miUML Metamodel Class Descriptions
State Subsystem

Leon Starr

Sunday, January 22, 2012

mint.miUMLmeta.td.4

Version 3.2.4

Table of Contents

Introduction

This metamodel captures the semantics of Executable UML as described in the book Executable UML: A Foundation for Model Driven Architectures. This document explains how the Class Model Metamodel supports these rules. Justification and explanation of Executable UML rules themselves are beyond the scope of this metamodel.

These descriptions are part of the Executable UML State Model Metamodel Class Model. Some of the Classes in this model are well known entities defined in [MB]. Many other classes have been added to more precisely define and constrain the Executable UML concepts and rules. In a few cases, and always noted, [OOA96] is also taken into consideration as it is a fundamental predecessor to [MB]. In the area of data type theory, [DATE1] leaned upon since it offers one consistent with the relational model of data upon which Executable UML is firmly rooted.

Key to Styles and Conventions

COLOR is used to convey information in this document. If you have the ability to print or display this document in color, it is highly recommended. Here are a few notes on the font, color and other styles used in class model descriptions.

Naming classes, types and subsystems

Modeled elements such as classes, types and subsystems are written with initial caps. The text Air Traffic Controller, for example, would probably be a class. If you see the text Application Domain, you know it is a modeled element, most likely a domain.

Naming attributes

Attributes are named with an initial cap followed by lower case, Time logged in, for example.

Instance values (M0)

Instance data, such as attribute values, are either in quote marks as in "Gwen" and "Owen" or represented in the following color/font: Gwen and Owen.

Model examples (M1)

Model examples, such as class and attribute names appear in the following color/font: Air Traffic Controller.Maximum session time

Whitespace

When programming, the squashedNamingStyle is optimal for easy typing and is adequately readable for short names. That's fine for programming, but analysis is all about easy readability and descriptive, sometimes verbose names. Programming style is retained for method names, but everywhere else whitespace or, when color/fonts are not an option, underscores are employed. For example: Air Traffic Controller, Time logged in or On_Duty_Controller.Time_logged_in.

hardcoreJavaProgrammersmayDisputethisPointandArguethatWhiteSpacesareEntirelyUnnecessaryforImprovingReadabilitybutIDisagree.

Attribute heading color

Attribute headings are colored according to the role the attribute plays in its class. Referential attributes are brown, **Logged in controller**, for example. Naming attributes are blue, **Number**, for example. Finally, descriptive attributes are red, **Time logged in**, for example. Derived attributes are purple **\Volume**, for example. (One of the nice things about color is that it helps eliminate the need for underscores).

Referential renaming

A referential attribute will often have a name that reflects its role rather than the name of the target attribute. Station.Logged in controller may refer to On Duty Controller.ID, for example. Any renaming will appear in the referential attribute's description.

MDA (Model Driven Architecture) Layers

The MDA defined layers are referenced from time to time. It is sometimes useful, especially in a metamodel, to distinguish the various meta layer names: M0, M1, M2 and M3. These are M0: data values (N3295Q, 27L, 490.7 Liters, ...) M1: domain specific model elements (class Aircraft, attribute Altitude, relationships R2 - is piloted by), M2: metamodel elements to define miUML (class: Class, class: Attribute, class: Relationship, attribute:Rnum...) and M3: meta-metamodel elements to define (ambitiously) any modeling language (class:Model Node, class:Structural Connection). For our purposes, we will seldom refer to the M3 layer and M2 will be relevant only if the subject matter at hand is a metamodel.

M2 language can be tricky as we sometimes employ metamodel terms to describe the model itself. This can be confusing, so when we refer to a modeled component in the miUML metamodel it will appear in all caps (as would be the case with any modeled subject matter) whereas lowercase is used when the term is not explicitly referring to a metamodel component. For example: 'The Attribute class will be instantiated for each attribute.' Better yet: 'Subclass is a subclass of Generalization Role.' and, just to mess with your head: 'The Identifier subclass Modeled Identifier has a single identifier'. Welcome to the special slice of hell known as metamodeling.

Keep in mind that these layers may be interpreted relative to the subject matter at hand. If you are modeling an air traffic control system, 490.7 Liters is likely data in the M0 layer with the class Aircraft at the M1 layer. 'Class' would be at M2 - the metamodel layer.

But if the subject matter is Executable UML, as is the case in the miUML metamodels, both the class Aircraft and 490.7 Liters could be compressed into the M0 layer. (Aircraft is data populating the Class and Attribute subsystem and 490.7 Liters populated into the Population subsystem). At the M1 layer we would possibly have Class and Value. And, since the metamodel will populate itself, as we bootstrap into code generation, the M2 layer is also 'Class' and 'Value'.

In other words, we can create a data base schema from the miUML metamodel and then insert the metamodel itself into that schema[1]. So if we define a table 'Class' based on our metamodel in the data base, we could then insert metamodel classes 'Class', 'Attribute', etc. into that table. Thus we see 'Class' both at the M0 (inserted data) and M1 (modeled) and M2 (metamodel) layers. Fun!

[1] The metamodel must be able to eat itself.

Local Data Types

The following common data types (not domain specific) are used in this subsystem. In all cases, type specific operators [DATE1] for get and set are assumed to be available. Other suitable type specific operators are listed. Possible representations (implementations) are suggested.

Name

This is text long enough to be readable and descriptive but of limited length to be suitable for naming things. A handful of words is okay, a full paragraph is too much. Since the exact value may vary to suit the application, it is represented symbolically as NAME_LENGTH.

Possible representations: String, Text, Varchar, ... limited to length NAME_LENGTH.

First character must be non-whitespace, followed by a mixture of whitespace and non-whitespace, terminated by non-whitespace. A single non-whitespace character is also acceptable.

Example operators: compare, ...

Nominal

This is a whole number used purely for naming with no ordinal or computational properties.

