miUML Metamodel Class Descriptions
Polymorphism Subsystem

Leon Starr

Saturday, March 3, 2012

mint.miUMLmeta.td.5

Version 3.1.4

Table of Contents

Introduction

miUML State Metamodel Subsystems

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.

Polymorphism Subsystem

This subsystem constitutes one of the miUML metamodel state model subsystems. It is necessitated by the polymorphism feature in miUML which is concerned primarily with the specification and detection of Events. With polymorphism, it is possible to address a signal to a specialized Object without requiring the sender to know which configuration of Subclasses are presently instantiated by that Object. For example, rather than having to choose between sending a Close Checking Account vs. Close Savings Account Event depending on the present specialization of an Account Object, where each of these Events must be handled differently, the sender can simply emit a generic Close Event to the Account Object. The polymorphism feature ensures that this Event will be specialized in transmission and delivered to the appropriate Subclass Lifecycle State Model.

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

Delegation Direction DEL_DIR

A Delegated Event is handed off to each Subclass along a single Generalization. Keep in mind that a single Class may play a Superclass role in more than one Generalization. This is referred to as 'compound generalization' in [MB].

Delegated event ID

Delegated Event.ID

Superclass

Delegated Event.Class

Domain

Delegated Event.Domain and Generalization.Domain

Generalization

Generalization.Rnum - constrained to be descended from the same hierarchy rooted in the defining Polymorphic Event Specification.Generalization.

IDENTIFIERS

I> Delegated event ID + Superclass + Domain

On a 1x:Mx association, the association class identifier is the identifier on the Mx side.

RELATIONSHIPS

R569

Delegated Event DELEGATED DOWN *exactly one* Generalization
Generalization ESTABLISHES DIRECTION OF DELEGATION FOR *zero, one or many* Delegated Event

From [MB] in 'Rules about Polymorphic Signals', rule #4 in section 13.4 says: 'At run time, the polymorphic signal occurrence is received as an event in exactly one state machine instance in a given hierarchy.' Here we enforce this constraint.

Consider the case where a single Class participates in two Superclass roles, each on a different Generalization (compound generalization). If it were possible to Delegate an Event in both directions, via each Generalization, the Event would be handled by two LIfecycles, each found down a different specialization hierarchy. To prevent this, delegation may pass through only one specialization direction and thus, through a single Generalization.

By definition, event delegation must be specified on a Generalization. A Generalization may or may not establish a conduit for delegation as specialized Lifecycles may or may not be defined.

Delegated Event DEL_EV

In the context of Events, 'delegated' means 'detectable here, but handled by a subclass'. An Event that will not be handled in its associated Superclass, but further down in the Superclass's Generalization is a *delegated event.*

Attributes

ID

Event.ID

Class

Event.State model — Polymorphism applies only to Lifecycle State Models so this attribute represents a Class name. In fact, it will always be one that participates as a Superclass.

Domain

Event.Domain

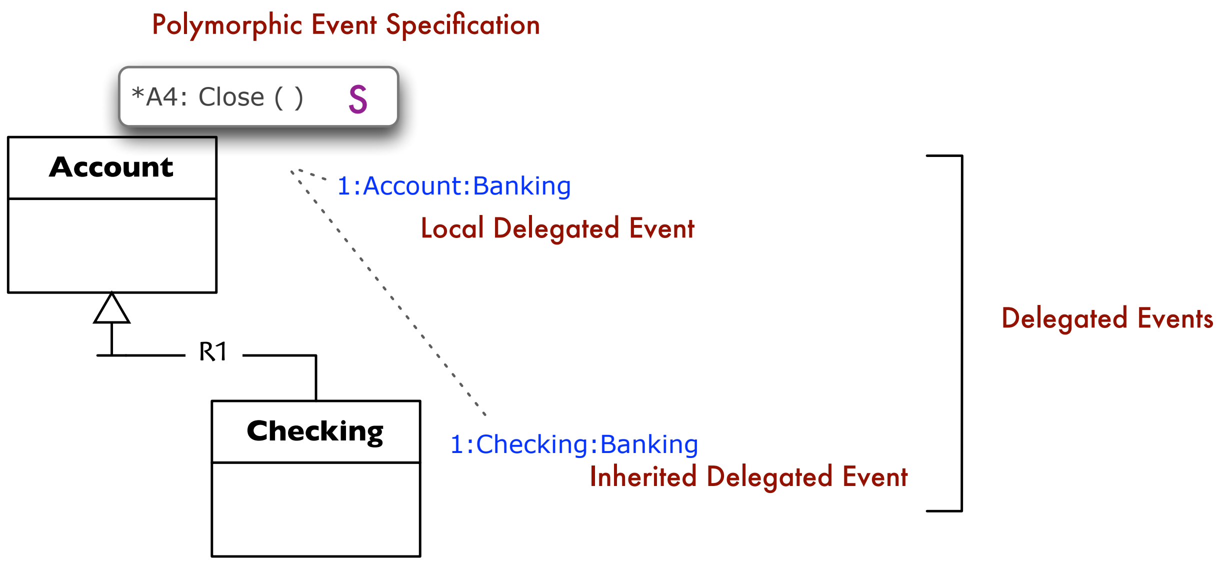
Identifiers

I> ID + Class + Domain

Relationships

R561

Delegated Event IS A Local or Inherited Delegated Event



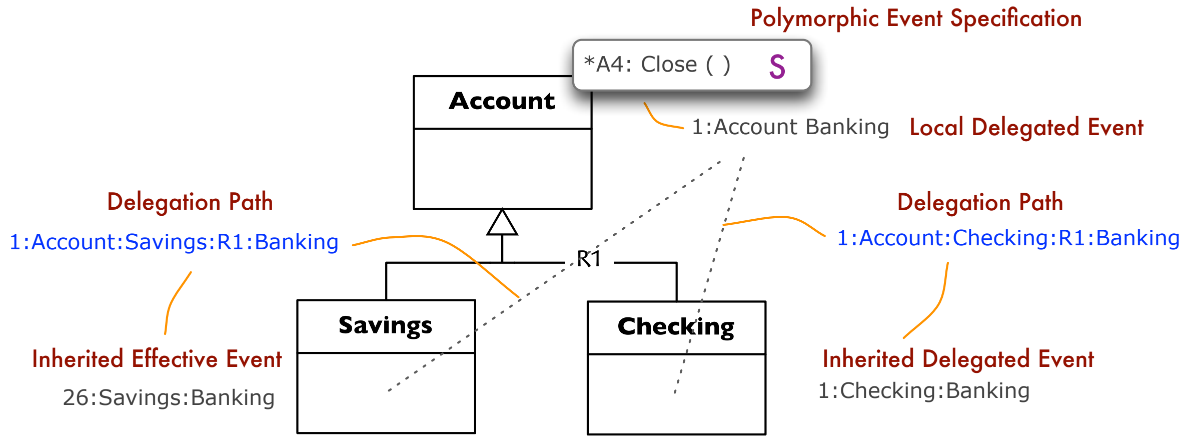
For an Event that will be delegated, there are two possibilities. The Event may be defined locally in which case it is at the topmost level of generalization. Otherwise, the Event corresponds to another Event at the next higher level of generalization that was delegated, in which case it is inherited. In either case, this Event will be further delegated to a lower level of specialization.

