_A1:create(name:"equal", symbol:"=", type:"boolean", result:"boolea
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"boolean", result:"b
generate TYOP_A1:create(name:"logical not", symbol:"not", type:"boolean", result
generate TYOP_A1:create(name:"logical and", symbol:"and", type:"boolean", result
generate TYOP_A1:create(name:"logical or", symbol:"or", type:"boolean", result:"
```

8.1.3.3 Instance State Model

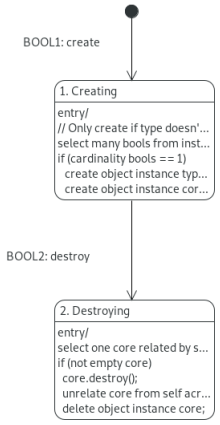

Figure 8.4: Boolean State Model

Algorithm 8.8 Creating

```
// Only create if type doesn''t exist
select many bools from instances of BOOL;
if (cardinality bools == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    relate core to self across R403;
    type.name = "boolean";
    core.type = data_t::BOOLEAN;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of BOOL attempted!");
    generate BOOL2:destroy to self;
end if;
```

Algorithm 8.9 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
   core.destroy();
   unrelate core from self across R403;
   delete object instance core;
end if;
```

8.1.4 Composite

"A type may be composite, but the corresponding attribute must always be treated by the domain as a single unit"[3]

The operations permitted using data types based on composite are limited to equality comparison and assignment to another composite type.

If individual elements of a composite type are to be operated on within the passed-to domain, then they must be sent individually. If they need to be treated as a group, then a class must be declared within the domain that supports them.

The operations external to the domain to support these two mechanisms consist of ungrouping the data from the composite, making the data available to the domain, and then regrouping the processed data. NOTE: while this process sounds onerous, the architecture can perform coping mechanisms, such as mapping instances of the class in the domain to point to memory locations in the composite external to the domain. In this case, the whole ungroup, pass-in, and regroup is done automatically and the external operation only needs to ensure the sequence occurs without interference.

8.1.4.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.4.2 Operations

Algorithm 8.10 void Composite:generateSupportedOperators()

The operations permitted using data types based on composite are limited to equality comparison and assignment to another composite type.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"composite", result
generate TYOP_A1:create(name:"equal", symbol:"=", type:"composite", result:"bool
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"composite", result:
```

8.1.4.3 Instance State Model

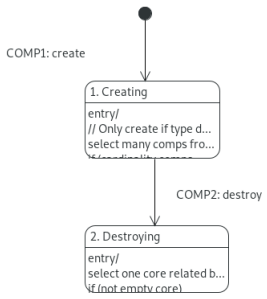

Figure 8.5: Composite State Model

Algorithm 8.11 Creating

```
// Only create if type doesn''t exist
select many comps from instances of COMP;
if (cardinality comps == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "composite";
   core.type = data_t::COMPOSITE;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of COMP attempted!");
   generate COMP2:destroy to self;
end if;
```

Algorithm 8.12 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.5 Core

Core types are defined within the context of the Shlaer-Mellor Method. Every Shlaer-Mellor model, adhering to this metamodel, will support these types. Core types are not intended to be multiply instantiated. Each core type is defined only once in the metamodel. Specialization of the core types are done through definitions based on the core types. These definitions are considered domain types, so they aren't modeled in this metamodel. The enumeration definition is an exception to this rule, as it has a known structure.

8.1.5.1 Attributes

***[2]type:data_t**

8.1.5.2 Relational Attributes

operator_name:same_as<Base_Attribute>

***name:same_as<Base_Attribute>**

8.1.5.3 Operations

Algorithm 8.13 void Core:destroy()

```
select one operator related by self->OPER[R419];
unrelate self from operator across R419;
select one type related by self->TYPE[R401];
if (not empty type)
  type.remove();
  unrelate type from self across R401;
  delete object instance type;
end if;
select many udts related by self->DOMD[R402];
for each udt in udts
  unrelate self from udt across R402;
  generate DOMD2:destroy() to udt;
end for;
```

8.1.6 Domain

Domain types are unique within the context of the domain. Domain types are often also called user-defined types, because they are defined by the analyst when modeling the domain. Often domain types will have a known name within the subject matter of the domain.

Domain types are formed using a core type as a base.

8.1.6.1 Relational Attributes

domain_name:same_as<Base_Attribute>

current_state:state<State_Model>

base_name:same_as<Base_Attribute>

***name:same_as<Base_Attribute>**

8.1.6.2 Instance State Model

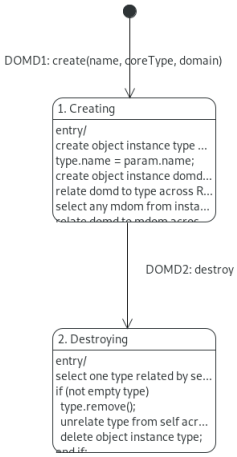

Figure 8.6: Domain State Model

Algorithm 8.14 Creating

```
create object instance type of TYPE;
type.name = param.name;
create object instance domd of DOMD;
relate domd to type across R401;
select any mdom from instances of MDOM where selected.name == param.domain;
relate domd to mdom across R412;
select many cores from instances of CORE;
for each core in cores
  if (core.type == param.coreType)
    relate core to domd across R402;
    break;
  end if;
end for;
```

Algorithm 8.15 Destroying

