14 Interactions

14.1 Overview

Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.

The Interaction package describes the concepts needed to express Interactions, depending on their purpose. An interaction can be displayed in several different types of diagrams: Sequence Diagrams, Interaction Overview Diagrams, and Communication Diagrams. Optional diagram types such as Timing Diagrams and Interaction Tables come in addition. Each type of diagram provides slightly different capabilities that makes it more appropriate for certain situations.

Interactions are a common mechanism for describing systems that can be understood and produced, at varying levels of detail, by both professionals of computer systems design, as well as potential end users and stakeholders of (future) systems.

Typically when interactions are produced by designers or by running systems, the case is that the interactions do not tell the complete story. There are normally other legal and possible traces that are not contained within the described interactions. Some projects do, however, request that all possible traces of a system shall be documented through interactions in the form of (e.g., sequence diagrams or similar notations).

The most visible aspects of an Interaction are the messages between the lifelines. The sequence of the messages is considered important for the understanding of the situation. The data that the messages convey and the lifelines store may also be very important, but the Interactions do not focus on the manipulation of data even though data can be used to decorate the diagrams.

In this chapter we use the term *trace* to mean “sequence of event occurrences,” which corresponds well with common use in the area of trace-semantics, which is a preferred way to describe the semantics of Interactions. We may denote this by <eventoccurrence1, eventoccurrence2, ...,eventoccurrence-n>. We are aware that other parts of the UML language definition of the term “trace” is used also for other purposes.

By *interleaving* we mean the merging of two or more traces such that the events from different traces may come in any order in the resulting trace, while events within the same trace retain their order. Interleaving semantics is different from a semantics where it is perceived that two events may occur at exactly the same time. To explain Interactions we apply an Interleaving Semantics.