Delegation Path DEL_PATH

A Delegated Event is defined on a Class that participates as a Superclass in at least one Generalization. From there, the Event is handed down through ONE of these Generalizations to EACH of its Subclass Lifecycle State Models. By 'handing down', we mean that a separate, but corresponding Event is created in each Subclass Lifecycle State Model.

A *delegation path* establishes the handoff between a Delegated Event and an Inherited Event via a Generalization traversal. This idea is illustrated with an example below.

Each Inherited Event is spawned by a Delegation Path



Let's start with a Polymorphic Event Specification such as *A4:Close(). It defines a Local Delegated Event 1:Account:Banking. Note that the 1 is an Event ID value whereas the 4 in A4 is an Event Specification.Number value. Only the latter should appear on a statechart. (See the Event.ID attribute description to see why this distinction is necessary). At this point one Delegation Path must be defined for each Subclass to guarantee that an occurrence of *A4: Close() will always be handled. These are 1:Account:Savings:R1:Banking and 1:Account:Checking:R1:Banking. Each Delegation Path then spawns a distinct Inherited Event on the associated Subclass's Lifecycle State Model, 26:Savings:Banking and 1:Checking:Banking, respectively. Since Event.ID's are generated separately for each State Model, the fact that the value 1 has been selected for both the 1:Account:Banking and 1:Checking:Banking Events is mere coincidence. In fact, the example was purposely contrived this way to clarify that potentially confusing point!

Attributes

Delegated event ID

Delegated Event.ID — An Event delegated from a Superclass Lifecycle State Model.

Superclass

Delegated Event.Class — The Class name associated on which the Delegated Event is defined.

Subclass

Subclass.Class — The Subclass to which the Delegated Event is handed off.

Generalization - constrained

Subclass.Rnum — The Generalization of the Subclass constrained to match that of the associated Delegation Direction.

Domain

Delegated Event.Domain and Subclass.Domain

Identifiers

I> ID + From superclass + To subclass + Generalization + Domain

Relationships

R553
Delegated Direction LEADS TO *one or many* Subclass
Subclass IS DESTINATION OF *zero, one or many* Delegation Direction

Executable UML rules [MB] require a Delegated Event to be delegated to each Subclass on a Generalization in which the Delegated Event's associated Superclass also participates. Since a Generalization defines at least one set partition and therefore at least two Subclasses, delegation always proceeds through more than one Subclass.

If a Subclass's Superclass does not define any Delegated Events, the Subclass won't need to handle any Inherited Events *for that Superclass-Subclass Generalization*.

Effective Event EFF_EV

An Event processed locally rather than delegated is an *effective event*. In other words, the Event will result in an effect in its State Model when an occurrence is detected.

See the Polymorphic Event Specification class description illustration for some Effective Event examples.

Attributes

ID

Event.ID

State model

Event.State model

Domain

Event.Domain

Identifiers

I> ID + State model + Domain

Relationships

R554

Effective Event IS A Local or Inherited Effective Event

An Event that is processed locally was either defined locally or inherited from a corresponding Delegated Event in a Superclass.

Event

'An *event* is the abstraction of an incident in the domain that tells us that something has happened.' [MB]

In other words, an Event is the abstraction of something that can happen at a specific point in time.

It is important to distinguish between an occurrence and the specification of something that can happen. An Event specifies a kind of thing that can happen, Door opened, for example. This is what would be defined on the State Model for the Door Class. During runtime, this Event may occur multiple times, for one or more Door Objects, Door opened on Door 3 on Tuesday at 15:17:03.21, could be one occurrence of the Door opened Event, for example. So the term 'event' as defined here is a part of its specification. The terms 'occurrence' or 'instance' should be used to clarify a particular occurrence of an Event.

To accommodate polymorphism, a further distinction must be made between where an Event is detected and where it is actually processed. The declaration of an Event on a State Model indicates only that the State Model recognizes the Event. If an Event will be processed on the same State Model where it is defined, it is an Effective Event. Otherwise, the Event may be delegated so that its processing is handled in multiple Subclass Lifecycle State Models.

Attributes

ID

Without polymorphism, all Events could be uniquely identified with a name or number coupled with the name of its State Model and Domain. The following non-polymorphic Events, for the elevator door lifecycle example, would be uniquely named: Door: Open: EV, Door: Close:EV, Door: Cabin:EV arrived, etc. (where 'EV' is the Domain alias) The alternate identifier would replace the Event name with the Event number: Door:1:EV, Door:2:EV, Door:3:EV assuming the Events are numbered Door1: Open, Door2: Close, Door3: Cabin arrived.

Sadly, this neat little scheme breaks under polymorphism. Consider a Polymorphic Event Specification identified as Close:Account:Banking, or alternately with its number, 4:Account:Banking. This specifies a Local Delegated Event at the top level of the Account R1 Generalization. Moving down to the Checking Subclass, the number/name must be preserved. But what if the number 4 is already assigned to some other, non-polymorphic Event in Checking? Or the name?

It gets worse. Consider a weird case where a Class plays the role of Subclass in two Generalizations so that the Subclass Lifecycle State Model inherits Events from two Superclass Lifecycle State Models, unusual, but perfectly legal. It would then be possible for each Superclass to define an Event with matching names and/or numbers as they need only be unique locally, again unusual, but perfectly legal. Creation of the Inherited Event in the one Subclass Lifecycle State Model would then result in an identity conflict.

One way to resolve all of this trouble would be to re-invent the naming mechanism described in [MB] and [OOA96]. A less methodologically disruptive solution is to generate unique identifier values for every Event, local to its State Model, regardless its polymorphic or non-polymorphic properties. And THAT is why there is a distinct Event.ID attribute.

Yeah, I don't like it either.

Type: nominal

State model

Union of the Effective Event.State Model and Delegated Event.Class value domains.