```
select one type related by self->TYPE[R401];
if (not empty type)
  type.remove();
  unrelate type from self across R401;
  delete object instance type;
end if;
select one core related by self->CORE[R402.''is based on''];
unrelate self from core across R402.''is based on'';
```

8.1.7 Duration

"Similarly, to define a data type that represents duration, write data type <data type name> is duration range is from <low limit> to <high limit> units are [year | month | day | hour | minute | second | millisec | microsec] precision is <smallest discriminated value>

The operations permitted using data types based on time and duration are: time := time ± duration duration := duration ± duration duration := duration * numeric duration := duration / numeric duration := time - time as well as the standard comparisons of < (read as "before"), >, $\leq$, and $\geq$. Each such comparison yields a data element of base type boolean. Comparisons are defined only between elements of the same base type."[6]

8.1.7.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.7.2 Operations

Algorithm 8.16 void Duration:generateSupportedOperators()

"The operations permitted using data types based on ... duration are:

- duration := duration ± duration
- duration := duration * numeric
- duration := duration / numeric
- duration := time - time

as well as the standard comparisons of < (read as "before"), >, $\leq$, and $\geq$. Each such comparison yields a data element of base type boolean. Comparisons are defined only between elements of the same base type."[6]
NOTE: For mixed type operations, conversion operators must be supported. The explicit conversion cases are:

- duration * numeric: duration->numeric result: numeric
- duration / numeric: duration->numeric result: numeric
- duration := numeric: numeric->duration
- duration := time: time->duration

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"duration", result:
generate TYOP_A1:create(name:"addition", symbol:"+", type:"duration", result:"du
generate TYOP_A1:create(name:"subtraction", symbol:"-", type:"duration", result:
generate TYOP_A1:create(name:"equal", symbol:"=", type:"duration", result:"boole
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"duration", result:"
generate TYOP_A1:create(name:"greater", symbol:">", type:"duration", result:"boo
generate TYOP_A1:create(name:"lesser", symbol:"<", type:"duration", result:"bool
generate TYOP_A1:create(name:"greater or equal", symbol:">=", type:"duration", r
generate TYOP_A1:create(name:"lesser or equal", symbol:"<=", type:"duration", re
// It is highly recommended that the architecture handle conversions auto-
// matically, so the analyst doesn''t have to explicitly add the conversion
// operator.
generate TYOP_A1:create(name:"convert to numeric", symbol:"->NUMI", type:"duratio
```

8.1.7.3 Instance State Model

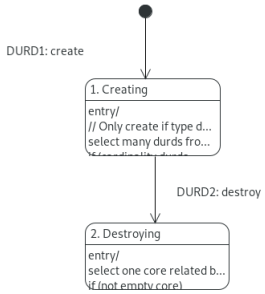

Figure 8.7: Duration State Model

Algorithm 8.17 Creating

```
// Only create if type doesn''t exist
select many durds from instances of DURD;
if (cardinality durds == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    type.name = "duration";
    core.type = data_t::DURATION;
    relate core to self across R403;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of DURD attempted!");
    generate DURD2:destroy to self;
end if;
```

Algorithm 8.18 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
   core.destroy();
   unrelate core from self across R403;
   delete object instance core;
end if;
```

8.1.8 Enumeration

"If a data type permits a finite set of values, define it as: data type <name> is enumerated values are <value 1 >, <value 2 >, . . . <value N > (default value is <value k >) 4 as in: data type IC color is enumerated values are red, blue, black, green, silver

The only operations permitted for data elements of an enumerated data type are the comparison operations, represented as = (identical in value) and != (not identical in value). The result of either comparison yields a data element of type boolean."[6]

8.1.8.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.8.2 Operations

Algorithm 8.19 void Enumeration:generateSupportedOperators()

["The only operations permitted for data elements of an enumerated data type are the comparison operations, represented as = (identical in value) and != (not identical in value). The result of either comparison yields a data element of type boolean."[6]

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"enumeration", resu
generate TYOP_A1:create(name:"equal", symbol:"=", type:"enumeration", result:"bo
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"enumeration", resul
```

8.1.8.3 Instance State Model

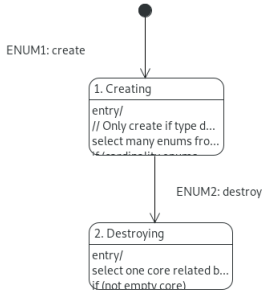

Figure 8.8: Enumeration State Model

Algorithm 8.20 Creating

```
// Only create if type doesn''t exist
select many enums from instances of ENUM;
if (cardinality enums == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "enumeration";
   core.type = data_t::ENUMERATION;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of ENUM attempted!");
   generate ENUM2:destroy to self;
end if;
```

Algorithm 8.21 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.9 Enumeration Definition

An enumeration definition bounds the set of enumeration values allowed when an enumeration type is used. A variable is declared as an enumeration definition as if the enumeration definition was the type. This is typical of how most programming languages support the use of enumerations. e.g., <enum keyword> <type name> <enumerator value list> <variable name>

8.1.9.1 Attributes

***name:string**

8.1.9.2 Relational Attributes

type_name:same_as<Base_Attribute>

8.1.10 Enumeration Value

An enumeration value is one of the enumerators that compose an enumeration. The uses of enumeration values are constrained by the operations supported by the enumeration type. An enumeration value can never be identified by just the value name; it must always use the enumeration definition name as well. e.g., def::value

8.1.10.1 Attributes

***[2]name:string**

8.1.10.2 Relational Attributes

***[2]enum_name:same_as<Base_Attribute>**

***id:same_as<Base_Attribute>**

8.1.11 Instance Reference

An instance reference is the type to which all object instances must conform. This is a core type that says object instances have a common set of operations defined for usage.

