Interactions

María Victoria Cengarle, Alexander Knapp, Heribert Mühlberger

Angaben zur Veröffentlichung / Publication details:

Cengarle, María Victoria, Alexander Knapp, and Heribert Mühlberger. 2009. "Interactions." In *UML 2 Semantics and Applications*, edited by Kevin Lano, 205–48. Hoboken, N.J.: Wiley. https://doi.org/10.1002/9780470522622.ch9.

CHAPTER 9

INTERACTIONS

MARÍA VICTORIA CENGARLE
Institut für Informatik, Technische Universität München, München, Germany

ALEXANDER KNAPP and HERIBERT MÜHLBERGER
Institut für Informatik, Universität Augsburg, Augsburg, Germany

9.1 INTRODUCTION

UML interactions describe possible message exchanges between system instances. The UML 2 [45] offers a powerful interaction language, which, besides integrating such standard operations as sequential, parallel, and iterative composition of interactions, provides means to specify recursive and negative behavior (i.e., behavior forbidden in system implementations).

The current UML 2 language for interactions is a complete overhaul of the interaction language of earlier versions. The UML 1 dialect was, on the one hand, based on the interaction diagrams of OOSE's [31], on the abstract, visual programming languages used by Fusion [13] and Syntropy [15], and also on ITU's message sequence charts (MSCs [30]). On the other hand, in the form of collaborations, it was also enriched with notions from role modeling in OORam [2]. Quite some effort has been spent on providing UML 1 sequence and collaboration diagrams with a formal semantics (see, e.g., [19,21,34,47]), thus making them amenable for use in formally based software development. However, it was realized that the language showed some defects in expressivity for more complex software engineering purposes, in particular with respect to modular modeling, describing alternatives, and combining interactions in different ways.

The UML 2 interaction language countered the deficiencies in expressivity of its previous version by incorporating and adapting many constructs of MSCs [30]. Additionally, means were introduced for distinguishing behavior that an implementing system should show and behavior that the system must not show, which was inspired by live sequence charts (LSCs [16]) and from software testing notions

UML 2 Semantics and Applications. Edited by Kevin Lano

and notations [54]. The increase in expressivity, and also in complexity, of the UML 2 interaction language spurred new efforts in providing it with a formal semantics [12,20,23,36,39,40,43,49,50]. In particular, the division of behaviors into being positive or valid for a system, negative or invalid, and finally, being inconclusive if it is neither positive nor negative, has received much attention. All these types of behavior are described by a single interaction, but it has not been clear how the different types are to be combined and how they interact [12,36,49].

In the present chapter we provide and discuss the formal semantics for UML 2 interactions following the UML specification [45] as closely as possible and also integrating the existing research results on the semantics of interactions. First, an interleaved, trace-based, denotational semantics is detailed which is built in several steps. The presentation starts from simple, basic interactions that are similar to what was present in UML 1. It is then extended by considering different message types, executions, combinations of interactions, and constraints. Finally, high-level interactions are integrated. A discussion of some alternative proposals to a formal semantics follows. In particular, an operational approach and a truly concurrent approach with event structures are considered. UML 2 interactions are related briefly with MSCs and LSCs. Finally, an overview of some notions of implementation and refinement of interactions and their role in verification and animation are given.

9.2 TRACE-BASED SEMANTICS

A trace-based formal semantics for UML 2 interactions is developed. According to the UML 2 specification document, an interaction describes *valid* (or *positive*) and *invalid* (or *negative*) traces of event occurrences. The union of the two sets of valid and invalid traces need not cover the entire universe of traces. A trace that is neither valid nor invalid for an interaction is said to be *inconclusive* for the interaction. Moreover, the semantics that we propose allows traces that are both valid and invalid for the same interaction. Hence, our semantics is based on a four-valued logic.

In developing the semantics, we proceed in a step-by-step manner, beginning with the core language constructs for describing basic interactions and then moving on to different communication types, combined fragments (including negation), constraints, and high-level interactions. For a start, however, in the following subsection we give a brief review of some mathematical concepts necessary to define appropriate semantic domains.

9.2.1 Pomsets

The formal semantics that we propose for UML 2 interactions employs partially ordered, labeled multisets which were introduced by Pratt [48] for modeling concurrency.

A *labeled partial order* (abbreviated lpo) $(X, \leq_X, \lambda_X)$ consists of a set X, a partial order $\leq_X$ on X (i.e., a relation on X that is reflexive, antisymmetric, and transitive), and a labeling function λ_X on X. An isomorphism between two lpos $(X, \leq_X, \lambda_X)$ and $(Y, \leq_Y, \lambda_Y)$ is a one-to-one mapping φ from X onto Y which is monotonic with respect to $\leq_X$ and $\leq_Y$, whose inverse mapping is also monotonic and which is label preserving [i.e., $\lambda_X(x) = \lambda_Y(\varphi(x))$ for all $x \in X$]. A *partially ordered, labeled multiset,* or *pomset* is the isomorphism class of an lpo, denoted $[(X, \leq_X, \lambda_X)]$.

A pomset p is said to be *finite* if for some (and hence, for all) $(X, \leq_X, \lambda_X) \in p$ the basic set X is finite. A pomset $p = [(X, \leq_X, \lambda_X)]$ is said to be *finitary* if for all $x \in X$ the set $\{x' \in X \mid x' \leq_X x\}$ is finite. A pomset p is said to be *linear* or *a trace* if for some $(X, \leq_X, \lambda_X) \in p$ the ordering $\leq_X$ is total on X. Let $\asymp$ be a binary, symmetric relation on labels. A pomset $p = [(X, \leq_X, \lambda_X)]$ is said to be $\asymp$*-linear* if it holds that $\forall x_1, x_2 \in X.\ \lambda_X(x_1) \asymp \lambda_X(x_2) \Rightarrow x_1 \leq_X x_2 \vee x_2 \leq_X x_1$. A pomset q is said to be *an extension* of a pomset p if there are two representatives $(X, \leq_X, \lambda_X) \in p$ and $(Y, \leq_Y, \lambda_Y) \in q$ such that $X = Y$ and $\leq_X \subseteq \leq_Y$ and $\lambda_X = \lambda_Y$. A pomset q is said to be a *linearization* of a pomset p if q is a linear extension of p. A pomset q is said to be a $\asymp$*-linearization* of a pomset p if q is a $\asymp$-linear extension of p. The set of all linearizations or $\asymp$-linearizations of p is denoted by $p\!\downarrow$ and $p \asymp\!\downarrow$, respectively. A function f that maps labels to labels is lifted to pomsets by defining $f([(X, \leq_X, \lambda_X)]) = [(X, \leq_X, f \circ \lambda_X)]$. Given a pomset $p = [(X, \leq_X, \lambda_X)]$ and a Boolean predicate π on labels, we define the *restriction* of p with respect to π by $p \restriction \pi = [(X', \leq_X \cap (X' \times X'), \lambda_X \restriction X')]$ with $X' = \{x \in X \mid \pi(\lambda_X(x))\}$.

The *empty* pomset, represented by $(\emptyset, \emptyset, \emptyset)$, is denoted by ε. Let $p = [(X, \leq_X, \lambda_X)]$ and $q = [(Y, \leq_Y, \lambda_Y)]$ be pomsets such that $X \cap Y = \emptyset$. The *concurrence* of p and q, written as $p \parallel q$, is given by $[(X \cup Y, \leq_X \cup \leq_Y, \lambda_X \cup \lambda_Y)]$. The *concatenation* of p and q, written as $p\,;q$, is given by $[(X \cup Y, \leq_X \cup \leq_Y \cup (X \times Y), \lambda_X \cup \lambda_Y)]$. Given a binary, symmetric relation $\asymp$ on labels, the $\asymp$*-concatenation* of p and q, written as $p\,;_{\asymp}\, q$, is given by $[(X \cup Y, (\leq_X \cup \leq_Y \cup \{(x, y) \in X \times Y \mid \lambda_X(x) \asymp \lambda_Y(y)\})^*, \lambda_X \cup \lambda_Y)]$. It is easy to ascertain that these definitions do not depend on the choice of representatives. Note that concatenation and $\asymp$-concatenation are associative, and concurrence is associative and commutative.

A *process* is a set of pomsets. An n-ary function f that maps pomsets to pomsets is lifted to processes $P_1, \ldots, P_n$ by defining

$$f(P_1, \ldots, P_n) = \{f(p_1, \ldots, p_n) \mid p_1 \in P_1, \ldots, p_n \in P_n\}$$

(e.g., $P_1 \asymp P_2 = \{p_1 \asymp p_2 \mid p_1 \in P_1 \wedge p_2 \in P_2\}$). For an n-ary function f that maps pomsets to processes, the image elements of the lifting of f are "flattened" [i.e., $f(P_1, \ldots, P_n) = \bigcup\{f(p_1, \ldots, p_n) \mid p_1 \in P_1, \ldots, p_n \in P_n\}$]. For instance, $P\!\downarrow = \bigcup\{p\!\downarrow \mid p \in P\}$. Furthermore, we define the *n-fold* $\asymp$*-iteration* of a process P, written $P^{(n)}$, as follows: $P^{(0)} = \{\varepsilon\}$ and $P^{(n+1)} = P \asymp P^{(n)}$.

9.2.2 Core Language

9.2.2.1 Basic Interactions

The sample basic interaction ex1 in Figure 9.1(a) specifies two instances x and y, which exchange messages a and b. The dispatch of a message (depicted by the arrow tail) and the arrival of a message (arrowhead) on the lifeline of an instance (dashed line) are called *event occurrences* or, more precisely, *message event occurrences*. The pictorial representation of a basic interaction carries the intuitive meaning of a partially ordered set of event occurrences: The dispatch of a message occurs before the arrival of the same message, and the event occurrences

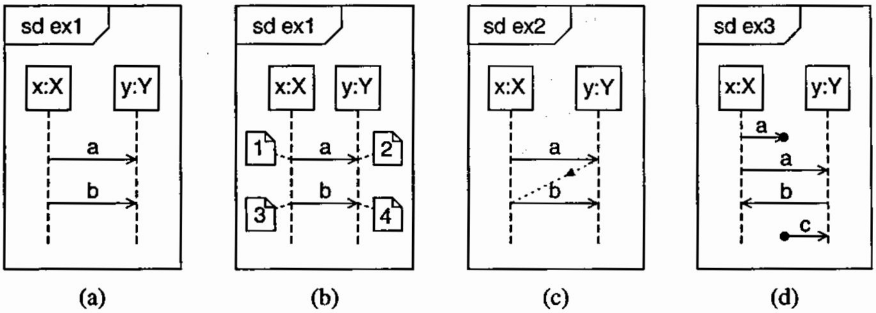


FIGURE 9.1 Basic interaction diagram (a), with labeled event occurrences (b), with an additional general ordering (c), and with lost and found messages (d).

on the lifeline of an instance are ordered from top to bottom. Thus, if we symbolize the dispatch of a from x by 1, the arrival of a at y by 2, the dispatch of b from x by 3, and the arrival of b at y by 4 [see Figure 9.1(b)], the interaction ex1 defines two valid traces: 1234 and 1324. All other traces are inconclusive for this interaction.

Additional ordering relations between event occurrences can be specified by means of general orderings. Interaction ex2, shown in Figure 9.1(c), is essentially equal to interaction ex1, except that a general ordering is added (depicted by a dotted line with an arrowhead placed somewhere in the middle of the dotted line). The general ordering in ex2 specifies that the arrival of message a has to occur before the dispatch of message b. Hence, only the trace 1234 remains valid for interaction ex2, whereas the trace 1324 is inconclusive. Finally, messages can get lost (depicted by a small black circle at the arrow end of the lost message) and messages can also be found (depicted by a small black circle at the arrow tail of the found message) [see Figure 9.1(d)]. We interpret a found message as a message whose origin lies outside the scope of the description.

9.2.2.2 Metamodel Figure 9.2 shows the fragment of the UML 2 metamodel that comprises the core language constructs for describing basic interactions. Metaclass Interaction is a subclass both of Behavior (from BasicBehaviors) and of InteractionFragment, the latter being an abstract notion of the most general interaction unit. An Interaction owns a set of Lifelines, a set of Messages, and an ordered set of InteractionFragments.