14.2 Abstract syntax

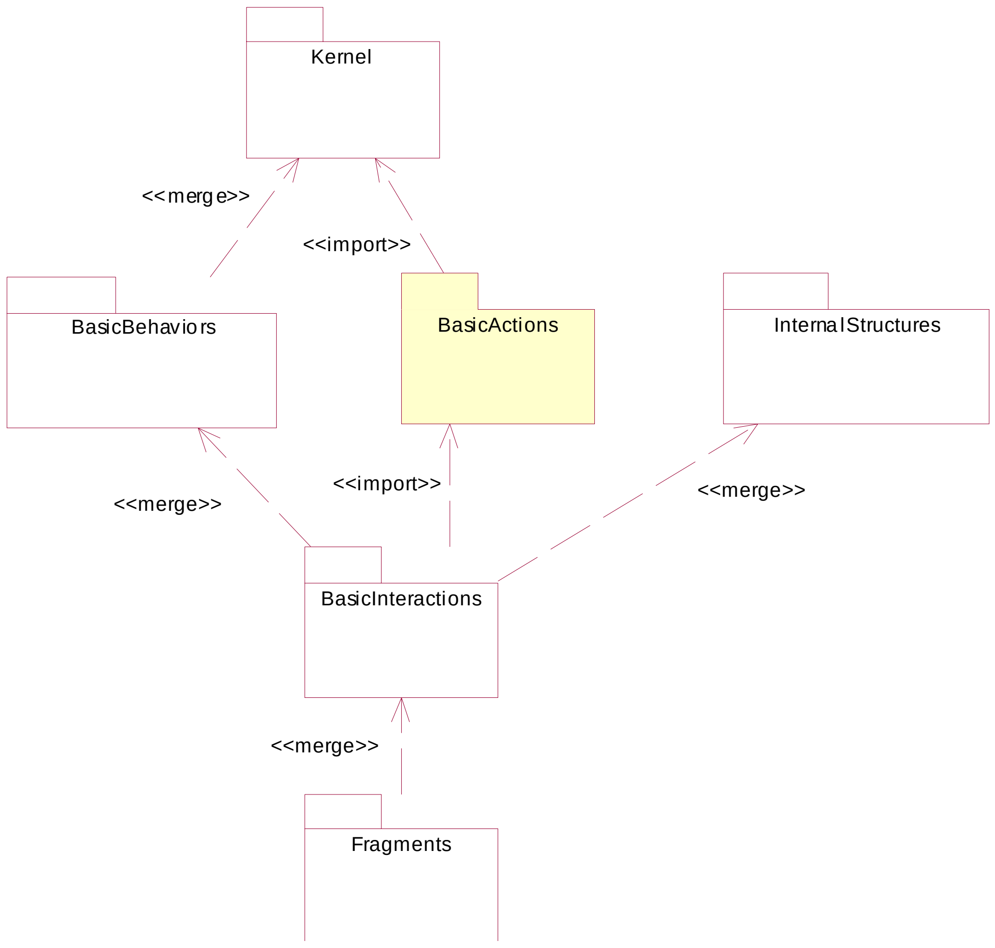


Figure 14.1 - Dependencies of the Interactions packages

Package BasicInteractions

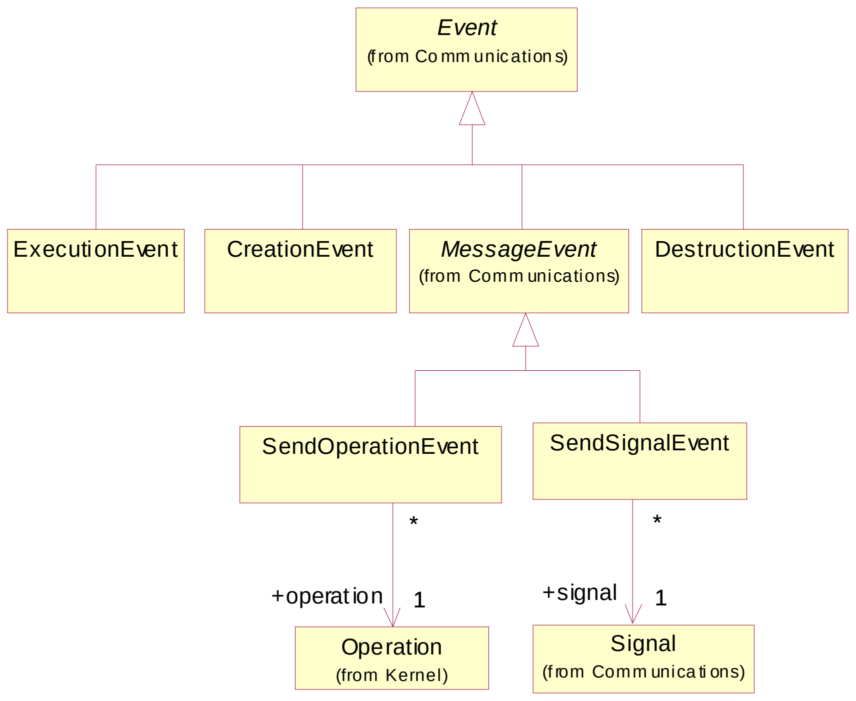


Figure 14.2 Events

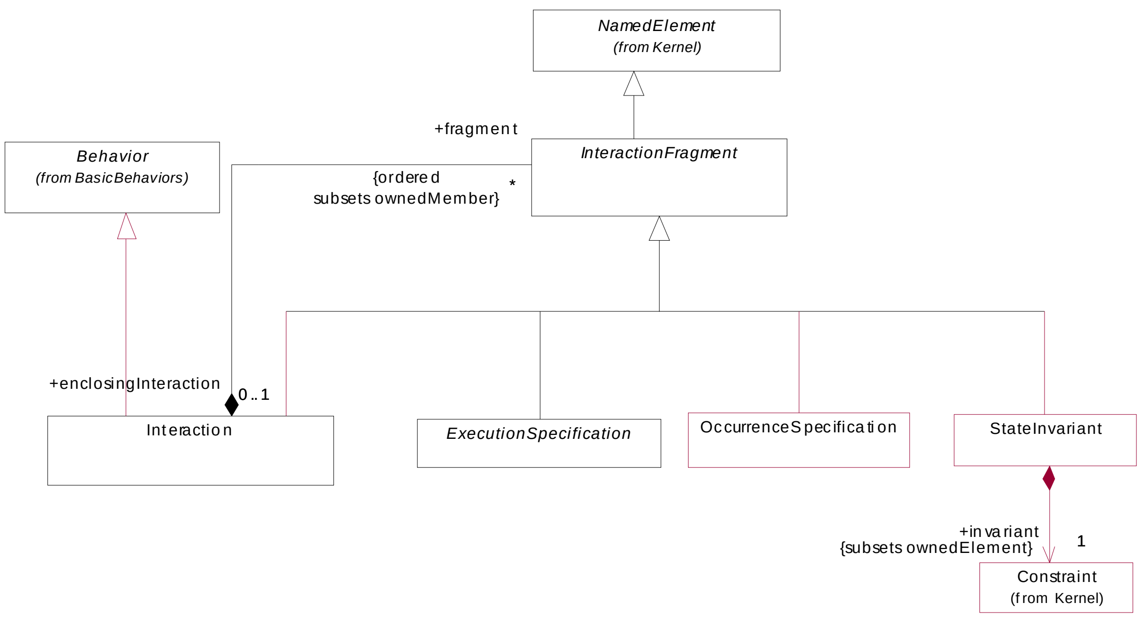


Figure 14.3 - Interactions

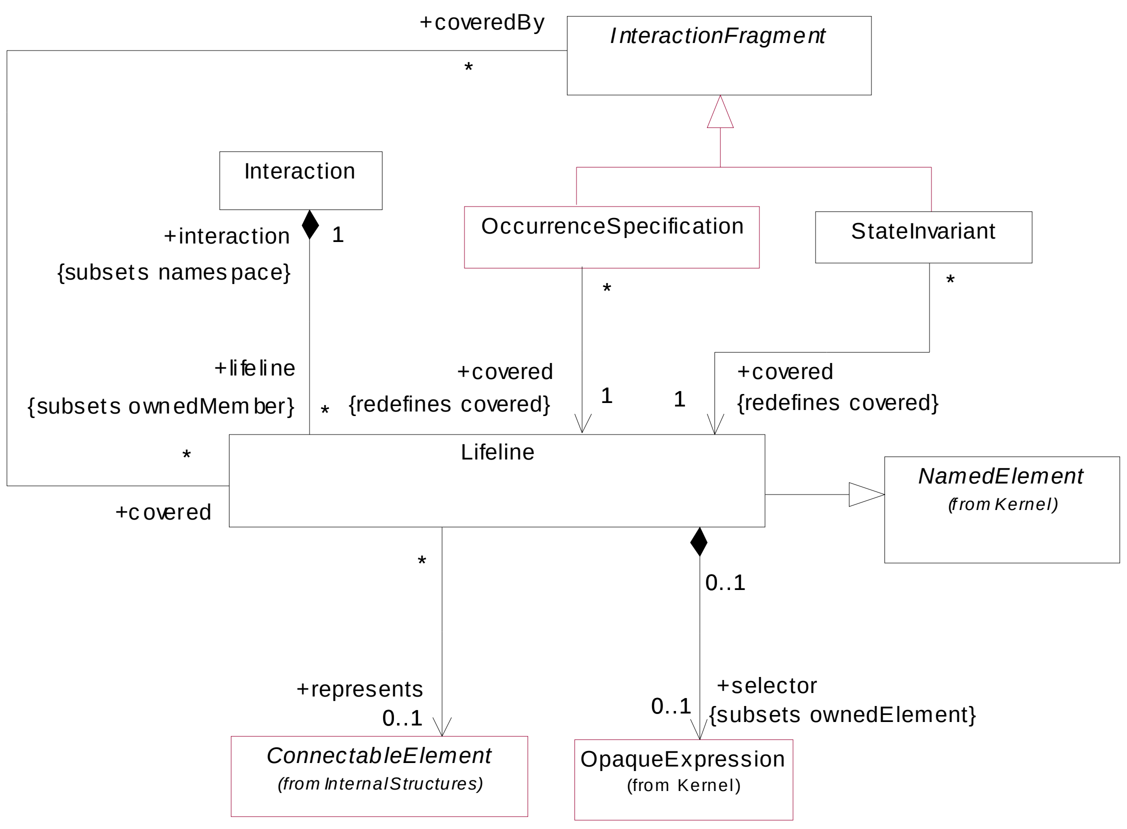


Figure 14.4 - Lifelines

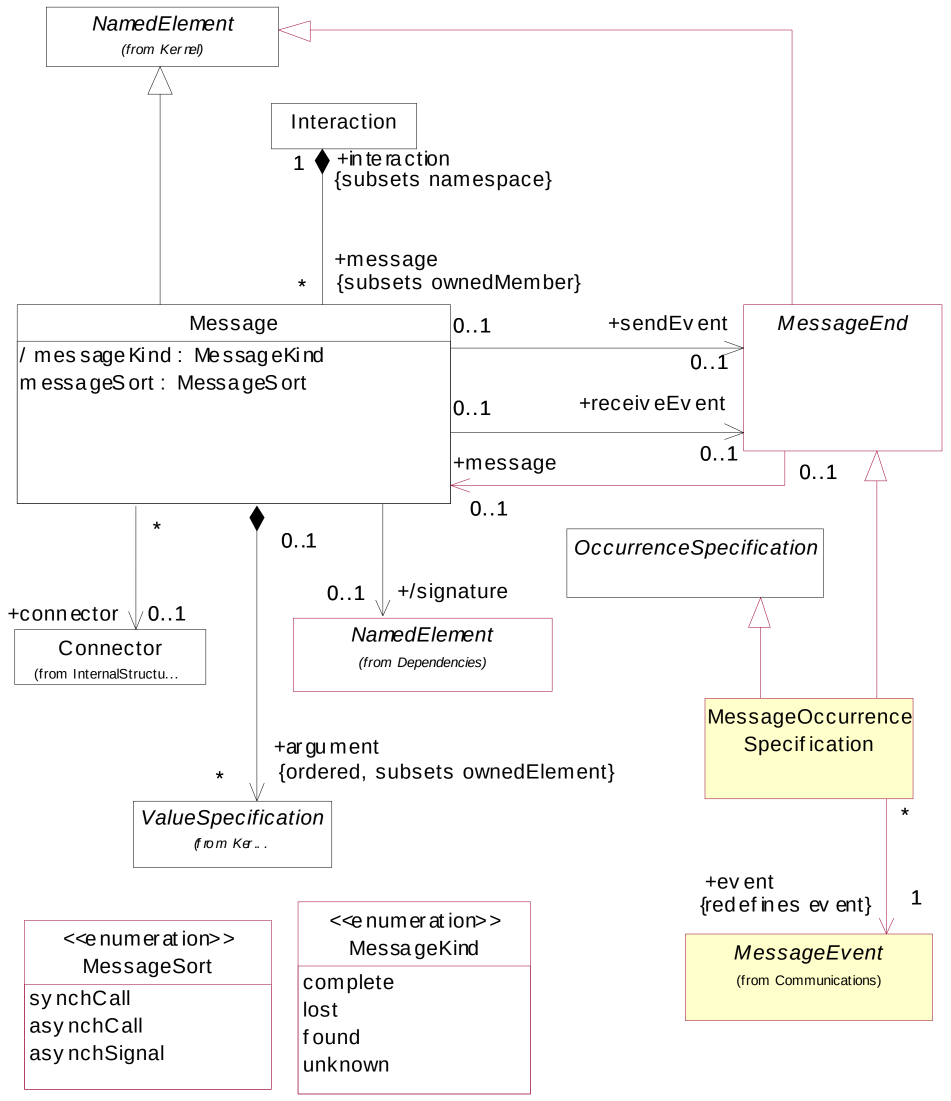


Figure 14.5 - Messages

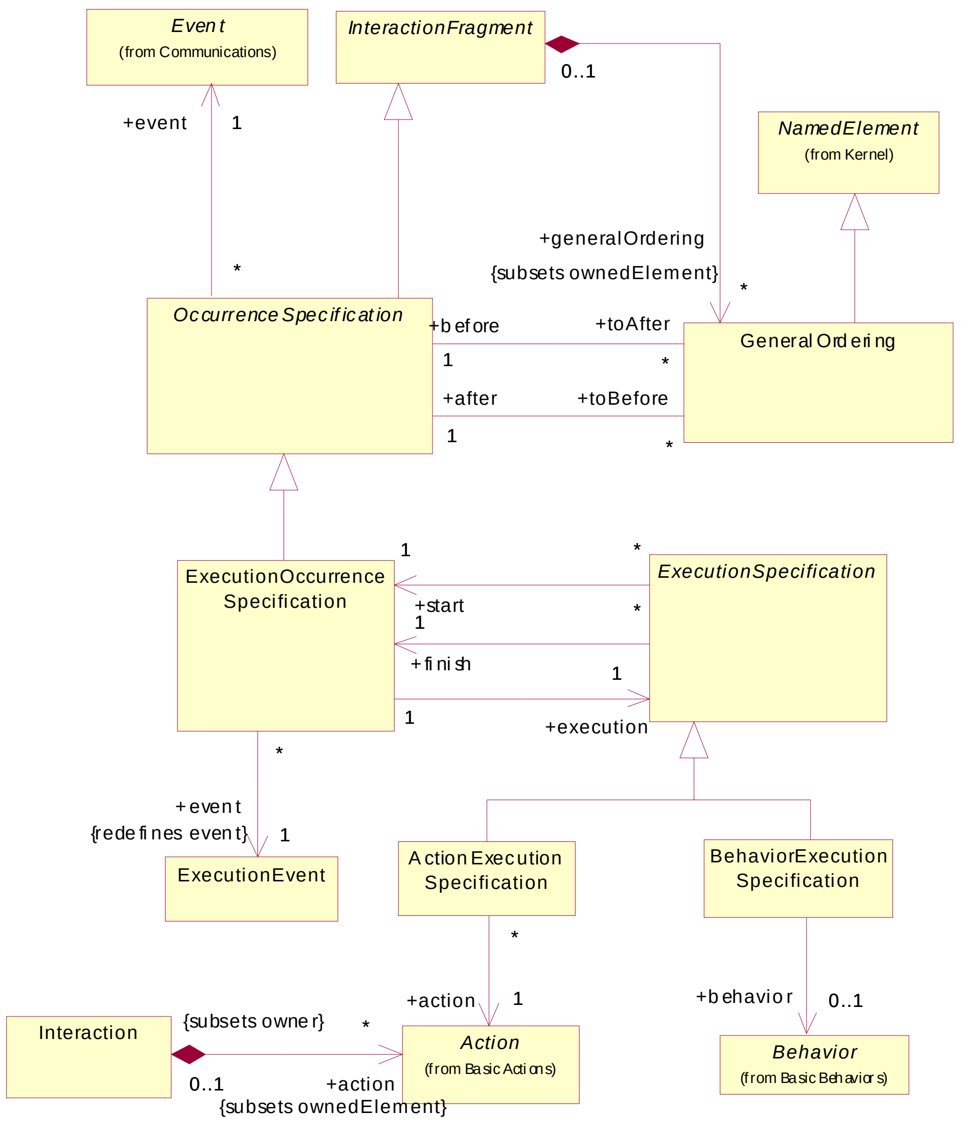


Figure 14.6 Occurrences

Package Fragments

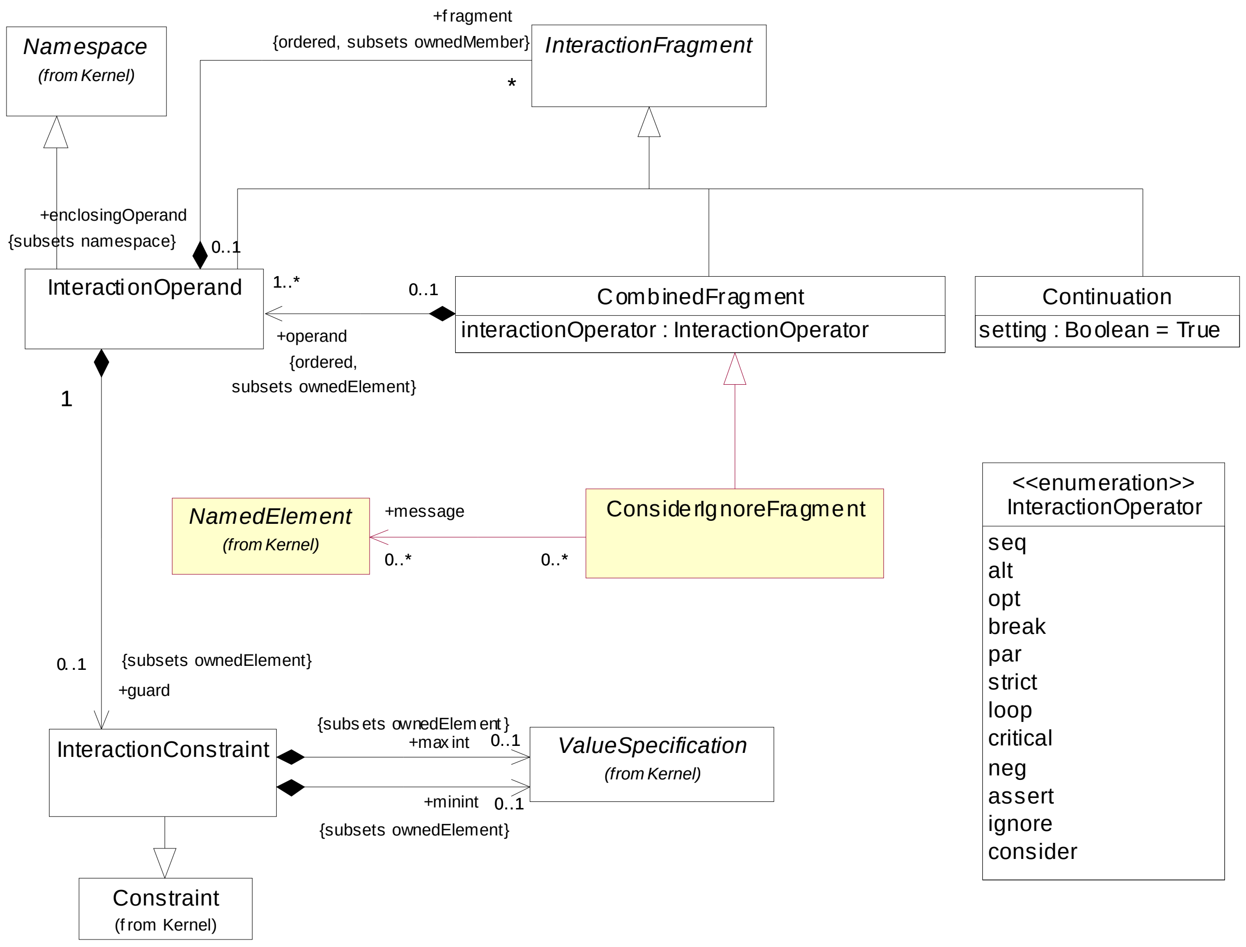


Figure 14.7 - CombinedFragments

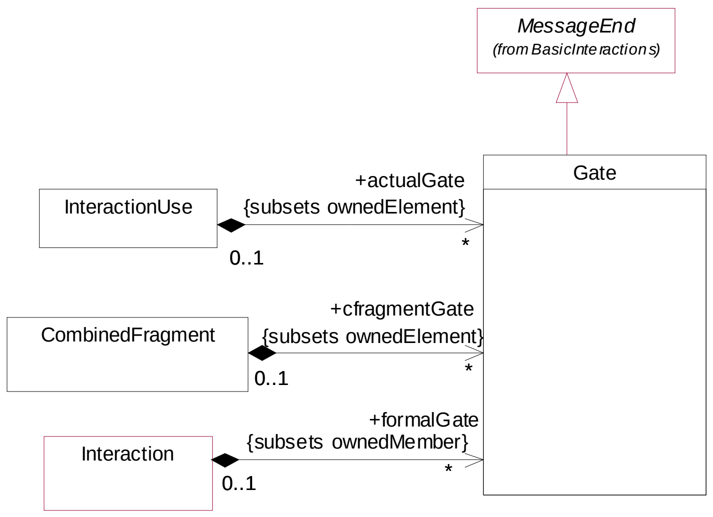


Figure 14.8 - Gates

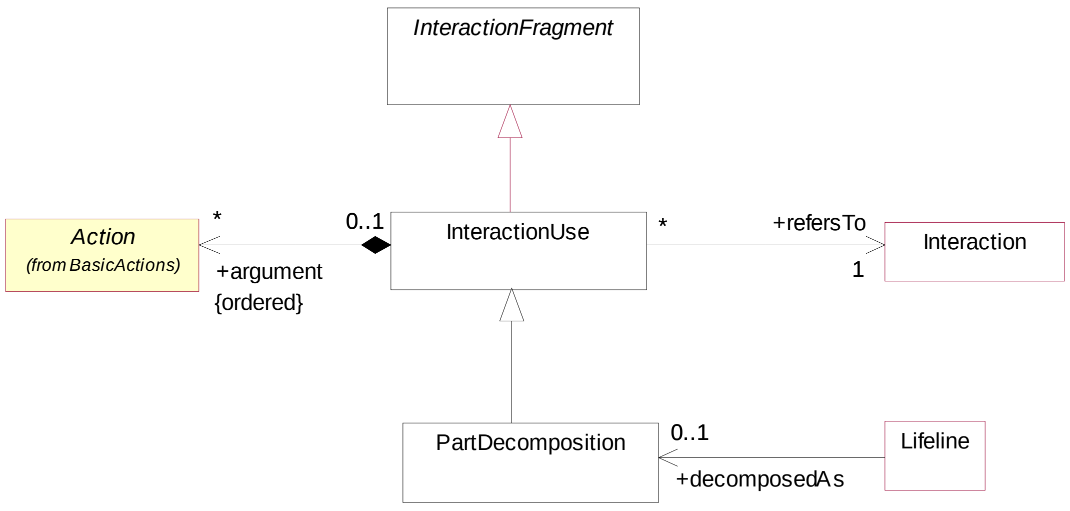


Figure 14.9 - InteractionUses

14.3 Class Descriptions

14.3.1 ActionExecutionSpecification (from BasicInteractions)

Generalizations

- "ExecutionSpecification (from BasicInteractions)" on page 464.

Description

ActionExecutionSpecification is a kind of ExecutionSpecification representing the execution of an action.

Attributes

No additional attributes

Associations

- action : Action [1] Action whose execution is occurring.

Constraints

[1] The Action referenced by the ActionExecutionOccurrence, if any, must be owned by the Interaction owning the ActionExecutionOccurrence.

Semantics

See "ExecutionSpecification (from BasicInteractions)" on page 464.

Notation

See "ExecutionSpecification (from BasicInteractions)" on page 464.

Rationale

ActionExecutionSpecification is introduced to support interactions specifying messages that result from actions, which may be actions owned by other behaviors.

14.3.2 BehaviorExecutionSpecification (from BasicInteractions)

Generalizations

- "ExecutionSpecification (from BasicInteractions)" on page 464

Description

BehaviorExecutionSpecification is a kind of ExecutionSpecification representing the execution of a behavior.

Attributes

No additional attributes

Associations

- behavior : Behavior [1] Behavior whose execution is occurring.

Constraints

No additional constraints

Semantics

See "ExecutionSpecification (from BasicInteractions)" on page 464.

Notation

See "ExecutionSpecification (from BasicInteractions)" on page 464.

Rationale

BehaviorExecutionSpecification is introduced to support interactions specifying messages that result from behaviors.

14.3.3 CombinedFragment (from Fragments)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 471

Description

A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner.

CombinedFragment is a specialization of InteractionFragment.

Attributes

• interactionOperator : InteractionOperator	Specifies the operation that defines the semantics of this combination of InteractionFragments.

Associations

• cfragmentGate : Gate[*]	Specifies the gates that form the interface between this CombinedFragment and its surroundings.
• operand: InteractionOperand[1..*]	The set of operands of the combined fragment.

Constraints

[1] If the interactionOperator is *opt*, *loop*, *break*, or *neg*, there must be exactly one operand.

[2] The InteractionConstraint with minint and maxint only apply when attached to an InteractionOperand where the interactionOperator is *loop*.

[3] If the interactionOperator is *break*, the corresponding InteractionOperand must cover all Lifelines within the enclosing InteractionFragment.

[4] The interaction operators 'consider' and 'ignore' can only be used for the CombineIgnoreFragment subtype of CombinedFragment.

((interactionOperator = #consider) **or** (interactionOperator = #ignore)) **implies** oclsisTypeOf(CombineIgnoreFragment)

Semantics

The semantics of a CombinedFragment is dependent upon the interactionOperator as explained below.

Alternatives

The interactionOperator **alt** designates that the CombinedFragment represents a choice of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the operand has no guard.

The set of traces that defines a choice is the union of the (guarded) traces of the operands.

An operand guarded by **else** designates a guard that is the negation of the disjunction of all other guards in the enclosing CombinedFragment.

If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing InteractionFragment is executed.

Option

The interactionOperator **opt** designates that the CombinedFragment represents a choice of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative CombinedFragment where there is one operand with non-empty content and the second operand is empty.

Break

The interactionOperator **break** designates that the CombinedFragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing InteractionFragment. A *break* operator with a guard is chosen when the guard is true and the rest of the enclosing Interaction Fragment is ignored. When the guard of the *break* operand is false, the break operand is ignored and the rest of the enclosing InteractionFragment is chosen. The choice between a *break* operand without a guard and the rest of the enclosing InteractionFragment is done non-deterministically.

A CombinedFragment with interactionOperator *break* should cover all Lifelines of the enclosing InteractionFragment.

Parallel

The interactionOperator **par** designates that the CombinedFragment represents a parallel merge between the behaviors of the operands. The EventOccurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

A parallel merge defines a set of traces that describes all the ways that EventOccurrences of the operands may be interleaved without obstructing the order of the EventOccurrences within the operand.

Weak Sequencing

The interactionOperator **seq** designates that the CombinedFragment represents a weak sequencing between the behaviors of the operands.

Weak sequencing is defined by the set of traces with these properties:

1. The ordering of EventOccurrences within each of the operands are maintained in the result.
2. OccurrenceSpecifications on different lifelines from different operands may come in any order.

3. OccurrenceSpecifications on the same lifeline from different operands are ordered such that an EventOccurrence of the first operand comes before that of the second operand.

Thus weak sequencing reduces to a parallel merge when the operands are on disjunct sets of participants. Weak sequencing reduces to strict sequencing when the operands work on only one participant.

Strict Sequencing

The interactionOperator **strict** designates that the CombinedFragment represents a strict sequencing between the behaviors of the operands. The semantics of strict sequencing defines a strict ordering of the operands on the first level within the CombinedFragment with interactionOperator *strict*. Therefore EventOccurrences within contained CombinedFragment will not directly be compared with other EventOccurrences of the enclosing CombinedFragment.

Negative

The interactionOperator **neg** designates that the CombinedFragment represents traces that are defined to be invalid.

The set of traces that defined a CombinedFragment with interactionOperator *negative* is equal to the set of traces given by its (sole) operand, only that this set is a set of invalid rather than valid traces. All InteractionFragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible.

Critical Region

The interactionOperator **critical** designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces. Even though enclosing CombinedFragments may imply that some OccurrenceSpecifications may interleave into the region, such as with *par*-operator, this is prevented by defining a region.

Thus the set of traces of enclosing constructs are restricted by critical regions.

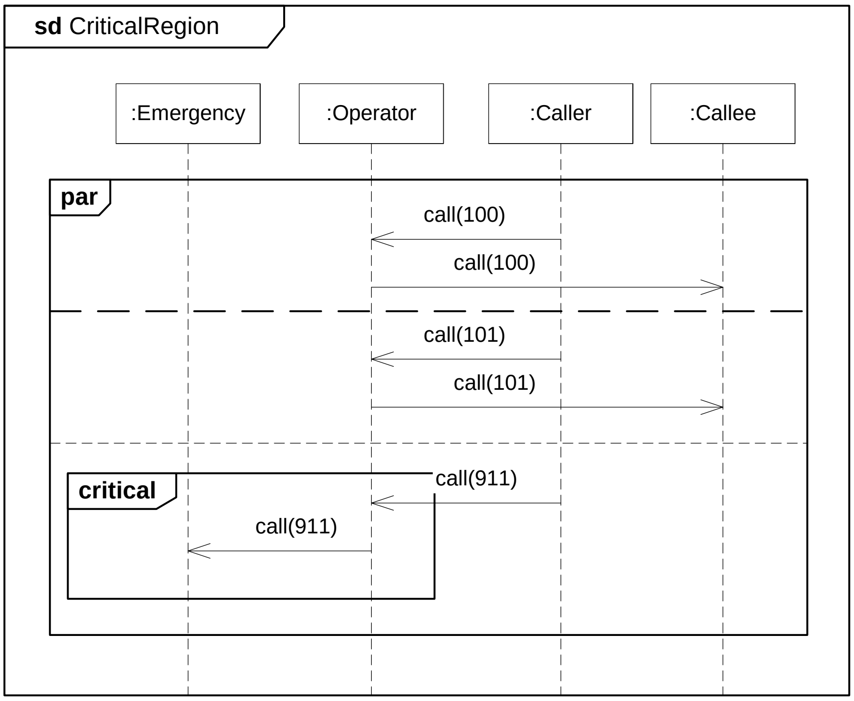


Figure 14.10 - Critical Region

The example, Figure 14.10 shows that the handling of a 911-call must be contiguously handled. The operator must make sure to forward the 911-call before doing anything else. The normal calls, however, can be freely interleaved.

Ignore / Consider

See the semantics of 14.3.4, “ConsiderIgnoreFragment (from Fragments),” on page 458.

Assertion

The interactionOperator **assert** designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.

Assertions are often combined with Ignore or Consider as shown in Figure 14.24.

Loop

The interactionOperator **loop** designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times.

The Guard may include a lower and an upper number of iterations of the loop as well as a Boolean expression. The semantics is such that a loop will iterate minimum the ‘minint’ number of times (given by the iteration expression in the guard) and at most the ‘maxint’ number of times. After the minimum number of iterations have executed and the boolean expression is false the loop will terminate. The loop construct represents a recursive application of the *seq* operator where the loop operand is sequenced after the result of earlier iterations.

The Semantics of Gates (see also “Gate (from Fragments)” on page 466)

The gates of a CombinedFragment represent the syntactic interface between the CombinedFragment and its surroundings, which means the interface towards other InteractionFragments.

The only purpose of gates is to define the source and the target of Messages.

Notation

The notation for a CombinedFragment in a Sequence Diagram is a solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle.

More than one operator may be shown in the pentagon descriptor. This is a shorthand for nesting CombinedFragments. This means that **sd strict** in the pentagon descriptor is the same as two CombinedFragments nested, the outermost with **sd** and the inner with **strict**.

The operands of a CombinedFragment are shown by tiling the graph region of the CombinedFragment using dashed horizontal lines to divide it into regions corresponding to the operands.

Strict

Notationally, this means that the vertical coordinate of the contained fragments is significant throughout the whole scope of the CombinedFragment and not only on one Lifeline. The vertical position of an OccurrenceSpecification is given by the vertical position of the corresponding point. The vertical position of other InteractionFragments is given by the topmost vertical position of its bounding rectangle.

Ignore / Consider

See the notation for “ConsiderIgnoreFragment (from Fragments)” on page 458.

Loop

Textual syntax of the loop operand:

*‘**loop**[‘(‘ <minint> [‘,’ <maxint>] ‘)’]*

<minint>	::= non-negative natural
<maxint>	::= non-negative natural (greater than or equal to *<minint>* | ‘*’

‘*’ means infinity.