Identifiers used only for referential constraints may be stripped out during implementation (assuming the constraints are still enforced). During simulation and debugging, however, these can be nice to have around for easy interpretation of runtime data sets without the need for elaborate debug environments. File-6:Drawer-2:Cabinet-Accounting is easier for a human to read than a bunch of pointer addresses.

Possible representations: int, string, number, ...

Example operators: get next value, get last assigned value, is value assigned, ...

Description

This is an arbitrarily long amount of text. The only length limit is determined by the platform. In fact, a zero-length description is acceptable. Therefore it should never be used as part of an identifier constraint.

Possible representations: String, Text, Varchar, ... no specific length limit

Example operators: is empty

References

[MB] Executable UML: A Foundation for Model-Driven Architecture, Stephen J. Mellor, Marc J. Balcer, Addison-Wesley, 2002, ISBN 0-201-74804-5

[OOA96] Shlaer-Mellor Method: The OOA96 Report, Sally Shlaer, Neil Lang, Project Technology, Inc, 1996

[DATE1] An Introduction to Database Systems, 8th Edition, C.J. Date, Addison-Wesley, 2004, ISBN 0-321-19784-4

[LSART] How to Build Articulate UML Class Models, Leon Starr, Model Integration, LLC, Google Knol, http://knol.google.com/k/how-to-build-articulate-uml-class-models

[SM1] Object-Oriented Systems Analysis, Modeling the World in Data, Sally Shlaer, Stephen J. Mellor, Yourdon Press, 1988, ISBN 0-13-629023-X

[HTBCM] Executable UML: How to Build Class Models, Leon Starr, Prentice-Hall, 2002, ISBN 0-13-067479-6

[LSSYNC] Time and Synchronization in Executable UML, Leon Starr, Model Integration LLC, Google Knol, http://knol.google.com/k/time-and-synchronization-in-executable-uml

Assigner ASGR

'An *assigner* is a state machine that serves as a single point of control for creating links on competitive associations.' [MB]

A detailed description of where Assigners are required, the problem they solve, and how to build them is available in [MB] and [OOA96]. Briefly though, on an Association where the Instances on each side are trying to link to each other, a kind of moderator may be required to ensure that multiple Instances on one side of the Association don't all try to grab the same Instance on the other side when only one (on one or both sides) may be linked at a time. The moderation task can't be delegated to a Lifecycle State Model on either side of the Association due to competition. In the naturally concurrent world of miUML [LSSYNC] we need to be explicit when platform-independent sequencing is required.

An Assigner solves the problem by creating a single, platform independent, point of control where the competition can be resolved deterministically. An Assigner is specified as a State Model with a single token (in the case of a Single Assigner) which steps through the process of verifying the existence of linkable Instances or waiting for an Instance to become available, choosing the Instances to link when sufficient Instances are available using relevant selection criteria, locking down the Instances to be linked to avoid a race condition (using status attributes) and then creating the Link in a concurrent-safe manner.

Whereas [MB] prescribes the use of a 'class-based state machine' for an Assigner to support UML compliance and easy implementation in existing UML tools, miUML parts ways and returns to the founding theory where the assigner state machine is bound directly to an Association.

'Associating assigners with (Associations) has two very desirable consequences. First, the analyst is now free to choose how to formalize the competitive (Association) and is no longer forced to do so with an (Association Class). Secondly, all (Classes) have, at most, one possible state model: a lifecycle model.' [OOA96]

While a class-based assigner idea works, in practice it is misappropriated to support all manner of class-based activity that is properly formalized with specification Classes and single Instance Classes. In fact, the concept of an Assigner is quickly forgotten and C++ style 'static class' behavior begins sprouting all over the class model. miUML is dedicated to providing a requirements analysis facility rather than an object-oriented code browsing environment of which there are already plenty.

The idea of a State Model built directly on an Association to manage Link competition avoids any confusion about the purpose of such a State Model. To map to standard UML, generate an extra UML singleton class with a statechart then tag that class as an {assigner}. In this way, miUML conceptual integrity is preserved and OMG UML standard conformance is possible.

Attributes

Rnum

Association.Rnum — An Association where the Instances on each side need to be linked together with competition resolved.

Domain

Association.Domain

Identifiers

1> Rnum + Domain

An Association may only have one Assigner.

Relationships

R501
Assigner RESOLVES COMPETITION ON *exactly one* Association
Association COMPETITION IS RESOLVED BY *zero or one* Assigner

Only those Associations where Instances on each side are competing to link to the other side require an Assigner. Typically, most Associations are non-competitive and can be easily managed with Class Methods or State Procedures on one or both sides.

On a given Association, there either is or isn't an Assigner defined.

R514
Assigner IS A Single or Multiple Assigner

A Single Assigner manages two sets of Instances, one on each side of an Association and executes with a single token. A Multiple Assigner relies on a Loop to subdivide the Instances on one side or the other into individually manages subsets of Instances and has a token for each subset.

Creation Event CEVT

An Event that causes the creation of a new Instance of a Class is a *creation event*. This type of Event may only appear on Lifecycle State Models and is never polymorphic.

(See also figure at Deletion Pseudo State description)

When a Creation Event occurs, it causes a new Instance of its Class to be created in the designated State (never a Deletion Pseudo State). Any parameter values delivered with that Event are provided as input to the new Destination State's Procedure executing on the new Instance.

The UML creation pseudo-state is a notational diagram artifact and is irrelevant to the miUML semantics. Nonetheless, a Creation Event may rendered on a UML statechart with the initial pseudo-state symbol with an arrow pointing to the designated State.

Why must a Creation Event always monomorphic? Consider the bank account example. You might want to send a close signal to an instance of Account without knowing or caring whether it is checking or savings, even though the resultant behavior might be handled in only one subclass. But you wouldn't ever try to create an instance of Account without knowing what type it should be. (Otherwise you would create only a portion of the object, which is illegal in miUML).