A Lifeline represents a system instance which participates in the Interaction. The mechanism by means of which these system instances are specified is not self-explanatory because it is interwoven with the concept of the context classifier of the Interaction (see BasicBehaviors::Behavior::context). Syntactically, a Lifeline references an instance of a concrete subclass of ConnectableElement (from InternalStructures). There are two such concrete subclasses specified in the package CompositeStructures, namely Property and Port. We discuss only the former here: A Property (from InternalStructures) is a specification of a set *S* of instances that are owned by a containing classifier instance. In the simplest case, this "containing classifier" coincides with the context classifier of the Interaction. If the Property concerned is multivalued (i.e., *S* may contain more than one instance), the Lifeline

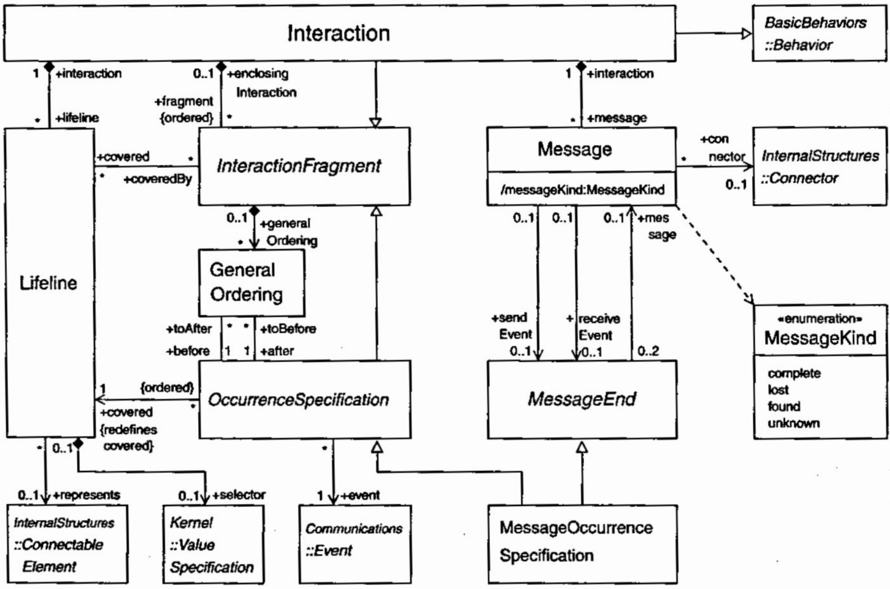


FIGURE 9.2 Fragment of the UML 2 metamodel that comprises the core language constructs for describing basic interactions.

may have an expression (the selector) that specifies which particular instance of *S* is represented by the Lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued Property is chosen. As already mentioned, Lifelines are depicted by vertical dashed lines. The left Lifeline of Interaction ex1, for instance, references a nonmultivalued Property named x which is typed by a Class named X; see Figure 9.1(a).

A Message defines a particular communication between Lifelines of an Interaction. A Message may, and usually does, associate two MessageEnds that are referenced by sendEvent and receiveEvent. A MessageEnd can either be a MessageOccurrenceSpecification or a Gate. The former specifies a message event occurrence, as mentioned above; Gates are dealt with in Sections 9.2.4 and 9.2.6. Message has a derived attribute messageKind whose value (complete, lost, found, or unknown) depends on the presence or absence of the MessageEnds. If both MessageEnds are present, messageKind is complete [see, e.g., message a in Figure 9.1(a)]. If only sendEvent or only receiveEvent is present, messageKind has the values lost and found, respectively [see message a (first dispatch) and message c in Figure 9.1(d), respectively]. If both MessageEnds are absent, which preferably should not occur, then messageKind is unknown. Message has also a second attribute called messageSort, which specifies the type of communication action used to generate the message. The present section deals with asynchronous communication only (i.e., messageSort is asynchCall or asynchSignal); synchronous communication is treated in Section 9.2.3. A Message may specify a Connector on which the Message is sent. If both MessageEnds of a Message are specified as MessageOccurrenceSpecifications, the Connector must link the ConnectableElements represented by the Lifelines that are covered by the two MessageEnds.

An InteractionFragment is an abstract notion of the most general interaction unit. InteractionFragment is the root class of a composite pattern (see Section 9.2.4) and has several direct subclasses. For the time being we are interested in basic interactions;[1] thus, Figure 9.2 shows only two of the direct subclasses of InteractionFragment: namely, OccurrenceSpecification and Interaction. The former is an abstract[2] class that specifies the occurrence of an Event. An OccurrenceSpecification covers (lies on) exactly one Lifeline, which represents the instance where the specified event is to occur. The order of OccurrenceSpecifications along a Lifeline is "significant, denoting the order in which these OccurrenceSpecifications will occur" [45, p. 491].[3]

As mentioned above, the semantics of Interactions is based on traces. A trace is a sequence of event occurrences. In general, the semantics of an Interaction or an InteractionFragment is given by a *pair* of sets of traces: namely, a set of valid (or positive) traces and a set of invalid (or negative) traces. However, negative traces are associated only with the use of negative CombinedFragments (see Section 9.2.4). In this section as well as in Section 9.2.3, the semantics of interactions are given solely by a set of positive traces P because the set of negative traces is always empty.

The UML specification document [45, p. 497] describes the semantics of an OccurrenceSpecification to be "just the trace of that single OccurrenceSpecification," thereby identifying an event that occurs in a (semantic) trace with its specifying (syntactic) OccurrenceSpecification. Although this identification is a legitimate approach, we prefer to consider an OccurrenceSpecification o as a *syntactic* unit which specifies (an occurrence of) a semantic event e, although this "event" is not only given by the Event that is referenced by o but also contains information about the role that o plays in the interaction, in particular, which Lifeline l is covered by o. Actually, e contains the same information as o. Bearing this in mind, we declare that an OccurrenceSpecification has only one trace, which consists of only one occurrence of the event e that is specified by the OccurrenceSpecification (i.e., $P = \{e\}$).

[1] A *basic interaction* is defined as an Interaction that does not own CombinedFragments. For the purposes of this section, however, a basic interaction is simply an Interaction whose constituent InteractionFragments are all OccurrenceSpecifications.

[2] Three class diagrams of the UML specification document [45] (the diagrams on pp. 460, 461, and 462) treat OccurrenceSpecification as an abstract class. In contrast, the class diagram (p. 463) as well as the specification text treat OccurrenceSpecification as a concrete class. We have decided to regard the metaclass OccurrenceSpecification as abstract.

[3] The authors of the present chapter are not absolutely certain of the necessity of the order designator at the nonnavigable end of the association between OccurrenceSpecification and Lifeline (see Figure 9.2). In our opinion, the order designator is redundant because the order of OccurrenceSpecifications along a Lifeline is completely specified by weak sequencing of the InteractionFragments that are owned by an Interaction.

The semantics of a basic interaction is specified as follows: Let I be an Interaction that owns pairwise distinct[4] OccurrenceSpecifications $o_1, \ldots, o_n$ (in this order) with positive trace sets $\{e_1\}, \ldots, \{e_n\}$, respectively. A binary, temporal relation $\rightarrow$ on $O = \{o_i | i = 1, \ldots, n\}$ is defined such that for all $i, j \in \{1, \ldots, n\}$, $o_i \rightarrow o_j$ if, and only if, at least one of the following conditions is satisfied:

1. o_i and o_j are referenced by a Message (with messageKind = complete) via sendEvent and receiveEvent, respectively.
2. o_i and o_j are referenced by a GeneralOrdering via before and after, respectively.
3. o_i and o_j cover the same Lifeline and $i < j$ (i.e., o_i lies above o_j).

The semantics of I can then be specified as the set of all traces $e_{\pi^{-1}(1)}\dot{e}_{\pi^{-1}(n)}$ with a permutation π on $\{1, \ldots, n\}$ such that for all $i, j \in \{1, \ldots, n\}$, $o_i \rightarrow o_j$ implies that $\pi(i) < \pi(j)$. Quite evidently, such a permutation π can exist only if $(O, \rightarrow)$ is a directed acyclic graph, with emphasis on *acyclic*. Calling $(O, \rightarrow)$ the *specification graph* of basic interaction I, we end up with the following constraint: *The specification graph of a basic interaction must not have (directed) cycles.*[5]

9.2.2.3 Example By instantiating the metamodel, we obtain the instance diagram in Figure 9.3, which is (a part of) the abstract syntax of the basic interaction ex1 in Figure 9.1(a). For reasons of readability, the model elements for message b are omitted. We assume that the collaboration C in Figure 9.4(a) underlies[6] the interaction ex1. Furthermore, we assume that the connector named "channel," which is part of C, is typed with association A of the class diagram in Figure 9.4(b).

9.2.2.4 Semantics We define a formal semantics of basic interactions by first mapping the metamodel in Figure 9.2 to an appropriate domain of pomsets and then defining valid traces as linearizations of these pomsets. For this purpose we assume

[4] In terms of object identity.

[5] Note that this constraint, albeit necessary, is not specified by the UML specification document. However, the idea of this constraint underlies several passages in the specification text, such as the following notation instruction: "A message is shown as a line from the sender message end to the receiver message end. The line must be such that every line fragment is either horizontal or downwards when traversed from send event to receive event" [45, p. 493].

[6] An Interaction is an *emergent* behavior. Emergent behavior results from the interaction of one or more participant objects. If the participating objects are parts of a larger composite object, an emerging behavior can be seen as indirectly describing the behavior of the container object also (cf. [45, p. 419]). In this case, the container object serves as a (pseudo-)execution context of the emergent behavior. The question arises from which metaclass a container object has to be selected if it is supposed to serve as a pseudoexecution context of an interaction of system instances contained in the container object. Since the object has to be referenced by Behavior::context it has to be selected as a BehavioredClassifier (from BasicBehaviors) Since the object contains system instances that are represented by ConnectableElements, it has to be selected as a StructuredClassifier (from InternalStructures). Consequently, the object must be an instance of a metaclass which is a specialization of both BehavioredClassifier and StructuredClassifier. Metaclass Collaboration (from Collaborations) meets these requirements.

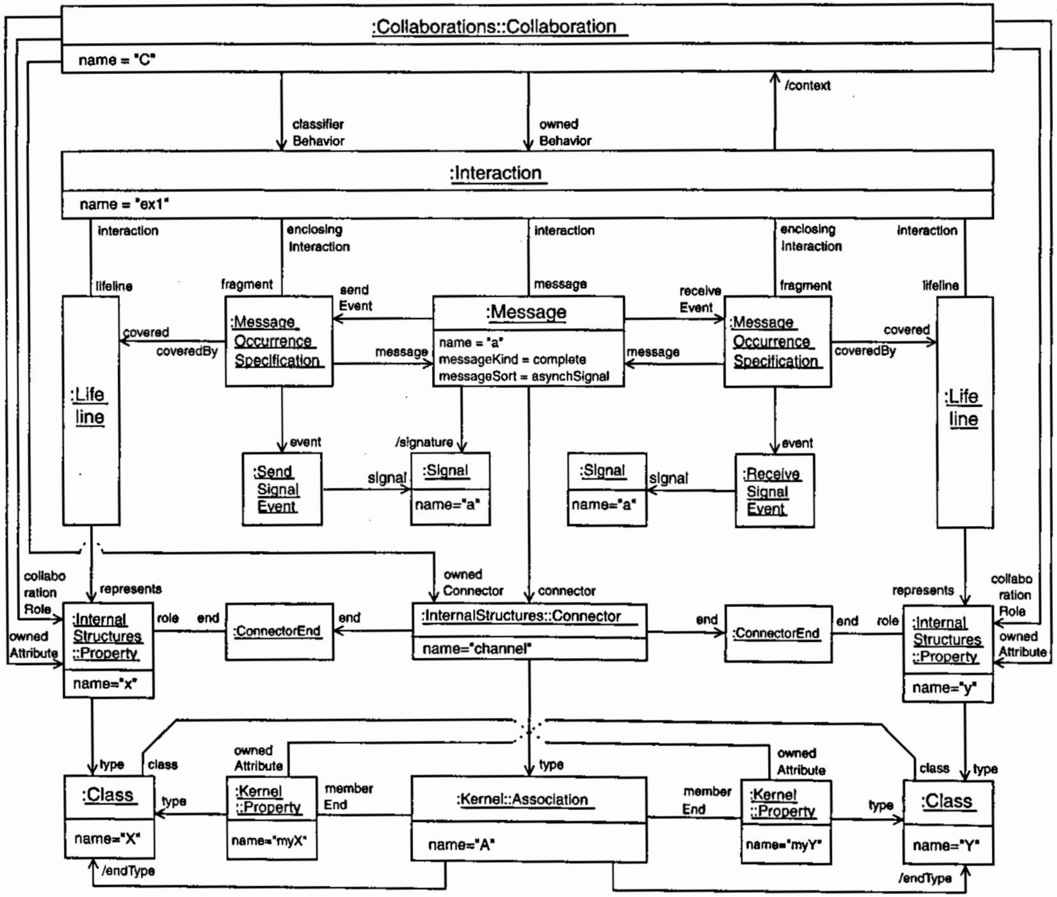


FIGURE 9.3 Fragment of the abstract syntax of basic interaction ex1 in Figure 9.1(a), with the collaboration C in Figure 9.4(a) serving as a (pseudo-)execution context of interaction ex1.