If only *<minint>* is present, this means that *<minint>* = *<maxint>* = *<integer>*.

If only **loop**, then this means a loop with infinity upper bound and with 0 as lower bound.

Presentation Options for “coregion area”

A notational shorthand for parallel combined fragments are available for the common situation where the order of event occurrences (or other nested fragments) on one Lifeline is insignificant. This means that in a given “coregion” area of a Lifeline all the directly contained fragments are considered separate operands of a parallel combined fragment. See example in Figure 14.12.

Examples

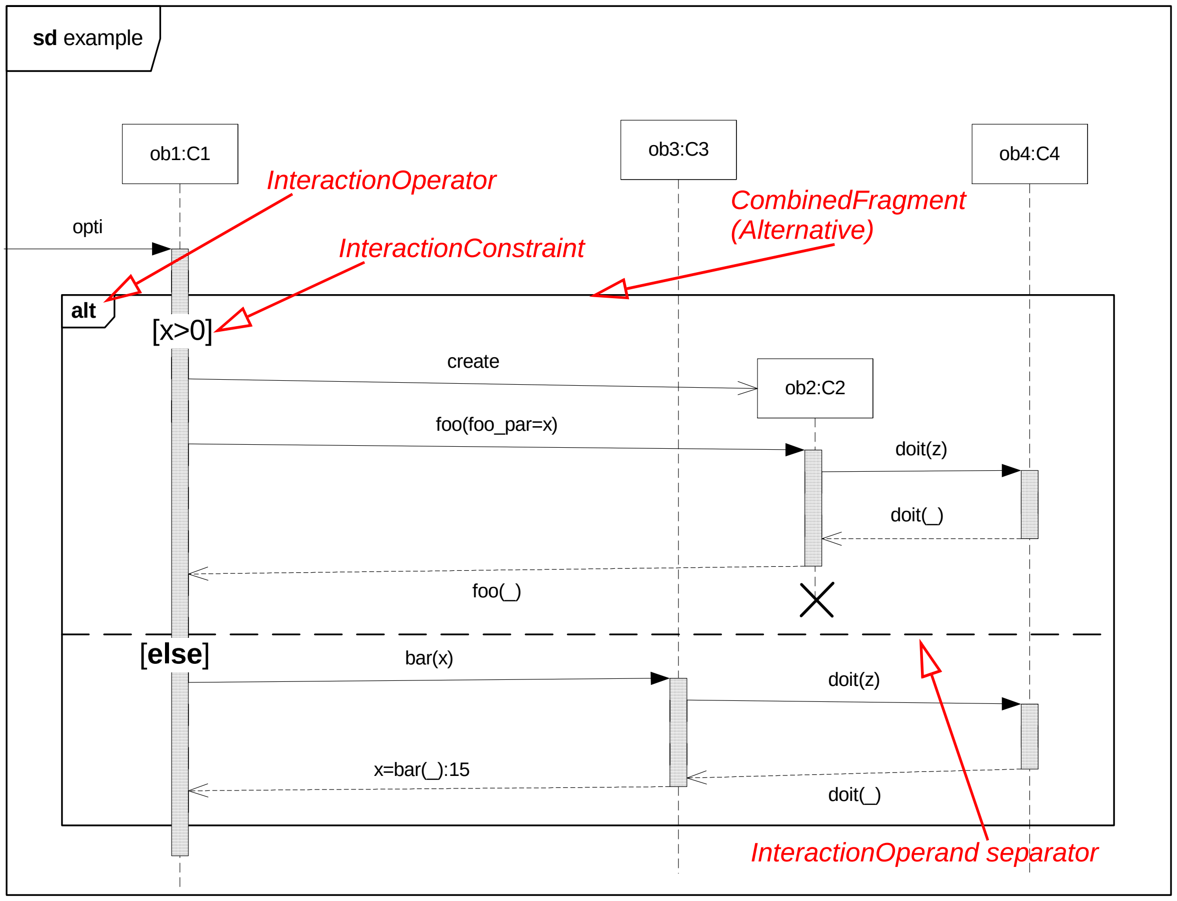


Figure 14.11 - CombinedFragment

Changes from previous UML

This concept was not included in UML 1.x.

14.3.4 ConsiderIgnoreFragment (from Fragments)

Generalizations

- "CombinedFragment (from Fragments)" on page 453

Description

A ConsiderIgnoreFragment is a kind of combined fragment that is used for the consider and ignore cases, which require lists of pertinent messages to be specified.

Attributes

- message : NamedElement [0..*]   The set of messages that apply to this fragment.

Constraints

[1] The interaction operator of a ConsiderIgnoreFragment must be either ‘consider’ or ‘ignore.’
(interactionOperator = #consider) **or** (interactionOperator = #ignore)

[2] The NamedElements must be of a type of element that identifies a message (e.g., an Operation, Reception, or a Signal).
message->forAll(m | m.oclIsKindOf(Operation) **or** m.oclIsKindOf(Reception) **or** m.oclIsKindOf(Signal))

Semantics

The interactionOperator **ignore** designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution. Alternatively, one can understand *ignore* to mean that the message types that are ignored can appear anywhere in the traces.

Conversely, the interactionOperator **consider** designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be *ignored*.

Notation

The notation for ConsiderIgnoreFragment is the same as for all CombinedFragments with the keywords **consider** or **ignore** indicating the operator. The list of messages follows the operand enclosed in a pair of braces (curly brackets) according to the following format:

*(‘**ignore**’ | ‘**consider**’) ‘{‘ <message-name> [‘,’ <message-name>]* ‘}’*

Note that ignore and consider can be combined with other types of operations in a single rectangle (as a shorthand for nested rectangles), such as **assert consider** {msgA, msgB}.

Examples

consider {m, s}: showing that only m and s messages are considered significant.

ignore {q,r}: showing that q and r messages are considered insignificant.

Ignore and consider operations are typically combined with other operations such as “**assert consider** {m, s}.”

Figure 14.24 on page 494 shows an example of consider/ignore fragments.

Changes from previous UML

This concept did not exist in UML 1.x.

14.3.5 Continuation (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

A Continuation is a syntactic way to define continuations of different branches of an Alternative CombinedFragment. Continuation is intuitively similar to labels representing intermediate points in a flow of control.

Attributes

• setting : Boolean	True when the Continuation is at the end of the enclosing InteractionFragment and False when it is in the beginning.

Constraints

[1] Continuations with the same name may only cover the same set of Lifelines (within one Classifier).

[2] Continuations are always global in the enclosing InteractionFragment (e.g., it always covers all Lifelines covered by the enclosing InteractionFragment).

[3] Continuations always occur as the very first InteractionFragment or the very last InteractionFragment of the enclosing InteractionFragment.

Semantics

Continuations have semantics only in connection with Alternative CombinedFragments and (weak) sequencing.

If an InteractionOperand of an Alternative CombinedFragment ends in a Continuation with name (say) X, only InteractionFragments starting with the Continuation X (or no continuation at all) can be appended.

Notation

Continuations are shown with the same symbol as States, but they may cover more than one Lifeline.

Continuations may also appear on flowlines of Interaction Overview Diagrams.

Continuations that are alone in an InteractionFragment are considered to be at the end of the enclosing InteractionFragment.

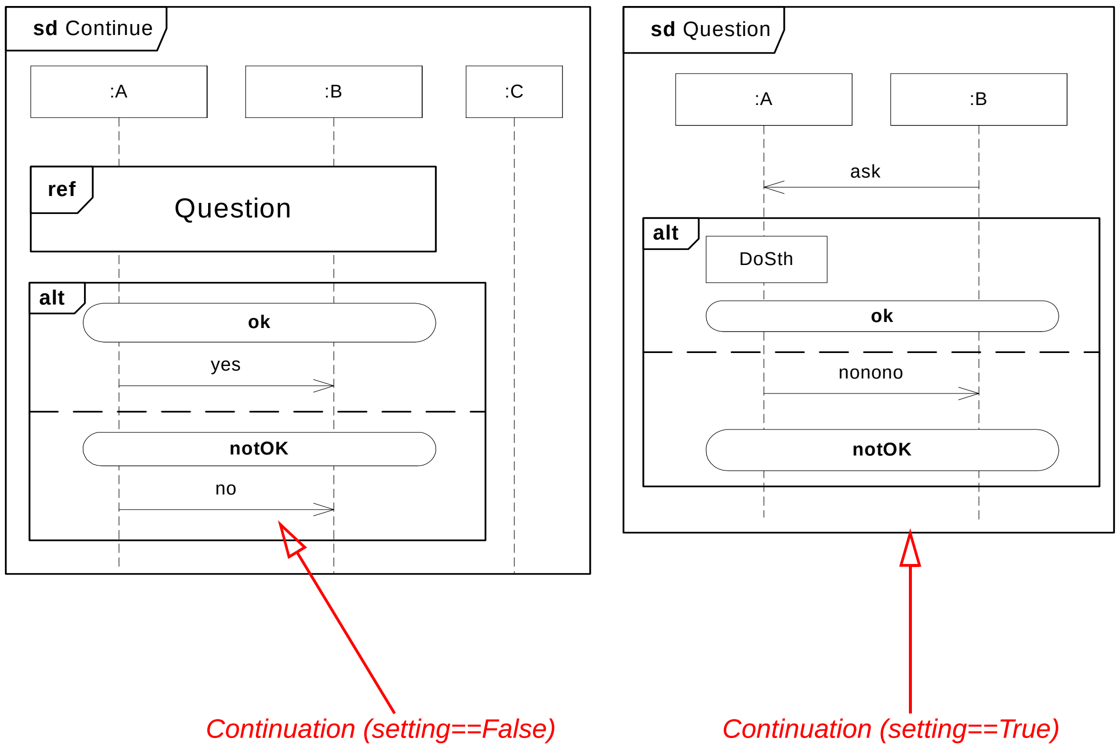


Figure 14.12 - Continuation

The two diagrams in Figure 14.12 are together equivalent to the diagram in Figure 14.13.

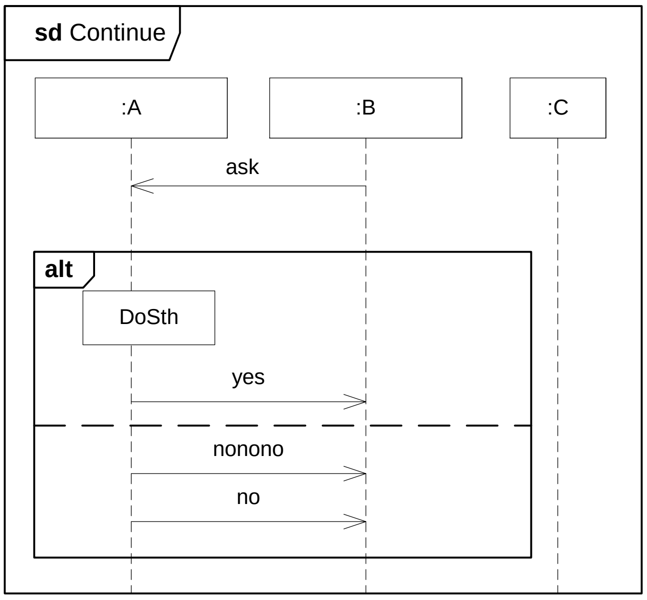


Figure 14.13 - Continuation interpretation

14.3.6 CreationEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

A CreationEvent models the creation of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A creation event represents the creation of an instance. It may result in the subsequent execution of initializations as well as the invocation of the specified classifier behavior (see “Common Behaviors” on page 407).

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.7 DestructionEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

A DestructionEvent models the destruction of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] No other OccurrenceSpecifications may appear below an OccurrenceSpecification that references a DestructionEvent on a given Lifeline in an InteractionOperand.

Semantics

A destruction event represents the destruction of the instance described by the lifeline containing the OccurrenceSpecification that references the destruction event. It may result in the subsequent destruction of other instances that this instance owns by composition (see “Common Behaviors” on page 407).

Notation

The DestructionEvent is depicted by a cross in the form of an X at the bottom of a Lifeline.

Figure 14.14 - DestructionEvent symbol

See example in Figure 14.11.

Changes from previous UML

DestructionEvent is new in UML 2.0.

14.3.8 ExecutionEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

An ExecutionEvent models the start or finish of an execution occurrence.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An execution event represents the start or finish of the execution of an action or a behavior.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.9 ExecutionOccurrenceSpecification (from BasicInteractions)

Generalizations

- "OccurrenceSpecification (from BasicInteractions)" on page 481

Description

An ExecutionOccurrenceSpecification represents moments in time at which actions or behaviors start or finish.

Attributes

No additional attributes

Associations

• event : ExecutionEvent [1]	Redefines the event referenced to be restricted to an execution event.
• execution : ExecutionSpecification [1]	References the execution specification describing the execution that is started or finished at this execution event.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.10 ExecutionSpecification (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 471

Description

An ExecutionSpecification is a specification of the execution of a unit of behavior or action within the Lifeline. The duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications, the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification.

Associations

• start : OccurrenceSpecification[1]	References the OccurrenceSpecification that designates the start of the Action or Behavior.
• finish: OccurrenceSpecification[1]	References the OccurrenceSpecification that designates the finish of the Action or Behavior.

Constraints

[1] The startEvent and the finishEvent must be on the same Lifeline.

start.lifeline = finish.lifeline

Semantics

The trace semantics of Interactions merely see an Execution as the trace <start, finish>. There may be occurrences between these. Typically the start occurrence and the finish occurrence will represent OccurrenceSpecifications such as a receive OccurrenceSpecification (of a Message) and the send OccurrenceSpecification (of a reply Message).

Notation

ExecutionOccurences are represented as thin rectangles (grey or white) on the lifeline (see “Lifeline (from BasicInteractions, Fragments)” on page 475).

We may also represent an ExecutionSpecification by a wider labeled rectangle, where the label usually identifies the action that was executed. An example of this can be seen in Figure 14.13 on page 461.

ExecutionSpecifications that refer to atomic actions such as reading attributes of a Signal (conveyed by the Message), the Action symbol may be associated with the reception OccurrenceSpecification with a line in order to emphasize that the whole Action is associated with only one OccurrenceSpecification (and start and finish associations refer the very same OccurrenceSpecification).

Overlapping execution occurrences on the same lifeline are represented by overlapping rectangles as shown in Figure 14.15.

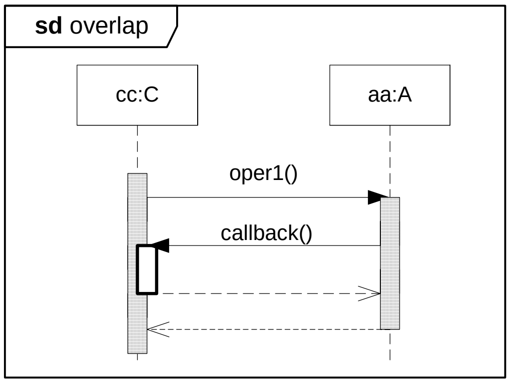


Figure 14.15 - Overlapping execution occurrences

14.3.11 Gate (from Fragments)

Generalizations

- “MessageEnd (from BasicInteractions)” on page 479

Description

A Gate is a connection point for relating a Message outside an InteractionFragment with a Message inside the InteractionFragment.

Gate is a specialization of MessageEnd.

Gates are connected through Messages. A Gate is actually a representative of an OccurrenceSpecification that is not in the same scope as the Gate.

Gates play different roles: we have formal gates on Interactions, actual gates on InteractionUses, expression gates on CombinedFragments.

Constraints

[1] The message leading to/from an actualGate of an InteractionUse must correspond to the message leading from/to the formalGate with the same name of the Interaction referenced by the InteractionUse.

[2] The message leading to/from an (expression) Gate within a CombinedFragment must correspond to the message leading from/to the CombinedFragment on its outside.

Semantics

The gates are named either explicitly or implicitly. Gates may be identified either by name (if specified), or by a constructed identifier formed by concatenating the direction of the message and the message name (e.g., *out_CardOut*). The gates and the messages between gates have one purpose, namely to establish the concrete sender and receiver for every message.

Notation

Gates are just points on the frame, the ends of the messages. They may have an explicit name (see Figure 14.19).

The same gate may appear several times in the same or different diagrams.

14.3.12 GeneralOrdering (from BasicInteractions)

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 93

Description

A GeneralOrdering represents a binary relation between two OccurrenceSpecifications, to describe that one OccurrenceSpecification must occur before the other in a valid trace. This mechanism provides the ability to define partial orders of OccurrenceSpecifications that may otherwise not have a specified order.

A GeneralOrdering is a specialization of NamedElement.

A GeneralOrdering may appear anywhere in an Interaction, but only between OccurrenceSpecifications.

Associations

• before: OccurrenceSpecification[1]	The OccurrenceSpecification referred comes before the OccurrenceSpecification referred by *after*.
• after:OccurrenceSpecification[1]	The OccurrenceSpecification referred comes after the OccurrenceSpecification referred by *before*.

Semantics

A GeneralOrdering is introduced to restrict the set of possible sequences. A partial order of OccurrenceSpecifications is defined by a set of GeneralOrdering.

Notation

A GeneralOrdering is shown by a dotted line connecting the two OccurrenceSpecifications. The direction of the relation from the *before* to the *after* is given by an arrowhead placed somewhere in the middle of the dotted line (i.e., not at the endpoint).

14.3.13 Interaction (from BasicInteraction, Fragments)

Generalizations

- "Behavior (from BasicBehaviors)" on page 416
- "InteractionFragment (from BasicInteractions, Fragments)" on page 471

Description

An interaction is a unit of behavior that focuses on the observable exchange of information between ConnectableElements.

An Interaction is a specialization of InteractionFragment and of Behavior.