Type: SM name

Domain

Union of the Effective Event.Domain and Delegated Event.Domain value domains.

Type: name

IDENTIFIERS

I> ID + State model + Domain

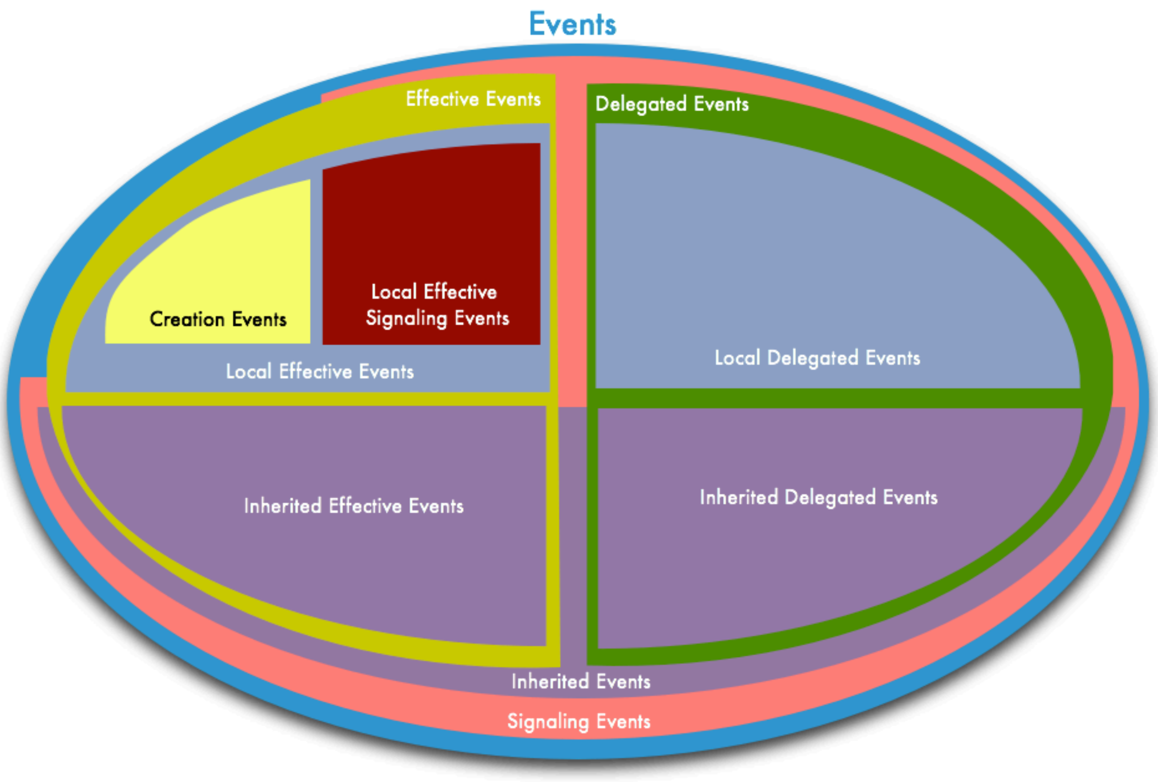
RELATIONSHIPS

R560
Event IS AN Effective or a Delegated Event

An Event is either processed in its own State Model or delegated, in the case of polymorphism, to Subclass Lifecycle State Models.

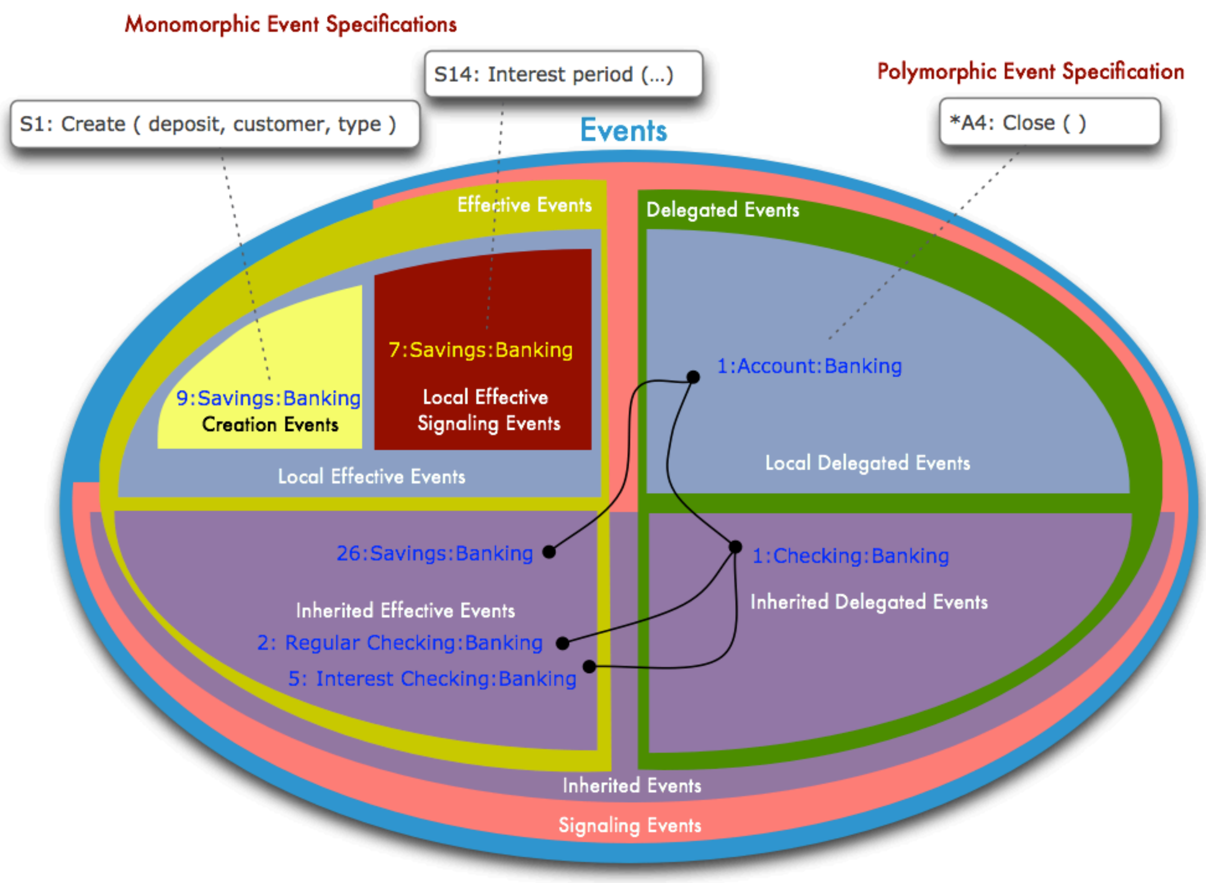
The following diagram illustrates the complete set of specializations used to characterize Events.