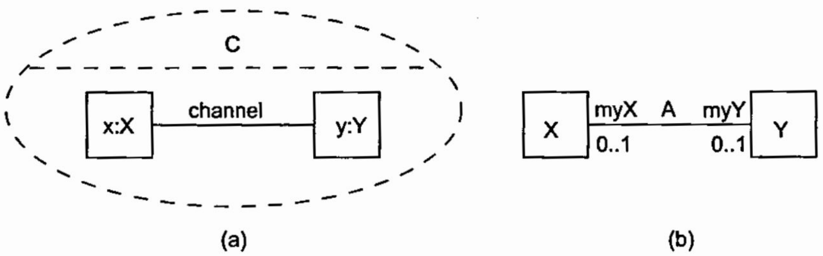


FIGURE 9.4 Underlying Collaboration C (a) of interaction ex1 and a class diagram (b) containing an association A for the purpose of typing the Connector "channel" in C and in Figure 9.3.

two primitive domains for instances $\mathbb{I}$ and messages $\mathbb{M}$. Metavariables s, r, l range over $\mathbb{I}$ and m ranges over $\mathbb{M}$. Using $\hat{s}, \hat{r} \in \hat{\mathbb{I}} ::= l \mid -$, the domain of *(message) events* $\mathbb{E}$ is defined as follows:

$$
\begin{aligned}
e \in \mathbb{E} ::= {} & \mathsf{snd}(s, \hat{r}, m) \\
\mid {} & \mathsf{rcv}(\hat{s}, r, m)
\end{aligned}
$$

An event of the form $\mathsf{snd}(s, r, m)$ or $\mathsf{rcv}(s, r, m)$ represents the dispatch and arrival of a message m (with messageKind = complete) from *sender* instance s to *receiver* instance r, respectively. An event of the form $\mathsf{snd}(s, -, m)$ represents the dispatch of a message m (with messageKind = lost) from sender instance s. An event of the form $\mathsf{rcv}(-, r, m)$ represents the arrival of a message m (with messageKind = found) at receiver instance r. We define:

$$
\begin{array}{ll}
\alpha : \mathbb{E} \longrightarrow \wp(\mathbb{I}) & \mu : \mathbb{E} \longrightarrow \mathbb{M} \\
\alpha(\mathsf{snd}(s, \hat{r}, m)) = \{s\} & \mu(\mathsf{snd}(s, \hat{r}, m)) = m \\
\alpha(\mathsf{rcv}(\hat{s}, r, m)) = \{r\} & \mu(\mathsf{rcv}(\hat{s}, r, m)) = m
\end{array}
$$

If $\alpha(e) = \{l\}$, the instance l is said to be *active* for event e. Since we identify instances with their representing lifelines, we call α the lifeline function. If instance l is active for event e, we also say that e *lies on* lifeline l. We define a binary, symmetric *conflict* relation $\not\asymp \subseteq \mathbb{E} \times \mathbb{E}$ as follows: $e_1 \not\asymp e_2$ if, and only if, $\alpha(e_1) \cap \alpha(e_2) \neq \emptyset$. Hence, two events are in conflict if, and only if, they lie on the same lifeline.

The domain $\mathbb{D}$ comprises all finitary pomsets $[(O, \leq_O, \lambda_O)]$ such that $\operatorname{ran}(\lambda_O) \subseteq \mathbb{E}$, with $\operatorname{ran}(\lambda_O)$ denoting the range of λ_O. An element $o \in O$ of the basic set of a representative $(O, \leq_O, \lambda_O)$ of a pomset $p \in \mathbb{D}$ denotes an *occurrence* of event $\lambda_O(o)$. A pomset $p \in \mathbb{D}$ is said to be *locally linear* if it is $\not\asymp$-linear. We define $\mathbb{P} = \{p \in \mathbb{D} \mid p \text{ is locally linear}\}$ and $\mathbb{T} = \{p \in \mathbb{D} \mid p \text{ is a trace}\}$. Clearly, $\mathbb{T} \subseteq \mathbb{P} \subseteq \mathbb{D}$ and $\varepsilon \in \mathbb{T}$. By identifying a pomset $[(\{o\}, \leq_{\{o\}}, \lambda_{\{o\}})]$ with event $\lambda_{\{o\}}(o)$ we can regard $\mathbb{E}$ as a subset of $\mathbb{T}$. Given n events $e_1, e_2, \ldots, e_n \in \mathbb{E}$ with $n \geq 1$, we also write the finite trace $e_1 \,;\, e_2 \,;\, \ldots \,;\, e_n$ as $e_1 e_2 \cdots e_n$.

TABLE 9.1 Formal Semantics of Basic Interactions

$\mathcal{P}[\![-]\!] : \text{Basic} \to \wp(\mathbb{T})$
$\mathcal{P}[\![\text{B}]\!] = B\downarrow$

A pomset $p \in \mathbb{D}$ is said to be *well formed* if there is $n \in \mathbb{N}$ and $m_1, \ldots, m_n \in \mathbb{M}$ and $(\hat{s}_1, \hat{r}_1), \ldots, (\hat{s}_n, \hat{r}_n) \in (\hat{\mathbb{I}} \times \hat{\mathbb{I}}) \setminus \{(-,-)\}$ such that[7]

$$p \in (M(\hat{s}_1, \hat{r}_1, m_1) \parallel \cdots \parallel M(\hat{s}_n, \hat{r}_n, m_n)) \asymp\downarrow$$

where $M(\hat{s}, \hat{r}, m)$ is defined as follows: $M(s,r,m) = \mathsf{snd}(s,r,m) \,;\, \mathsf{rcv}(s,r,m)$, $M(s,-,m) = \mathsf{snd}(s,-,m)$, and $M(-,r,m) = \mathsf{rcv}(-,r,m)$. Well-formed pomsets are obviously finite and locally linear.

For the purpose of developing our semantics, basic interactions are given syntactically by well formed pomsets. We define $\text{Basic} = \{B \in \mathbb{P} \mid B \text{ is well formed}\}$ and use B as a metavariable that ranges over Basic. The formal semantics of basic interactions is given by a semantic function $\mathcal{P}[\![-]\!]$ which maps basic interactions to sets of positive (valid) traces (see Table 9.1). As mentioned above, basic interactions do not have negative traces.

The question remains how the metamodel in Figure 9.2 is to be mapped into the new (syntactic) domain Basic: Let I be a basic interaction and $(O, \dashrightarrow)$ the specification graph of I; see Section 9.2.2.2. Since $(O, \dashrightarrow)$ is an acyclic graph, the O-reflexive-transitive closure of $\dashrightarrow$ is a partial order on O; we define $\leq_O$ by $\dashrightarrow^*$. Basic interaction I is then mapped to $[(O, \leq_O, \lambda_O)]$, where λ_O is a labeling function $O \to \mathbb{E}$, whose definition is straightforward; in particular, synchronous messages (see Section 9.2.3) and coregions (see [45]) can be dealt with easily. The only interesting issue in defining λ_O is how a Message M is to be mapped to a message identificator $m \in \mathbb{M}$. One possibility would be to use the Message M itself (i.e., the object identificator). However, we define m as the set of any information that is conveyed by the message and can be used by the receiver to distinguish between two messages coming from the same sender. In particular, m comprises the name of M, arguments, and any kind of message content. Note that the resulting mapping of the metamodel into Basic is not injective (see Figure 9.5). A receiver instance cannot determine the order in which it receives two completely identical messages.

9.2.3 Synchronous and Asynchronous Messages

9.2.3.1 Communication Types The sample interaction ex4 in Figure 9.6 models the establishing of a connection between a client instance x and a server instance y and the subsequent processing of a client request by an instance z that has been created by y for this very purpose. Unlike the previous examples, which used only asynchronous communication (depicted by open arrowheads), interaction ex4

[7] The concurrence of $n = 0$ pomsets is defined by the empty pomset ε.

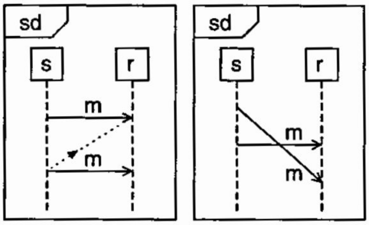


FIGURE 9.5 Two structurally distinct basic interaction diagrams that are both mapped to the same element snd(s, r, m) snd(s, r, m) rcv(s, r, m) rcv$(s, r, m) \in$ Basic.

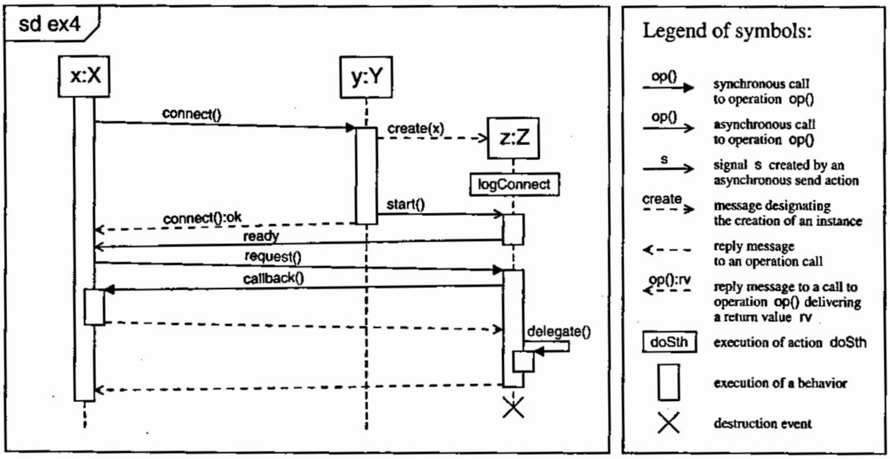

sd ex4
x:X
y:Y
z:Z
connect()
create(x)
logConnect
start()
connect():ok
ready
request()
callback()
delegate()
Legend of symbols:
op()
synchronous call to operation op()
op()
asynchronous call to operation op()
s
signal s created by an asynchronous send action
create
message designating the creation of an instance
reply message to an operation call
op():rv
reply message to a call to operation op() delivering a return value rv
doSth
execution of action doSth
execution of a behavior
destruction event

FIGURE 9.6 Sample interaction diagram that uses different types of communication.

also specifies messages reflecting *synchronous calls to operations* (depicted by filled arrowheads). These messages are connect(), request(), callback(), and delegate(). A synchronous call to an operation typically results in a reply message, which is shown graphically by a dashed line. Reply message connect():ok, for instance, delivers a return value ok, indicating that a connection with the server has been established successfully. Messages start() and ready represent an asynchronous call to an operation and a signal, respectively. Message create(x) designates the creation of a new instance z, with the argument x informing z what its communication partner is. The ✕ at the bottom of the lifeline of z depicts a *destruction event* that represents the destruction of instance z.

An *execution specification* (also known as *activation bar* or *focus of control*) is a notation that can appear on a lifeline to indicate the time during which an instance is *active* (i.e., executes a behavior or performs an action). In the case of a behavior, the execution specification is called a *behavior execution specification*, which is depicted by a thin rectangle that covers a part of a lifeline (e.g., instance x is active right from the start). In the case of an action, the execution specification is called an *action execution specification*, which is depicted by a wider, labeled rectangle, where the

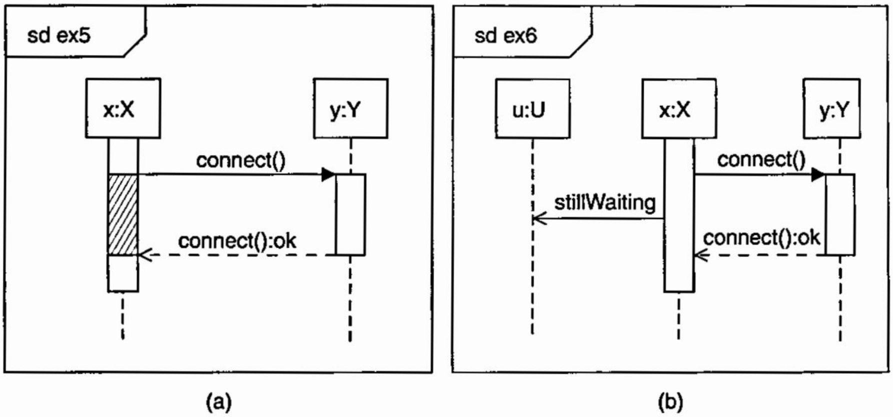


FIGURE 9.7 In diagram (a), which is not a legal UML 2 diagram, we used shading to indicate that an execution specification is blocked. In contrast, diagram (b) is legal UML 2, but it may be considered bad style.

label identifies the action that is executed (see action logConnect, which represents the writing of log data to a database).

The intuitive semantics of synchronous calls to operations is that after dispatching the synchronous message connect() the behavior execution specification on lifeline x is blocked until the corresponding reply message connect():ok is received. In Figure 9.7(a), shading is used to indicate the part of the behavior execution specification on lifeline x that is blocked by message connect(). Note that this form of shading is not a legal UML 2 notation, although it has actually been used in the literature (see, e.g., [51]). Regardless of how a blocked execution specification is depicted, the question arises whether message arrows may depart from positions on a lifeline where the lifeline is covered by a blocked execution specification; see, for example, message stillWaiting in Figure 9.7(b), which informs a user u that client x is still waiting for a reply message from server y. At first sight, from a sequential operational point of view, the dispatch of message stillWaiting would be unimplementable because the behavior execution specification on lifeline x is blocked by the synchronous message connect(). This view on synchronous messages, although legitimate, is by no means mandatory, because one and the same execution specification may represent behavior that emerges from several concurrent subbehaviors (e.g., parallel regions of a state machine). Even if one of the concurrent subbehaviors is blocked by a synchronous call, the other subbehaviors can still be active and send messages (although this may be considered bad style).