Associations

• formalGate: Gate[*]	Specifies the gates that form the message interface between this Interaction and any InteractionUses that reference it.
• lifeline: LifeLine[0..*]	Specifies the participants in this Interaction.
• event:MessageEnd[*]	MessageEnds (e.g., OccurrenceSpecifications or Gates) owned by this Interaction.
• message:Message[*]	The Messages contained in this Interaction.
• fragment:InteractionFragment[*]	The ordered set of fragments in the Interaction.
• action:Action[*]	Actions owned by the Interaction. See "ActionExecutionSpecification (from BasicInteractions)" on page 452.

Semantics

Interactions are units of behavior of an enclosing Classifier. Interactions focus on the passing of information with Messages between the ConnectableElements of the Classifier.

The semantics of an Interaction is given as a pair of sets of traces. The two trace sets represent valid traces and invalid traces. The union of these two sets need not necessarily cover the whole universe of traces. The traces that are not included are not described by this Interaction at all, and we cannot know whether they are valid or invalid.

A trace is a sequence of event occurrences, each of which is described by an OccurrenceSpecification in a model. The semantics of Interactions are compositional in the sense that the semantics of an Interaction is mechanically built from the semantics of its constituent InteractionFragments. The constituent InteractionFragments are ordered and combined by the seq operation (weak sequencing) as explained in "Weak Sequencing" on page 454.

The invalid set of traces are associated only with the use of a Negative CombinedInteraction. For simplicity we describe only valid traces for all other constructs.

As Behavior an Interaction is generalizable and redefineable. Specializing an Interaction is simply to add more traces to those of the original. The traces defined by the specialization is combined with those of the inherited Interaction with a union.

The classifier owning an Interaction may be specialized, and in the specialization the Interaction may be redefined. Redefining an Interaction simply means to exchange the redefining Interaction for the redefined one, and this exchange takes effect also for InteractionUses within the supertype of the owner. This is similar to redefinition of other kinds of Behavior.

Basic trace model: The semantics of an Interaction is given by a pair [P, I] where P is the set of valid traces and I is the set of invalid traces. $P \cup I$ need not be the whole universe of traces.

A trace is a sequence of event occurrences denoted <e1, e2, ... , en>.

An event occurrence will also include information about the values of all relevant objects at this point in time.

Each construct of Interactions (such as CombinedFragments of different kinds) are expressed in terms of how it relates to a pair of sets of traces. For simplicity we normally refer only to the set of valid traces as these traces are those mostly modeled.

Two Interactions are equivalent if their pair of trace-sets are equal.

Relation of trace model to execution model: In Chapter 13, "Common Behaviors" we find an Execution model, and this is how the Interactions Trace Model relates to the Execution model.

An Interaction is an Emergent Behavior.

An InvocationOccurrence in the Execution model corresponds with an (event) Occurrence in a trace. Occurrences are modeled in an Interaction by OccurrenceSpecifications. Normally in Interaction the action leading to the invocation as such is not described (such as the sending action). However, if it is desirable to go into details, a Behavior (such as an Activity) may be associated with an OccurrenceSpecification. An occurrence in Interactions is normally interpreted to take zero time. Duration is always between occurrences.

Likewise a ReceiveOccurrence in the Execution model is modeled by an OccurrenceSpecification. Similarly the detailed actions following immediately from this reception is often omitted in Interactions, but may also be described explicitly with a Behavior associated with that OccurrenceSpecification.

A Request in the Execution model is modeled by the Message in Interactions.

An Execution in the Execution model is modeled by an ExecutionSpecification in Interactions. An Execution is defined in the trace by two Occurrences, one at the start and one at the end. This corresponds to the StartOccurrence and the CompletionOccurrence of the Execution model.

Notation

The notation for an Interaction in a Sequence Diagram is a solid-outline rectangle. The keyword **sd** followed by the Interaction name and parameters is in a pentagon in the upper left corner of the rectangle. The notation within this rectangular frame comes in several forms: Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams, and Timing Diagrams.

The notation within the pentagon descriptor follows the general notation for the name of Behaviors. In addition the Interaction Overview Diagrams may include a list of Lifelines through a lifeline-clause as shown in Figure 14.28. The list of lifelines is simply a listing of the Lifelines involved in the Interaction. An Interaction Overview Diagram does not in itself show the involved lifelines even though the lifelines may occur explicitly within inline Interactions in the graph nodes.

An Interaction diagram may also include definitions of local attributes with the same syntax as attributes in general are shown within class symbol compartments. These attribute definitions may appear near the top of the diagram frame or within note symbols at other places in the diagram.

Please refer to Section 14.4 to see examples of notation for Interactions.

Examples

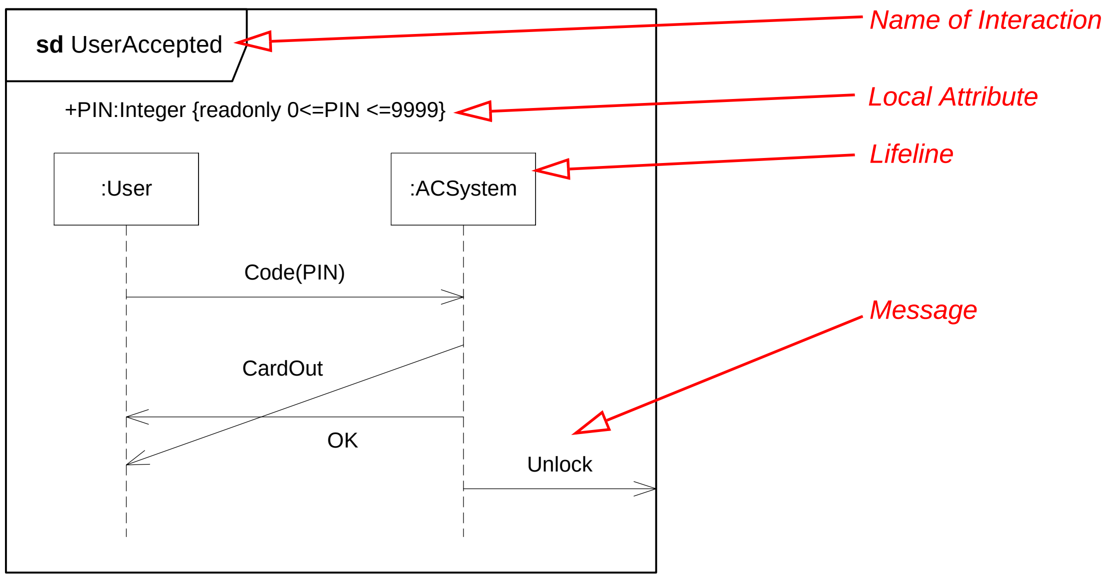


Figure 14.16 - An example of an Interaction in the form of a Sequence Diagram

The example in Figure 14.16 shows three messages communicated between two (anonymous) lifelines of types *User* and *ACSystem*. The message *CardOut* overtakes the message *OK* in the way that the receiving event occurrences are in the opposite order of the sending EventOccurrences. Such communication may occur when the messages are asynchronous. Finally a fourth message is sent from the *ACSystem* to the environment through a gate with implicit name *out_Unlock*. The local attribute *PIN* of *UserAccepted* is declared near the diagram top. It could have been declared in a Note somewhere else in the diagram.

Changes from previous UML

Interactions are now contained within Classifiers and not only within Collaborations. Their participants are modeled by Lifelines instead of ClassifierRoles.

14.3.14 InteractionConstraint (from Fragments)

Generalizations

- "Constraint (from Kernel)" on page 54

Description

An InteractionConstraint is a boolean expression that guards an operand in a CombinedFragment.

InteractionConstraint is a specialization of Constraint.

Furthermore the InteractionConstraint contains two expressions designating the minimum and maximum number of times a loop CombinedFragment should execute.

Associations

• minint: ValueSpecification[0..1]	The minimum number of iterations of a loop.
• maxint: ValueSpecification[0..1]	The maximum number of iterations of a loop.

Constraints

[1] The dynamic variables that take part in the constraint must be owned by the ConnectableElement corresponding to the covered Lifeline.

[2] The constraint may contain references to global data or write-once data.

[3] Minint/maxint can only be present if the InteractionConstraint is associated with the operand of a loop CombinedFragment.

[4] If minint is specified, then the expression must evaluate to a non-negative integer.

[5] If maxint is specified, then the expression must evaluate to a positive integer.

[6] If maxint is specified, then minint must be specified and the evaluation of maxint must be >= the evaluation of minint.

Semantics

InteractionConstraints are always used in connection with CombinedFragments, see "CombinedFragment (from Fragments)" on page 453.

Notation

An InteractionConstraint is shown in square brackets covering the lifeline where the first event occurrence will occur, positioned above that event, in the containing Interaction or InteractionOperand.

*<interactionconstraint> ::= ['[' (<Boolean-expression' | '**else**') ']']*

When the InteractionConstraint is omitted, true is assumed.

Please refer to an example of InteractionConstraints in Figure 14.11 on page 458.

14.3.15 InteractionFragment (from BasicInteractions, Fragments)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 93.

Description

InteractionFragment is an abstract notion of the most general interaction unit. An interaction fragment is a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself.

InteractionFragment is an abstract class and a specialization of NamedElement.

Associations

• enclosingOperand: InteractionOperand[0..1]	The operand enclosing this InteractionFragment (they may nest recursively).
• covered : Lifeline[*]	References the Lifelines that the InteractionFragment involves.
• generalOrdering:GeneralOrdering[*]	The general ordering relationships contained in this fragment.
• enclosingInteraction: Interaction[0..1]	The Interaction enclosing this InteractionFragment.

Semantics

The semantics of an InteractionFragment is a pair of set of traces. See "Interaction (from BasicInteraction, Fragments)" for explanation of how to calculate the traces.

Notation

There is no general notation for an InteractionFragment. The specific subclasses of InteractionFragment will define their own notation.

Changes from previous UML

This concept did not appear in UML 1.x.

14.3.16 InteractionOperand (from Fragments)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 471
- "Namespace (from Kernel)" on page 95

Description

An InteractionOperand is contained in a CombinedFragment. An InteractionOperand represents one operand of the expression given by the enclosing CombinedFragment.

An InteractionOperand is an InteractionFragment with an optional guard expression. An InteractionOperand may be guarded by an InteractionConstraint. Only InteractionOperands with a guard that evaluates to true at this point in the interaction will be considered for the production of the traces for the enclosing CombinedFragment.

InteractionOperand contains an ordered set of InteractionFragments.

In Sequence Diagrams these InteractionFragments are ordered according to their geometrical position vertically. The geometrical position of the InteractionFragment is given by the topmost vertical coordinate of its contained EventOccurrences or symbols.

Associations

• fragment: InteractionFragment[*]	The fragments of the operand.
• guard: InteractionConstraint[0..1]	Constraint of the operand.

Constraints

[1] The guard must be placed directly prior to (above) the EventOccurrence that will become the first EventOccurrence within this InteractionOperand.

[2] The guard must contain only references to values local to the Lifeline on which it resides, or values global to the whole Interaction (See "InteractionConstraint (from Fragments)" on page 470).

Semantics

Only InteractionOperands with true guards are included in the calculation of the semantics. If no guard is present, this is taken to mean a true guard.

The semantics of an InteractionOperand is given by its constituent InteractionFragments combined by the implicit *seq* operation. The seq operator is described in "CombinedFragment (from Fragments)" on page 453.

Notation

InteractionOperands are separated by a dashed horizontal line. The InteractionOperands together make up the framed CombinedFragment.

Within an InteractionOperand of a Sequence Diagram the order of the InteractionFragments are given simply by the topmost vertical position.

See Figure 14.11 for examples of InteractionOperand.

14.3.17 InteractionOperator (from Fragments)

Generalizations

None

Description

Interaction Operator is an enumeration designating the different kinds of operators of CombinedFragments. The InteractionOperand defines the type of operator of a CombinedFragment. The literal values of this enumeration are:

- alt
- opt
- par
- loop
- critical
- neg

- assert
- strict
- seq
- ignore
- consider

Semantics

The value of the interactionOperator is significant for the semantics of “CombinedFragment (from Fragments)” on page 453.

Notation

The value of the InteractionOperand is given as text in a small compartment in the upper left corner of the CombinedFragment frame. See Figure 14.11 on page 458 for examples of InteractionOperator.

14.3.18 InteractionUse (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones.

It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification.

Description

InteractionUse is a specialization of InteractionFragment.

An InteractionUse has a set of actual gates that must match the formal gates of the referenced Interaction.

Associations

• refersTo: Interaction[1]	Refers to the Interaction that defines its meaning.
• argument:InputPin[*]	The actual arguments of the Interaction.
• actualGate:Gate[*]	The actual gates of the InteractionUse.

Constraints

[1] Actual Gates of the InteractionUse must match Formal Gates of the referred Interaction. Gates match when their names are equal.

[2] The InteractionUse must cover all Lifelines of the enclosing Interaction that appear within the referred Interaction.

[3] The arguments of the InteractionUse must correspond to parameters of the referred Interaction.

[4] The arguments must only be constants, parameters of the enclosing Interaction or attributes of the classifier owning the enclosing Interaction.

Semantics

The semantics of the InteractionUse is the set of traces of the semantics of the referred Interaction where the gates have been resolved as well as all generic parts having been bound such as the arguments substituting the parameters.

Notation

The InteractionUse is shown as a CombinedFragment symbol where the operator is called *ref*. The complete syntax of the name (situated in the InteractionUse area) is:

<name> ::=[<attribute-name> '='] [<collaboration-use> '.'] <interaction-name>
['(' <io-argument> [',' <io-oargument>] ')'] [':' <return-value>*
*<io-argument> ::= <in-argument> | **'out'** <out-argument>]*

The *<attribute-name>* refers to an attribute of one of the lifelines in the Interaction.

<collaboration-use> is an identification of a collaboration use that binds lifelines of a collaboration. The interaction name is in that case within that collaboration. See example of collaboration uses in Figure 14.25.

The *io-arguments* are most often arguments of IN-parameters. If there are OUT- or INOUT-parameters and the output value is to be described, this can be done following an **out** keyword.

The syntax of *argument* is explained in the notation section of Messages ("Message (from BasicInteractions)" on page 477).

If the InteractionUse returns a value, this may be described following a colon at the end of the clause.

Examples

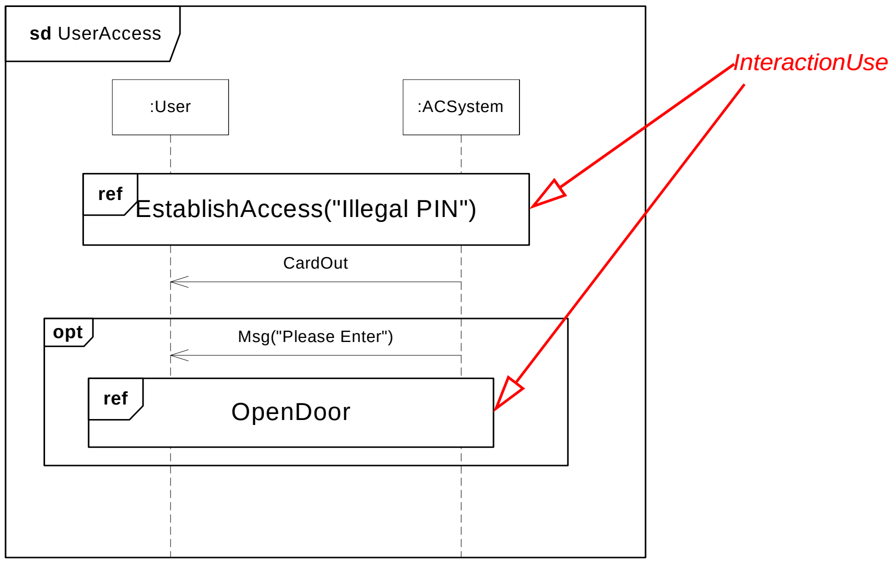
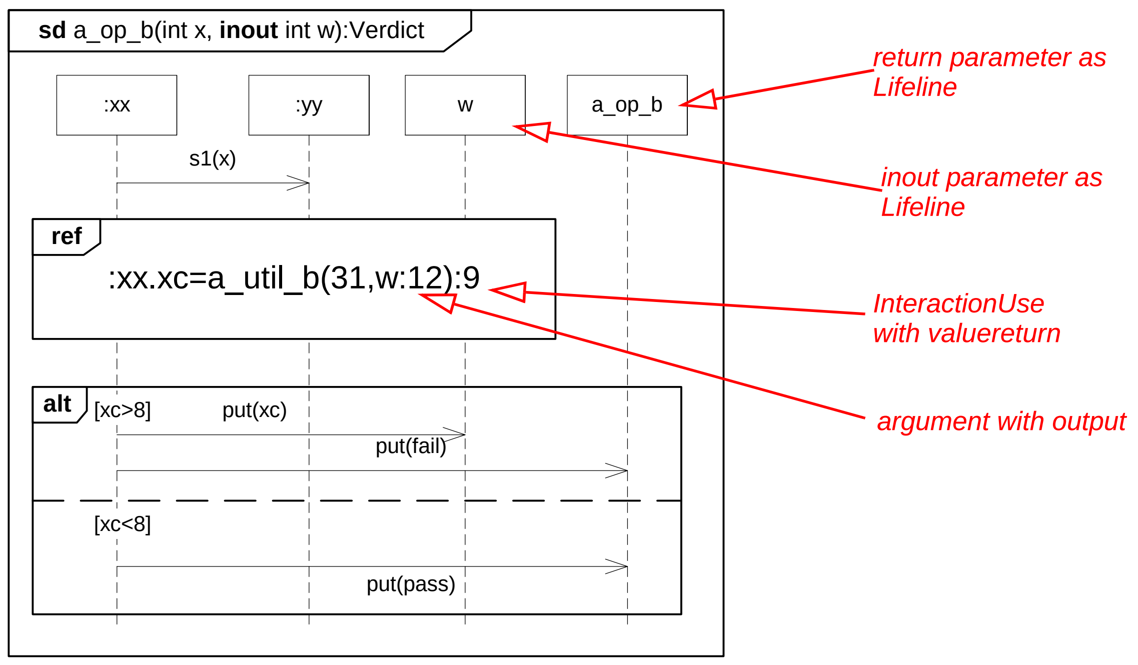


Figure 14.17 - InteractionUse

In Figure 14.17 we show an *InteractionUse* referring the Interaction *EstablishAccess* with (input) argument "Illegal PIN." Within the optional CombinedFragment there is another InteractionUse without arguments referring *OpenDoor*.



sd a_op_b(int x, inout int w):Verdict
:xx
:yy
w
a_op_b
s1(x)
ref
:xx.xc=a_util_b(31,w:12):9
alt
[xc>8]
put(xc)
put(fail)
[xc<8]
put(pass)
return parameter as Lifeline
inout parameter as Lifeline
InteractionUse with valuereturn
argument with output

Figure 14.18 - InteractionUse with value return

In Figure 14.18 we have a more advanced Interaction that models a behavior returning a *Verdict* value. The return value from the Interaction is shown as a separate Lifeline *a_op_b*. Inside the Interaction there is an InteractionUse referring *a_util_b* with value return to the attribute *xc* of *:xx* with the value 9, and with inout parameter where the argument is w with returning out-value 12.