8.1.11.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.11.2 Operations

Algorithm 8.22 void Instance Reference:generateSupportedOperators()

The operations permitted for instance reference data types are

- the comparisons = and != (identical and not identical in value)
- the set existence checks of empty and not empty.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"instance reference
generate TYOP_A1:create(name:"equal", symbol:"=", type:"instance reference", res
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"instance reference"
generate TYOP_A1:create(name:"empty", symbol:"empty", type:"instance reference",
generate TYOP_A1:create(name:"not empty", symbol:"not empty", type:"instance ref
```

8.1.11.3 Instance State Model

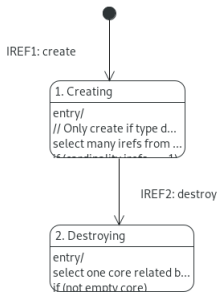

Figure 8.9: Instance Reference State Model

Algorithm 8.23 Creating

```
// Only create if type doesn''t exist
select many irefs from instances of IREF;
if (cardinality irefs == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    type.name = "instance reference";
    core.type = data_t::INSTANCE_REFERENCE;
    relate core to self across R403;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of type IREF attempted!");
    generate IREF2:destroy to self;
end if;
```

Algorithm 8.24 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
   core.destroy();
   unrelate core from self across R403;
   delete object instance core;
end if;
```

8.1.12 Numeric

Numeric is as described below. NOTE: The description isn't a model. Some seeming attributes of numeric in the description might be modeled as their own objects. e.g., "units" aren't always used, so cannot be an attribute.

"If a data type is numeric in nature, write: data type <data type name> is numeric (base <N>) range is from <low limit> to <high limit> units are <unit symbol> precision is <smallest discriminant> (default value is <value>) where base N specifies the base of the quantities <low limit>, <high limit>, <smallest discriminant> and <value>. If base N is omitted, base 10 is assumed. Hence: data type ring diameter is numeric range is from 0 to 39 units are cm precision is 0.01 data type bit pattern is numeric base 8 range is from 0 to 177777 units are octal bits precision is 1 Note that the analyst does not specify whether a numeric data type will be implemented as an integer or a real number. This will ultimately be determined by the architecture, based on the native types available in the implementation language, the word length of these native types, and the range and precision required for the data type. As a result, the OOA models of any domain are entirely decoupled from the implementation technology, thereby maximizing the potential for reuse across a wide range of platforms and implementation languages.

The operations permitted for numeric data types are: - the standard arithmetic operations +, -, * (multiplication), / (division), %% (division modulo N), and ** (exponentiation). The result of such an operation is again of base type numeric. - the standard arithmetic comparisons of = , != , <, >, <= , and >= . The result of such a comparison yields a data element of base type boolean."[6]

8.1.12.1 Attributes

Keyword A keyword is a special processing directive.

8.1.12.2 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.12.3 Operations

Algorithm 8.25 void Numeric:generateSupportedOperators()

"the standard arithmetic operations +, -, * (multiplication), / (division), %% (division modulo N), and ** (exponentiation). The result of such an operation is again of base type numeric. the standard arithmetic comparisons of = , != , <, >, <= , and >=. The result of such an operation is of base type boolean."[6]

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"numeric", result:"
generate TYOP_A1:create(name:"addition", symbol:"+", type:"numeric", result:"nun
generate TYOP_A1:create(name:"subtraction", symbol:"-", type:"numeric", result:"
generate TYOP_A1:create(name:"multiplication", symbol:"*", type:"numeric", resul
generate TYOP_A1:create(name:"division", symbol:"/", type:"numeric", result:"nun
generate TYOP_A1:create(name:"modulo", symbol:"%%", type:"numeric", result:"nume
generate TYOP_A1:create(name:"exponentiation", symbol:"**", type:"numeric", resu
generate TYOP_A1:create(name:"equal", symbol:"=", type:"numeric", result:"boolea
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"numeric", result:"b
generate TYOP_A1:create(name:"greater", symbol:">", type:"numeric", result:"bool
generate TYOP_A1:create(name:"lesser", symbol:"<", type:"numeric", result:"boole
generate TYOP_A1:create(name:"greater or equal", symbol:">=", type:"numeric", re
generate TYOP_A1:create(name:"lesser or equal", symbol:"<=", type:"numeric", res
```

8.1.12.4 Instance State Model

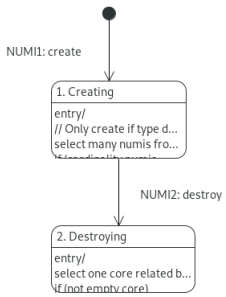

Figure 8.10: Numeric State Model

Algorithm 8.26 Creating

Responsible for instantiating this core data type and all operations associated with it. NOTE: At the time of this writing, core data types are never expected to be deleted, as deleting a core type would require deletion of all user data types based on the core type and all attributes and synchronous functionality ultimately based on the core type. There is no logical default value to use as a fall back.

```
// Only create if type doesn''t exist
select many numis from instances of NUMI;
if (cardinality numis == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "numeric";
   core.type = data_t::NUMERIC;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of NUMI attempted!");
   generate NUMI2:destroy to self;
end if;
```

Algorithm 8.27 Destroying

Responsible for instantiating this core data type and all operations associated with it. NOTE: At the time of this writing, core data types are never expected to be deleted, as deleting a core type would require deletion of all user data types based on the core type and all attributes and synchronous functionality ultimately based on the core type. There is no logical default value to use as a fall back.