9.2.3.2 Metamodel Figure 9.8 shows the fragment of the UML 2 metamodel that is relevant to the new language constructs introduced by Figure 9.6. Model elements that have already appeared in our previous metamodel diagrams are shaded.

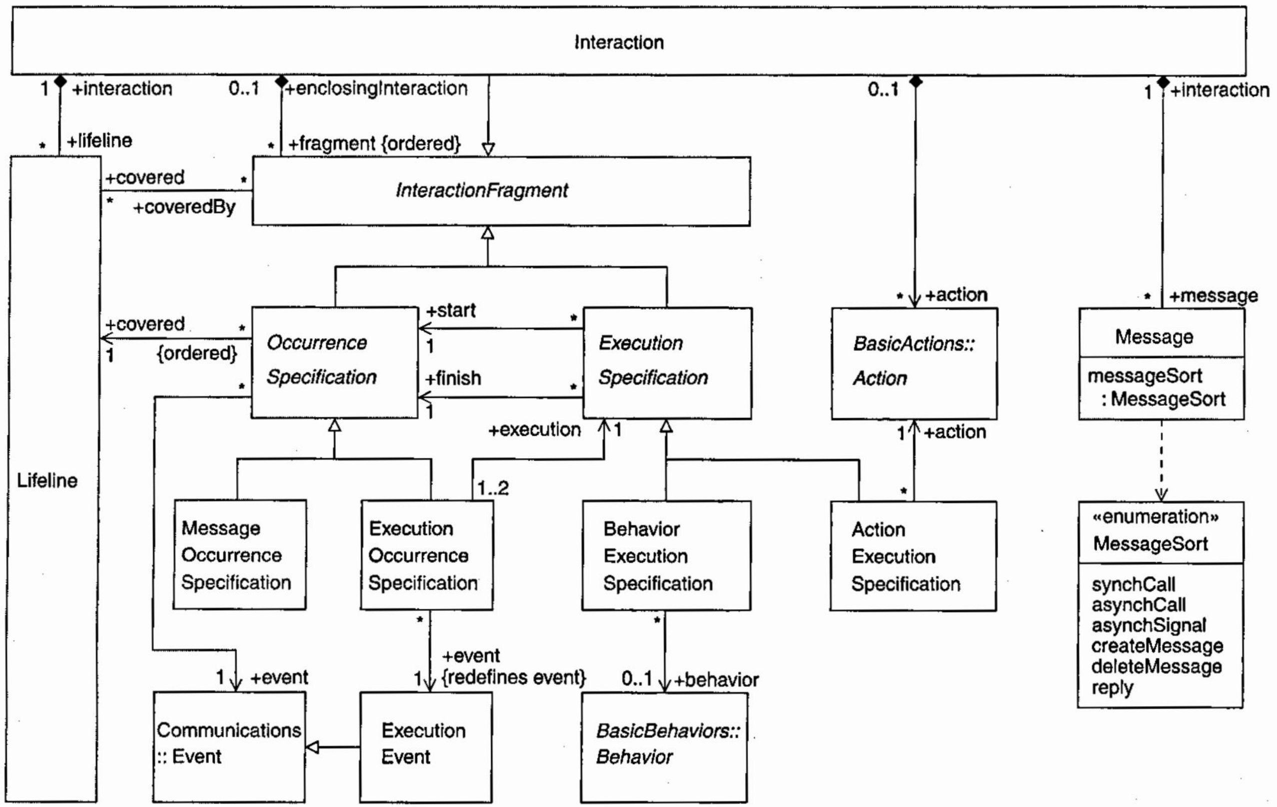


FIGURE 9.8 Fragment of the UML 2 metamodel that is relevant to different communication types and execution specification bars.

Attribute Message::messageSort specifies the type of communication action that was used to generate a message. It has the enumeration type MessageSort with values synchCall, asynchCall, asynchSignal, createMessage, deleteMessage, and reply. These values are read, respectively, as follows: A synchronous call to an operation as in the case of message connect() (see Figure 9.6); an asynchronous call to an operation such as start(); an asynchronous send action as in the case of ready; a pseudomessage standing for the creation of another lifeline instance such as create(x); a pseudomessage standing for the termination of another lifeline (not shown); and a reply message to an operation call such as connect():ok.

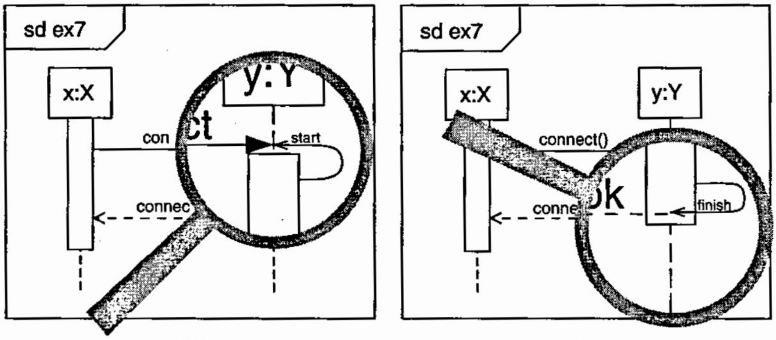
Metaclasses ExecutionSpecification, BehaviorExecutionSpecification, and ActionExecutionSpecification correspond directly to the graphical model elements of the same name (written as separate, uncapitalized words). An ExecutionOccurrenceSpecification represents a point in time at which an action or a behavior starts or finishes. The duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications: the "start ExecutionOccurrenceSpecification" (upper end of an activation bar) and the "finish ExecutionOccurrenceSpecification" (lower end of an activation bar). These two ExecutionOccurrenceSpecifications reference the ExecutionSpecification to which they belong via ExecutionOccurrenceSpecification::execution.[8] Start ExecutionOccurrenceSpecification and finish ExecutionOccurrenceSpecification may coincide[9] if they belong to an ActionExecutionSpecification.

The semantics of an ExecutionSpecification is given by the trace sf where s and f are the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification of the ExecutionSpecification, respectively (and $s \neq f$). In the case of $s = f$, the semantics is simply given by the trace s. An ExecutionSpecification references two OccurrenceSpecifications via ExecutionSpecification::start and ExecutionSpecification::finish (for short: start and finish). Based on our interpretation,[10] start and

[8] We adjusted the multiplicity value at the nonnavigable end of the directed association between ExecutionOccurrenceSpecification and ExecutionSpecification; compare Figure 9.8 with the class diagram of the UML specification document [45, p. 463].

[9] In terms of object identity.

[10] To the authors of the present chapter, the text of the UML specification document [45] that refers to metaclass ExecutionSpecification has appeared difficult to interpret consistently. On the one hand, the specification document states that "the duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications" (p. 478). This is in line with the description of ExecutionOccurrenceSpecifications as "moments in time at which actions or behaviors start or finish" (p. 478). On the other hand, the class diagram on p. 463 of the specification document specifies a multiplicity of 1 at the nonnavigable end of the directed association between ExecutionOccurrenceSpecification and ExecutionSpecification. Furthermore, the type of the association ends ExecutionSpecification::start and ExecutionSpecification::finish is specified as OccurrenceSpecification—not as ExecutionOccurrenceSpecification, as one might expect. This means that MessageOccurrenceSpecifications as well as ExecutionOccurrenceSpecifications may "designate" (p. 479) the start or the finish of a behavior or an action. The question arises whether this actually means that MessageOccurrenceSpecifications may specify the boundary points of the time interval during which a behavior is executed. Two considerations weigh against this interpretation. First, a MessageOccurrenceSpecification cannot reference an ExecutionSpecification. Second, even if the arrival of a message causes execution of a behavior, typically some time elapses between the arrival of the message and the start of the execution.



(a) (b)

FIGURE 9.9 Suggested interpretation of (a) start and (b) finish.

finish generally do *not* coincide with *s* and *f*, respectively. Instead, start references an OccurrenceSpecification which lies so closely above *s* that the time interval between start and *s* is smaller than the resolution limit of the diagram. Typically start is a MessageOccurrenceSpecification of an incoming Message which causes the specified behavior to start [see Figure 9.9(a)]. Association end finish has a similar meaning: It references an OccurrenceSpecification which lies so closely above *f* that the time interval between finish and *f* is smaller than the resolution limit of the diagram. Typically, finish is a MessageOccurrenceSpecification of an outgoing reply message with a return value [see Figure 9.9(b)]. The authors of the present chapter want to point out that the meaning of start and finish which is conveyed by Figure 9.9 is just a *suggestion* for a consistent interpretation of the UML specification text. We further wish to stress that UML interactions do not imply statements about causality. They merely deal with temporal relationships.

A few notes about delete messages and destruction events are necessary. Since DestructionEvent is a specialization of Event – and not of ExecutionEvent – and since ExecutionOccurrenceSpecification::event redefines OccurrenceSpecification::event, a DestructionEvent cannot be referenced by an ExecutionOccurrenceSpecification. Since we decided[11] to regard the metaclass OccurrenceSpecification as abstract, a DestructionEvent can only be referenced by a MessageOccurrenceSpecification. This means that a destruction event cannot occur separately on a lifeline (as it is depicted, for example, in Figure 9.6). Actually, a message head is supposed to point at the destruction event. A delete message lends itself to this purpose. Whenever an instance decides to destruct itself, it has to send itself a delete message.[12]

9.2.3.3 *Semantics* We define a domain *MSort* = {sc, ac, as, cm, dm, r} that corresponds directly to enumeration type MessageSort. The values listed between the

[11] For a (very) short discussion of this question, see footnote 2.

[12] As a matter of fact, this pseudocommunication has little to do with actual processes in a runtime environment.

braces stand for synchCall, asynchCall, asynchSignal, createMessage, deleteMessage, and reply, respectively. Futhermore, we assume a domain for *executions* $\mathbb{X}$ which is the union of two disjoint, primitive subdomains for *behavior executions* and *action executions*. To facilitate a stepwise expansion of our semantics, we introduce a domain of *information sets* $i \in \mathit{Info}$, which for the purposes of this section is defined as $\mathit{Info} = \wp_{\mathrm{fin}}(\mathbb{X}) \times \wp_{\mathrm{fin}}(\mathbb{X})$. If (*start*, *finish*) is the information set of an event occurrence *o*, *start* and *finish* represent the sets of all executions whose ExecutionSpecification references the OccurrenceSpecification of *o* via ExecutionSpecification::start and ExecutionSpecification::finish, respectively. The domain of *events* $\mathbb{E}$ is defined as follows:

$$\begin{aligned} e \in \mathbb{E} ::= {} & \mathsf{snd}(s, \hat{r}, m, ms, i) \\ \mid {} & \mathsf{rcv}(\hat{s}, r, m, ms, i) \\ \mid {} & \mathsf{exec}(l, x, i) \end{aligned}$$

An occurrence of an event of the form $\mathsf{snd}(s, \hat{r}, m, ms, (\mathit{start}, \mathit{finish}))$ means:

1. In the case of $\hat{r} = r$: Dispatch of a message m (with messageKind = complete and messageSort = ms) from sender instance s to receiver instance r.
2. In the case of $\hat{r} = -$: Dispatch of a message m (with messageKind = lost and messageSort = ms) from sender instance s.

An occurrence of an event of the form $\mathsf{rcv}(\hat{s}, r, m, ms, (\mathit{start}, \mathit{finish}))$ means:

3. In the case of $\hat{s} = s$: Arrival of a message m (with messageKind = complete and messageSort = ms) from sender instance s at receiver instance r.
4. In the case of $\hat{s} = -$: Arrival of a message m (with messageKind = found and messageSort = ms) at receiver instance r.

Finally, an occurrence of an event of the form $\mathsf{exec}(l, x, (\mathit{start}, \mathit{finish}))$ means:

5. If x is a behavior execution, the occurrence denotes an (upper or lower) boundary point of the time interval during which the behavior is executed by instance l.
6. If x is an action execution, the occurrence denotes the (idealized) point in time at which the action is performed by instance l.