Changes from previous UML

InteractionUse was not a concept in UML 1.x.

14.3.19 Lifeline (from BasicInteractions, Fragments)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 93.

Description

A lifeline represents an individual participant in the Interaction. While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity.

Lifeline is a specialization of NamedElement.

If the referenced ConnectableElement is multivalued (i.e, has a multiplicity > 1), then the Lifeline may have an expression (the 'selector') that specifies which particular part is represented by this Lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued ConnectableElement is chosen.

Associations

• selector : Expression[0..1]	If the referenced ConnectableElement is multivalued, then this specifies the specific individual part within that set.
• interaction: Interaction[1]	References the Interaction enclosing this Lifeline.
• represents: ConnectableElement[0..1]	References the ConnectableElement within the classifier that contains the enclosing interaction.
• decomposedAs : PartDecomposition[0..1]	References the Interaction that represents the decomposition.

Constraints

[1] If two (or more) InteractionUses within one Interaction, refer to Interactions with common Lifelines, those Lifelines must also appear in the Interaction with the InteractionUses. By ‘common Lifelines’ we mean Lifelines with the same selector and represents associations.

[2] The selector for a Lifeline must only be specified if the referenced Part is multivalued.

(self.selector->isEmpty() **implies not** self.represents.isMultivalued()) **or**

(**not** self.selector->isEmpty() **implies** self.represents.isMultivalued())

[3] The classifier containing the referenced ConnectableElement must be the same classifier, or an ancestor, of the classifier that contains the interaction enclosing this lifeline.

```
if (represents->notEmpty()) then
   (if selector->notEmpty() then represents.isMultivalued() else not represents.isMultivalued())
```

Semantics

The order of OccurrenceSpecifications along a Lifeline is significant denoting the order in which these OccurrenceSpecifications will occur. The absolute distances between the OccurrenceSpecifications on the Lifeline are, however, irrelevant for the semantics.

The semantics of the Lifeline (within an Interaction) is the semantics of the Interaction selecting only OccurrenceSpecifications of this Lifeline.

Notation

A Lifeline is shown using a symbol that consists of a rectangle forming its “head” followed by a vertical line (which may be dashed) that represents the lifetime of the participant. Information identifying the lifeline is displayed inside the rectangle in the following format:

<lifelineident> ::= ([<connectable-element-name>[‘[‘ <selector> ‘]’]] [: <class_name>] [decomposition]) | ***‘self’***
<selector> ::= <expression>
<decomposition> ::= ***‘ref’*** *<interactionident> [‘****strict’****]*

where *<class-name>* is the type referenced by the represented ConnectableElement. Note that, although the syntax allows it, *<lifelineident>* cannot be empty.

The Lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Often the head is a white rectangle containing the name.

If the name is the keyword **self**, then the lifeline represents the object of the classifier that encloses the Interaction that owns the Lifeline. Ports of the encloser may be shown separately even when *self* is included.

To depict method activations we apply a thin grey or white rectangle that covers the Lifeline line.

Examples

See Figure 14.16 where the Lifelines are pointed to.

See Figure 14.11 to see method activations.

Changes from previous UML

Lifelines are basically the same concept as before in UML 1.x.

14.3.20 Message (from BasicInteractions)

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 93

Description

A Message defines a particular communication between Lifelines of an Interaction.

A Message is a NamedElement that defines one specific kind of communication in an Interaction. A communication can be, for example, raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication given by the dispatching ExecutionSpecification, but also the sender and the receiver.

A Message associates normally two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification.

Attributes

• messageKind:MessageKind	The derived kind of the Message (*complete*, *lost*, *found*, or *unknown*).
• messageSort:MessageSort	The sort of communication reflected by the Message.

Associations

• interaction:Interaction[1]	The enclosing Interaction owning the Message.
• sendEvent : MessageEnd[0..1]	References the Sending of the Message.
• receiveEvent: MessageEnd[0..1]	References the Receiving of the Message.
• connector: Connector[0..1]	The Connector on which this Message is sent.
• argument:ValueSpecification[*]	The arguments of the Message.
• /signature:NamedElement[0..1]	The definition of the type or signature of the Message (depending on its kind). The associated named element is derived from the message end that constitutes the sending or receiving message event. If both a sending event and a receiving message event are present, the signature is obtained from the sending event.

Constraints

[1] If the sending MessageEvent and the receiving MessageEvent of the same Message are on the same Lifeline, the sending MessageEvent must be ordered before the receiving MessageEvent.

[2] The signature must either refer to an Operation (in which case messageSort is either synchCall or asynchCall) or a Signal (in which case messageSort is asynchSignal). The name of the NamedElement referenced by signature must be the same as that of the Message.

[3] In the case when the Message signature is an Operation, the arguments of the Message must correspond to the parameters of the Operation. A Parameter corresponds to an Argument if the Argument is of the same Class or a specialization of that of the Parameter.

[4] In the case when the Message signature is a Signal, the arguments of the Message must correspond to the attributes of the Signal. A Message Argument corresponds to a Signal Attribute if the Argument is of the same Class or a specialization of that of the Attribute.

[5] Arguments of a Message must only be:
i) attributes of the sending lifeline.
ii) constants.
iii) symbolic values (which are wildcard values representing any legal value).
iv) explicit parameters of the enclosing Interaction.
v) attributes of the class owning the Interaction.

[6] Messages cannot cross boundaries of CombinedFragments or their operands.

[7] If the MessageEnds are both OccurrenceSpecifications, then the connector must go between the Parts represented by the Lifelines of the two MessageEnds.

Semantics

The semantics of a *complete* Message is simply the trace <sendEvent, receiveEvent>.

A *lost* message is a message where the sending event occurrence is known, but there is no receiving event occurrence. We interpret this to be because the message never reached its destination. The semantics is simply the trace <sendEvent>.

A *found* message is a message where the receiving event occurrence is known, but there is no (known) sending event occurrence. We interpret this to be because the origin of the message is outside the scope of the description. This may for example be noise or other activity that we do not want to describe in detail. The semantics is simply the trace <receiveEvent>.

A Message reflects either an Operation call and start of execution - or a sending and reception of a Signal.

When a Message represents an Operation invocation, the arguments of the Message are the arguments of the CallOperationAction on the sending Lifeline as well as the arguments of the CallEvent occurrence on the receiving Lifeline.

When a Message represents a Signal, the arguments of the Message are the arguments of the SendAction on the sending Lifeline and on the receiving Lifeline the arguments are available in the SignalEvent.

If the Message represents a CallAction, there will normally be a reply message from the called Lifeline back to the calling lifeline before the calling Lifeline will proceed.

Notation

A message is shown as a line from the sender message end to the receiver message end. The form of the line or arrowhead reflect properties of the message:

- Asynchronous Messages have an open arrow head.
- Synchronous Messages typically represent method calls and are shown with a filled arrow head. The reply message from a method has a dashed line.
- Object creation Message has a dashed line with an open arrow.
- Lost Messages are described as a small black circle at the arrow end of the Message.

- Found Messages are described as a small black circle at the starting end of the Message.
- On Communication Diagrams, the Messages are decorated by a small arrow along the connector close to the Message name and sequence number in the direction of the Message.

Syntax for the Message name is the following:

<messageident> ::= ([<attribute> '='] <signal-or-operation-name> ['(' [<argument> [','<argument>] ')']*
[':' <return-value>]) | ''*
<argument> ::= (<[parameter-name> '='] <argument-value>) | (<attribute> '=' <out-parameter-name>
[':' <argument-value>] | ' -'

Messageident equaling '*' is a shorthand for more complex alternative CombinedFragment to represent a message of any type. This is to match asterisk triggers in State Machines.

Return-value and attribute assignment are used only for reply messages. Attribute assignment is a shorthand for including the Action that assigns the return-value to that attribute. This holds both for the possible return value of the message (the return value of the associated operation), and the out values of (in)out parameters.

When the argument list contains only argument-values, all the parameters must be matched either by a value or by a dash (-). If parameter-names are used to identify the argument-value, then arguments may freely be omitted. Omitted parameters get an unknown argument-value.

Examples

In Figure 14.16 we see only asynchronous Messages. Such Messages may overtake each other.

In Figure 14.11 we see method calls that are synchronous accompanied by replies. We also see a Message that represents the creation of an object.

In Figure 14.27 we see how Messages are denoted in Communication Diagrams.

Examples of syntax:

mymessage(14, - , 3.14, "hello") *// second argument is undefined*
v=mymsg(16, variab):96 *// this is a reply message carrying the return value 96 assigning it to v*
mymsg(myint=16) *// the input parameter 'myint' is given the argument value 16*

See Figure 14.11 for a number of different applications of the textual syntax of message identification.

Changes from previous UML

Messages may have Gates on either end.

14.3.21 MessageEnd (from BasicInteractions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 93.

Description

A MessageEnd is an abstract NamedElement that represents what can occur at the end of a Message.

Associations

- message : Message [1] References a Message.

Semantics

Subclasses of MessageEnd define the specific semantics appropriate to the concept they represent.

14.3.22 MessageKind (from BasicInteractions)

This is an enumerated type that identifies the type of message.

Generalizations

None

Description

MessageSort is an enumeration of the following values:

- complete = sendEvent and receiveEvent are present.
- lost = sendEvent present and receiveEvent absent.
- found = sendEvent absent and receiveEvent present.
- unknown = sendEvent and receiveEvent absent (should not appear).

14.3.23 MessageOccurrenceSpecification (from BasicInteractions)

Generalizations

- “MessageEnd (from BasicInteractions)” on page 479
- “OccurrenceSpecification (from BasicInteractions)” on page 481

Description

Specifies the occurrence of message events, such as sending and receiving of signals or invoking or receiving of operation calls. A message occurrence specification is a kind of message end. Messages are generated either by synchronous operation calls or asynchronous signal sends. They are received by the execution of corresponding accept event actions.

Attributes

No additional attributes

Associations

- event : MessageEvent [1] Redefines the event referenced to be restricted to a message event.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.24 MessageSort (from BasicInteractions)

This is an enumerated type that identifies the type of communication action that was used to generate the message.

Generalizations

None

Description

MessageSort is an enumeration of the following values:

- synchCall - The message was generated by a synchronous call to an operation.
- asynchCall - The message was generated by an asynchronous call to an operation (i.e., a CallAction with "isSynchronous = false").
- asynchSignal - The message was generated by an asynchronous send action.

14.3.25 OccurrenceSpecification (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 471

Description

An OccurrenceSpecification is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions.

OccurrenceSpecifications are ordered along a Lifeline.

The namespace of an OccurrenceSpecification is the Interaction in which it is contained.

Associations

• event : Event [1]	References a specification of the occurring event.
• covered: Lifeline[1]	References the Lifeline on which the OccurrenceSpecification appears. Redefines InteractionFragment.covered.
• toBefore:GeneralOrdering[*]	References the GeneralOrderings that specify EventOcurrences that must occur before this OccurrenceSpecification.
• toAfter: GeneralOrdering[*]	References the GeneralOrderings that specify EventOcurrences that must occur after this OccurrenceSpecification.

Semantics

The semantics of an OccurrenceSpecification is just the trace of that single OccurrenceSpecification.

The understanding and deeper meaning of the OccurrenceSpecification is dependent upon the associated Message and the information that it conveys.

Notation

OccurrenceSpecifications are merely syntactic points at the ends of Messages or at the beginning/end of an ExecutionSpecification.

Examples

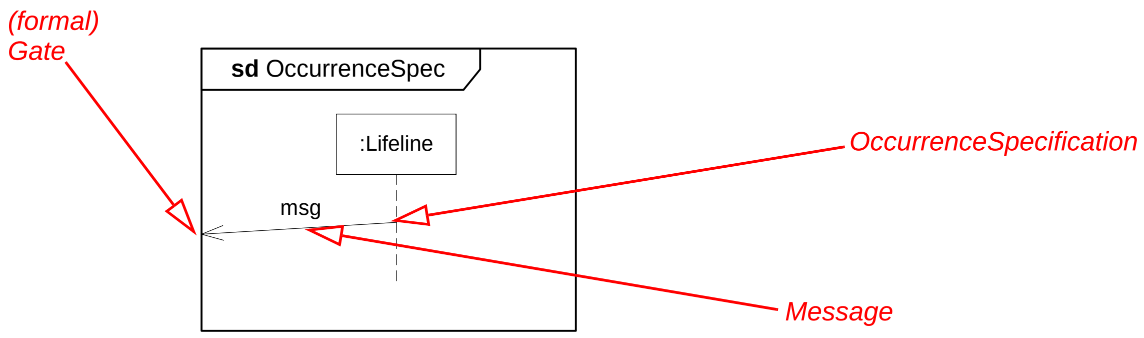


Figure 14.19 - OccurrenceSpecification

14.3.26 PartDecomposition (from Fragments)

Generalizations

- "InteractionUse (from Fragments)" on page 473

Description

PartDecomposition is a description of the internal interactions of one Lifeline relative to an Interaction.

A Lifeline has a class associated as the type of the ConnectableElement that the Lifeline represents. That class may have an internal structure and the PartDecomposition is an Interaction that describes the behavior of that internal structure relative to the Interaction where the decomposition is referenced.

A PartDecomposition is a specialization of InteractionUse. It associates with the ConnectableElement that it decomposes.

Constraints

[1] PartDecompositions apply only to Parts that are Parts of Internal Structures not to Parts of Collaborations.

[2] Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Within X there is a sequence of constructs along L (such constructs are CombinedFragments, InteractionUse and (plain) OccurrenceSpecifications). Then a corresponding sequence of constructs must appear within D, matched one-to-one in the same order.
i) CombinedFragment covering L are matched with an extra-global CombinedFragment in D.
ii) An InteractionUse covering L are matched with a global (i.e., covering all Lifelines) InteractionUse in D.
iii) A plain OccurrenceSpecification on L is considered an actualGate that must be matched by a formalGate of D.

[3] Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Assume also that there is within X an InteractionUse (say) U that covers L. According to the constraint above U will have a counterpart CU within D. Within the Interaction referenced by U, L should also be decomposed, and the decomposition should reference CU. (This rule is called commutativity of decomposition.)

Semantics

Decomposition of a lifeline within one Interaction by an Interaction (owned by the type of the Lifeline's associated ConnectableElement), is interpreted exactly as an InteractionUse. The messages that go into (or go out from) the decomposed lifeline are interpreted as actual gates that are matched by corresponding formal gates on the decomposition.

Since the decomposed Lifeline is interpreted as an InteractionUse, the semantics of a PartDecomposition is the semantics of the Interaction referenced by the decomposition where the gates and parameters have been matched.

That a CombinedFragment is extra-global depicts that there is a CombinedFragment with the same operator covering the decomposed Lifeline in its Interaction. The full understanding of that (higher level) CombinedFragment must be acquired through combining the operands of the decompositions operand by operand.

Notation

PartDecomposition is designated by a referencing clause in the head of the Lifeline as can be seen in the notation section of "Lifeline (from BasicInteractions, Fragments)" on page 475. See also Figure 14.20.

If the part decomposition is denoted inline under the decomposed lifeline and the decomposition clause is the keyword "strict," this indicates that the constructs on all sub lifelines within the inline decomposition are ordered in strict sequence (see "CombinedFragment (from Fragments)" on page 453 on the "strict" operator).

Extraglobal CombinedFragments have their rectangular frame go outside the boundaries of the decomposition Interaction.

Style Guidelines

The name of an Interaction that is involved in decomposition would benefit from including in the name, the name of the type of the Part being decomposed and the name of the Interaction originating the decomposition. This is shown in Figure 14.20 where the decomposition is called *AC_UserAccess* where 'AC' refers to *ACSystem*, which is the type of the Lifeline and *UserAccess* is the name of the Interaction where the decomposed lifeline is contained.