Venn diagram based on Event specializations



Note that a Creation Event must always be a Local Effective Event and is therefore never associated with a Polymorphic Event Specification.

To further illustrate, we can populate this diagram (see below) with instances from the example shown in the Polymorphic Event Specification class description along with an example Creation Event and a non-polymorphic signal Event.


Monomorphic Event Specifications
S14: Interest period (...)
Polymorphic Event Specification
S1: Create (deposit, customer, type)
Events
*A4: Close ( )
Effective Events
Delegated Events
7:Savings:Banking
Local Effective Signaling Events
9:Savings:Banking
Creation Events
1:Account:Banking
Local Effective Events
Local Delegated Events
26:Savings:Banking
1:Checking:Banking
Inherited Effective Events
Inherited Delegated Events
2: Regular Checking:Banking
5: Interest Checking:Banking
Inherited Events
Signaling Events

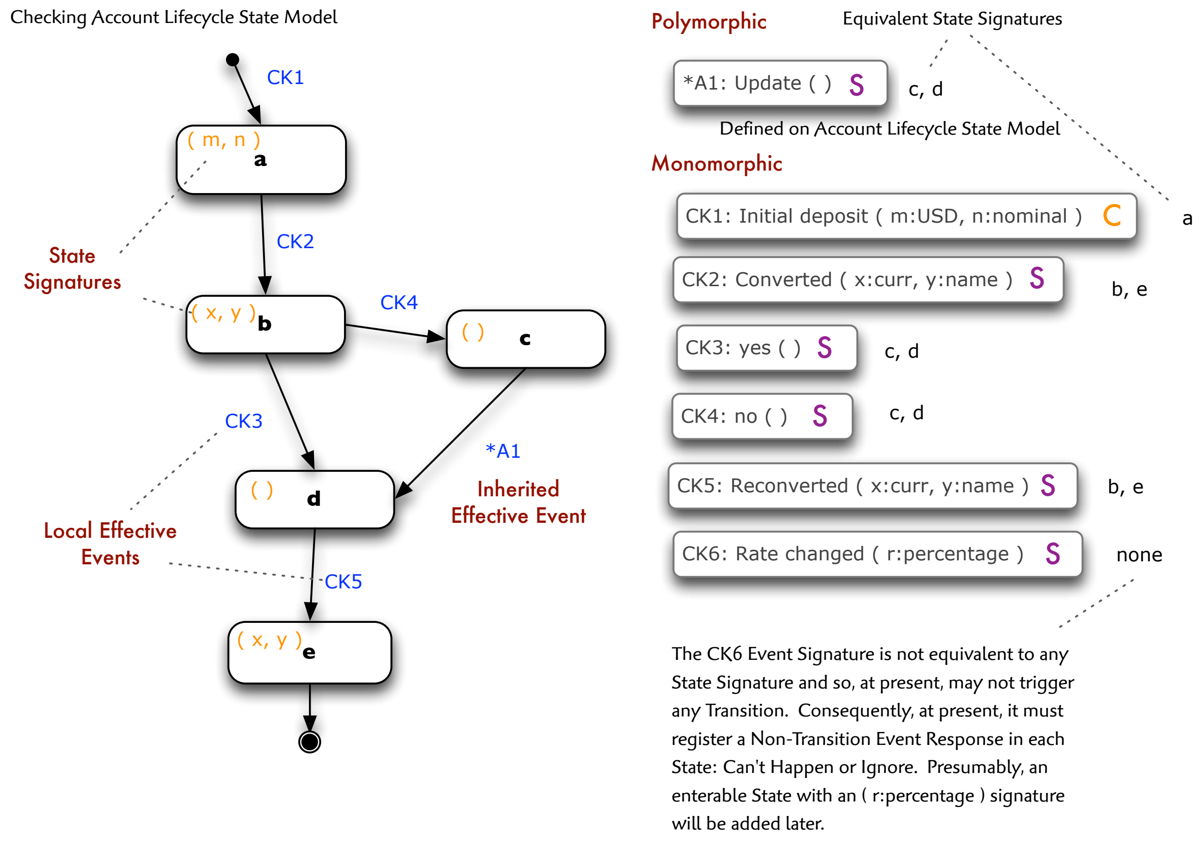
Event Specification and Event examples

Note above the chain of Delegated Events (via Delegation Paths - not shown) leading ultimately to Inherited Effective Events in appropriate subclasses.

Event Signature ESIG

The parameters, if any, supplied by an Event are defined in an *event signature*. This signature must match the State Signature of every State entered via Transitions triggered by a specified Effective Event.

Event and State Signatures


Checking Account Lifecycle State Model
CK1
(m, n)
a
CK2
State Signatures
(x, y)
b
CK4
( )
c
CK3
( )
d
*A1
Inherited Effective Event
Local Effective Events
CK5
(x, y)
e
Polymorphic
Equivalent State Signatures
*A1: Update ( ) S
c, d
Defined on Account Lifecycle State Model
Monomorphic
CK1: Initial deposit (m:USD, n:nominal) C
a
CK2: Converted (x:curr, y:name) S
b, e
CK3: yes ( ) S
c, d
CK4: no ( ) S
c, d
CK5: Reconverted (x:curr, y:name) S
b, e
CK6: Rate changed (r:percentage) S
none

The CK6 Event Signature is not equivalent to any State Signature and so, at present, may not trigger any Transition. Consequently, at present, it must register a Non-Transition Event Response in each State: Can't Happen or Ignore. Presumably, an enterable State with an (r:percentage) signature will be added later.

Attributes

Event

Event Specification.Number and State Model Signature.Name

State model

State.State model and State Model Signature.State Model

Domain

State.Domain and State Model Signature.Domain

Identifiers

I> Event + State model + Domain

Relationships

R556
Event Signature SPECIFIES PARAMETERS DELIVERED BY *exactly one* Event Specification
Event Specification DELIVERS PARAMETERS SPECIFIED BY *exactly one* Event Signature

An Event Signature is created as an essential component of an Event Specification. When any Effective Event is assigned to a Transition, the parameter set of the destination State's State Signature must be equivalent (same names and types) to the Event's Event Signature.