Attributes

ID

Local Effective Event.ID

Class

Lifecycle.Class, Local Effective Event.State model and State.State model — A Creation Event is constrained to create new Instances only on the same LIfecycle where it is defined and only in a State also belonging to the same State Model.

Domain

Lifecycle.Domain, Local Effective Event.Domain and State.Domain — This is part of the constraint on Class above.

To state

State.Name — The newly created Instance is placed in this State.

Identifiers

1> ID + Class + Domain

An Event has a unique number on its State Model, which, in this case must be a Lifecycle.

2> To state + Class + Domain

Only one Creation Event may lead into a State.

Relationships

R519

Creation Event TRIGGERS CREATION OF NEW INSTANCE ON *exactly one* Lifecycle
Lifecycle SPECIFIES NEW INSTANCE CREATION WITH *zero, one or many* Creation Event

A Lifecycle specifies one or more Creation Events only if it permits the creation of new Instances during runtime.

There may be multiple Creation Events to accommodate alternate creation Procedures. For example, both a Create new and a Create by copy Creation Event may be specified on the same Lifecycle. Each Creation Event would be associated with a separate Initial Transition and destination State in this case.

R508

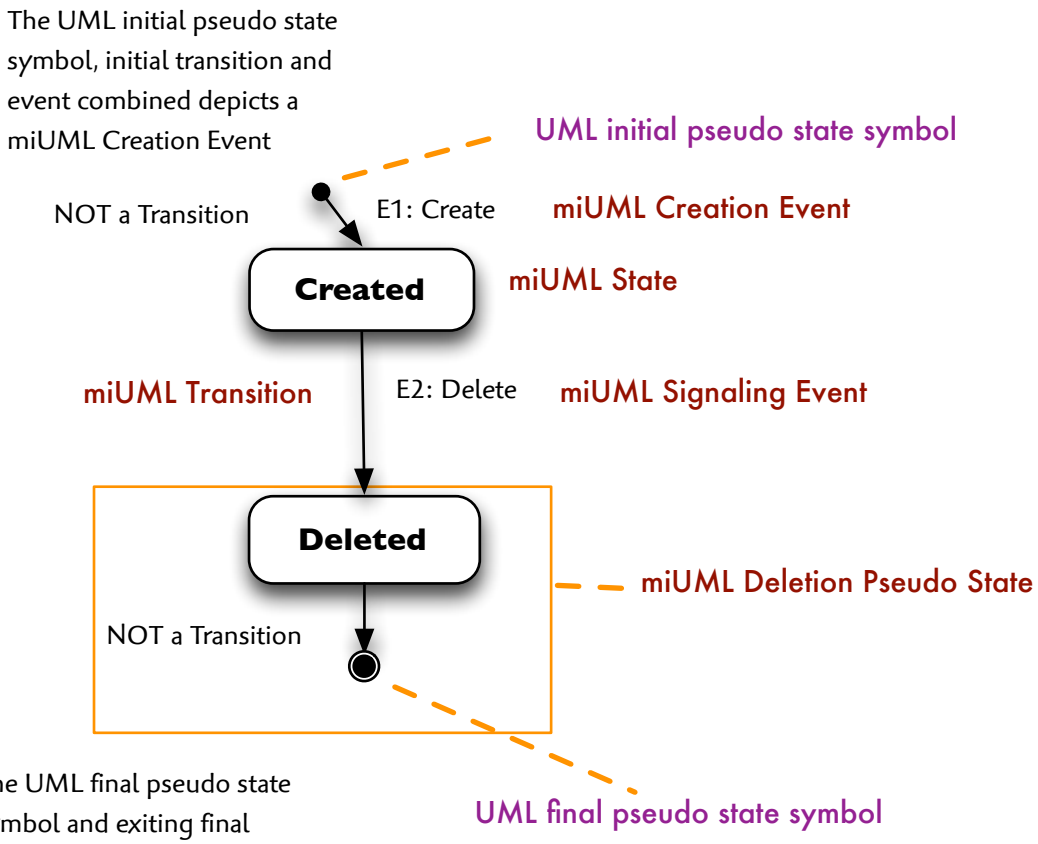
Creation Event CREATES INSTANCE IN *exactly one* State
State IS WHERE A NEW INSTANCE MAY BE CREATED BY *zero or one* Creation Event

To avoid ambiguity, a Creation Event must specify the target State where the new Instance will be created.

A State may or may not be a location where new Instances can be created. In fact, there is typically only one such State on a Lifecycle State Model where runtime creation of Instances is specified. In fact, there must be one such distinct State per Creation Event on any Lifecycle State Model.

Deletion Pseudo State DSTATE

An Object on a Lifecycle State Model may be deleted during runtime. To specify this behavior, a *deletion pseudo state* is placed at the end of a Transition from some State.



Attributes

Name

Destination.Name — This name should either be Deleted, indicating the condition of non-existence or it may be named according to the Procedure that cleans up, such as Cleaning Up. Either is fine as this is not a real State.

Class

Destination.State model and Lifecycle.Class — Since only an Object may be deleted during run-time, a Deletion Pseudo State can only be specified as part of a Lifecycle State Model (not an Assigner).

Domain

Destination.Domain and Lifecycle.Domain

Identifiers

1> Name + Class + Domain

Each Destination is uniquely named within a State Model.

Relationships

R513

Deletion Pseudo State DELETES INSTANCES IN *exactly one* Lifecycle

Lifecycle INSTANCES ARE DELETED UPON ENTRY TO *zero, one or many* Deletion Pseudo State

A Deletion Pseudo State may exist only in a Lifecycle State Model.

A Lifecycle will have one or more Deletion Pseudo States if its Objects will be deleted during runtime. Multiple are possible to accommodate more than one deletion Procedure. The final State of an Object may require different cleanup Actions prior to deletion, for example.

A Lifecycle without any Deletion Pseudo States would represent Objects that always exist, at least during runtime.

Destination DEST

From the perspective of a Transition (State or Initial) a *destination* is a place to go where a Procedure is executed. There are only two such possibilities: a State or a Deletion Pseudo State. Thus, there is nothing special about a 'deletion transition' other than its Destination.