Examples

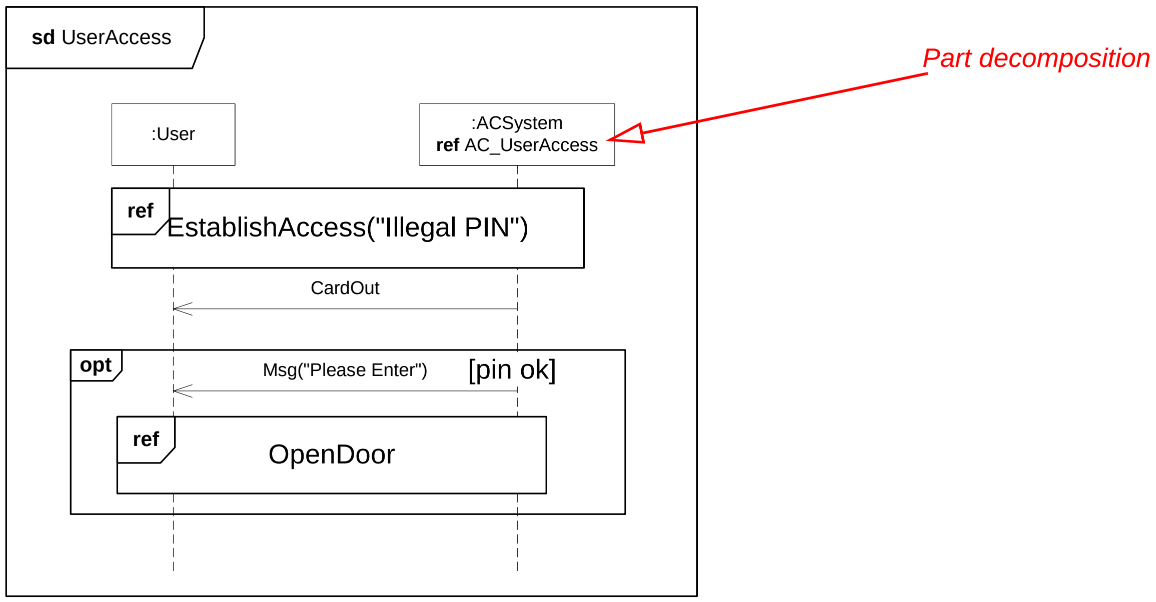


Figure 14.20 - Part Decomposition - the decomposed part

In Figure 14.20 we see how *ACSystem* within *UserAccess* is to be decomposed to *AC_UserAccess*, which is an Interaction owned by class *ACSystem*.

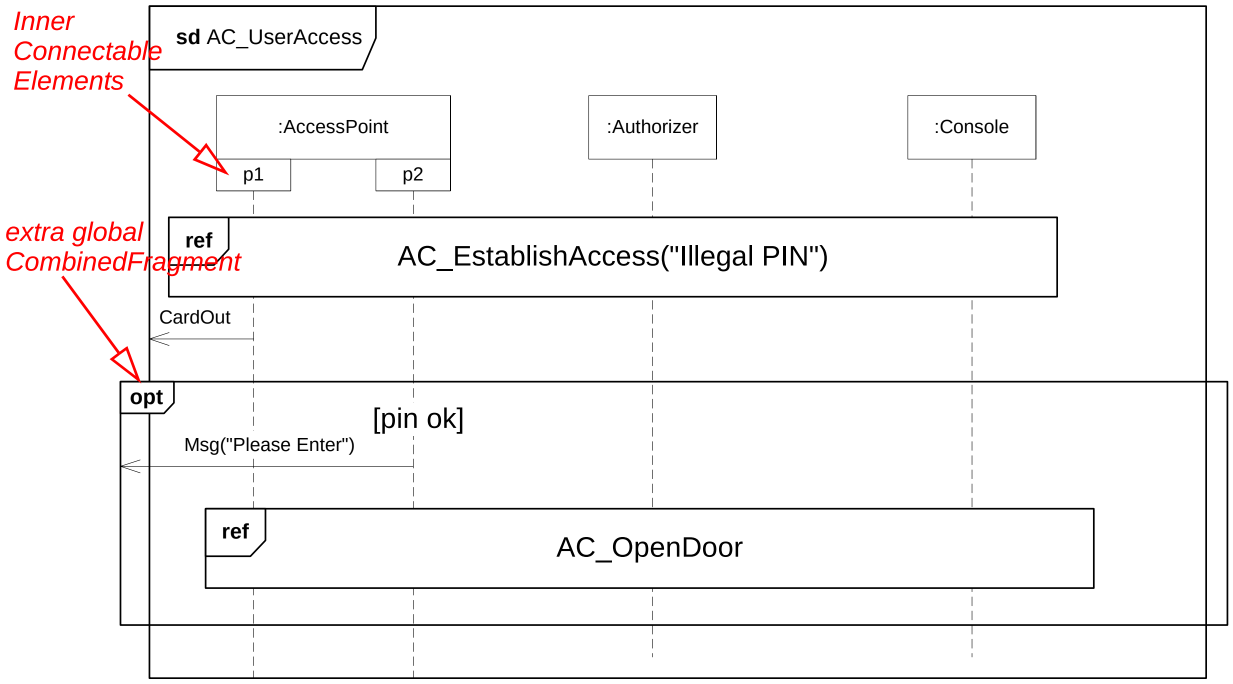


Figure 14.21 - PartDecomposition - the decomposition

In Figure 14.21 we see that *AC_UserAccess* has global constructs that match the constructs of *UserAccess* covering *ACSystem*.

In particular we notice the "extra global interaction group" that goes beyond the frame of the Interaction. This construct corresponds to a CombinedFragment of *UserAccess*. However, we want to indicate that the operands of extra global interaction groups are combined one-to-one with similar extra global interaction groups of other decompositions of the same original CombinedFragment.

As a notational shorthand, decompositions can also be shown "inline." In Figure 14.21 we see that the inner ConnectableElements of *:AccessPoint* (p1 and p2) are represented by Lifelines already on this level.

Changes from previous UML

PartDecomposition did not appear in UML 1.x.

14.3.27 SendOperationEvent (from BasicInteractions)

Generalizations

- "MessageEvent (from Communications)" on page 431

Description

A SendOperationEvent models the invocation of an operation call.

Attributes

No additional attributes

Associations

- operation : Operation [1] The operation associated with this event.

Constraints

No additional constraints

Semantics

A send operation event specifies the sending of a request to invoke a specific operation on an object. The send operation event may result in the occurrence of a call event in the receiver object (see “Common Behaviors” on page 407), and may consequentially cause the execution of a behavior by the receiver object.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.28 SendSignalEvent (from BasicInteractions)

Generalizations

- “MessageEvent (from Communications)” on page 431

Description

A SendSignalEvent models the sending of a signal.

Attributes

No additional attributes

Associations

- signal : Signal [1] The signal associated with this event.

Constraints

No additional constraints

Semantics

A send signal event specifies the sending of a message to a receiver object. The send signal event may result in the occurrence of a signal event in the receiver object (see “Common Behaviors” on page 407), and may consequentially cause the execution of a behavior by the receiver object. The sending object will not block waiting for a reply, but will continue its execution immediately.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.29 StateInvariant (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 471

Description

A StateInvariant is a runtime constraint on the participants of the interaction. It may be used to specify a variety of different kinds of constraints, such as values of attributes or variables, internal or external states, and so on.

A StateInvariant is an InteractionFragment and it is placed on a Lifeline.

Associations

• invariant: Constraint[1]	A Constraint that should hold at runtime for this StateInvariant.
• covered: Lifeline[1]	References the Lifeline on which the StateInvariant appears. Specializes InteractionFragment.covered

Semantics

The Constraint is assumed to be evaluated during runtime. The Constraint is evaluated immediately prior to the execution of the next OccurrenceSpecification such that all actions that are not explicitly modeled have been executed. If the Constraint is true, the trace is a valid trace; if the Constraint is false, the trace is an invalid trace. In other words all traces that have a StateInvariant with a false Constraint are considered invalid.

Notation

The possible associated Constraint is shown as text in curly brackets on the lifeline. See example in Figure 14.24 on page 494.

Presentation Options

A StateInvariant can optionally be shown as a Note associated with an OccurrenceSpecification.

The state symbol represents the equivalent of a constraint that checks the state of the object represented by the Lifeline. This could be the internal state of the classifierBehavior of the corresponding Classifier, or it could be some external state based on a "black-box" view of the Lifeline. In the former case, and if the classifierBehavior is described by a state machine, the name of the state should match the hierarchical name of the corresponding state of the state machine.

The regions represent the orthogonal regions of states. The identifier need only define the state partially. The value of the constraint is true if the specified state information is true.

The example in Figure 14.24 also shows this presentation option.

14.4 Diagrams

Interaction diagrams come in different variants. The most common variant is the Sequence Diagram ("Sequence Diagrams" on page 488) that focuses on the Message interchange between a number of Lifelines. Communication Diagrams ("Communication Diagrams" on page 496) show interactions through an architectural view where the arcs between the communicating Lifelines are decorated with description of the passed Messages and their sequencing. Interaction Overview Diagrams ("Interaction Overview Diagrams" on page 499) are a variant of Activity Diagrams that define interactions in a way that promotes overview of the control flow. In the Annexes one may also find optional diagram notations such as Timing Diagrams and Interaction Tables.

Sequence Diagrams

The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the Message interchange between a number of Lifelines.

A sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the Lifelines. The Interactions that are described by Sequence Diagrams are described in this chapter.

Graphic Nodes

The graphic nodes that can be included in sequence diagrams are shown in Table 14.1.

Table 14.1 - Graphic nodes included in sequence diagrams

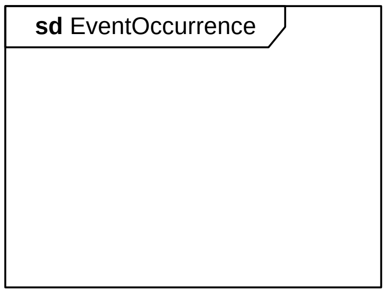
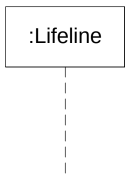
Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 467.
Lifeline		See "Lifeline (from BasicInteractions, Fragments)" on page 475.

Table 14.1 - Graphic nodes included in sequence diagrams

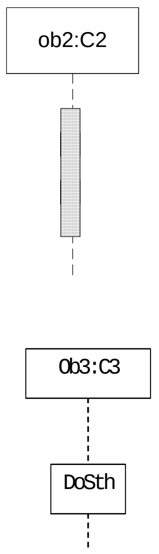
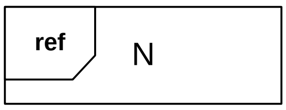
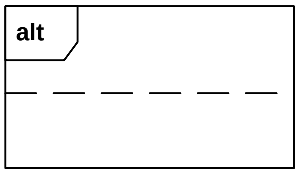
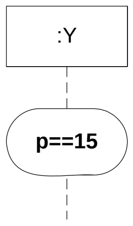
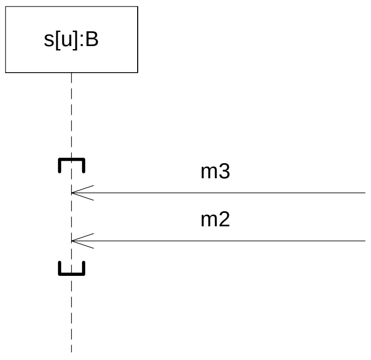
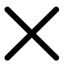
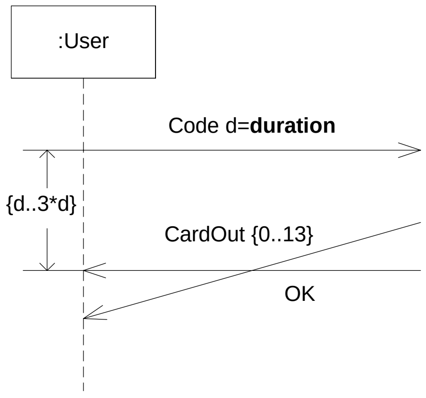
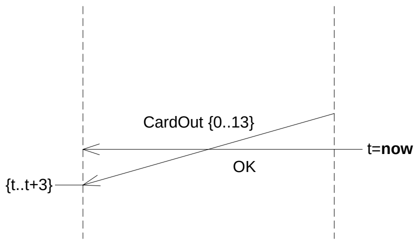
Node Type	Notation	Reference
Execution Specification		See “CombinedFragment (from Fragments)” on page 453. See also “Lifeline (from BasicInteractions, Fragments)” on page 475 and “ExecutionSpecification (from BasicInteractions)” on page 464.
InteractionUse		See “InteractionUse (from Fragments)” on page 473.
CombinedFragment		See “CombinedFragment (from Fragments)” on page 453.
StateInvariant / Continuations		See “Continuation (from Fragments)” on page 459 and “StateInvariant (from BasicInteractions)” on page 487.

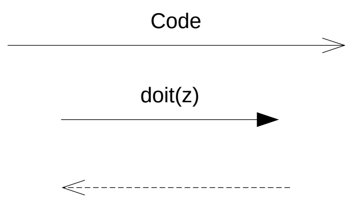
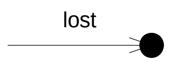
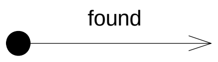
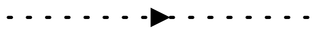
Table 14.1 - Graphic nodes included in sequence diagrams

Node Type	Notation	Reference
Coregion		See explanation under *parallel* in "CombinedFragment (from Fragments)" on page 453.
Stop		See Figure 14.11 on page 458.
DurationConstraint Duration Observation	 :User Code d=**duration** {d..3*d} CardOut {0..13} OK	See Figure 14.26 on page 496.
TimeConstraint TimeObservation		See Figure 14.26 on page 496.

Graphic Paths

The graphic paths between the graphic nodes are given in Table 14.2

Table 14.2 - Graphic paths included in sequence diagrams

Node Type	**Notation**	**Reference**
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all *complete* messages. See “Message (from BasicInteractions)” on page 477.
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen. See “Message (from BasicInteractions)” on page 477.
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification. See “Message (from BasicInteractions)” on page 477.
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 466.

Examples

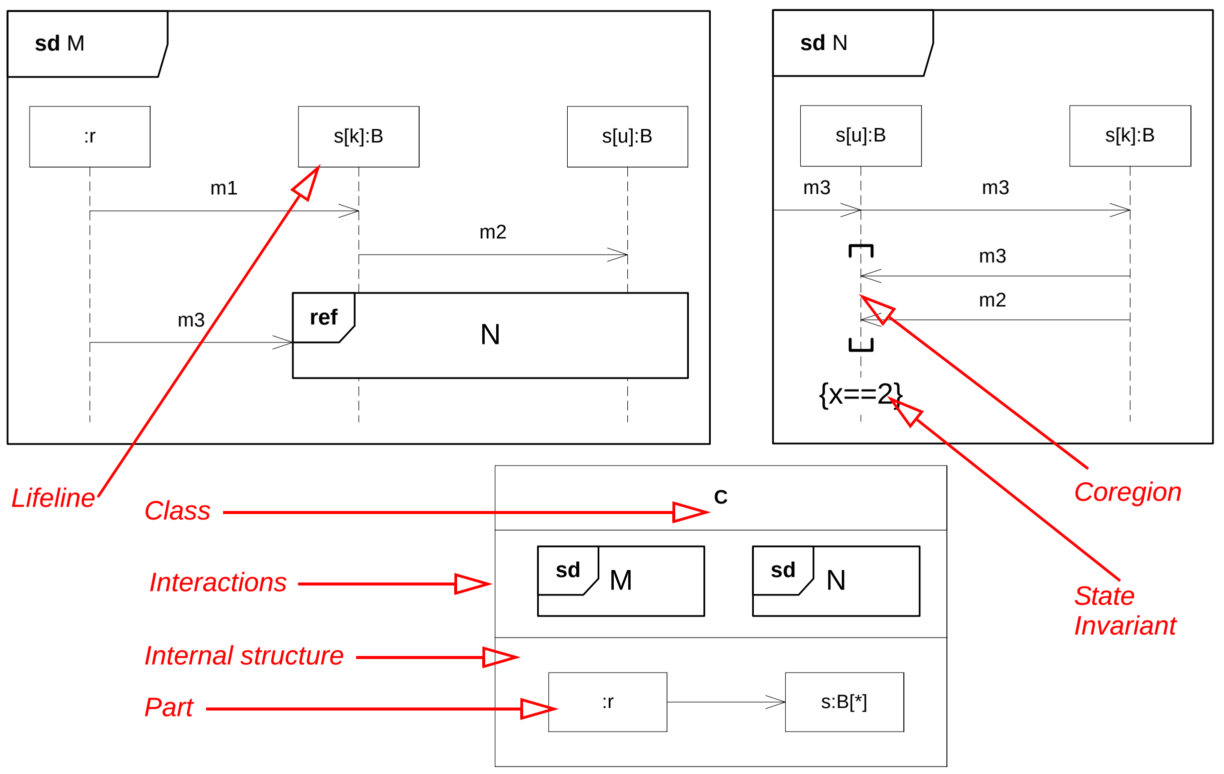


Figure 14.22 - Sequence Diagrams where two Lifelines refer to the same set of Parts (and Internal Structure)

The sequence diagrams shown in Figure 14.22 show a scenario where r sends m1 to s[k] (which is of type B), and s[k] sends m2 to s[u]. In the meantime independent of s[k] and s[u], r may have sent m3 towards the InteractionUse N through a gate. Following the m3 message into N we see that s[u] then sends another m3 message to s[k]. s[k] then sends m3 and then m2 towards s[u]. s[u] receives the two latter messages in any order (coregion). Having received these messages, we state an invariant on a variable x (most certainly owned by s[u]).

In order to explain the mapping of the notation onto the metamodel we have pointed out areas and their corresponding metamodel concept in Figure 14.23. Let us go through the simple diagram and explain how the metamodel is built up. The whole diagram is an Interaction (named N). There is a formal gate (with implicit name in_m3) and two Lifelines (named s[u] and s[k]) that are contained in the Interaction. Furthermore the two Messages (occurrences) both of the same type m3, implicitly named m3_1 and m3_2 here, are also owned by the Interaction. Finally there are the three OccurrenceSpecifications.

We have omitted in this metamodel the objects that are more peripheral to the Interaction model, such as the Part s and the class B and the connector referred by the Message.