Using $\hat{m} \in \hat{\mathbb{M}} ::= m \mid -$, the lifeline function α and the message function μ are redefined as follows:

$$\begin{aligned} &\alpha : \mathbb{E} \longrightarrow \wp(\mathbb{I}) && \mu : \mathbb{E} \longrightarrow \hat{\mathbb{M}} \\ &\alpha(\mathsf{snd}(s, \hat{r}, m, ms, i)) = \{s\} && \mu(\mathsf{snd}(s, \hat{r}, m, ms, i)) = m \\ &\alpha(\mathsf{rcv}(\hat{s}, r, m, ms, i)) = \{r\} && \mu(\mathsf{rcv}(\hat{s}, r, m, ms, i)) = m \\ &\alpha(\mathsf{exec}(l, x, i)) \quad\;\; = \{l\} && \mu(\mathsf{exec}(l, x, i)) \quad\;\; = - \end{aligned}$$

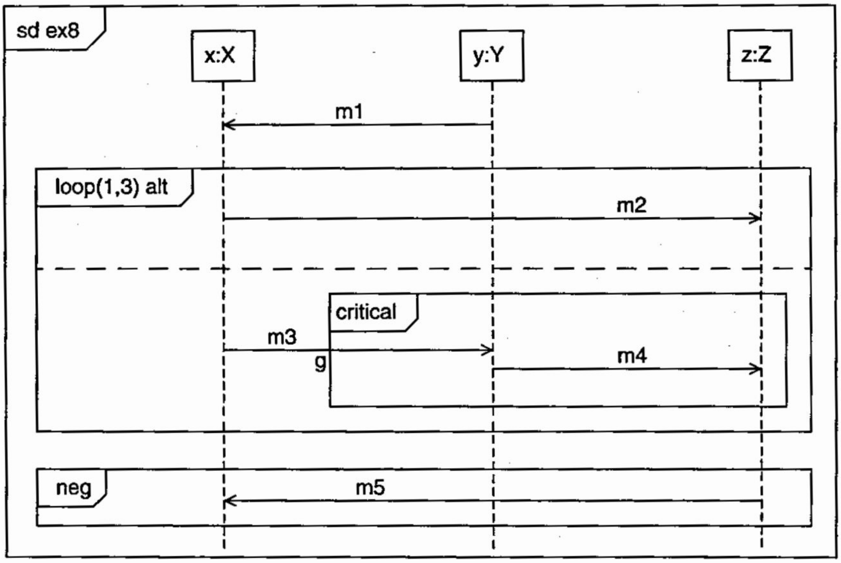


FIGURE 9.10 Sample interaction diagram using combined fragments.

9.2.4 Combined Fragments

9.2.4.1 Complex Interactions All interaction diagrams that have been discussed in previous sections are basic interaction diagrams. In the present section, we turn to *complex interactions*, which are characterized by the presence of *combined fragments*. A combined fragment defines an expression of interaction fragments. It is depicted by a solid-outline rectangle, in which an *interaction operator* is specified in a pentagon in the upper left-hand corner of the rectangle (pentagon descriptor). If the arity of this operator is greater than 1, the *interaction operands* are separated from each other graphically by dashed horizontal lines. More than one operator may be specified in the pentagon descriptor. This is shorthand for nesting combined fragments.

In addition to message m1, the sample interaction ex8 in Figure 9.10 specifies four combined fragments of kind loop(1,3), alt, critical, and neg.[13] The combined fragment with operator kind loop(1,3) has only one operand, which is a combined fragment with operator kind alt. The latter, in turn, has two operands: (1) a basic interaction consisting of message m2, and (2) a combined fragment of type critical together with a message m3 which enters the combined fragment via a *combined fragment gate* named g. No notation is specified for gates. They are merely points on the frame of a combined fragment. However, they may have explicit names.

[13] We postpone an explanation of the semantics of these operators to the following section.

9.2.4.2 Metamodel Figure 9.11 shows the fragment of the UML 2 metamodel that comprises the language constructs for describing complex interactions with gates. As already mentioned in Section 9.2.2.2, InteractionFragment is the abstract notion of the most general interaction unit. InteractionFragment is the root class of a composite pattern and has seven direct subclasses: Interaction, OccurrenceSpecification, ExecutionSpecification (see Section 9.2.3), CombinedFragment, InteractionOperand, StateInvariant (see Section 9.2.5), and Continuation.[14] A CombinedFragment references at least one InteractionOperand via CombinedFragment::operand. Each InteractionOperand is itself an InteractionFragment, and may, moreover, reference any finite number of InteractionFragments via InteractionOperand::fragment.

The semantics of an InteractionFragment is given by a pair of sets of traces: namely, a set of valid (or positive) traces and a set of invalid (or negative) traces. These sets need not be disjoint nor their union cover the entire universe of traces. The semantics of InteractionOperands as well as the semantics of Interactions are *compositional* in the sense that the semantics of an InteractionOperand (or an Interaction) is built mechanically from the semantics of its constituent InteractionFragments.[15] The constituent InteractionFragments are ordered and combined by an implicit seq-operation (weak sequencing).

Given an ordered set of traces $t_1 = e_{1,1}e_{1,2}\dots e_{1,l_1}, \dots, t_n = e_{n,1}e_{n,2}\dots e_{n,l_n}$, the *weak sequencing of* $t_1, \dots, t_n$ is defined by the set of all traces $e_{\pi(1)}e_{\pi(2)}\cdots e_{\pi(l)}$, where $l = l_1 + \cdots + l_n$, and π is a bijection (i.e., a one-to-one and onto mapping), $\pi : \{1, 2, \dots, l\} \to \{(1,1), \dots, (1, l_1), \dots, (n, 1), \dots, (n, l_n)\}$, $i \mapsto (\pi_1(i), \pi_2(i))$ such that for all $1 \le i < j \le l$, the following conditions hold:

1. If $e_{\pi(i)}$ and $e_{\pi(j)}$ lie on the same lifeline, then $\pi_1(i) \le \pi_1(j)$.
2. If $\pi_1(i) = \pi_1(j)$, then $\pi_2(i) < \pi_2(j)$.

The semantics of a CombinedFragment depends on the value of its attribute interactionOperator. This attribute has the enumeration type InteractionOperatorKind with the values strict, seq, par, loop, alt, ignore, neg, assert, critical, break, opt, and consider. A description of the semantics of these operators can be found in the UML specification document 2.1.2 [45, pp. 468–470].[16] We restrict ourselves to citing some defining phrases in Table 9.2, and refer to our formal semantics in Section 9.2.4.4 for the rest.

9.2.4.3 Abstract (Term) Syntax We assume a primitive domain for *gates* $\mathbb{G}$ and use g as a metavariable that ranges over $\mathbb{G}$. The domains $\mathbb{I}_{\mathbb{G}}$ and $\hat{\mathbb{I}}_{\mathbb{G}}$ are defined by $r_{\mathbb{G}}, s_{\mathbb{G}} \in \mathbb{I}_{\mathbb{G}} ::= l \mid g$ and by $\hat{r}_{\mathbb{G}}, \hat{s}_{\mathbb{G}} \in \hat{\mathbb{I}}_{\mathbb{G}} ::= l \mid g \mid -$, respectively. The definition of the

[14] A Continuation allows the concatenation of branches in alternatives; as it is a mere syntactic entity, it is not covered in this chapter.

[15] See UML specification document 2.1.2 [45, pp. 482, 486].

[16] These three pages are a veritable wellspring of hermeneutical problems.

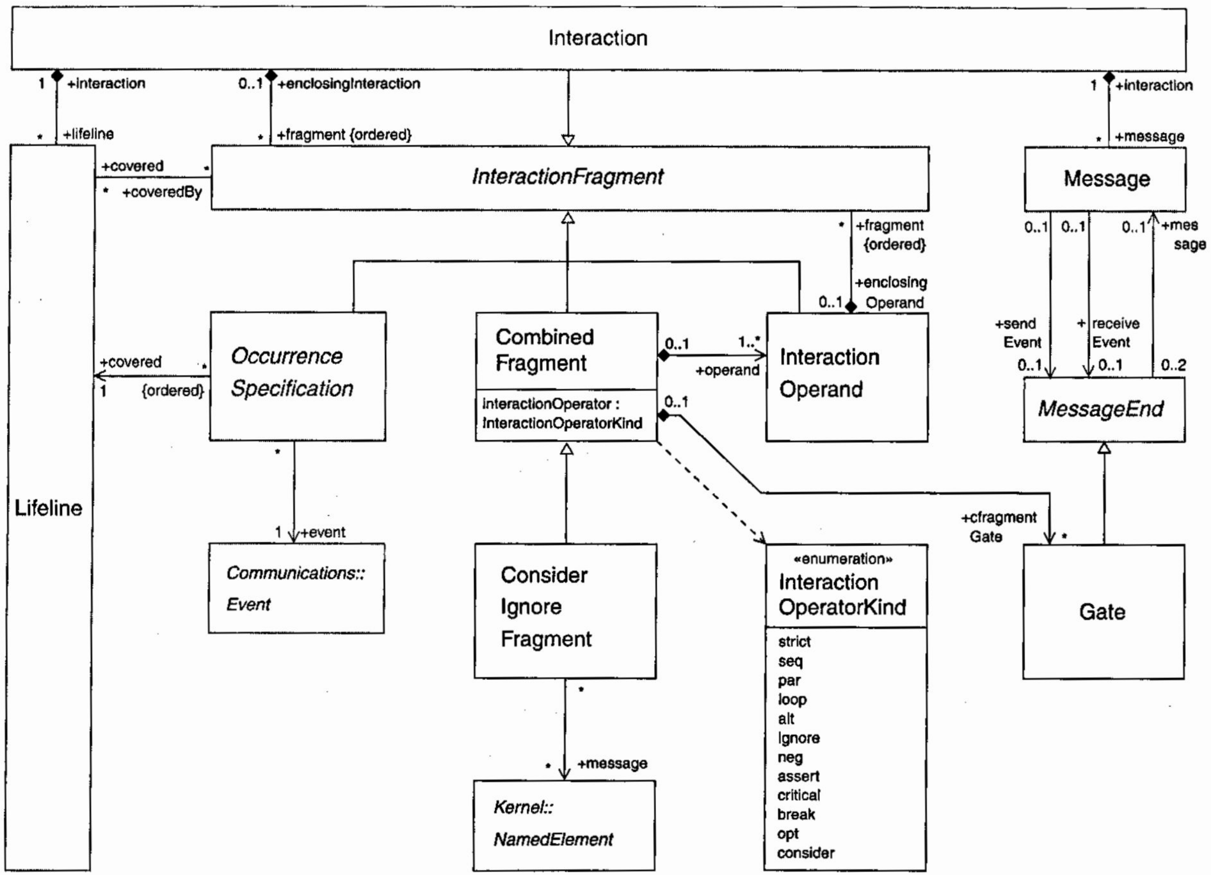


FIGURE 9.11 Fragment of the UML 2 metamodel that comprises the language constructs for describing complex interactions with gates.

TABLE 9.2 Semantics of Interaction Operators in Combined Fragments

strict	"strict designates [read means] that the CombinedFragment represents a strict sequencing between the behaviors of the operands."
seq	"seq designates that the CombinedFragment represents a weak sequencing between the behaviors of the operands."
par	"par designates that the CombinedFragment represents a parallel merge between the behaviors of the operands."
loop	"loop designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times."
alt	"alt designates that the CombinedFragment represents a choice of behavior."
ignore	"ignore designates that there are some message types that are not shown within this combined fragment."
neg	"neg designates that the CombinedFragment represents traces that are defined to be invalid."
assert	"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."
critical	"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."
break	"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."
opt	"opt designates that the CombinedFragment represents a choice of behavior where either the (sole) operand happens or nothing happens."
consider	"consider designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be *ignores*."