Attributes

Name

This is the union of State and Deletion Pseudo State names. See each subclass for a full description of how these names might be formed.

Type: Name

State model

Union of Deletion Pseudo State.Class and State.State model. So it is either a Class name or an rnum.

Domain

Union of Deletion Pseudo State.Domain and State Model.Domain

Identifiers

1> Name + State model + Domain

Each Destination on a State Model is named uniquely.

Relationships

R510
Destination IS A Deletion Pseudo State or State

A Transition may lead to either a State or a Deletion Pseudo State. A State is real in the sense that it represents a period of time where something is happening. A Deletion Pseudo State is really just the home of a Procedure that performs any necessary clean-up before an Object is deleted. (The Procedure need not specify the actual deletion action as this is assumed).

Effective Signaling Event EFF_SEVT

As opposed to a Creation Event, an Event used for signaling is addressed to an existing Object or Assigner State Model. The signal / creation nature of an Event is established in its Event Specification. An Event used for signaling may be polymorphic, which means that a signaling Event may be delegated. If an Event is not a Creation Event and it is handled, rather than delegated, in its State Model, it is an *effective signaling event*. As such, it requires an Event Response on every State of that State Model.

In casual use, the term 'event' generally refers to this particular classification.

Attributes

ID

Union of the Inherited Effective Event.ID and Local Effective Signaling Event.ID value domains. (Note that this union excludes any Created Event.ID's).

State model

Union of the Inherited Effective Event.Class and Local Effective Signaling Event.State model value domains.

Domain

Union of the Inherited Effective Event.Domain and Local Effective Signaling Event.Domain value domains.

Identifiers

1> ID + State model + Domain

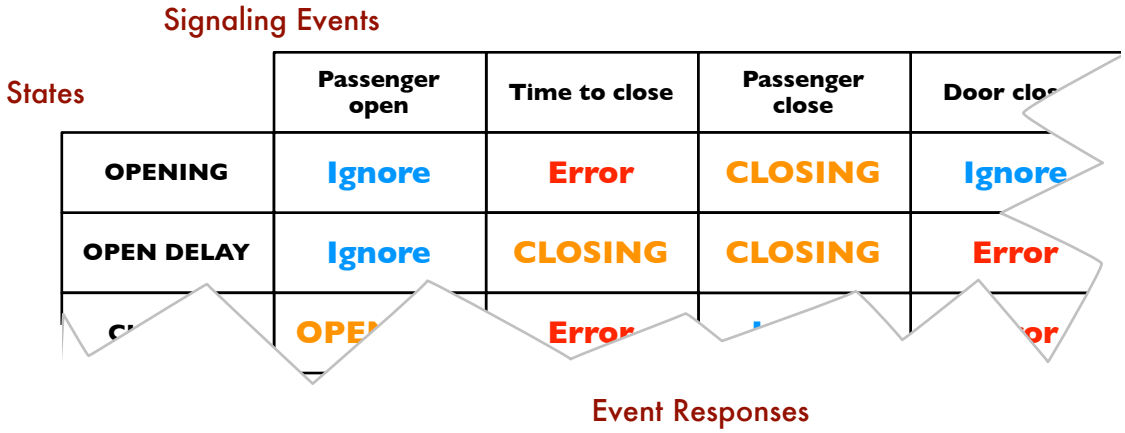
Relationships

(See Polymorphism Subsystem)

Event Response ERESP

Each State must provide a response for each Signaling Event defined on a State Model. This correspondence is commonly represented with a state table like the following:

Moore State Table (partial)



An *event response* must be defined for each cell Event-State intersection in the state table. [LSSYNC] Three standard miUML responses are possible: Transition, Ignore and Can't Happen. The second two options are grouped together as Non-Transition Responses since they don't require any special data or relationships.

Note that it is possible to define a Signaling Event which is not associated with any Transition. It may, for example, be ignored in all States. There would be no utility to such a non-transitioning Event in a delivered software product. But during the model editing and diagnostic phases, it might be useful to 'disable' an Event by setting all responses to Ignore. Alternatively, an Event marked Can't happen in all States could be specified for the diagnostic purpose of making it possible to easily crash a State Model. Neither use is necessarily recommended, but it is worthwhile to point out the possibility.

Attributes

State

State.Name — The name of the State where the Event is handled.

Event

Signaling Event.ID — The Event to be handled.

State model

State.State model and Signaling Event.State model — Both the Event and State must be defined in the same State Model. The {x} tag indicates a cross-product constraint - see R505.

Domain

State.Domain and Signaling Event.Domain — Same constraint as State model above. The {x} tag indicates a cross-product constraint - see R505.

Identifiers

1> State + Event + State model + Domain

Standard many to many formulation of an identifier in an association class with the State model and Domain attributes each merged.

Relationships

R505

Cross Product

State SPECIFIES RESPONSE TO *zero, one or many* Signaling Event
Signaling Event IS RESPONDED TO IN *one or many* State

Cross Product on subsets identified by: **State Model + Domain**

In the simplest possible state table we would have one State and no Signaling Events. In that case, there would be no Event Responses necessary. If we add a Signaling Event, a column is added to the table and one Event Response is required. As we add more States and Signaling Events, more Event Responses are required, one for each cell added to the state table.

Note that Creation Events are excluded as the Object does not exist and is therefore not in any State when they occur.

This is a cross-product association so each Signaling Event must have an Event Response defined in every State within the same State Model.