Note that an Effective Event may not necessarily trigger any Transition in its State Model. All States may mark the Event as Can't Happen or Ignore, for example. In this case State Signatures are irrelevant.

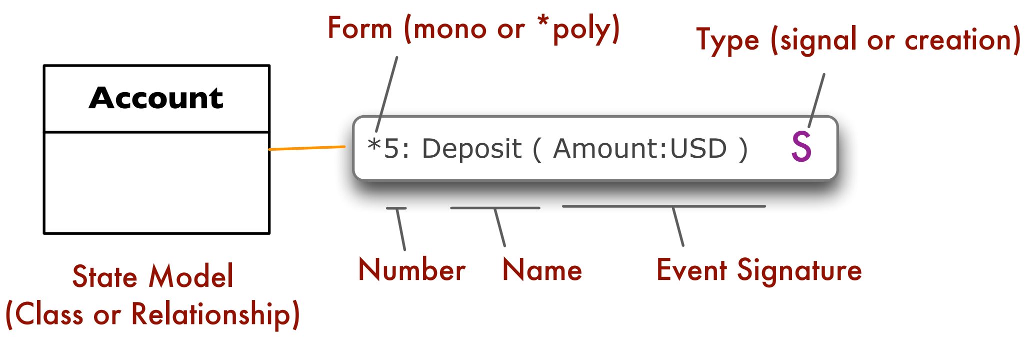
All this implies that when a State Signature changes (its parameter set edited), all incoming Transitions must either force any incoming triggering Events to conform (update their Event Signatures) or delete themselves.

When an Event is unbound from all Transitions, no State Signatures will be relevant, but its Event Signature will always remain as part of the Event's Event Specification.

Event Specification ESPEC

Something that can happen and can be detected on a given State Model is defined with an *event specification*. An occurrence of an Event Specification may be handled locally, on the State Model in which it is specified, or, only in the case of a Lifecycle Statechart, delegated to a Subclass's State Model.

Anatomy of an Event Specification



An Event Specification does not itself indicate where an Event may be detected. It merely defines the event structure which includes its name, type (creation or signal), form (mono or poly), number, parameters and State Model association.

ATTRIBUTES

Number

Each Event is numbered uniquely within a State Model.

Was tempted to follow the cnum, rnum pattern and call this an 'enum', but that might lead to some programmer confusion! Event.Number it is.

Type: nominal

Name

The meaning of an Event is conveyed by its name. Example Events for an elevator door lifecycle might be Open requested, Cabin arrived, etc.

Type: name

State model

State Model.Name — The State Model on which this Event Specification is defined.

Domain

State Model.Domain

Identifiers

1> Number + State model + Domain

2> Name + State model + Domain

Relationships

R565
Event Specification IS DEFINED ON *exactly one* State Model
State Model DEFINES *zero, one or many* Event Specification

The definition (specification) of an Event refers to where an Event is named and where it is assigned a number. With polymorphism, an Event will be detectable on State Models other than where it is defined (subclass lifecycles). So definition implies detection, but detection does not imply definition. The example A4: Close Event is defined on the Account object with a Polymorphic Event Specification. But it is detected on the Account, Checking, Savings, Checking with Interest and Regular Checking Lifecycle State models.

Since it is okay for a State Model to be event-less (just a State with nothing happening), a State Model might not react to any Events. Also a subclass Lifecycle State Model might detect one or more Events, with absolutely none specified on that particular State Model. In this case they will all be Inherited Events.

By definition, an Event Specification must be defined on a single State Model.

R550
Event Specification IS A Monomorphic or Polymorphic Event Specification

If something that can happen is both defined and handled by the same Lifecycle or it is defined on an Assigner State Model, it is Monomorphic. But if an Event is defined on a Superclass Lifecycle State Model and delegated through one or more layers of Subclass Lifecycle State Models, then it is Polymorphic. So if it weren't for polymorphism, all Event Specifications would be Monomorphic and the distinction between Event and Event Specification would be superfluous.

Inherited Delegated Event INDEL_EV

An Event detectable on a Class participating as a Subclass that was created to handle a corresponding Delegated Event on a Superclass, with both Classes participating in the same Generalization, where the Event is also delegated through a Generalization in which the Event's Class participates as a Superclass is an *inherited delegated event*.

See the example illustrated in the Polymorphic Event Specification class description to help parse the above sentence.

Attributes

ID

Delegated Event.ID and Inherited Event.ID

Class

Delegated Event.Class and Inherited Event.Subclass —This reflects the fact that an Inherited Delegated Event must be defined on a Class that participates as a Subclass since event inheritance requires a Superclass.

Domain

Delegated Event.Domain and Inherited Event.Domain

Identifiers

I> ID + Class + Domain

Relationships

None.

Inherited Effective Event INEFF_EV

An Event is 'effective' if it is not only detected in a given State Model, but it is also processed by that State Model without further delegation. This means that each State in the State Model must have an Event Response defined.

An Event that has been inherited (corresponds to a Delegated Event from a Superclass Lifecycle State Model) and is not delegated any further is an *inherited effective event.*

Attributes

ID

Inherited Event.ID and Effective Event.ID and Signaling Event.ID

Class

Inherited Event.Subclass and Effective Event.State model and Signaling Event.State model — An Inherited Event must be defined on a Subclass, so the name of the Effective Event State Model is the Subclass name.