Source: Excerpts from UML specification document 2.1.2 [45, pp. 468–470, 473].

domain of events $\mathbb{E}$ is extended as follows:

$$\begin{aligned}
e \in \mathbb{E} &::= e_{\mathrm{r}} \mid e_{\mathrm{p}} \\
e_{\mathrm{r}} \in \mathbb{E}_{\mathrm{r}} &::= \mathsf{snd}(s, \hat{r}_{\mathrm{G}}, m, ms, i) \\
&\quad\mid \mathsf{rcv}(\hat{s}_{\mathrm{G}}, r, m, ms, i) \\
&\quad\mid \mathsf{exec}(l, x, i) \\
e_{\mathrm{p}} \in \mathbb{E}_{\mathrm{p}} &::= \mathsf{gsnd}(g, r_{\mathrm{G}}, m, ms, i) \\
&\quad\mid \mathsf{grcv}(s_{\mathrm{G}}, g, m, ms, i)
\end{aligned}$$

An event e is either a real event e_{r} or a pseudoevent e_{p}, where real events are of the form $\mathsf{snd}(s, \hat{r}, m, ms, i)$, $\mathsf{rcv}(\hat{s}, r, m, ms, i)$, and $\mathsf{exec}(l, x, i)$, as introduced in Section 9.2.3.3. Now, a real event of the form $\mathsf{snd}(s, g, m, ms, i)$ represents the dispatch of a message m (with messageKind = complete and messageSort = ms) from sender instance s to gate g. Similarly, a real event of the form $\mathsf{rcv}(g, r, m, ms, i)$ represents the arrival of a message m (with messageKind = complete and messageSort = ms)

TABLE 9.3 Abstract Syntax of Interaction Terms

$T \in$ IFragment	::=	B_q
	$\mid$	CF
	$\mid$	O
$CF \in$ CFragment	::=	$\mathsf{strict}_q(O_1, O_2)$
	$\mid$	$\mathsf{seq}_q(O_1, O_2)$
	$\mid$	$\mathsf{par}_q(O_1, O_2)$
	$\mid$	$\mathsf{loop}_q(m, \hat{n}, O)$
	$\mid$	$\mathsf{alt}_q(O_1, O_2)$
	$\mid$	$\mathsf{ignore}_q(M, O)$
	$\mid$	$\mathsf{neg}_q(O)$
	$\mid$	$\mathsf{assert}_q(O)$
	$\mid$	$\mathsf{critical}_q(O)$
	$\mid$	$\mathsf{break}_q(O_1, O_2)$
$O \in$ IOperand	::=	T

from gate g at receiver instance r. A pseudoevent of the form $\mathsf{grcv}(s_G, g, m, ms, i)$ occurs whenever a message m coming from sender s_G enters (i.e., arrives at) a gate g. A pseudoevent of the form $\mathsf{gsnd}(g, r_G, m, ms, i)$ occurs whenever a message m has passed through a gate g and leaves the gate on the other side in the direction of r_G. The definitions of the lifeline function α and the message function μ are extended as follows:

$$
\begin{array}{lll@{\qquad}lll}
\alpha : \mathbb{E} \to \wp(\mathbb{I}_G) & & & \mu : \mathbb{E} \to \hat{\mathbb{M}} & & \\
\alpha(\mathsf{snd}(s, \hat{r}, m, ms, i)) & = & \{s\} & \mu(\mathsf{snd}(s, \hat{r}, m, ms, i)) & = & m \\
\alpha(\mathsf{rcv}(\hat{s}, r, m, ms, i)) & = & \{r\} & \mu(\mathsf{rcv}(\hat{s}, r, m, ms, i)) & = & m \\
\alpha(\mathsf{exec}(l, x, i)) & = & \{l\} & \mu(\mathsf{exec}(l, x, i)) & = & - \\
\alpha(\mathsf{gsnd}(g, r_G, m, ms, i)) & = & \{g\} & \mu(\,\mathsf{gsnd}\,(g, r_G, m, ms, i)) & = & m \\
\alpha(\mathsf{grcv}(s_G, g, m, ms, i)) & = & \{g\} & \mu(\,\mathsf{grcv}\,(s_G, g, m, ms, i)) & = & m
\end{array}
$$

The definition of the conflict relation $\asymp$ given in Section 9.2.2 remains unchanged. Note that we consider gate identifiers g as a form of pseudolifelines.

The abstract syntax of interactions is given by the grammar in Table 9.3. Therein, T ranges over terms representing interaction fragments (*terms* for short), B ranges over terms representing basic interactions (*basic terms* or *leaf terms* for short), CF ranges over terms representing combined fragments (*combined terms* for short), O ranges over terms representing interaction operands,[17] m ranges over natural numbers, $\hat{n}$ ranges over natural numbers or ∞, and M ranges over $\wp_{\text{fin}}(\mathbb{M})$. The occurrences of metavariable q that adorn each operator symbol constitute a numbering schema that allows us to identify each basic term uniquely and each loop-operator inside a term. For this purpose we define the domain of paths by $q \in \text{Path} ::= \epsilon \mid q\,n,\ n \in \{1, 2\}$, with 1 denoting a left (or sole) operand and with 2 denoting a right operand. The concatenation of two paths q, q' is written $q.q'$. Each operator symbol inside a term is annotated

[17] In Section 9.2.5, constraints (i.e., guard expressions) are added to the syntax of these terms.

with a unique path identifier q (see Table 9.3). The function top : IFragment → Path ("topmost operator path") maps a term T to the path q with the topmost (or outermost) operator of T is annotated. We inductively define a unary predicate "is well numbered" on terms as follows:

1. B_q is well numbered.
2. If T is a well-numbered term and $uop \in \{\text{loop}, \text{ignore}, \text{neg}, \text{assert}, \text{critical}\}$, and if there is a path r such that $\text{top}(T) = q.1.r$, then $uop_q(T)$ is a well-numbered term.
3. If T_1 and T_2 are well-numbered terms and $bop \in \{\text{strict}, \text{seq}, \text{par}, \text{alt}, \text{break}\}$, and if there are paths r_1 and r_2 such that $\text{top}(T_1) = q.1.r_1$ and $\text{top}(T_2) = q.2.r_2$, then $bop_q(T_1, T_2)$ is a well-numbered term.

In the following, we always use terms with the implicit understanding that these terms are well numbered. Furthermore, we use the name Empty for the term that represents the empty (basic) interaction; Empty is given by $[(\emptyset, \emptyset, \emptyset)]$. opt($T$) abbreviates alt(Empty, T), and consider(M, T) abbreviates ignore($\mathbb{M} \setminus M, T$).

9.2.4.4 *Intermediate Semantics* We define a formal semantics of complex interactions by employing a two-step approach: First, we compositionally define two semantic functions $\mathcal{P}_i[\![-]\!], \mathcal{N}_i[\![-]\!]$: IFragment → $\wp(\mathbb{T})$, which map interaction terms to sets of positive and negative traces, respectively. The pair $(\mathcal{P}_i[\![T]\!], \mathcal{N}_i[\![T]\!])$ is said to be the *intermediate semantics* of a term T. In a second step, filtering functions $\wp(\mathbb{T}) \to \wp(\mathbb{T})$ are employed to map the intermediate semantics to the (definitive) semantics. Pseudoevents and gate identifiers may occur in the intermediate semantics, but they do not occur in the (definitive) semantics.

Both specification of the formal semantics of critical regions that occur in the body of a loop and of the handling of gates require event occurrences to be equipped with additional semantic information. For this purpose, we redefine the domain of information sets as follows:

$$Info = \wp_{\text{fin}}(\mathbb{X}) \times \wp_{\text{fin}}(\mathbb{X}) \times \wp_{\text{fin}}(\text{Path}) \times [\text{Path} \to \mathbb{N}] \times \text{Path}$$

Let (*start*, *finish*, *region*, *loop*, *basic*) be the information set of an event occurrence o. Then *region* is the set of all paths that identify critical-operators whose operands contain the syntactic specification of o. The function *loop* maps a path that identifies a loop-operator to the iteration number of the loop to which the event occurrence o belongs. *loop* is a partial function, with $loop(q) = 0$ meaning "*loop* is not defined at q" because either q does not identify a loop-operator or the operand of the loop-operator does not contain the syntactic specification of o. The path *basic* identifies the basic term that contains the syntactic specification of o. The meanings of *start* and *finish* remain unchanged (see Section 9.2.3).

Let $q \in$ Path and $n \in \mathbb{N}$. Moreover, let f_q, $g_{q,n}$, and h_q denote three functions $\mathbb{E} \to \mathbb{E}$, which are defined as follows. If $e \in \mathbb{E}$ is an event with information set

$i = (\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}, \mathit{basic})$, then $f_q(e)$ is the event that is obtained from e by substituting $(\mathit{start}, \mathit{finish}, \mathit{region} \cup \{q\}, \mathit{loop}, \mathit{basic})$ for i; $g_{q,n}(e)$ is obtained from e by substituting $(\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}[q \mapsto n], \mathit{basic})$ for i; and $h_q(e)$ is obtained from e by substituting $(\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}, q)$ for i. We lift f_q, $g_{q,n}$, and h_q to pomsets (see Section 9.2.1), and for each $p \in \mathbb{D}$ we let $(p)_q$ be defined by $f_q(p)$, $p[q \mapsto n]$ be defined by $g_{q,n}(p)$, and $[p]_q$ be defined by $h_q(p)$. Furthermore, we define the *n-fold iteration* of a process $P \subseteq \mathbb{D}$ with respect to q, written $P_q^{(n)}$, as follows: $P_q^{(0)} = \{\varepsilon\}$ and $P_q^{(n+1)} = P[q \mapsto n+1] \,;_{\bowtie}\, P_q^{(n)}$.

Let $M \subseteq \mathbb{M}$ be a set of messages. On pomsets in $\mathbb{D}$, the filtering relation $\mathit{mfilter}(M) : \mathbb{D} \to \wp(\mathbb{D})$ removes some elements of p whose labels show a message in M. More precisely, we first define $\mathit{mfilter}(M)$ on event-labeled sets as follows. Let O be a set and $\lambda : O \to \mathbb{E}$ a labeling function. Then $O' \in \mathit{mfilter}(M)(O, \lambda)$ if $O' \subseteq O$ and $\mu(\lambda(o)) \in M$ for any $o \in O \setminus O'$. For an event-labeled partial order $(O, \leq_O, \lambda_O)$, we set $(O', \leq_O \cap (O' \times O'), \lambda_O \restriction O') \in \mathit{mfilter}(M)(O, \leq_O, \lambda_O)$ if $O' \in \mathit{mfilter}(M)(O, \lambda_O)$. Finally, we extend these definitions to event-labeled pomsets $p \in \mathbb{D}$ by setting $p \in \mathit{mfilter}(M)([(O, \leq_O, \lambda_O)])$ if there is $(O', \leq_{O'}, \lambda_{O'}) \in \mathit{mfilter}(M)(O, \leq_O, \lambda_O)$ such that $p = [(O', \leq_{O'}, \lambda_{O'})]$. The relation $\mathit{mfilter}(M)$ obviously is well defined. Given a pomset $p \in \mathbb{D}$, by $\mathit{mfilter}(M)^{-1}(p)$ we denote $\{q \in \mathbb{D} \mid p \in \mathit{mfilter}(M)(q)\}$. This "inverse relation" is lifted to processes in the usual way (see Section 9.2.1). Given a process $P \subseteq \mathbb{D}$, we write $P\langle M\rangle$ for $\mathit{mfilter}(M)^{-1}(P)$. Furthermore, we define $\lhd(P)$ to be the prefix closure of P.

The intermediate semantics of complex interactions is given by a pair of semantic functions $\mathcal{P}_{\mathrm{i}}[\![-]\!]$ and $\mathcal{N}_{\mathrm{i}}[\![-]\!]$ that map interaction terms to sets of positive and negative traces, respectively; see Tables 9.4 and 9.5. These sets constitute an intermediate semantics since their traces may contain pseudoevents as well as gate identifiers.

The semantics of the positive fragment of the language closely follows the textual description of the specification (see Table 9.2). Indeed, the literature [12,36,49] shows a broad consensus on the semantics of this fragment. On the contrary, for the negative fragment the specification leaves room for different interpretations, and consequently diverging proposals have been made (see also [27]). In line with Kobro Runde et al. [36], we have adopted the view that a trace is negative for an interaction fragment whenever it has exhaustively traversed a negative subfragment. Only assert is an exception to this rationale, since, following the specification, it makes negative everything that is not explicitly positive.

9.2.4.5 *Filtering* Let $\mathbb{T}_{\mathrm{r}}$ be the set of all traces of occurrences of real events that do not contain gate identifiers. We define a filter $\mathcal{F}_{\mathrm{gate}} : \wp(\mathbb{T}) \to \wp(\mathbb{T}_{\mathrm{r}})$. This filter (1) removes all pseudoevents from a trace, (2) replaces all gate identifiers in events of the form $\mathsf{snd}(s, g, m, ms, i)$ and $\mathsf{rcv}(g, r, m, ms, i)$ with the lifelines of the actual receiver and the actual sender of the message, respectively, and (3) discards the trace if the actual sender or the actual receiver of a message cannot be determined or if the trace is malformed for some other reason. This "gate filter" works as follows. Let $P \subseteq \mathbb{T}$ be a process consisting of traces. For each trace $t \in P$, the following rewriting rule (R)

TABLE 9.4 Intermediate Semantics of Complex Interactions (Positive Fragment)

$$
\begin{aligned}
&\mathcal{P}_{\mathrm{i}}[\![-]\!] : \mathrm{IFragment} \to \wp(\mathbb{T}) \\
&\mathcal{P}_{\mathrm{i}}[\![B_q]\!] &&= [B{\downarrow}]_q \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{strict}_q(O_1, O_2)]\!] &&= \mathcal{P}_{\mathrm{i}}[\![O_1]\!] \,;\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!] \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{seq}_q(O_1, O_2)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O_1]\!] \,;_{\asymp}\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{par}_q(O_1, O_2)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O_1]\!] \parallel \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{loop}_q(m, \hat{n}, O)]\!] &&= \textstyle\bigcup_{m \le i < \hat{n}+1} ((\mathcal{P}_{\mathrm{i}}[\![O]\!])_q^{(i)}){\downarrow} \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{alt}_q(O_1, O_2)]\!] &&= \mathcal{P}_{\mathrm{i}}[\![O_1]\!] \cup \mathcal{P}_{\mathrm{i}}[\![O_2]\!] \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{ignore}_q(M, O)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O]\!]\langle M \rangle){\downarrow} \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{neg}_q(O)]\!] &&= \{\varepsilon\} \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{assert}_q(O)]\!] &&= \mathcal{P}_{\mathrm{i}}[\![O]\!] \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{critical}_q(O)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O]\!])_q \\
&\mathcal{P}_{\mathrm{i}}[\![\mathsf{break}_q(O_1, O_2)]\!] &&= \mathcal{P}_{\mathrm{i}}[\![O_1]\!] \cup (\lhd(\mathcal{P}_{\mathrm{i}}[\![O_1]\!]) \,;_{\asymp}\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow}
\end{aligned}
$$