R506

Event Response IS A Transition or Non-Transition Response

Executable UML [MB] specifies three possible responses to an Event. These are:

1) Take a Transition

2) Ignore the Event — remain in the current State

3) Can't Happen — raise an exception and defer to the MEE (model execution environment)

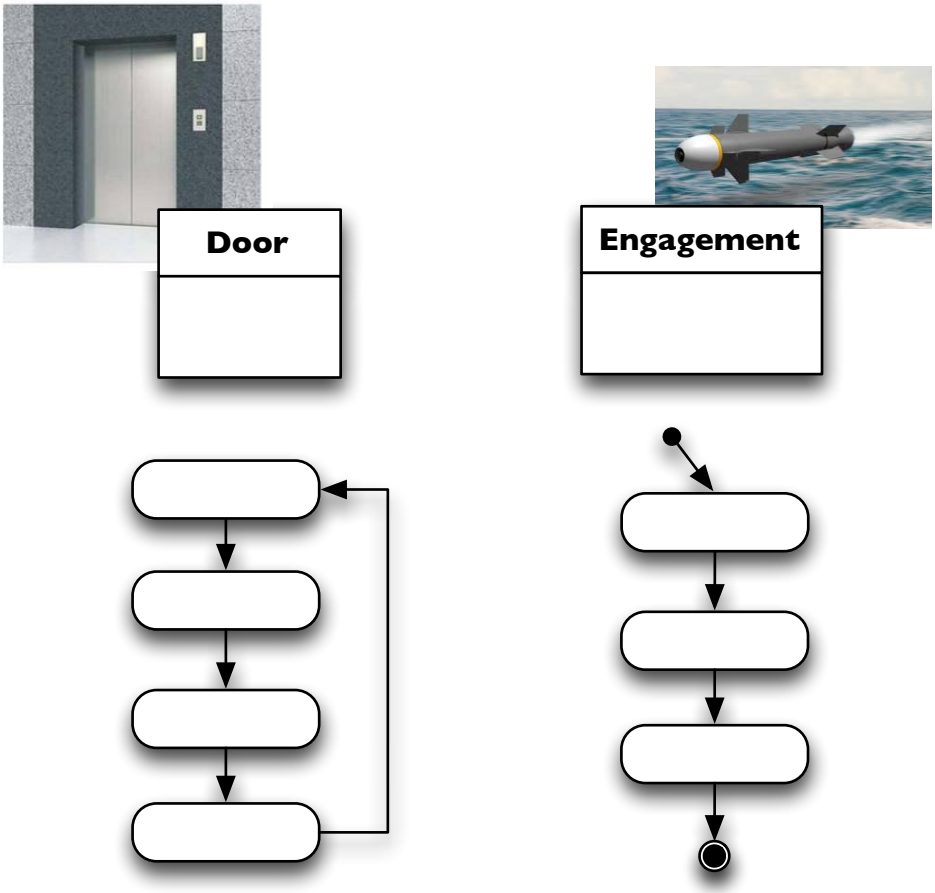
In all cases, the Event occurrence is discarded.

Since the Non-Transition responses (2 and 3) require no further relationships, they are lumped together as a single subclass.

Lifecycle LCYC

"Each instance of a Class generally has a lifetime. Objects are created, they progress through certain stages, and then they go away. Their behavior is a progression through various stages over time. These changes over time are known as the *lifecycle* of the object." [MB]

Since a Class is the definition of a set of things with the same behavior, it should be possible to specify a model of that behavior that applies equally to each and every Object member of that Class. A Lifecycle is just such a model.



Lifecycle State Models

A classic Lifecycle example is that of an elevator Door with a circular pattern of stages such as Open, Closing, Closed, Locked, Opening. A non-physical Class such as missile Engagement might involve a linear pattern of stages of target acquisition, tracking, shooting and either success or failure. In each of these cases it may be necessary to create the relevant Object at the beginning of its Lifecycle and then delete it at the end. Persistent Objects may be defined as well which permanently exist without the need for runtime creation and deletion.

Attributes

Class

Class.Name and State Model.Name — The name of the Class who's behavior is described by this Lifecycle.

Domain

Class.Domain

Identifiers

1> Class + Domain

Since a Lifecycle defines the behavior of exactly one Class, the Class's identifier also serves as its Lifecycle identifier.

Relationships

R500

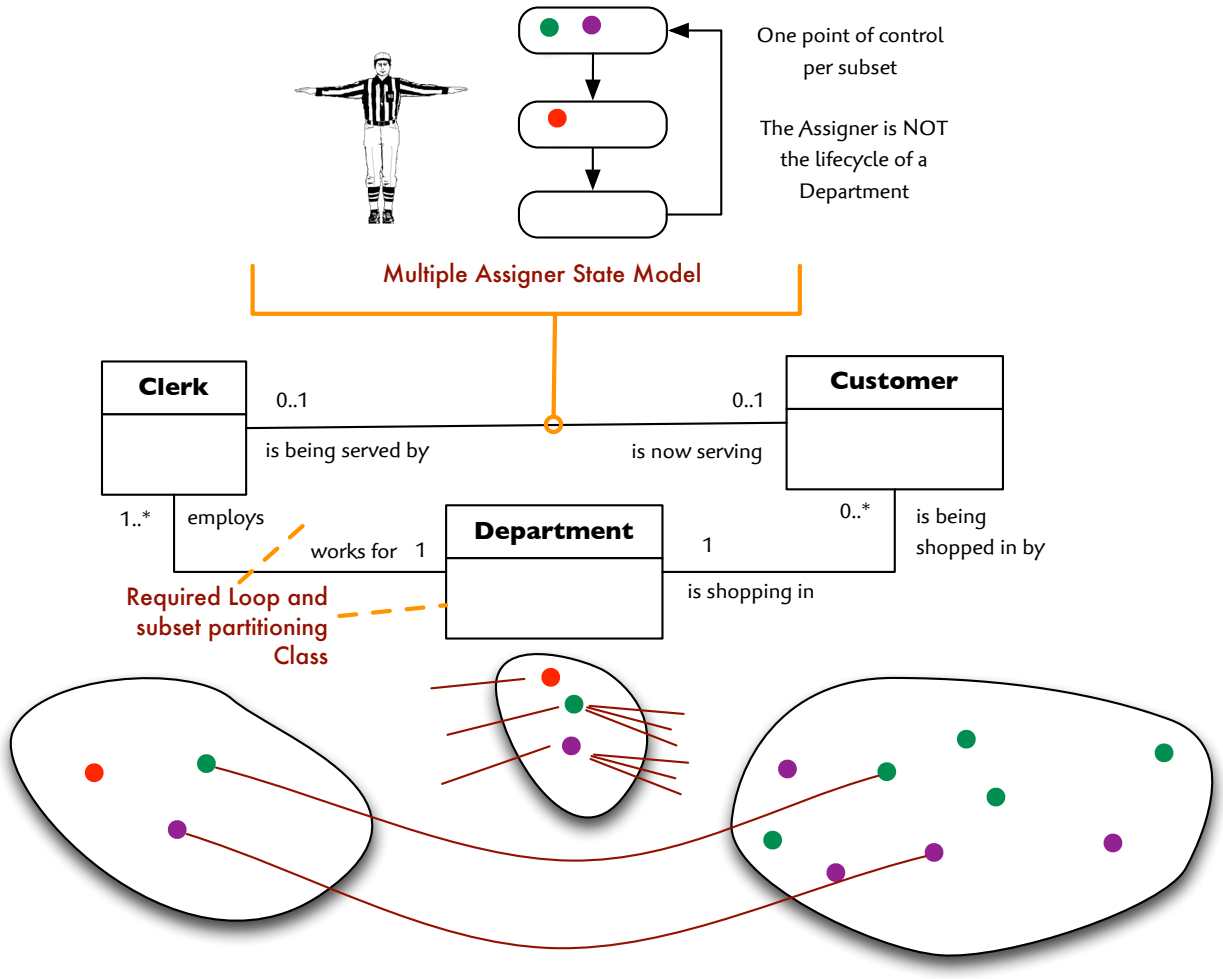
Lifecycle SPECIFIES PATTERN OF BEHAVIOR FOR *exactly one* Class
Class BEHAVIOR PATTERN IS MODELED BY *zero or one* Lifecycle