```
select one core related by self->CORE[R403];
if (not empty core)
        core.destroy();
        unrelate core from self across R403;
        delete object instance core;
end if;
```

8.1.13 Operator

An Operator represents a function that may be defined for a Type. Not all Operators will be visible in implementation. e.g., an action language may choose to allow automatic conversion from one type to another, 5 + "5" = 10. In such a case, a string to numeric Operator must be defined as part of the instantiation of the metamodel.

8.1.13.1 Attributes

***[2]symbol:string** "A short, often single character, name to represent an Operator in a written expression. Common examples are + , - , * , ++ , next , etc."[8]

***name:string** "A descriptive name applicable to multiple Types such as add, multiply, increment, etc."[8]

8.1.13.2 Operations

Algorithm 8.28 void Operator:associate(rh_element:inst_ref, lh_element:inst_ref)

Operator association associates the operator for use in an equation.
rh_element: The element on the right hand side of the operator.
lh_element: The element on the left hand side of the operator.

Algorithm 8.29 void Operator:unassign(type:string)

```
select any type related by self->TYPE[R406] where selected.name == param.type;
select many tyops related by self->TYOP[R406] where selected.type_name == param.
for each tyop in tyops
        unrelate self from type across R406 using tyop;
        select one restype related by tyop->TYPE[R415];
        unrelate restype from tyop across R415;
        delete object instance tyop;
end for;
```

8.1.14 Ordinal

The ordinal core type automatically populated in the metamodel is specified as ascending and based upon identifier, because the only known operation is for traversing a collection of object instances.

To support other ordinal uses in a process model, the user must create a user-defined ordinal type.

The below gives more detail on ordinals:

"Ordinal data types are used to express order, such as first, second, and so on. However, the subject of ordering is a lot more interesting that just this common example; hence the following digression.

An ordering is always applied to a set of elements. The set can be finite or infinite. There are two types of orderings to consider. The first is the most familiar; it is a complete ordering. What this means is that you can express the concept of "before" (represented as <) between any two members of the set. Hence, 7 is before 26 (7 < 26). A complete ordering has the property of transitivity:

If A is before B, and B is before C, then A is before C.

A practical example would be the ordering of the cars that make up a freight train. Assume we define a first car. Then we could pick any two cars and easily determine which one was before the other. Far more interesting are the partial orderings. Consider this sketch of a partial ordering.

A -> B -> C -> D + -> E -> F -> G

Using the obvious interpretation, we can say that A < B (A is before B), C < D, C < E, and E < F. But we cannot say anything about the relationship between D and F: They are non-comparable.

Examples of structures that are partially ordered include PERT charts, trees used for any purpose, interlock chains, the connectivity of an electric grid, and the like. All of these can be modeled in complete detail using standard OOA relationships; for examples see [4] and Chapter 4 of Shlaer-Mellor Method: The OOA96 Report[5]. Note, however, that when modeling such a structure, one frequently finds it necessary to employ quite a number of ancillary objects (such as root node, parent node, child node, and leaf node) together with a significant set of relationships all required to express a generally well-known concept. While this can be quite satisfying when one is in a purist frame of mind, the pragmatist points out that such constructions are often of limited value, obscuring, as they can, the true purpose of the model. This becomes particularly pertinent when constructing models of an architecture, where ordering is a particularly prominent theme (see The Timepoint Architecture chapter). Hence we have defined the ordinal base data type, leaving it to the judgment of the analyst as for when to use an ordinal attribute as opposed to using more fully expressive OOA objects and relationships.

Returning now to the main theme, an ordinal data type is defined by:

data type <data type name> is ordinal

The operations permitted for ordinal data types are:

- the comparisons = and != (identical and not identical in value)
- the comparisons < (read as "before"), >, <= , and >= . Each such comparison yields a data element of base type boolean if the ordering is complete, and of base type extended boolean if the ordering is partial."[6]
  NOTE: This metamodel doesn't support an extended boolean type, so a comparison of partial ordering will always yield false.
- the set existence checks of empty and not empty"[6]

Ordering implies a direct path where one can follow the path in any direction to determine what is before and what is after. Changing direction isn't permitted (i.e., taking another path), and traversing multiple paths is another operation (e.g., get number of items after C on path 1 yields 1 (D), and get number of items after C on path 2 yields 3 (E, F, G); the total number of items after C is 4 (result 1 + result 2).

Any comparison operations have to be done on the same path, so an "illegal" comparison across two paths will yield a false. The analyst must be cautioned about this rule, or the tool implementer could make such comparisons result in an error. This rule on paths obviates the need for an "extended boolean" as described in [6].

Ordering is done based upon a common index value. This means that all the members of the ordinal must be of the same type, but not all types can be part of an ordinal. e.g., composites, booleans, ordinals, and enumerations.

8.1.14.1 Attributes

direction:integer Direction of ordering is either ascending or descending. The direction can be dynamically changed in the process models. A loop operation on the ordinal will follow the set direction.

order_specifier:string The order specifier attribute determines the basis of the ordering. The order specifier can be an object attribute, if the ordinal is composed of objects, a meta-attribute, e.g., identifier, name, etc., common ordering based upon data type, i.e., numeric +/- 1, alphabetical for strings, etc., or physical placement. (A TBD symbol or keyword will be defined to indicate placement.) Placement is to be considered ordered by a means external to the data. e.g., an array. A placement ordinal will have means of inserting, moving, and removing ordered elements. Placement ordinals will also have associated marks, so placement can be based on platform considerations. e.g., memory location, timestamp, etc. All ordinals will have the ability to add and remove elements. The order specifier can be dynamically changed in the process models.

8.1.14.2 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.14.3 Operations

Algorithm 8.30 void Ordinal:generateSupportedOperators()

The operations permitted for instance reference data types are

- the comparisons = and != (identical and not identical in value)
- the set existence checks of empty and not empty.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"ordinal", result:"
generate TYOP_A1:create(name:"join", symbol:"+", type:"ordinal", result:"ordinal
generate TYOP_A1:create(name:"equal", symbol:"=", type:"ordinal", result:"boolea
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"ordinal", result:"b
generate TYOP_A1:create(name:"empty", symbol:"empty", type:"ordinal", result:"bo
generate TYOP_A1:create(name:"not empty", symbol:"not empty", type:"ordinal", re
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"numeric", res
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"numeric", res
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"symbolic", re
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"symbolic", re
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"boolean", res
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"boolean", res
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"arbitrary", r
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"arbitrary", r
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"duration", re
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"duration", re
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"time", result
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"time", result
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"instance refe
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"instance refe
generate TYOP_A1:create(name:"insert", symbol:"insert into", type:"function refe
generate TYOP_A1:create(name:"remove", symbol:"remove from", type:"function refe
```