Domain

Inherited Event.Domain and Effective Event.Domain and Signaling Event.Domain

Identifiers

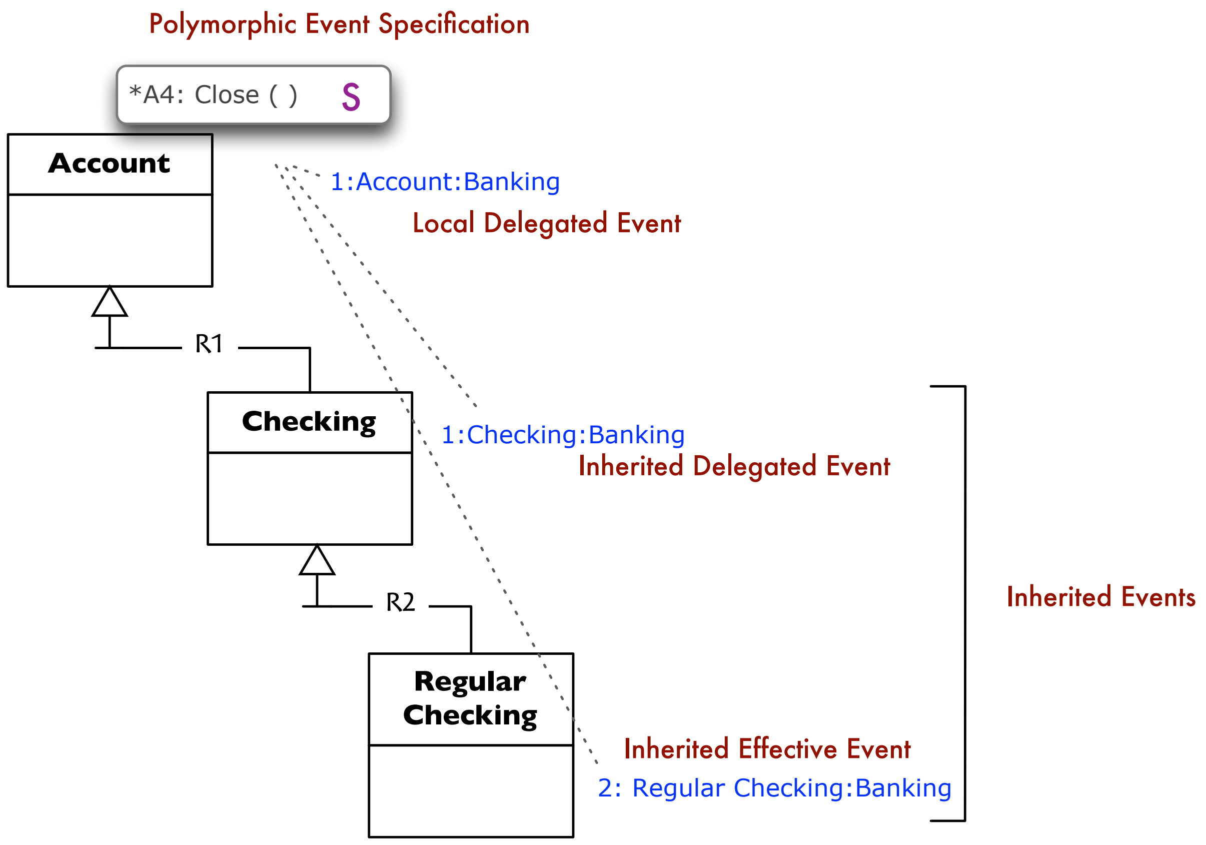
I> ID + Class + Domain

Relationships

None.

Inherited Event IN_EV

An Event that has been created on a Subclass Lifecycle State Model to handle a corresponding Delegated Event defined on a Superclass Lifecycle State Model is an *inherited event.*



Attributes

ID

Union of Inherited Effect Event.ID and Inherited Delegated Event.ID value domains — See the description for Event.ID.

Type: nominal

Subclass

Delegation Path.Subclass — The Inherited Event is defined on this Class.

Domain

Delegation Path.Domain

Parent event ID

Delegation Path.Delegated event ID — This is the Delegated Event in the Superclass being delegated to the Inherited Event.

Superclass

Delegation Path.Superclass — This is the home of the parent (Delegated Event) that is being inherited.

Generalization

Delegation Path.Generalization — This is the Generalization that includes both the Superclass Delegated Event and the Subclass Inherited Event.

Identifiers

1> ID + Subclass + Domain

This identifier is essential for the R552 specialization. The ID component ensures that there is no collision in the rare, but possible, case were a Subclass with two Superclasses (each in a separate Generalization) inherits an Event from each Superclass where the two Events are named or numbered identically.

2> Parent event ID + Superclass + Subclass + Generalization + Domain

Since Association R559 is 1:1 and unconditional on the referenced side, the reference to the Delegation Path must form a complete identifier.

Relationships

R559
Delegation Path SPAWNS *exactly one* Inherited Event
Inherited Event IS SPAWNED BY *exactly one* Delegation Path

The path from a Superclass to a Subclass on a Generalization for a given Delegated Event must lead to a corresponding Inherited Event in the Subclass.

A parent Delegated Event is always found by tracing the path from an Inherited Event through a Generalization from the Inherited Event's Class, which must be a Subclass in the Generalization, to the Superclass.

R552
Inherited Event IS AN Inherited Effective or Inherited Delegated Event

An Event that has been inherited can either be processed in the Event's Subclass Lifecycle State Model or it may be passed along (delegated further) if the Event's Class participates as a Superclass in another Generalization. If the Event is passed along, it is an Inherited Delegated Event. If processed, it is an Inherited Effective Event.

Local Delegated Event LD_EV

The ability of a Superclass Lifecycle State Model to detect an occurrence defined by a Polymorphic Event Specification is represented by a *local delegated event.*

By 'local', we mean that the Event is detectable on the very same Class where it is defined. By 'delegated', we mean that the Event is not actually handled by this Superclass, but is delegated to multiple Subclass Lifecycle State Models.

See the illustration in the Delegated Event class description for an example.

Attributes

ID

Delegated Event.ID

Number

Polymorphic Event Specification.Number

Class

Polymorphic Event Specification.Superclass and Delegated Event.Superclass — This is a class name which belong to a Class that participates in at least one Generalization as a Superclass.

Domain

Polymorphic Event Specification.Domain and Delegated Event.Domain

Identifiers

1> ID + Class + Domain

2> Number + Class + Domain

Relationships

R558
Local Delegated Event DETECTS *exactly one* Polymorphic Event Specification
Polymorphic Event Specification IS DETECTABLE AS *exactly one* Local Delegated Event

Whereas the Polymorphic Event Specification numbers and names a type of occurrence that may be seen throughout a Generalization, a Local Delegated Event specifies its 'detectability' on a single Lifecycle State Model, the one where it is originally defined. In other words, a Local

Delegated Event is the manifestation of a Polymorphic Event Specification in the originating Superclass Lifecycle State Model.

The need for this specification/event distinction results entirely from polymorphism. It addresses the need to define a type of occurrence in one place and then specify its detectability in multiple places. In a non-polymorphic miUML, event and specification would be fused together into a single ‘event’ concept.

Local Effective Event LEFF_EV

A non-polymorphic Event is both defined (local) and processed (effective) on the same State Model (Assigner or Lifecycle). Fuse this concept with a Monomorphic Event Specification and you have what is commonly, albeit imprecisely, understood as an 'event' in Executable UML.