By definition, a Lifecycle defines the common behavior on all Objects belonging to a specific Class. A Class, may exhibit such limited behavior (a single state, for example) that it does not warrant the specification of a Lifecycle. So while we can imagine a Lifecycle on every Class, in practice we build them only for Classes with interesting behavior.

Since a Class is defined as a set of things with the same behavior, it stands to reason that a Class can specify at most one Lifecycle.

Multiple Assigner MASGR

A *multiple assigner* will link any Instance within a subset on one side of an Association to any Instance in a corresponding subset on the other side A partitioning Class, other than the one on each side of the Association, but all belonging to the same constrained Loop, establishes the subsets with its Instances. Thus each Instance of the partitioning Class will be related to a subset of Instances on either side of the Association through a constrained Loop. Multiple instances of the Assigner (assigner tokens) will each manage competition within a single subset.



See [MB] and [OOA96] for a complete description and an example of a Multiple Assigner. Briefly though, consider the Single Assigner example of Clerks and Customers. Now add a third partitioning Class: Department. Each Department Instance will then establish a subset of Clerks and Customers that may be linked together. A separate instance of the Multiple Assigner will manage the linking within each Department.

Attributes

Rnum

Assigner.Rnum

Domain

Assigner.Domain, Loop.Domain and Class.Domain

Loop - constrained

Loop.Element — This must be a Loop to which the partitioning Class and the Class(es) on each side of the Association managed by the Multiple Assigner belong.

Partitioning class - constrained

Class.Name — This must be a Class within the Loop and not on either side of the Association managed by the Multiple Assigner.

Identifiers

1> Rnum + Domain

Same as superclass.

Relationships

R517

Loop ESTABLISHES CONSTRAINT FOR *zero, one or many* Multiple Assigner
Multiple Assigner MANAGES WITHIN CONSTRAINT SET BY *exactly one* Loop

A Multiple Assigner may not be specified in absence of a constrained Loop. It is this Loop which establishes concurrently manageable subsets of Instances.

A Loop may or may not intersect a competitive Association. It is also possible, though probably rare, for a Loop to intersect multiple such Associations.

It is true that R518 already makes it possible to find this Loop. But R517 ensures that a Multiple Assigner cannot be created outside of a constrained Loop.

R518

Class PARTITIONS CONCURRENTLY LINKABLE INSTANCE SUBSETS FOR *zero, one or many* Multiple Assigner

Multiple Assigner CONCURRENTLY MANAGES INSTANCE LINKAGE SUBSET PARTITIONED BY *exactly one* Class

A Multiple Assigner must be instantiated for each subset of Instances that can be managed concurrently. Each subset is established by an Instance of a partitioning Class in the same constrained Loop as the Multiple Assigner's Association. This Class may not be on either of the managed Association.

A given Class may or may not take on the partitioning role for a Multiple Assigner. Since a Class may be part of more than one constrained Loop, it is possible (though highly unlikely) for it to create partitions for more than one Multiple Assigner.

Non Transition Response NTRESP

There may be no Transition desired or possible when an Event occurs in a particular State.

In the first case, the Event is simply ignored and the Instance remains in the current State. For example, an elevator door in the Locked state (moving between floors) should ignore the Open requested Event (open button pushed). This behavior is every bit as normal and anticipated as a Transition Response.

In the second case, an Event may occur in a State that does not expect the Event to ever happen. As far as the State Model is concerned, the Event should not and cannot happen in a particular State. So if the Event does, it means that there is a bug that must be handled outside the scope of the State Model's Domain. An exception is raised in the MEE (Model Execution Environment) which may take whatever actions are appropriate for its platform and overall product requirements. This can range from a complete system halt, hard or soft system reset, Domain reset, simple logging, memory buffer swap, and so on. The point is, something has seriously gone wrong!