8.1.14.4 Instance State Model

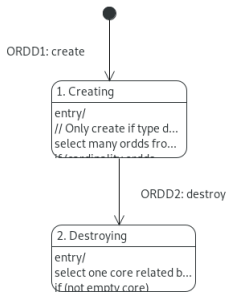

Figure 8.11: Ordinal State Model

Algorithm 8.31 Creating

```
// Only create if type doesn''t exist
select many ordds from instances of ORDD;
if (cardinality ordds == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    type.name = "ordinal";
    core.type = data_t::ORDINAL;
    relate core to self across R403;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of ORDD attempted!");
    generate ORDD2:destroy to self;
end if;
```

Algorithm 8.32 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
   core.destroy();
   unrelate core from self across R403;
   delete object instance core;
end if;
```

8.1.15 Relationship Instance Reference

A relationship instance reference is the type to which all relationship instances must conform. This is a core type that says relationship instances have a common set of operations defined for usage.

8.1.15.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.15.2 Operations

Algorithm 8.33 void Relationship Instance Reference:generateSupportedOperators()

The operations permitted for instance reference data types are

- the comparisons = and != (identical and not identical in value)
- the set existence checks of empty and not empty.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"relationship instan
generate TYOP_A1:create(name:"equal", symbol:"=", type:"relationship instance re
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"relationship instan
generate TYOP_A1:create(name:"empty", symbol:"empty", type:"relationship instanc
generate TYOP_A1:create(name:"not empty", symbol:"not empty", type:"relationship
```

8.1.15.3 Instance State Model

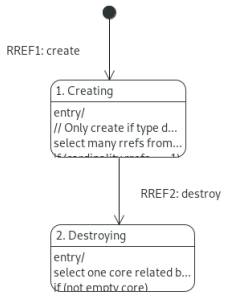

Figure 8.12: Relationship Instance State Model

Algorithm 8.34 Creating

```
// Only create if type doesn''t exist
select many rrefs from instances of RREF;
if (cardinality rrefs == 1)
   create object instance type of TYPE;
   create object instance core of CORE;
   relate type to core across R401;
   type.name = "relationship instance reference";
   core.type = data_t::RELATIONSHIP_INSTANCE_REFERENCE;
   relate core to self across R403;
   self.generateSupportedOperators();
else
   LOG::LogFailure(message:"Multiple creation of type RREF attempted!");
   generate RREF2:destroy to self;
end if;
```

Algorithm 8.35 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.16 Symbolic

"For data elements that have the nature of names, we need to be able to define symbolic data types: data type <data type name> is symbolic length is (from <minimum number of characters> to) <maximum number of characters> (default value is <character string>) The analyst specifies the maximum and minimum number of characters required based on his or her knowledge of the longest and shortest plausible values. Hence: data type gas name is symbolic length is from 2 to 15 default value is Helium

The operations defined for symbolic data types are: - concatenate (represented as +); the result of concatenation is a data element of base type symbolic. - comparison for identical value, represented as = (identical in value) and != (not identical in value). The result of such a comparison yields a data element of base type boolean. - comparison for position in a collating sequence[5], represented as < (before), > (after), <= (before or identical), and >= (identical or after). The result of such a comparison yields a data element of base type boolean.

[5] A collating sequence prescribes the order of all the characters in a specified character set, typically including letters, numbers, and punctuation marks. Collating

sequences are defined in the implementation environment, and may vary from country to country depending on the concept of "alphabetical order" and the repertoire of characters or symbols used in the natural language."[6]

8.1.16.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.16.2 Operations

Algorithm 8.36 void Symbolic:generateSupportedOperators()

"The operations defined for symbolic data types are:

- concatenate (represented as +); the result of concatenation is a data element of base type symbolic.
- comparison for identical value, represented as = (identical in value) and != (not identical in value). The result of such a comparison yields a data element of base type boolean.
- comparison for position in a collating sequence[see definition below], represented as < (before), > (after), <= (before or identical), and >= (identical or after). The result of such a comparison yields a data element of base type boolean.