Attributes

ID

Effective Event.ID

Number

Monomorphic Event Specification.Number

State model

Monomorphic Event Specification.State model and Effective Event.State model

Domain

Monomorphic Event Specification.Domain and Effective Event.Domain

Identifiers

1> ID + State model + Domain

2> Number + State model + Domain

Relationships

R557
Local Effective Event DETECTS *exactly one* Monomorphic Event Specification
Monomorphic Event Specification IS DETECTED VIA *exactly one* Local Effective Event

In fact, a Monomorphic Event Specification and a Local Effective Event are essentially the same thing since a non-polymorphic event is always specified and detected on the same State Model. They are kept separate for two reasons: 1) to help make the distinction between polymorphic and non-polymorphic Events explicit and 2) generalization-wide unique identification of Events as described in the Event.ID attribute description.

R567

Local Effective Event IS A Creation or Local Effective Signaling Event

Local Effective Events are non-polymorphic Events. A given State Model may handle only these two kinds, creation or signaling, non-polymorphic Events. In fact, a Lifecycle State Model may handle both, but an Assigner State Model is limited to Local Effective Signaling Events only. (An Assigner represents an Association which always exists, so there is nothing to create).

A Creation Event cannot be polymorphic (see Creation Event class description). This means that it must be a Local Effective Event. Furthermore, a Creation Event must be defined on a Lifecycle and it must be associated with a Destination that is a State. To enforce these constraints it is essential to abstract a 'Local Effective Creation Event' as distinct from a Local Effective Signaling Event. Since this it the only possible type of Creation Event, the title has been simplified. Not so for Signaling Event since they may also be polymorphic and hence inherited.

Local Effective Signaling Event LEFF_SEV

A Signaling Event which is neither delegated or inherited, in short, non-polymorphic, is a *local effective signaling event*.

Attributes

ID

Local Effective Event.ID and Signaling Event.ID

State model

Local Effective Event.State model and Signaling Event.State model

Domain

Local Effective Event.Domain and Signaling Event.Domain

Identifiers

I> ID + State model + Domain

Same as superclass.

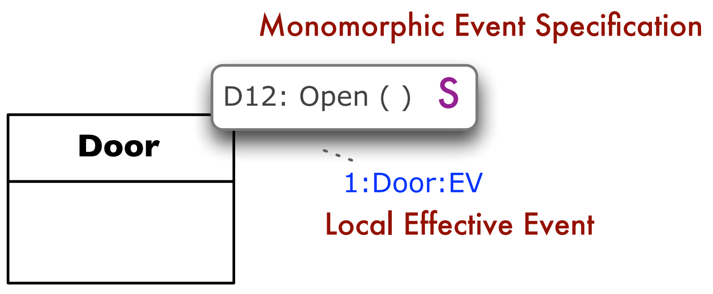
Relationships

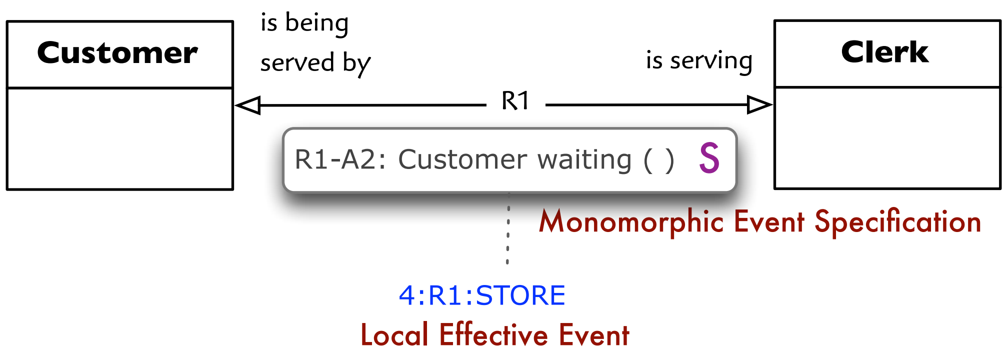
None.

Monomorphic Event Specification MESPEC

A non-polymorphic event is defined with a *monomorphic event specification*. It establishes the number, name, signature and associated Lifecycle or a Assigner State Model.

A Monomorphic Event Specification may be defined on a Class's Lifecycle State Model or an Association's Assigner State Model.





Attributes

Number

Event Specification.Number

State Model

Event Specification.State model — The name of the Class or an Rnum.

Domain

Event Specification.Domain

Identifiers

I> Number + State model + Domain

Relationships

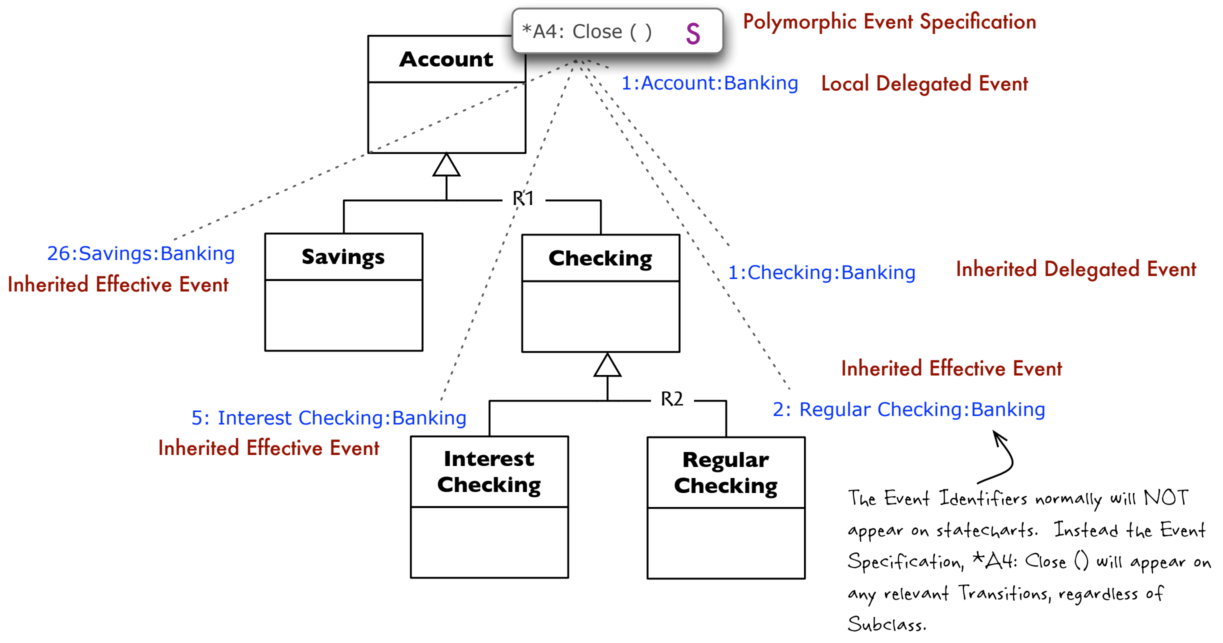
None.