Attributes

State

Event Response.State

Event

Event Response.Event

State model

Event Response.State model

Domain

Event Response.Domain

Behavior

One of the two possible non-transition responses: Ignore or Can't Happen

Type: [IGN | CH]

Reason

The reason why this response is Ignore vs. Can't Happen. There should always be a good reason. By specifying a Reason, the modeler assures the reader that he or she has thought this

through and not just provided a default value. (A Transition drawn on a statechart, on, the other hand, is generally assumed to be intentional).

Type: description

Identifiers

1> State + Event + State model + Domain

Same as superclass.

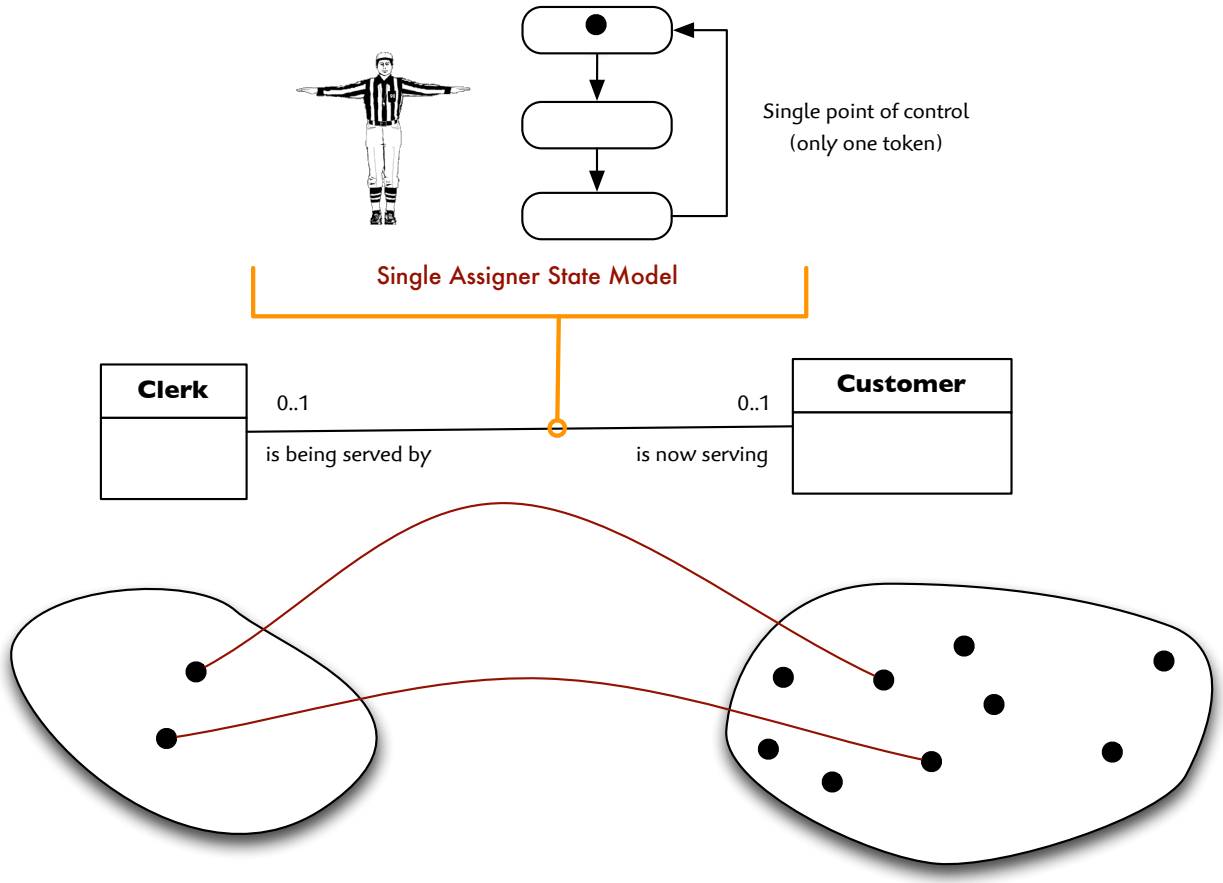
Relationships

None.

Single Assigner SASGR

A *single assigner* will link any Instance on one side of an Association to any Instance on the other side, selection algorithm permitting.

See [MB] and [OOA96] for an example and detailed explanation. Briefly though, consider the example of the two Classes: Clerk and Customer. Only one Clerk may be assigned to a Customer at a time and vice versa. When a Clerk becomes available, the miUML concurrency rules [LSSYNC] would otherwise permit each waiting Customer to simultaneously grab that same Clerk, violating the one-to-one Association multiplicity constraint. The Single Assigner specifies a single, platform independent, point of control to resolve the conflict. The Assigner State Model and its Procedures will ensure that only one Customer and Clerk are linked at a time. Just as importantly, the Assigner assures that no Clerk or Customer waits when a Link can be made.



The 'single' in Single Assigner means that any Customer may potentially link with any Clerk provided both are available and the selection algorithm doesn't exclude either Instance for some reason. So, there is only a 'single' pool of linkable Instances and only one token required on the Assigner State Model.

Attributes

Rnum

Assigner.Rnum

Domain

Assigner.Domain

Identifiers

1> Rnum + Domain

Same as superclass.

Relationships

None.

State S

'A *state* represents the condition of an object in which a defined set of rules, policies, regulations, and physical laws applies.' [MB]

In other words, we can think of a State as representing some period of time during which an Object is behaving in a particular way.

Attributes

Name

A descriptive name characterizing the overall behavior occurring at the time. An elevator door, for example, might have states named Open, Closed, Closing, Opening, Locked, etc.

The name should NOT describe the Actions triggered in the associated Procedure. For example, Closing means, the door is physically closing. It represents the period of time while this is happening. In incorrect name for such a State like Outputting Closing Command refers to the Action that initiates the behavior, but this takes only a fraction of the time experienced during the State. In other words, don't confuse a State with an Action.

Destination.Name

Snum

From [MB] we have: 'States are also numbered. The number of a state is arbitrary and does not imply any sequencing to the states.' These are analogous to class and relationship numbers.