A collating sequence prescribes the order of all the characters in a specified character set, typically including letters, numbers, and punctuation marks. Collating sequences are defined in the implementation environment, and may vary from country to country depending on the concept of "alphabetical order" and the repertoire of characters or symbols used in the natural language."[6]

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"symbolic", result:
generate TYOP_A1:create(name:"concatenate", symbol:"+", type:"symbolic", result:
generate TYOP_A1:create(name:"equal", symbol:"=", type:"symbolic", result:"boole
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"symbolic", result:"
generate TYOP_A1:create(name:"greater", symbol:">", type:"symbolic", result:"boo
generate TYOP_A1:create(name:"lesser", symbol:"<", type:"symbolic", result:"bool
generate TYOP_A1:create(name:"greater or equal", symbol:">=", type:"symbolic", r
generate TYOP_A1:create(name:"lesser or equal", symbol:"<=", type:"symbolic", re
```

8.1.16.3 Instance State Model

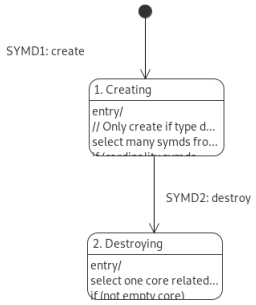

Figure 8.13: Symbolic State Model

Algorithm 8.37 Creating

```
// Only create if type doesn''t exist
select many symds from instances of SYMD;
if (cardinality symds == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    type.name="symbolic";
    core.type = data_t::SYMBOLIC;
    relate core to self across R403;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of SYMD attempted!");
    generate SYMD2:destroy to self;
end if;
```

Algorithm 8.38 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
  core.destroy();
  unrelate core from self across R403;
  delete object instance core;
end if;
```

8.1.17 Time

"To define a data type that represents calendar-clock time, write: data type <data type name> is time range is from <year-mon-day> (<hour:min:sec >) to <year-mon-day> (<hour:min:sec >) precision is <smallest discriminated value> [year | month | day | hour | minute | second | millisec | microsec]

The operations permitted using data types based on time and duration are: time := time ± duration duration := duration ± duration duration := duration * numeric duration := duration / numeric duration := time - time as well as the standard comparisons of < (read as "before"), >, $\leq$, and $\geq$. Each such comparison yields a data element of base type boolean. Comparisons are defined only between elements of the same base type."[6]

8.1.17.1 Relational Attributes

current_state:state<State_Model>

***name:same_as<Base_Attribute>**

8.1.17.2 Operations

Algorithm 8.39 void Time:generateSupportedOperators()

"The operations permitted using data types based on time ... are:

- time := time ± duration as well as the standard comparisons of < (read as "before"), >, $\leq$, and $\geq$. Each such comparison yields a data element of base type boolean. Comparisons are defined only between elements of the same base type."[6]
- time := time ± time

NOTE: For mixed type operations, conversion operators must be supported. The explicit conversion cases are:

- duration := time: time->duration
  NOTE: The architecture should add boundary checks on this conversion, as durations are often expressed in (e.g.,) microseconds. The system needs to be able to support very large numbers if values are more than one hour.

```
generate TYOP_A1:create(name:"assignment", symbol:":=", type:"time", result:"tim
generate TYOP_A1:create(name:"addition", symbol:"+", type:"time", result:"time")
generate TYOP_A1:create(name:"subtraction", symbol:"-", type:"time", result:"tim
generate TYOP_A1:create(name:"equal", symbol:"=", type:"time", result:"boolean")
generate TYOP_A1:create(name:"not equal", symbol:"!=", type:"time", result:"bool
generate TYOP_A1:create(name:"greater", symbol:">", type:"time", result:"boolean
generate TYOP_A1:create(name:"lesser", symbol:"<", type:"time", result:"boolean"
generate TYOP_A1:create(name:"greater or equal", symbol:">=", type:"time", resul
generate TYOP_A1:create(name:"lesser or equal", symbol:"<=", type:"time", result
```

8.1.17.3 Instance State Model

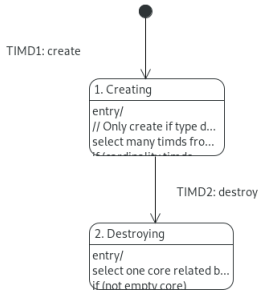

Figure 8.14: Time State Model

Algorithm 8.40 Creating

```
// Only create if type doesn''t exist
select many timds from instances of TIMD;
if (cardinality timds == 1)
    create object instance type of TYPE;
    create object instance core of CORE;
    relate type to core across R401;
    type.name = "time";
    core.type = data_t::TIME;
    relate core to self across R403;
    self.generateSupportedOperators();
else
    LOG::LogFailure(message:"Multiple creation of TIMD attempted!");
    generate TIMD2:destroy to self;
end if;
```

Algorithm 8.41 Destroying

```
select one core related by self->CORE[R403];
if (not empty core)
   core.destroy();
   unrelate core from self across R403;
   delete object instance core;
end if;
```

8.1.18 Type

"A Type is a named finite or infinite set of values."[8]

"RULE: All data elements that appear in the OOA models of a domain must be typed."[6]

8.1.18.1 Attributes

***name:string** Types are uniquely identified by a string designation. The type name is unique within its context.

8.1.18.2 Operations

Algorithm 8.42 void Type:remove()

```
select many operators related by self->OPER[R406];
for each operator in operators
  operator.unassign(type:self.name);
  select any type related by operator->TYPE[R406.''is used in a context establisl
  if (empty type)
    delete object instance operator;
  end if;
end for;
```

8.1.19 Typed Operator

"An Operator is polymorphic in that it may be applied to multiple Types."[8] The Typed Operator is the mapping of a Type to an Operator. The context of the result of the operation is the same as the context of the Typed Operator chosen. This implies precedence must be built into the process model, if automatic type conversion is desired, because the result of e.g., 5 + "5" is different if '+' is a string operator vs a numeric operator. (This also requires that the corresponding conversion operator is defined.)