Polymorphic Event Specification PESPEC

An entity (anything capable of emitting a Signaling Event) may want to tell a bank account to close without regard to the particular type of account. But the activity that results may vary depending on the type. So we want the ability to send a generic 'close' Event, without knowledge of the target instance Subclass and yet have that Event processed by the appropriate Subclass Lifecycle State Model.

This is accomplished by defining the generic event on a Superclass Lifecycle State Model, such as that for the example class Account. This event will then trickle down through one or more Generalizations to ultimately be resolved by activity defined in the appropriate Subclass Lifecycle.

A Polymorphic Event Specification and all corresponding Events across multiple Generalizations


*A4: Close ( ) S
Polymorphic Event Specification
Account
1:Account:Banking
Local Delegated Event
R1
26:Savings:Banking
Inherited Effective Event
Savings
Checking
1:Checking:Banking
Inherited Delegated Event
Inherited Effective Event
R2
5: Interest Checking:Banking
Inherited Effective Event
2: Regular Checking:Banking
Interest Checking
Regular Checking
The Event Identifiers normally will NOT appear on statecharts. Instead the Event Specification, *A4: Close () will appear on any relevant Transitions, regardless of Subclass.

An Event is defined on a Superclass Lifecycle State Model but delegated through one or more Subclass Lifecycle State Models with a *polymorphic event specification*.

Attributes

Number

Event Specification.Number

Superclass

Event Specification.State model and Superclass.Class —The name of the Superclass and its Lifecycle State Model are the same.

Domain

Event Specification.Domain and Superclass.Domain — Both the Event Specification and Superclass are in the same Domain.

Generalization

Superclass.Rnum — Polymorphism will descend down through this Generalization

Identifiers

I> Number + Superclass + Domain

Relationships

R551

Superclass DEFINES *zero, one or many* Polymorphic Event Specification
Polymorphic Event Specification IS DEFINED ON *exactly one* Superclass

Polymorphism always starts on a Superclass. In other words, you cannot define a polymorphic event by starting with a leaf Subclass or on a Class that does not participate in any Generalization.

Any given Superclass may or may not define polymorphic events or, in fact, have a Lifecycle State Model at all.

>> Signaling Event (imported)

Relationships

R568
Signaling Event IS A Local Effective Signaling Event or an Inherited Effective Event

A Signaling Event is either monomorphic or polymorphic. If it is monomorphic, the Signaling Event's specification (Monomorphic Event Specification) is on the same State Model where the Signaling Event is defined. If it is polymorphic, the SIgnaling Event inherits a Polymorphic Event Specification through a chain of one or more Delegated Events.

In both cases, the Signaling Event is an Effective Event which means that the Event is processed on its State Model and is not further delegated.

State Model Parameter SM_PARAM

A parameter is a variable (name + type) that specifies a required value. A *state model parameter* is a parameter defined within the context of a State Model. In this context, it specifies data required by a State which must be delivered by any Event triggering a transition into the State.

Attributes

Name

This is a descriptive variable name. An Event Specification named Set desired pressure, for example, might required a parameter named Desired pressure.

Type: name

Signature

State Model Signature.Name

State model

State Model Signature.State model

Domain

State Model Signature.Domain

Type

Constrained Type.Name — This is the data type constraining the set of values that may be assigned to this State Model Parameter.

Identifiers

I> Name + Signature + State model + Domain

A State Model Parameter is named uniquely within its State Model Signature

Relationships

R563

State Model Parameter IS REQUIRED IN *exactly one* State Model Signature

State Model Signature SPECIFIES REQUIREMENT OF *zero, one or many* State Model Parameter

Since State Model Parameters are relatively easy to specify, requiring only a name and type, there is little value in putting them into any sort of shareable library. So the naming of a State Model Parameter is entirely local to a State Model Signature. Therefore, two separate States on the same State Model might specify a parameter such as Duration which would in fact be two separate State Model Parameters.

A State Model Signature may be empty which indicates that a State, and hence an Event Specification, does not require any parameters.

R566

Constrained Type DEFINES RANGE OF VALUES ASSIGNABLE TO *zero, one or many* State Model Parameter

State Model Parameter MAY ASSUME VALUES DEFINED BY *exactly one* Constrained Type

A State Model Parameter is a variable and like any variable it needs a name and a type.

A Constrained Type may be defined without necessarily being used.

State Model Signature SMSIG

A set of named and typed Parameters defined on a State Model is a *state model signature*. This signature defines incoming Parameters that will be accessible in the State's Procedure. Consequently, any Transition entering a State must be triggered by an Event with an equivalent Event Signature. Equivalence, in this case, means that each Signature shares exactly the same quantity, names and types of Parameters.

See the illustration in the Event Signature class description for examples.

Attributes

Name

Depending on the subclass, this is either a State name or an Event number.

Type: SM sig name, defined as the union of the name and nominal common types

State model

The State Model upon which the signature is defined. The signature cannot be used outside the context of this State Model.

Type: SM name

Domain

The State Model is defined in this Domain.

Type: name

Identifiers

I> Name + State model + Domain

Relationships

R562
State Model Signature IS A State or Event Signature

A State Model Signature is either defined on a State, to express the parameter set required on any Event triggering a Transition that enters the State, or it is defined as part of an Event Specification.

In either case, a State Model Signature specifies a set of named and typed State Model Parameters.

State Signature SSIG

The Procedure executed upon entry into a State requires a specific set of parameters defined by a *state signature*. A value for each of these parameters must be supplied by any active entity (Instance or assigner token) that arrives. In Executable UML, this ensures that the Procedure in a State operates without regard to which Transition triggered it.

To ensure that this is the case, a signature is established for each State. The Event on every Transition leading into a State must be specified with an Event Signature that matches the State Signature.

Attributes

State

State.Name and State Model Signature.Name

State model

State.State model and State Model Signature.State Model

Domain

State.Domain and State Model Signature.Domain

Identifiers

I> State + State model + Domain

Relationships

R564

State Signature SPECIFIES PARAMETERS REQUIRED IN *exactly one* Destination
Destination REQUIRES PARAMETERS SPECIFIED BY *exactly one* State Signature

To enforce the rule that all Events leading into a State must provide the same set of State Model Parameters, it is necessary to define a single parameter signature on each State.

Since a Deletion Pseudo State can define a Procedure which may require State Model Parameters, a State Signature is required for both Destination types.