Type: nominal

State model

State Model.Name and Destination.State model — The State is on a particular State Model which also defines its superclass Destination.

Domain

State Model.Domain and Destination.Domain — These match because a State is a Destination.

Identifiers

1> Name + State model + Domain

Each State on a State Model is named uniquely.

2> Snum + State model + Domain

Relationships

R503
State PARTITIONS TIME PERIOD ON *exactly one* State Model
State Model SPECIFIES PERIOD OF TIME AS *one or many* State

A State Model consisting of a single State represents the total duration of time experienced by a dynamic entity (Object or Assigner). If there were no States, no duration would be represented and the dynamic entity could never exist. So, polymorphism aside, a State Model should require at least one State.

Now consider polymorphism in the context of the checking account example. A superclass Account class exists which does not specify any behavior, but does define a set of Polymorphic Event Specifications. Corresponding Events will be delegated to a deeper level of specialization so it would seem that Account has no need for any Lifecycle or States. But at the most general level, an Account simply 'Exists'. So, even in this case, there is a State, Exists, albeit a boring one.

So, in the case of polymorphism, at minimum, a single comprehensive State must be defined in the most general superclass where polymorphic events are defined even if no Effective Events are handled in that Lifecycle. States within Subclass Lifecycle Models would then constitute relative substates. This may seem like an odd requirement, but it is preferable to the admission of nonsensical stateless lifecycles.

A State is defined in a specific State Model and cannot exist independently.

State Model SM

It is best not to confuse the general idea of a statechart, state model or state machine (in the software industry these terms are often interchanged without clear definition or distinction) with the state modeling application prescribed in miUML.

UML statecharts and state machines in general can be used for many purposes. For example, state machines (UML and otherwise) are often used in regular expression parsers. They are also used to define hardware logic. In these cases and others, the rules for creating, executing and applying these models varies.

In miUML, we have two distinct purposes in mind. Primarily we want to specify the pattern of behavior followed by all Objects in a Class. We also have another purpose which is to resolve competition with a single point of control in competitive Associations.

In miUML the term *state model* refers to the semantic mechanism for specifying behavior known as a Moore machine. A State Model is typically shown in diagram form as a UML statechart, but this representation is almost always incomplete (which events are ignored? unexpected? is a response defined for each Event in every State?). Furthermore, statecharts do not expose the execution semantics required to drive a State Model. So rule one in metamodeling statecharts is: 'Don't metamodel statecharts'. Rule two is: 'Don't metamodel statecharts!'[2]

Instead, we model the Moore state table, because it best embodies the state execution rules. Understand how the state table works, and this metamodel should make some sense. See the Event Response class description for an example table.

Attributes

Name

This is either a Class name or an Association's rnum. It represents the union of values taken from both the Lifecycle and Assigner Element numbers. There cannot be a collision since Element numbers all originate from Element.Number.

Type: SM name as union of Lifecycle.Class and Assigner.Rnum types

Domain

This is the union of the Lifecycle and Assigner Domain names.

Type: Name

[2] You have, of course, seen the movie 'Fight Club', right?

Identifiers

I> Name + Domain

Since Name is the union of two different types there is no possibility of a name collision between the two subclasses. Say, for example, that the modeler gives a Class the bizarre name 307 which appears to collide with an Rnum 307. In fact, the underlying type system will be able to distinguish these since each value has a different underlying type. Therefore 307:name is a complete different value from 307:nominal.

Relationships

R502
State Model IS A LIfecycle or Assigner

These are the only two uses of State Models in miUML. While these uses are entirely different, the underlying Moore finite state machine mechanics are almost identical.

From a use perspective, a Lifecycle represents the systematic behavior of Objects belonging to a Class whereas an Assigner resolves competition on a specific Association.

A Lifecycle specifies the behavior of Objects and, on some Classes, Objects may be created and deleted during runtime. So, on a Lifecycle State Model, it is possible for Objects to come into existence and disappear at designated points on the Lifecycle.

An Assigner, by contrast, specifies competition resolution for the community Objects on each side of an Association. So for the entire community, there is only ever one manifestation of the Assigner running. Like the Association it manages, an Assigner is part of the model structure and always exists (even though the Links it manages may individually come and go).

Transition T

'A *transition* abstracts the progression from state to state as caused by an event.' [MB]

An Event may cause an Instance to advance to a new State or back to the current State, triggering the destination State's Procedure in either case. A Transition is always triggered as an Event Response.

Attributes

State

Event Response.State

Event

Event Response.Event

State model

Event Response.State model and Destination.State model — Both the Destination and Transition must be part of the same State Model.

Domain

Event Response.Domain and Destination.Domain — This is part of the constraint described in State model above.

Destination

Destination.Name — The dynamic entity (Instance or Assigner token) will move to this Destination when the Transition is enabled in response to an Event occurrence.

Identifiers

I> State + Event + State model + Domain

Same as superclass. The Destination attribute cannot form part of an identifier since a State may be arrived via multiple Transitions all enabled by the same Event but originating from different States.

RELATIONSHIPS

R507

Transition SPECIFIES PATH TO *exactly one* Destination
Destination IS ARRIVED AT VIA *zero, one or many* Transition

A Transition defines a path from a State to a Destination (State or Deletion Pseudo State).

A Destination may be reached from multiple Transitions from, possibly, multiple States. It is possible for a Destination to be unreachable via a Transition on a LIfecycle State Model. If the Destination is a State, it might still be reachable from an Initial Transition via a Creation Event. Or perhaps the dynamic entity (Instance or Assigner token) will be initialized directly in the State prior to runtime.

Otherwise, a modeler may still create a State Model with one or more unreachable Destinations. The metamodel does not prevent this case, but supporting tools can warn the model about this situation in case it is unintentional. An incomplete State Model may have one or more unreachable States which are allowed as an editing convenience.

That said, there is no utility in specifying an unreachable Destination.