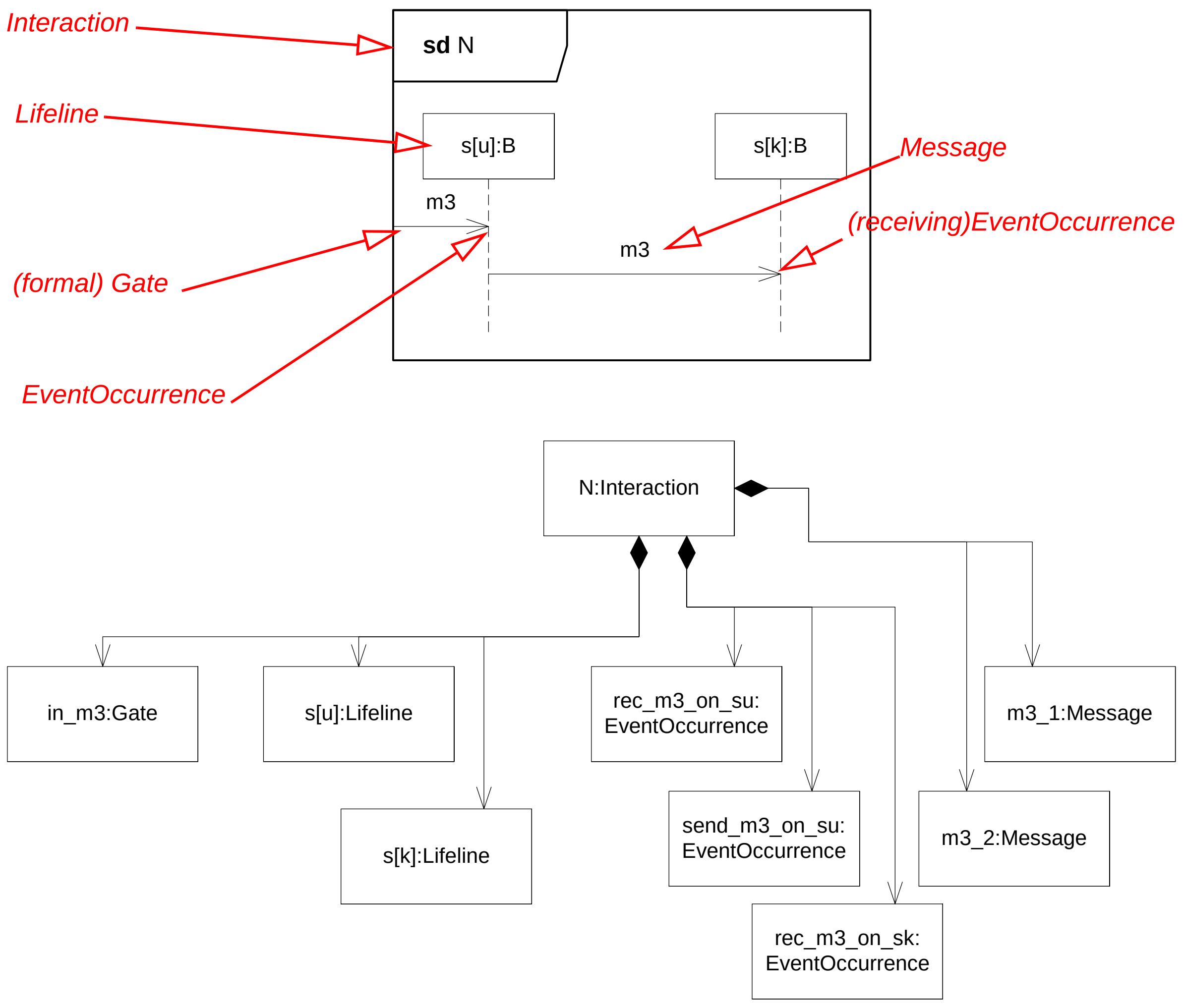


Figure 14.23 - Metamodel elements of a sequence diagram

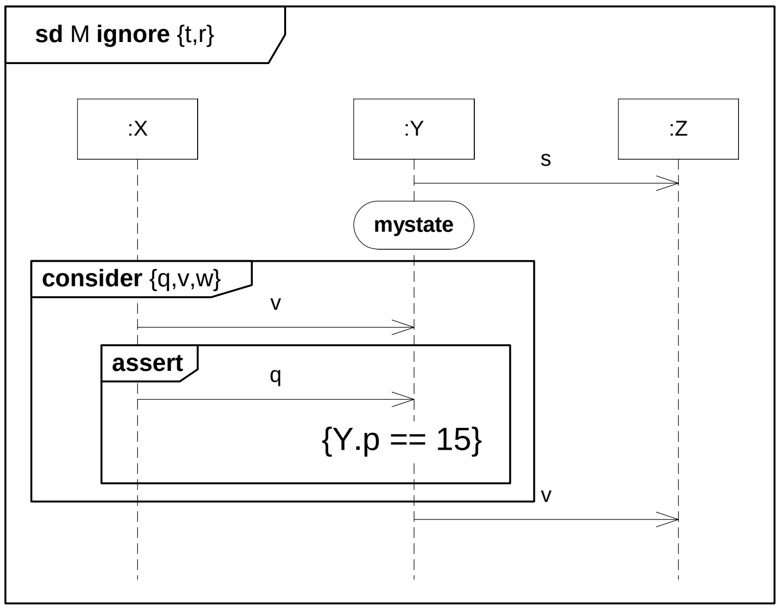


Figure 14.24 - Ignore, Consider, assert with State Invariants

In Figure 14.24 we have an Interaction M, which considers message types other than t and r. This means that if this Interaction is used to specify a test of an existing system and when running that system a t or an r occurs, these messages will be ignored by this specification. t and r will of course be handled in some manner by the running system, but how they are handled is irrelevant for our Interaction shown here.

The State invariant given as a state “mystate” will be evaluated at runtime directly prior to whatever event occurs on Y after “mystate.” This may be the reception of q as specified within the assert-fragment, or it may be an event that is specified to be insignificant by the filters.

The **assert** fragment is nested in a **consider** fragment to mean that we expect a q message to occur once a v has occurred here. Any occurrences of messages other than v, w, and q will be ignored in a test situation. Thus the appearance of a w message after the v is an invalid trace.

The state invariant given in curly brackets will be evaluated prior to the next event occurrence after that on Y.

Internal Structure and corresponding Collaboration Uses

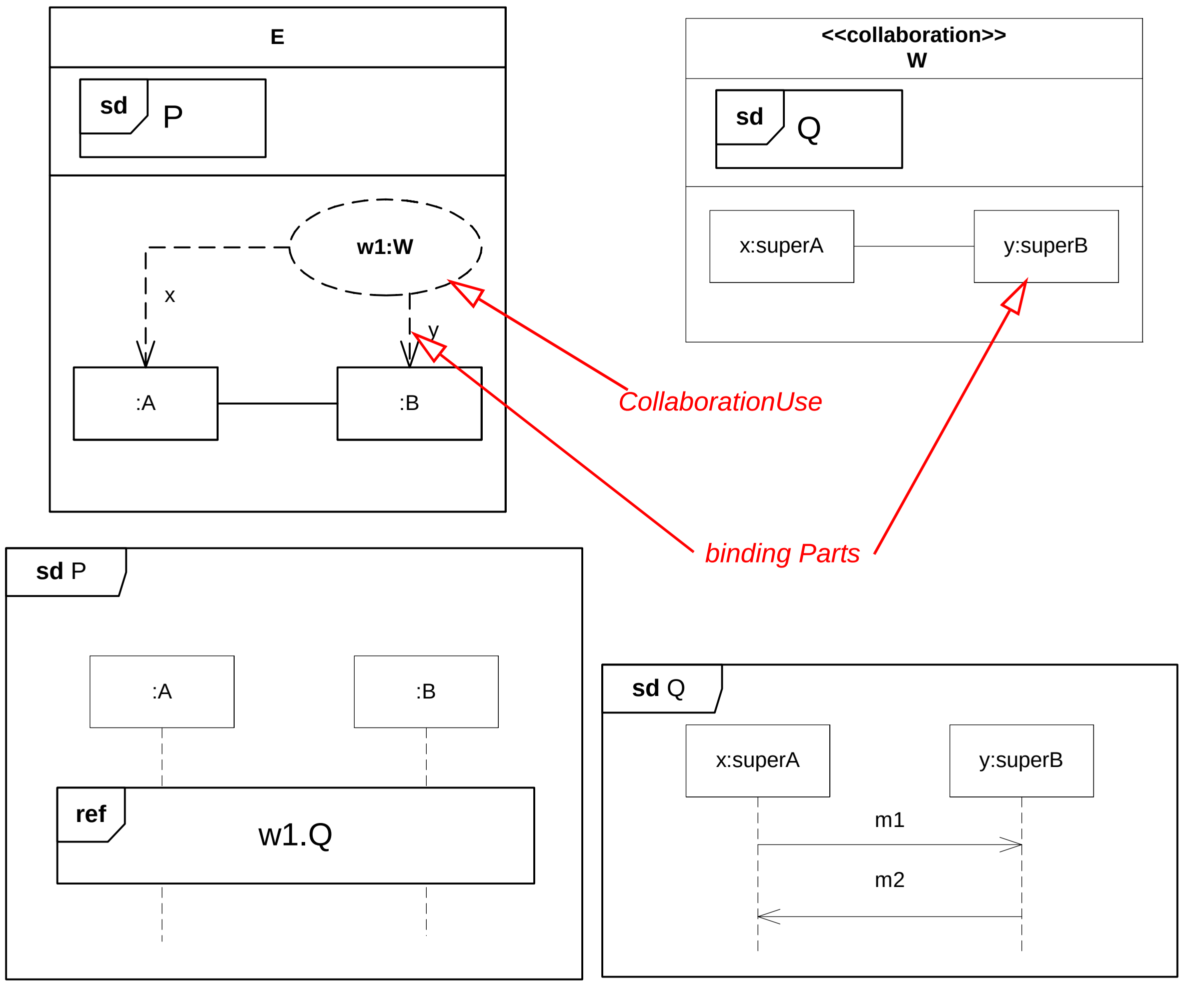


Figure 14.25 - Describing collaborations and their binding

The example in Figure 14.25 shows how collaboration uses are employed to make Interactions of a Collaboration available in another classifier.

The collaboration W has two parts x and y that are of types (classes) superA and superB respectively. Classes A and B are specializations of superA and superB respectively. The Sequence Diagram Q shows a simple Interaction that we will reuse in another environment. The class E represents this other environment. There are two anonymous parts :A and :B and the CollaborationUse w1 of Collaboration W binds x and y to :A and :B respectively. This binding is legal since :A and :B are parts of types that are specializations of the types of x and y.

In the Sequence Diagram P (owned by class E) we use the Interaction Q made available via the CollaborationUse w1.

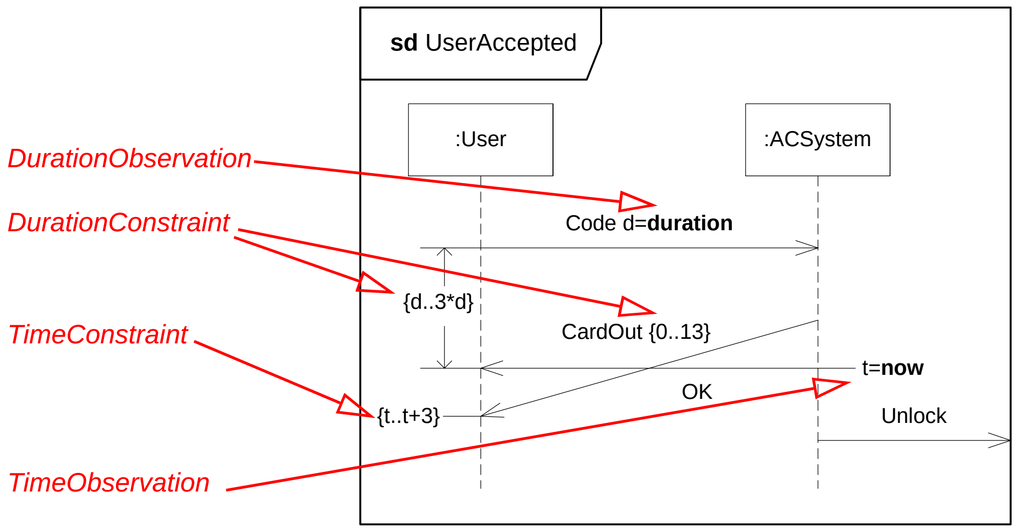


Figure 14.26 - Sequence Diagram with time and timing concepts

The Sequence Diagram in Figure 14.26 shows how time and timing notation may be applied to describe time observation and timing constraints. The :User sends a message Code and its duration is measured. The :ACSystem will send two messages back to the :User. CardOut is constrained to last between 0 and 13 time units. Furthermore the interval between the sending of Code and the reception of OK is constrained to last between d and 3*d where d is the measured duration of the Code signal. We also notice the observation of the time point t at the sending of OK and how this is used to constrain the time point of the reception of CardOut.

Communication Diagrams

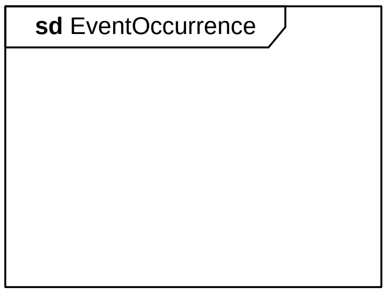
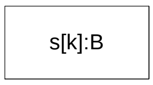
Communication Diagrams focus on the interaction between Lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of Messages is given through a sequence numbering scheme.

Communication Diagrams correspond to simple Sequence Diagrams that use none of the structuring mechanisms such as InteractionUses and CombinedFragments. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

Graphical Nodes

Communication diagram nodes are shown in Table 14.3.

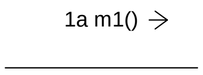
Table 14.3 - Graphic nodes included in communication diagrams

Node Type	**Notation**	**Reference**
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 467.
Lifeline		See "Lifeline (from BasicInteractions, Fragments)" on page 475.

Graphic Paths

Graphic paths of communication diagrams are given in Table 14.4.

Table 14.4 - Graphic paths included in communication diagrams

Node Type	**Notation**	**Reference**
Message		See "Message (from BasicInteractions)" on page 477 and "Sequence expression" on page 498. The arrow shown here indicates the communication direction.

Examples

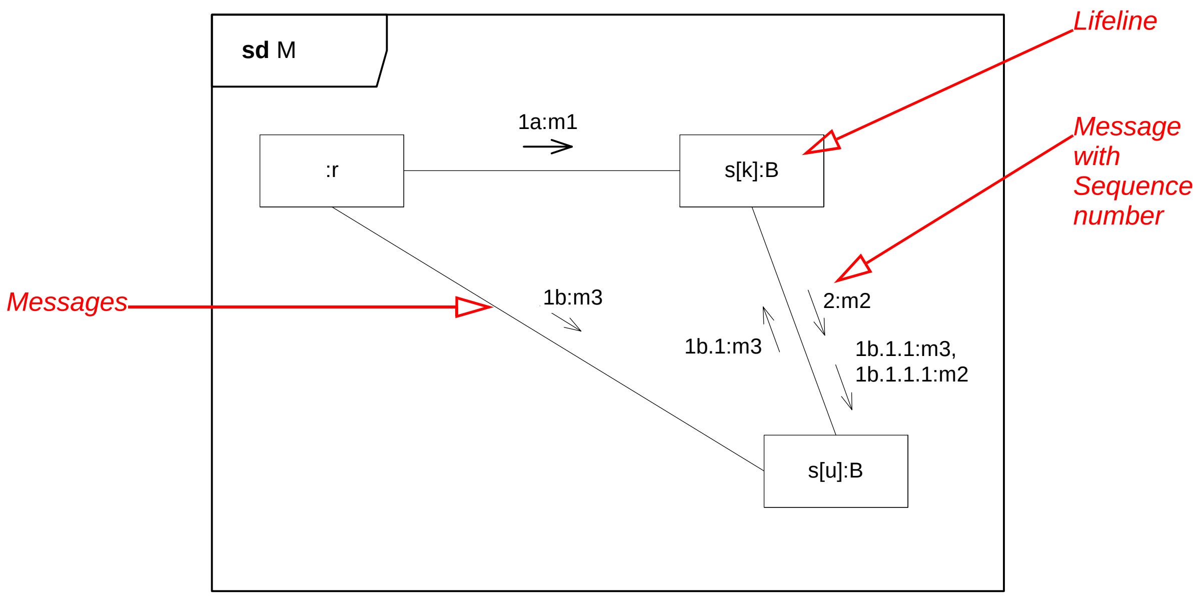


Figure 14.27 - Communication diagram

The Interaction described by a Communication Diagram in Figure 14.27 shows messages m1 and m3 being sent concurrently from :r towards two instances of the part s. The sequence numbers show how the other messages are sequenced. 1b.1 follows after 1b and 1b.1.1 thereafter etc. 2 follows after 1a and 1b.

Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (‘:’).

sequence-term ‘.’ . . . ‘:’

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[integer | name] [recurrence]

The *integer* represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The *name* represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

‘’ ‘[’ iteration-clause ‘]’an iteration*
‘[’ guard ‘]’a branch

An iteration represents a sequence of Messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: **[i := 1..n]*.

A guard represents a Message whose execution is contingent on the truth of the condition clause. The guard is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: *[x > y]*.

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the Messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): *||.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

Interaction Overview Diagrams

Interaction Overview Diagrams define Interactions (described in Chapter 14, “Interactions”) through a variant of Activity Diagrams (described in Chapter 6, “Activities”) in a way that promotes overview of the control flow.

Interaction Overview Diagrams focus on the overview of the flow of control where the nodes are Interactions or InteractionUses. The Lifelines and the Messages do not appear at this overview level.