TABLE 9.5 Intermediate Semantics of Complex Interactions (Negative Fragment)

$$
\begin{aligned}
&\mathcal{N}_{\mathrm{i}}[\![-]\!] : \mathrm{IFragment} \to \wp(\mathbb{T}) \\
&\mathcal{N}_{\mathrm{i}}[\![B_q]\!] &&= \emptyset \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{strict}_q(O_1, O_2)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O_1]\!] \,;\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]) \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \,;\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!]) \\
& &&\quad \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \,;\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]) \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{seq}_q(O_1, O_2)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O_1]\!] \,;_{\asymp}\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \,;_{\asymp}\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
& &&\quad \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \,;_{\asymp}\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{par}_q(O_1, O_2)]\!] &&= (\mathcal{P}_{\mathrm{i}}[\![O_1]\!] \parallel \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \parallel \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
& &&\quad \cup (\mathcal{N}_{\mathrm{i}}[\![O_1]\!] \parallel \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{loop}_q(m, \hat{n}, O)]\!] &&= \textstyle\bigcup_{m \le i < \hat{n}+1} ((\mathcal{P}_{\mathrm{i}}[\![O]\!] \cup \mathcal{N}_{\mathrm{i}}[\![O]\!])_q^{(i)}){\downarrow} \setminus ((\mathcal{P}_{\mathrm{i}}[\![O]\!])_q^{(i)}){\downarrow} \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{alt}_q(O_1, O_2)]\!] &&= \mathcal{N}_{\mathrm{i}}[\![O_1]\!] \cup \mathcal{N}_{\mathrm{i}}[\![O_2]\!] \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{ignore}_q(M, O)]\!] &&= (\mathcal{N}_{\mathrm{i}}[\![O]\!]\langle M \rangle){\downarrow} \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{neg}_q(O)]\!] &&= \mathcal{P}_{\mathrm{i}}[\![O]\!] \cup \mathcal{N}_{\mathrm{i}}[\![O]\!] \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{assert}_q(O)]\!] &&= (\mathbb{T} \setminus \mathcal{P}_{\mathrm{i}}[\![O]\!]) \cup \mathcal{N}_{\mathrm{i}}[\![O]\!] \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{critical}_q(O)]\!] &&= (\mathcal{N}_{\mathrm{i}}[\![O]\!])_q \\
&\mathcal{N}_{\mathrm{i}}[\![\mathsf{break}_q(O_1, O_2)]\!] &&= \mathcal{N}_{\mathrm{i}}[\![O_1]\!] \cup (\lhd(\mathcal{P}_{\mathrm{i}}[\![O_1]\!]) \,;_{\asymp}\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
& &&\quad \cup (\lhd(\mathcal{N}_{\mathrm{i}}[\![O_1]\!]) \,;_{\asymp}\, \mathcal{P}_{\mathrm{i}}[\![O_2]\!]){\downarrow} \\
& &&\quad \cup (\lhd(\mathcal{N}_{\mathrm{i}}[\![O_1]\!]) \,;_{\asymp}\, \mathcal{N}_{\mathrm{i}}[\![O_2]\!]){\downarrow}
\end{aligned}
$$

is iteratively applied to t as long as the rule matches:

$$
\begin{aligned}
&(R) \qquad t_1\, a\, t_2\, b\, t_3\, c\, t_4\, d\, t_5 \longrightarrow t_1\, a'\, t_2\, t_3\, t_4\, d'\, t_5 \\
&\text{where } a = \mathsf{snd}(s, g, m, ms, (\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}, q)), \\
&\phantom{\text{where }} b = \mathsf{grcv}(s, g, m, ms, (\_, \_, \_, \_, q)), \\
&\phantom{\text{where }} c = \mathsf{gsnd}(g, l_{\mathrm{G}}, m, ms, (\_, \_, \_, \_, q')), \\
&\phantom{\text{where }} d = k(g, l_{\mathrm{G}}, m, ms, (\mathit{start}', \mathit{finish}', \mathit{region}', \mathit{loop}', q')) \\
&\phantom{\text{where }} a' = \mathsf{snd}(s, l_{\mathrm{G}}, m, ms, (\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}, q)), \\
&\phantom{\text{where }} d' = k(s, l_{\mathrm{G}}, m, ms, (\mathit{start}', \mathit{finish}', \mathit{region}', \mathit{loop}', q')),
\end{aligned}
$$

TABLE 9.6 Semantics of Complex Interactions

$\mathcal{P}\llbracket - \rrbracket, \mathcal{N}\llbracket - \rrbracket$: IFragment $\longrightarrow \wp(\mathbb{T}_{\mathrm{r}})$		
$\mathcal{P}\llbracket - \rrbracket$	$=$	$\mathcal{F}_{\mathrm{crit}} \circ \mathcal{F}_{\mathrm{gate}} \circ \mathcal{P}_{\mathrm{i}}\llbracket - \rrbracket$
$\mathcal{N}\llbracket - \rrbracket$	$=$	$\mathcal{F}_{\mathrm{crit}} \circ \mathcal{F}_{\mathrm{gate}} \circ \mathcal{N}_{\mathrm{i}}\llbracket - \rrbracket$

with (($k =$ rcv and $l_{\mathrm{G}} \in \mathbb{I}$) or ($k =$ grcv and $l_{\mathrm{G}} \in \mathbb{G}$)), and $t_1, t_2, t_3, t_4 \in \mathbb{T}$ do not contain a, b, c, d, respectively. If the resulting trace is an element of $\mathbb{T}_{\mathrm{r}}$ (i.e., if it does not contain pseudoevents or gate identifiers), the trace is retained as an element of $\mathcal{F}_{\mathrm{gate}}(P)$; otherwise, the trace is discarded.

A filter $\mathcal{F}_{\mathrm{crit}} : \wp(\mathbb{T}_{\mathrm{r}}) \rightarrow \wp(\mathbb{T}_{\mathrm{r}})$ is required to prevent traces from violating atomicity constraints specified by critical-constructs. For the purpose of defining this filter, let $\varrho : \mathbb{E} \rightarrow \wp_{\mathrm{fin}}(\mathrm{Path})$ and $\ell : \mathbb{E} \rightarrow [\mathrm{Path} \rightarrow \mathbb{N}]$ be two functions that map an event e to the third and fourth components of the information set of e, respectively. For each path q, the set $\mathrm{prefix}(q) = \{q' \mid \exists q''.\, q = q'q''\}$ contains all prefixes of q; in particular, $q \in \mathrm{prefix}(q)$. We say that a (finite) trace $t = e_1 e_2 \cdots e_n \in \mathbb{T}_{\mathrm{r}}$ *preserves atomicity* if for all $1 \leq i \leq j \leq k \leq n$ and for all $q \in \mathrm{Path}$, $q \in \varrho(e_j)$ and $\ell(e_i) \restriction \mathrm{prefix}(q) = \ell(e_j) \restriction \mathrm{prefix}(q)$ whenever $q \in \varrho(e_i) \cap \varrho(e_k)$ and $\ell(e_i) \restriction \mathrm{prefix}(q) = \ell(e_k) \restriction \mathrm{prefix}(q)$. Given a process $P \subseteq \mathbb{T}_{\mathrm{r}}$, we define $\mathcal{F}_{\mathrm{crit}}(P)$ by $\{t \in P \mid t \text{ preserves atomicity}\}$.

9.2.4.6 Semantics The semantics of complex interactions is given by a pair of semantic functions $\mathcal{P}\llbracket - \rrbracket$ and $\mathcal{N}\llbracket - \rrbracket$ that map terms to sets of positive and negative traces of occurrences of real events, respectively (see Table 9.6). Therein, $\mathcal{P}_{\mathrm{i}}\llbracket - \rrbracket$ and $\mathcal{N}_{\mathrm{i}}\llbracket - \rrbracket$ are the semantic functions defined in Tables 9.4 and 9.5, respectively. The filtering functions $\mathcal{F}_{\mathrm{gate}}$ and $\mathcal{F}_{\mathrm{crit}}$ are defined in Section 9.2.4.5.

9.2.5 Constraints

9.2.5.1 Guards and State Invariants The sample interaction ex9 in Figure 9.12 illustrates the use of guards and state invariants. The combined fragment of type alt which is contained in ex9 has two guarded operands: The first (upper) operand has the guard z.p >= 1; the second (lower) operand has an else-guard. These guards have the following meanings:

1. If the upper operand is chosen and the dispatch of m2 occurs before the dispatch of m3, the Boolean expression z.p >= 1 has to be true with respect to the global state of the system that exists directly before the dispatch of m2.
2. If the upper operand is chosen and the dispatch of m3 occurs before the dispatch of m2, the Boolean expression z.p >= 1 has to be true with respect to the global state that exists directly before the dispatch of m3.
3. If the lower operand is chosen, the Boolean expression z.p >= 1 has to be false with respect to the global state directly before the dispatch of m4.

The state invariant 2 > z.p >= 0 on the lifeline of z is evaluated in the global state of the system that exists directly before the next event occurs on the same lifeline

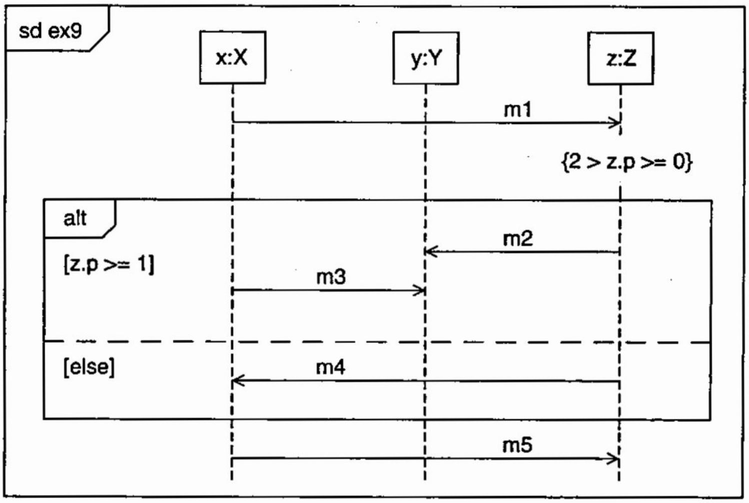


FIGURE 9.12 Sample interaction diagram using guards and state invariants.

after the state invariant. This may be the dispatch of m2, the dispatch of m4, or the arrival of m5.

9.2.5.2 Metamodel Figure 9.13 shows the fragment of the UML 2 metamodel that is relevant to guards and state invariants. Considering the class diagram, an

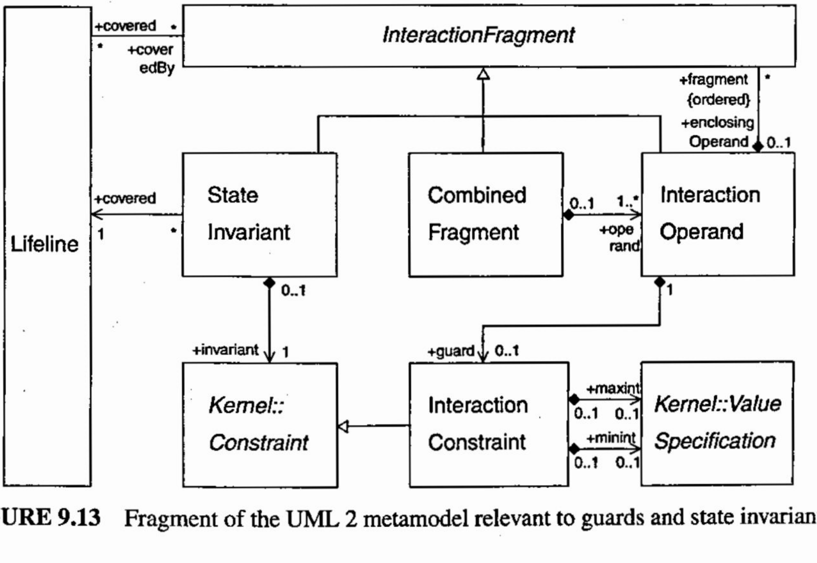


FIGURE 9.13 Fragment of the UML 2 metamodel relevant to guards and state invariants.

InteractionConstraint may be assigned only to an InteractionOperand (as a whole).[18] We therefore recommend placing a guard anywhere in the frame of an interaction operand.