8.1.19.1 Attributes

definition:boolean "This is a precise specification of the processing required to transform the Operands to produce either a return value or an update result, depending on the Typed Operator specialization."[8]

NOTE: miUML has this typed as "psuedocode", because of not yet having an "Operator definition language". In BridgePoint, it's declared as a derived type, so OAL can be used to define the operator.

Algorithm 8.43 Typed Operator:defined Derived Attribute

```
self.definition=false;
```

8.1.19.2 Relational Attributes

***result_type_name:same_as<Base_Attribute>**

***type_name:same_as<Base_Attribute>**

***operation_name:same_as<Base_Attribute>**

8.1.19.3 Object State Model

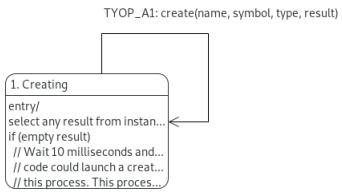

Figure 8.15: Typed Operator State Model

Algorithm 8.44 Creating

```
select any result from instances of TYPE where selected.name == rcvd_evt.result;
if (empty result)
  // Wait 10 milliseconds and retry, since type creation is asynchronous. This
  // code could launch a creation event, but the responsibility lies outside of
  // this process. This process only has to understand the asynchronous nature
  // of type creation and acommodate it.
  create event instance recreate of TYOP_A1:create(name:rcvd_evt.name, symbol:rc
  bridge retry_timer = TIM::timer_start(microseconds:10000, event_inst:recreate)
else
  select any operator from instances of OPER where (selected.name == rcvd_evt.na
  if (empty operator)
    create object instance operator of OPER;
    operator.name = rcvd_evt.name;
    operator.symbol = rcvd_evt.symbol;
  end if;
  select any type from instances of TYPE where selected.name == rcvd_evt.type;
  create object instance typed_op of TYOP;
  relate operator to type across R406 using typed_op;
  relate typed_op to result across R415;
  // For assignment operators, make them the required operator if requestor is
  // a core type (which it probably is).
  if (operator.name == "assignment")
     select one core related by type->CORE[R401];
     if (not empty core)
        relate core to operator across R419;
     end if;
  end if;
end if;
```

8.2 Relationship Descriptions

R401 Data types in OOA are all based on a set of core data types defined by the Shlaer-Mellor method, but the method allows the types to be further constrained to meet the requirements of the domain being modeled.

R402 A domain data type is based on one of the core data types. A domain data type is used to further constrain the core types to meet the requirements of the domain more closely.

R403 The core data type is subtyped into all the supported core types defined by the Shlaer-Mellor method.

R406 "An Operator has no utility unless it is relevant to at least one Type. The same Operator may be used with many Types making it polymorphic.

A Type has no utility without at least one Operator. Since assignment and equality must be defined for all Types, there should at the very least be two Operators defined for any given Type."[6]
A Domain Type can have its own defined operators, or it can just make use of its associated Core Type operators, so "Type" in the second paragraph quoted above should be read as "Core Type" with respect to this metamodel. This makes the relationship conditional on the Operator end.

R412 A domain type is defined and unique within a modeled domain. The modeled domain encapsulates the definition of the domain type, such that the domain type can only be used within the modeled domain.

R415 A typed operator will result in a type that can differ from it's assigned type. A type can be a result type for many typed operators.

R416 Enumeration definitions are typed as an enumeration, but the enumeration type can exist without any enumeration definitions.

R417 An Instance Reference types an Object Instance used as a value. Many Object Instances can be typed as Instance Reference.

R419 A core type always has at least one operator defined for it, but an operator might be defined for a domain type instead of a core type.
The assignment operator would be good to establish this relationship, as all types have an assignment operator. i.e., what good is a datatype that can't be assigned?

R420 An enumeration value is assigned to one enumeration definition, and an enumeration has one or more enumeration values.

R421 A Relationship Instance Reference types a Relationship Instance used as a value. Many Relationship Instances can be typed as a Relationship Instance Reference.

R422 An Accessor Reference types an Accessor Instance used as a value. Many Accessor Instances can be typed as Accessor Reference.

R423 A Keyword types a Keyword Instance used as a value. Many Keyword Instances can be typed as a Keyword.

8.3 Domain Services

8.3.1 Provided Services

Algorithm 8.45 void createCoreTypes()

```
// Each core type is responsible for it''s own creation.
generate BOOL1:create() to BOOL creator;
generate COMP1:create() to COMP creator;
generate DURD1:create() to DURD creator;
generate TIMD1:create() to TIMD creator;
generate NUMI1:create() to NUMI creator;
generate ARID1:create() to ARID creator;
generate ENUM1:create() to ENUM creator;
generate ORDD1:create() to ORDD creator;
generate SYMD1:create() to SYMD creator;
generate IREF1:create() to IREF creator;
generate AREF1:create() to AREF creator;
```

8.3.1.1 Object Model Bridges

Algorithm 8.46 void ()

8.3.1.2 Process Model Bridges

Incoming bridges for creating model elements of the Process Model subsystem as meta-model instances.

Algorithm 8.47 void ()

8.3.1.3 Dynamic Model Bridges

Algorithm 8.48 void ()

8.3.1.4 Data Model Bridges

Algorithm 8.49 void ()

8.4 Domain Datatypes

User defined data types for the operation of the Shlaer-Mellor Metamodel domain.

data_t An enumeration of the core data types defined in the metamodel.