Graphic Nodes

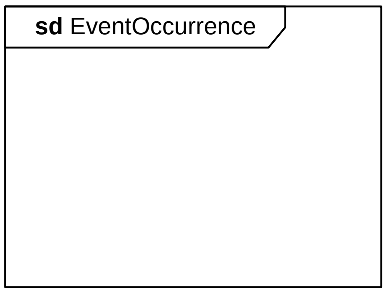
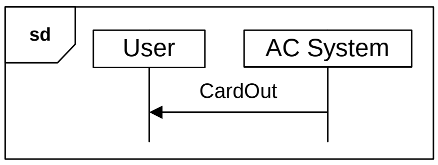
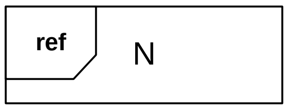
Interaction Overview Diagrams are specialization of Activity Diagrams that represent Interactions.

Interaction Overview Diagrams differ from Activity Diagrams in some respects.

1. In place of ObjectNodes of Activity Diagrams, Interaction Overview Diagrams can only have either (inline) Interactions or InteractionUses. Inline Interaction diagrams and InteractionUses are considered special forms of ActivityInvocations.
2. Alternative Combined Fragments are represented by a Decision Node and a corresponding Merge Node.
3. Parallel Combined Fragments are represented by a Fork Node and a corresponding Join Node.
4. Loop Combined Fragments are represented by simple cycles.
5. Branching and joining of branches must in Interaction Overview Diagrams be properly nested. This is more restrictive than in Activity Diagrams.
6. Interaction Overview Diagrams are framed by the same kind of frame that encloses other forms of Interaction Diagrams. The heading text may also include a list of the contained Lifelines (that do not appear graphically).

.

Table 14.5 - Graphic nodes included in Interaction Overview Diagrams in addition to those borrowed from Activity Diagrams

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 467.
Interaction		An Interaction diagram of any kind may appear inline as an ActivityInvocation. See “Interaction (from BasicInteraction, Fragments)” on page 467. The inline Interaction diagrams may be either anonymous (as here) or named.
InteractionUse		ActivityInvocation in the form of InteractionUse. See “InteractionUse (from Fragments)” on page 473. The tools may choose to “explode” the view of an InteractionUse into an inline Interaction with the name of the Interaction referred by the occurrence. The inline Interaction will then replace the occurrence by a replica of the definition Interaction where arguments have replaced parameters.

Examples

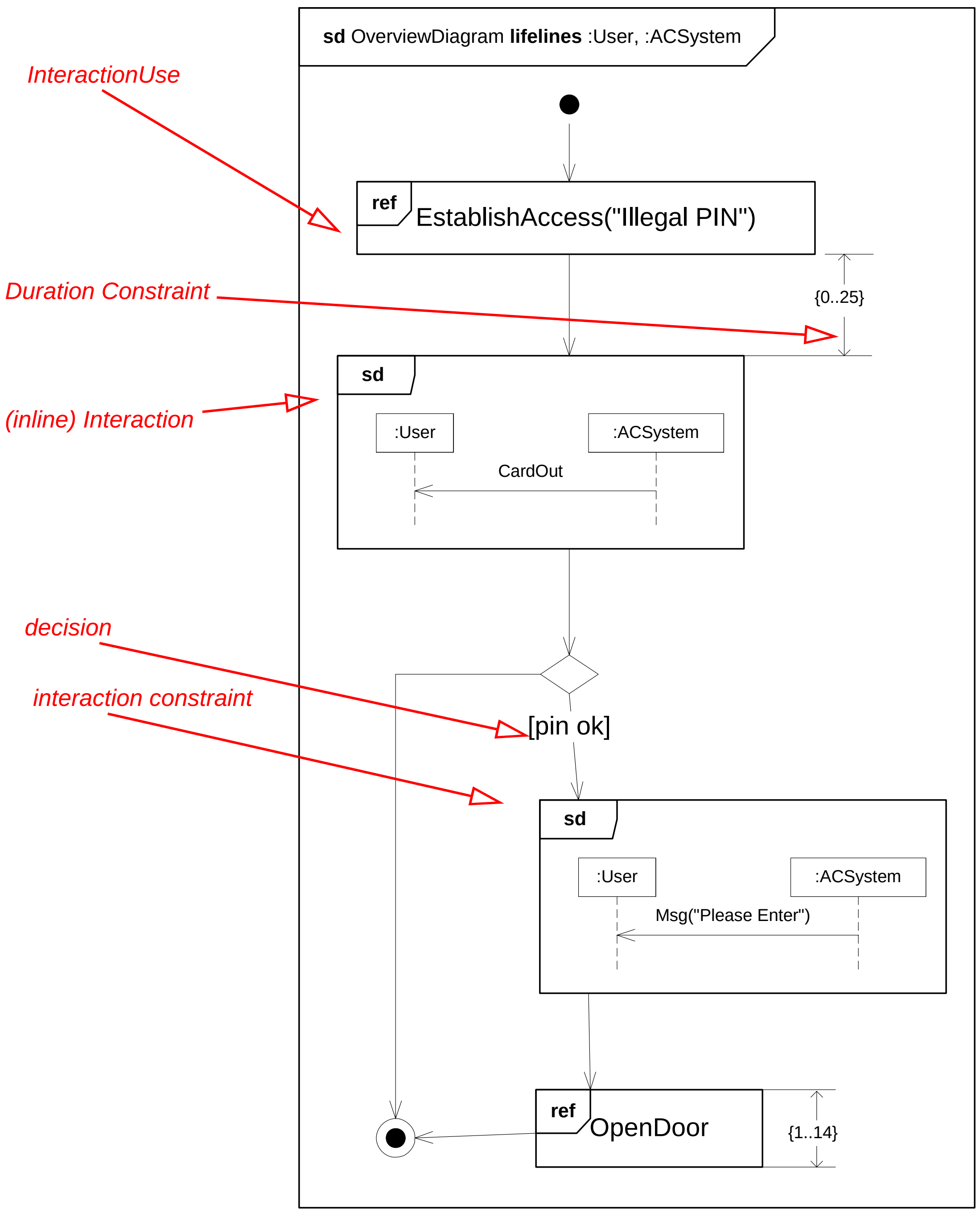


Figure 14.28 - Interaction Overview Diagram representing a High Level Interaction diagram

Interaction Overview Diagrams use Activity diagram notation where the nodes are either Interactions or InteractionUses. Interaction Overview Diagrams are a way to describe Interactions where Messages and Lifelines are abstracted away. In the purest form all Activities are InteractionUses and then there are no Messages or Lifelines shown in the diagram at all.

Figure 14.28 is another way to describe the behavior shown in Figure 14.17, with some added timing constraints. The Interaction *EstablishAccess* occurs first (with argument "Illegal PIN") followed by weak sequencing with the message *CardOut* which is shown in an inline Interaction. Then there is an alternative as we find a decision node with an InteractionConstraint on one of the branches. Along that control flow we find another inline Interaction and an InteractionUse in (weak) sequence.

Timing Diagram

Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Timing diagrams focus on conditions changing within and among Lifelines along a linear time axis.

Timing diagrams describe behavior of both individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifelines.

Graphic Nodes

The graphic nodes and paths that can be included in timing diagrams are shown in Table 14.6.

Table 14.6 - Graphic nodes and paths included in timing diagrams

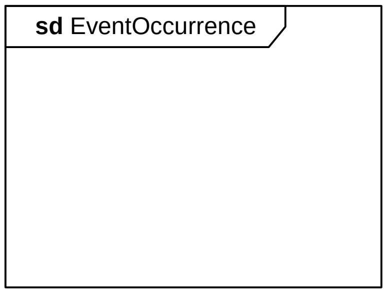
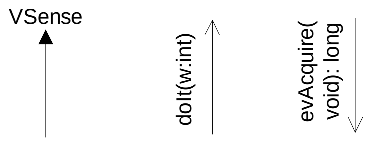
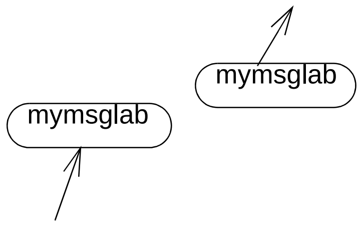
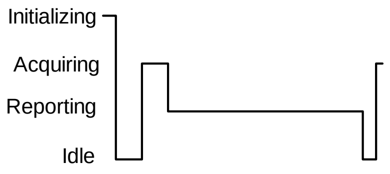
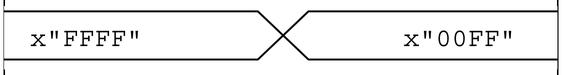
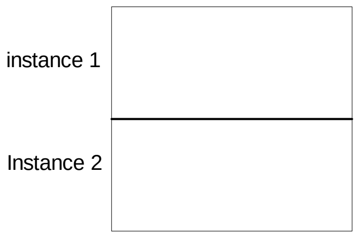
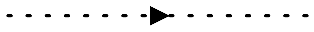
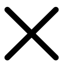
Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 467.
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. See "Message (from BasicInteractions)" on page 477.

Table 14.6 - Graphic nodes and paths included in timing diagrams

Node Type	Notation	Reference
Message label		Labels are only notational shorthands used to prevent cluttering of the diagrams with a number of messages crisscrossing the diagram between Lifelines that are far apart. The labels denote that a Message may be disrupted by introducing labels with the same name.
State or condition timeline		This is the state of the classifier or attribute, or some testable condition, such as an discrete enumerable value. See also “StateInvariant (from BasicInteractions)” on page 487. It is also permissible to let the state-dimension be continuous as well as discrete. This is illustrative for scenarios where certain entities undergo continuous state changes, such as temperature or density.
General value lifeline		Shows the value of the connectable element as a function of time. Value is explicitly denoted as text. Crossing reflects the event where the value changed.
Lifeline		See “Lifeline (from BasicInteractions, Fragments)” on page 475.
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 466
DestructionEvent		See “DestructionEvent (from BasicInteractions)” on page 462.

Examples

Timing diagrams show change in state or other condition of a structural element over time. There are a few forms in use. We shall give examples of the simplest forms.

Sequence Diagrams as the primary form of Interactions may also depict time observation and timing constraints. We show in Figure 14.26 an example in Sequence Diagram that we will also give in Timing Diagrams.

The :User of the Sequence Diagram in Figure 14.26 is depicted with a simple Timing Diagram in Figure 14.29.

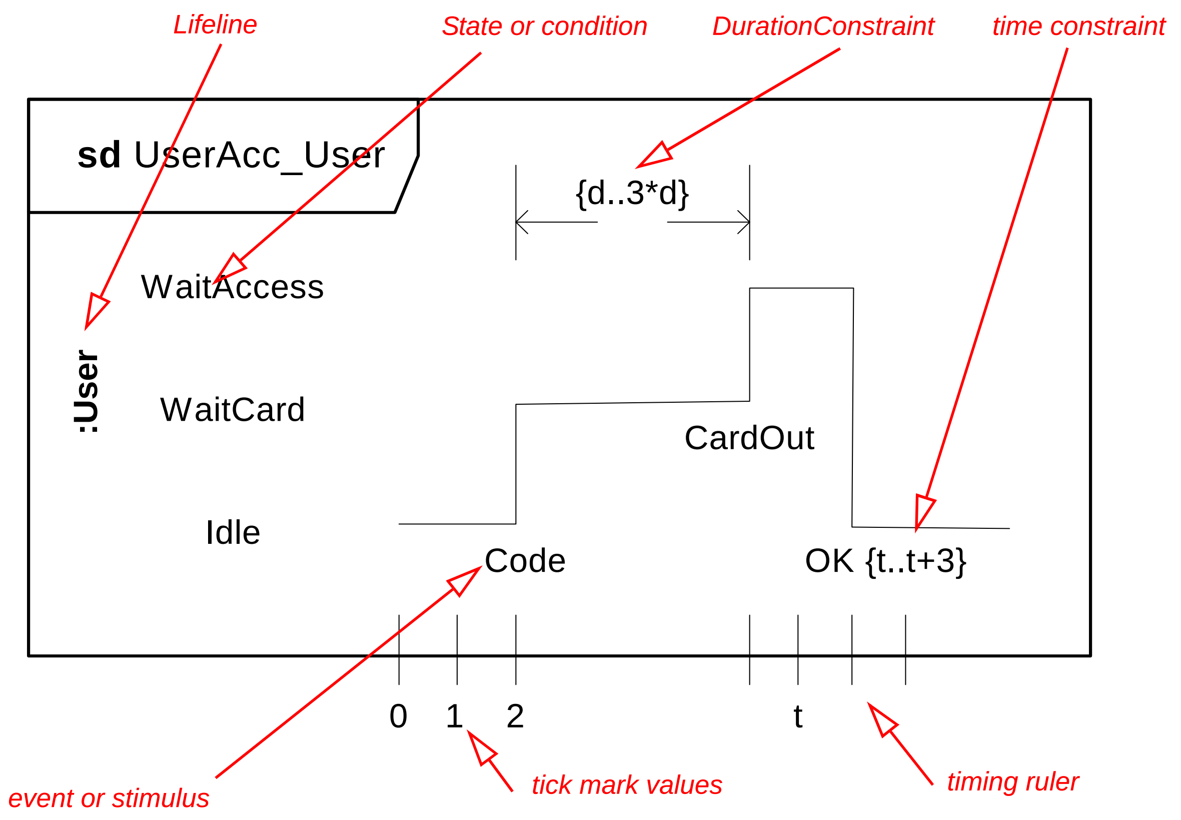


Figure 14.29 - A Lifeline for a discrete object

The primary purpose of the timing diagram is to show the change in state or condition of a lifeline (representing a Classifier Instance or Classifier Role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli. The received events are annotated as shown when it is desirable to show the event causing the change in condition or state.

Sometimes it is more economical and compact to show the state or condition on the vertical Lifeline as shown in Figure 14.30.

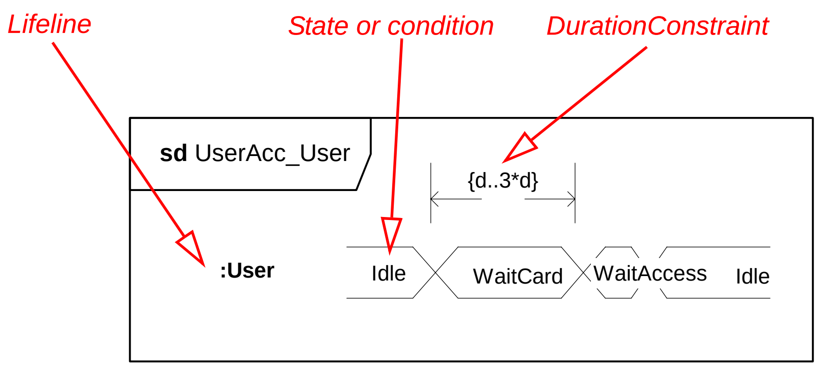


Figure 14.30 - Compact Lifeline with States

Finally we may have an elaborate form of TimingDiagrams where more than one Lifeline is shown and where the messages are also depicted. We show such a Timing Diagram in Figure 14.31 corresponding to the Sequence Diagram in Figure 14.26.

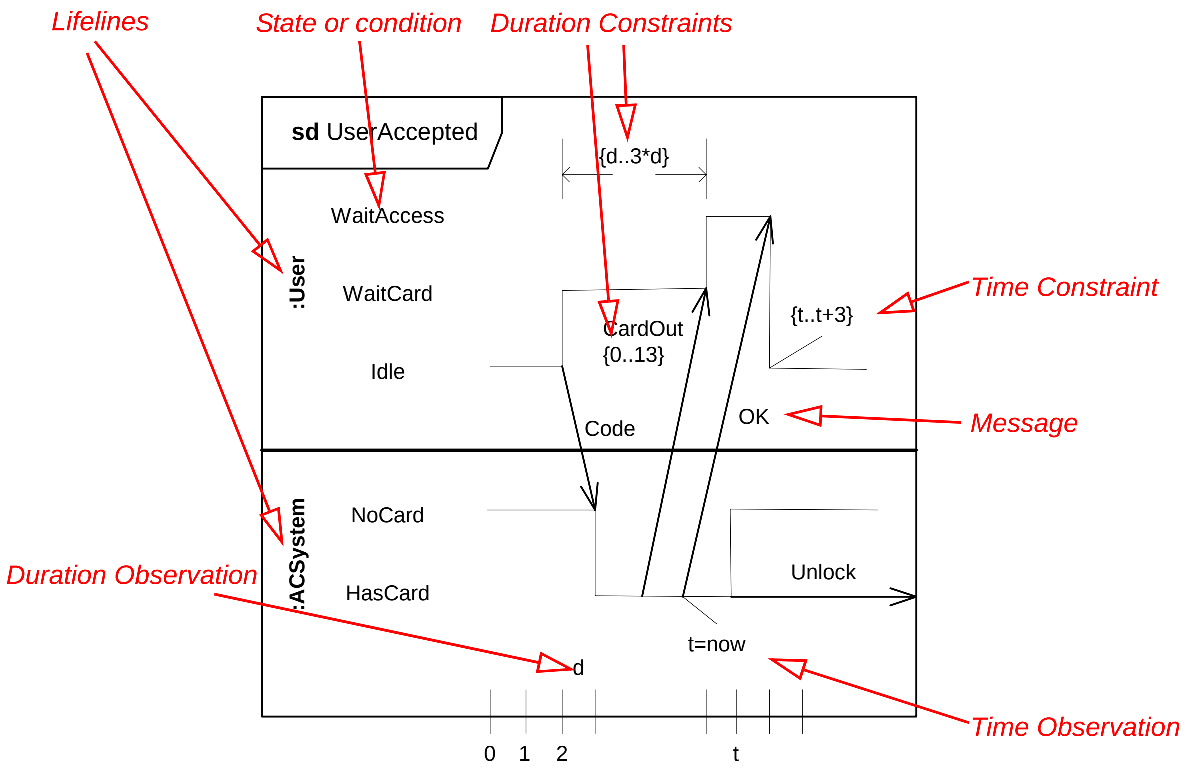


Figure 14.31 - Timing Diagram with more than one Lifeline and with Messages

Changes from previous UML

The Timing Diagrams were not available in UML 1.4.