- **ACCESSOR_REFERENCE** A data type that represents a handle to an accessor instance.
- **ARBITRARY** A data type that acts as a unique specifier.
- **BOOLEAN** A data type that can only take one of two values. e.g., true or false
- **COMPOSITE** An unordered grouping of data. e.g., the struct type in C
- **DURATION** A specified period in units of time.
- **ENUMERATION** A data type that represents a finite set of unique values, specified explicitly.
- **INSTANCE_REFERENCE** A data type that represents a handle to a single instantiation of an object.
- **KEYWORD** Data that belongs to the set of special processing directives, that should not be defined as variable names.
- **NUMERIC** A data type used to express some number. i.e., real, integer, ...
- **ORDINAL** A data type used to express order. e.g., an array
- **RELATIONSHIP_INSTANCE_REFERENCE** A data type that represents a handle to a single instantiation of a relationship.
- **SYMBOLIC** A data type that represents a non-empty finite set of symbols (e.g., an alphabet), combined and used to convey meaning. Typically known as a string in various programming languages.
- **TIME** A calendar-clock time data type.

creationResult_t The creation result type is used in conjunction with creating a Shlaer-Mellor domain model to verify it is compatible with the metamodel. The result enumerator directs the verifier toward further action.

- **SUCCESS** Creation of model element satisfies all constraints in the metamodel. No further action recommended for the verifier.
- **DUPLICATE** A duplicate of the specified model element has already been created for this domain. The result is a new copy isn't created, and a log of the failure will be made.

The verifier can ignore this result, if duplication is expected in its operation, or further action can be taken.
If this occurs when creating a domain, then it could be that an old verification attempt is still in existence.

FAILURE This means the creation resulted in a failure with respect to constraints in the metamodel. The result is some elements might have been created in the metamodel, and a log entry of the constraint failure is made. This result is to be expected at certain points of model verification. e.g., add_domain will fail due to the constraint that a domain model must contain at least one object, but you also can't create an object without a domain. In unexpected cases, the log entry should be consulted before proceeding.

state_t An enumeration of the possible purposes for a state in a state model.

CREATION A creation state is a start state that causes instance creation upon entry.

DELETION A deletion state is a state that causes instance deletion upon exit.

MIDDLE A middle state is a state that isn't a start, creation, or deletion state.

START A start state is the residing state upon state machine creation.

transition_t An enumeration of the types of entries that are added to cells of a state transition table (STT). "the STT is a far superior representation for verifying the completeness of and consistency of the transition rules. Filling out the STT requires you to consider the effect of every event-state combination."[2]

CANNOT_HAPPEN The transition results in an "can't happen"."If an event cannot happen when the instance is in a particular state, record the fact by entering can't happen in the appropriate cell [of the state transition table]."[2]

EVENT_IGNORED The transition results in an "event ignored"."If an object refuses to respond to a particular event when it is in a certain state, enter event ignored in the appropriate cell [of the state transition table]."[2]

NEW_STATE The transition results in a new state. "The cell [of the state transition table] is filled in with the name of the new state that results when an instance in the state specified by the row receives the event specified by the column."[2]

Part II

Templates

Chapter 9

Subsystem Template

<description of subsystem>

<picture of object model for subsystem>NOTE: Scale to 70% page width.

Figure 9.1: <name> Subsystem Diagram

9.1 Object and Attribute Descriptions

<see chapter 10>

9.2 Relationship Descriptions

<see chapter 11>

Chapter 10

Object and Attribute List Template[1]

10.0.1 <Object Name>

10.0.1.1 Attributes

[*][2]<name>:<type> <description>

[Derived attributes]

Algorithm 10.1 <return type> <object>:<operation name>(<parameter name>:<type>[, . . .])

< a c t i o n l a n g u a g e >

10.0.1.2 Relational Attributes

[*]<name>:<relationship number> [description][3]

---

[1]Imported objects (from another subsystem) aren't included.

[2]Optional asterisk to indicate identifier. Additional identifiers should have a superscript number on the right side of the asterix. There shouldn't be a space between asterix and name.

[3]Descriptions for relational attributes are optional and assumed to be the same as the actual attribute. They can be repeated here, or used to describe unique characteristics of the relational attribute in this object.

10.0.1.3 Operations

Algorithm 10.2 <return type> <object>:<operation name>(<parameter name>:<type>[, . . .])

[Description]

```
<action language>
```

10.0.1.4 Instance State Model

<state model graphics>NOTE: Scale graphics to 30%.

Figure 10.1: <object> State Model

Algorithm 10.3 <state name>

[Description]

```
<action language>
```

<state name>

Chapter 11

Relationship Template

R<number> <description>

Bibliography

[1] Object-Oriented Systems Analysis: Modeling the World in Data. Sally Shlaer and Stephen J. Mellor. 1988. Prentice-Hall, Inc. ISBN-13: 978-0136290230

[2] Object Lifecycles: Modeling the World in States. Sally Shlaer and Stephen J. Mellor. 1992. Prentice-Hall, Inc. ISBN-13: 978-0136299400

[3] Executable UML: A Foundation for Model-driven Architecture. Stephen J. Mellor and Marc J. Balcer. 2002. Addison-Wesley Professional. ISBN-13: 978-0201748048.

[4] How to Build Shlaer-Mellor Object Models. Leon Starr. 1996.

[5] OOA '96. Sally Shlaer and Neil Lang. 1996. Prentice-Hall, Inc. ISBN:0-13-207663-2

[6] Data Types in OOA. Sally Shlaer and Stephen J. Mellor.

[7] Bridges and Wormholes. Sally Shlaer and Stephen J. Mellor.

[8] miUML open source executable uml. Leon Starr and Andrew Mangogna.