9.2.5.3 *Abstract Syntax* We assume a domain of constraints $C \in \text{Constraint}$ whose syntax is left unspecified, except for the requirement that the domain contains the logical connectives $\vee$, $\neg$, and the logical expression true. Interaction operands are equipped with these constraints as follows (see Table 9.3):

$$O \in \text{IOperand} ::= [C]T$$

The constraint C that occurs in an interaction operand $[C]T$ acts as a guard for T. If an InteractionOperand (for fragments represented by T) does not specify a guard, the InteractionOperand is translated into our term syntax as $[\mathsf{true}]T$. A CombinedFragment of type $k \in \{\mathsf{strict}, \mathsf{seq}, \mathsf{par}\}$ with more than two InteractionOperands is translated using the following syntactic transformation:

$$\begin{aligned}&k([C_1]T_1, \ldots, [C_n]T_n)\\ &= k([C_1]T_1, [\mathsf{true}]k([C_2]T_2, \ldots, [\mathsf{true}]k([C_{n-1}]T_{n-1}, [C_n]T_n)\ldots))\end{aligned}$$

A CombinedFragment of type alt with $n \geq 0$ InteractionOperands is interpreted as $\mathsf{alt}([C_1]T_1, \ldots, [C_n]T_n, [\neg(C_1 \vee \cdots \vee C_n)]\mathsf{Empty})$.[19] If $n \geq 2$, the latter is translated using the syntactic transformation given above. A CombinedFragment of type loop with operand $[C]T$, lower bound m, and upper bound $\hat{n}$ is translated using the following syntactic transformation:[20]

$$\mathsf{loop}(m, \hat{n}, [C]T) = \underbrace{\mathsf{seq}(T, \ldots, \mathsf{seq}(T,}_{m \text{ times}} \mathsf{loop}(0, \hat{n} - m, [C]T))\ldots)$$

Furthermore, a new sort of pseudoevent is introduced, which represents a state invariant C lying on lifeline l (see Section 9.2.4.3):

$$e_p \in \mathbb{E}_p \ ::= \ \ldots \mid \mathsf{stateinv}(l, C, i)$$

We define $\alpha(\mathsf{stateinv}(l, C, i))$ by $\{l\}$, and $\mu(\mathsf{stateinv}(l, C, i))$ by $-$.

[18] However, several passages in the UML specification document indicate that an InteractionConstraint is—at least graphically—assigned to a particular lifeline: namely, "the lifeline where the first event occurrence [of the interaction operand] will occur" [45, p. 484]. In addition, there is a formal constraint specifying that a "guard must be placed directly prior to (above) the OccurrenceSpecification that will become the first OccurrenceSpecification within this InteractionOperand" [45, p. 486]. Since the minimum of a partial order of event occurrences is, in general, not determined uniquely, the quoted passages of the specification document appear ill-formed to the authors of the present chapter.

[19] UML specification document 2.1.2 [45, p. 468] states that if "none of the operands [of a combined fragment of kind alt] has a guard that evaluates to true, [...] the remainder of the enclosing InteractionFragment is executed." This means that the set of positive traces of such an alt-fragment is $\{\varepsilon\}$ (and not $\emptyset$).

[20] UML specification document 2.1.2 [45, p. 470] states that "a loop will iterate minimum 'minint' number of times [...]. After the minimum number of iterations have executed and the Boolean expression is false the loop will terminate." In our opinion, this means that the Boolean expression is not to be evaluated during the first 'minint' iterations.

9.2.5.4 Semantics Let Σ be the set of all global states of the overall system. We assume a semantic function $\mathcal{C}[\![-]\!]$: Constraint $\rightarrow (\Sigma \rightarrow \{tt, f\!f\})$ which maps a constraint and a global state to a Boolean value.[21] Furthermore, we assume that $\mathcal{C}[\![-]\!]$ interprets $C_1 \vee C_2$, $\neg\, C$, and true in the canonical way.

A pair $(\sigma, e) \in \Sigma \times \mathbb{E}$ is said to be a *stateful event*. We define $\underline{\mathbb{E}}$ by $\Sigma \times \mathbb{E}$ and use $\underline{e}$ as a metavariable that ranges over $\underline{\mathbb{E}}$. The lifeline function α, the message function μ, and the functions ϱ and ℓ (see Section 9.2.4.5) are defined on stateful events (σ, e) by applying the respective functions to the second component e. Two stateful events $\underline{e}_1$ and $\underline{e}_2$ are in conflict, written $\underline{e}_1 \asymp \underline{e}_2$, if $\alpha(\underline{e}_1) \cap \alpha(\underline{e}_2) \neq \emptyset$. The domains of $\underline{\mathbb{E}}$-labeled pomsets $\underline{\mathbb{D}}$, $\underline{\mathbb{P}}$, and $\underline{\mathbb{T}}$ correspond directly to $\mathbb{D}$, $\mathbb{P}$, and $\mathbb{T}$, respectively. For each (σ, e) that occurs in a trace $\underline{t} \in \underline{\mathbb{T}}$, the state σ denotes the global state of the overall system *directly before* the event e occurs. If the occurrence of e depends on whether a certain constraint C evaluates to tt in the state σ, the evaluation of C (to tt) and the event e occur *atomically*.

A compositional definition of the semantics of interaction operands $[C]T$—namely, the one whose term T produces an empty trace ε—requires a refined definition of events:

$$e \in \mathbb{E} \ ::= \ e_{\mathrm{r}} \mid e_{\mathrm{p}} \mid \varepsilon(i)$$

Therein, e_{r} and e_{p} range over real events and pseudoevents, respectively, as they are defined in Section 9.2.4.3. A stateful event $(\sigma, \varepsilon(i))$ is said to be a *state marker*. A state marker is a special form of pseudoevent that occurs only in traces, but not in syntactic terms. We set $\alpha(\sigma, \varepsilon(i)) = \emptyset$ (i.e., a state marker does not conflict with any event). An occurrence of a state marker $(\sigma, \varepsilon(i))$ in a trace $\underline{t} \in \underline{\mathbb{T}}$ indicates that the state of the system at this point of the trace is σ. A state marker $(\sigma, \varepsilon(i))$ is inserted in a trace automatically whenever the constraint C of an interaction operand $[C]T$ is evaluated in a state σ, the result of the evaluation is tt, and the term T produces an empty trace.[22]

For each $C \in$ Constraint, we define a filter $\mathcal{F}_C : \wp(\underline{\mathbb{T}}) \rightarrow \wp(\underline{\mathbb{T}})$ which (1) discards each nonempty trace that starts with a stateful event (σ, e) such that C is false in σ, and (2) replaces the empty trace ε (if any) with the set of all state markers $(\sigma, \varepsilon(\emptyset))$ such that C is true in σ. The symbol $\emptyset$ denotes an information set with an empty *region* component and a totally undefined *loop* component. Formally, we define

$$\begin{aligned}\mathcal{F}_C(P) = &\{\underline{t} \in P \mid \exists \sigma \in \Sigma, e \in \mathbb{E}, \underline{t}' \in \underline{\mathbb{T}}.\, \underline{t} = (\sigma, e)\underline{t}' \wedge \mathcal{C}[\![C]\!]\sigma = tt\} \\ &\cup \{(\sigma, \varepsilon(\emptyset)) \mid \varepsilon \in P \wedge \sigma \in \Sigma \wedge \mathcal{C}[\![C]\!]\sigma = tt\}\end{aligned}$$

[21] The proposal by Calegari García et al. [9] uses OCL/RT, an extension of OCL for real time (see [11,44]), which in fact is based on a three-valued logic.

[22] The latter is indicated by the symbol ε in the state marker.

TABLE 9.7 Intermediate Semantics of Interactions with Constraints (Positive Fragment)[a]

$\mathcal{P}_{\mathrm{i}}\llbracket - \rrbracket : \mathbb{I}\mathrm{Fragment} \to \wp(\mathbb{T})$	
$\mathcal{P}_{\mathrm{i}}\llbracket B_q \rrbracket$	$= \Sigma([B{\downarrow}]_q)$
$\mathcal{P}_{\mathrm{i}}\llbracket [C]T \rrbracket$	$= \mathcal{F}_C\,(\mathcal{P}_{\mathrm{i}}\llbracket T \rrbracket)$

[a] Only clauses differing from Table 9.4 are shown.

TABLE 9.8 Intermediate Semantics of Interactions with Constraints (Negative Fragment)[a]

$\mathcal{N}_{\mathrm{i}}\llbracket - \rrbracket : \mathbb{I}\mathrm{Fragment} \to \wp(\underline{\mathbb{T}})$	
$\mathcal{N}_{\mathrm{i}}\llbracket \mathsf{assert}_q(O) \rrbracket$	$= \underline{\mathbb{T}} \setminus \mathcal{P}_{\mathrm{i}}\llbracket O \rrbracket$
$\mathcal{N}_{\mathrm{i}}\llbracket [C]T \rrbracket$	$= \mathcal{F}_{\neg C}\,(\mathcal{P}_{\mathrm{i}}\llbracket T \rrbracket) \cup \mathcal{N}_{\mathrm{i}}\llbracket T \rrbracket$

[a] Only clauses differing from Table 9.5 are shown.

A mapping $\Sigma(-)$ is defined that transforms an $\mathbb{E}$-labeled pomset $p \in \mathbb{D}$ into a set of $\underline{\mathbb{E}}$-labeled pomsets, thereby pairing off the labels e of p with all possible combinations of states $\sigma \in \Sigma$, that is, $\Sigma(\varepsilon) = \{\varepsilon\}$ and $\Sigma(p) = \{[(O, \leq_O, \lambda'_O)] \mid \exists \sigma_O : O \to \Sigma . \lambda'_O = (\sigma_O, \lambda_O)\}$ for each $p = [(O, \leq_O, \lambda_O)] \in \mathbb{D} \setminus \{\varepsilon\}$.

The intermediate semantics of interactions with constraints is given by a pair of semantic functions $\mathcal{P}_{\mathrm{i}}\llbracket - \rrbracket$ and $\mathcal{N}_{\mathrm{i}}\llbracket - \rrbracket$ that map interaction terms to sets of positive and negative traces, respectively (see Tables 9.7 and 9.8).[23]

9.2.6 High-Level Interactions

9.2.6.1 References to Interactions The sample interaction ex10 shown in Figure 9.14(a) references another interaction ex11 shown in Figure 9.14(b) in a ref. Intuitively, the referenced interaction is expanded into the place where it is referred to. In fact, the UML specification also allows to use arguments for formal parameters of interactions; we do not handle parameters in this chapter.

9.2.6.2 Metamodel Figure 9.15 shows the fragment of the UML 2 metamodel that is relevant to high-level interactions. An InteractionUse refers to an Interaction via refersTo. The InteractionUse must cover all Lifelines of the enclosing Interaction that appear within the referred Interaction. An InteractionUse has an ordered set of arguments that must correspond to the parameters of the referred Interaction. Furthermore, an InteractionUse has a set of actualGates that must match the formalGates of

[23] The set of traces positively associated with a constrained interaction coincides with the definition by Calegari García et al. [9]. On the contrary, the set of traces negatively associated with a constrained interaction does not; in that work, $\mathcal{N}_{\mathrm{i}}\llbracket [C]T \rrbracket = \mathcal{F}_{\neg C}\,(\underline{\mathbb{T}}) \cup \mathcal{N}_{\mathrm{i}}\llbracket T \rrbracket$.

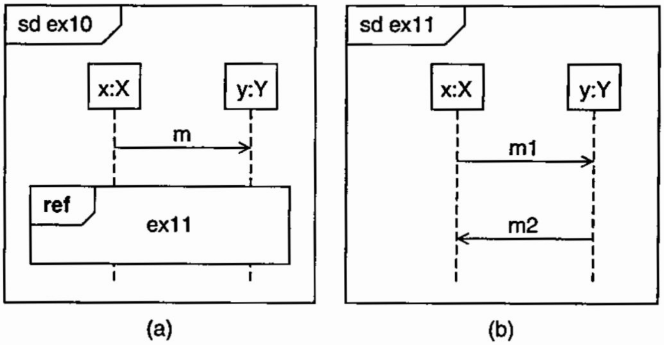


FIGURE 9.14 (a) Sample interaction diagram ex10 using references to (b) another interaction (ex11).

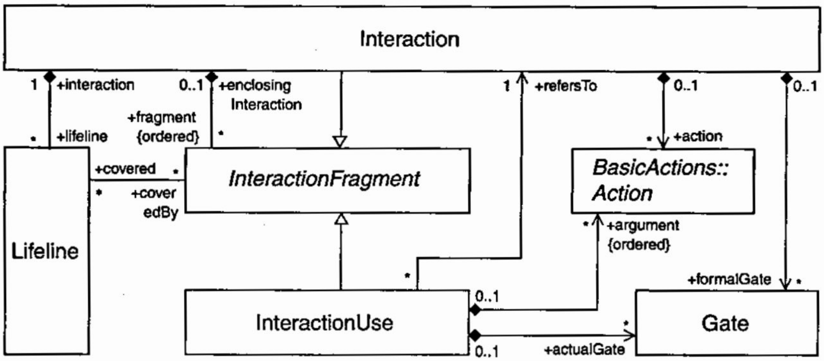


FIGURE 9.15 Fragment of the UML 2 metamodel relevant to high-level interactions.

the referred Interaction. Since parameters and gates are mere syntactic constructs, we do not handle them in our formal semantics.

9.2.6.3 Semantics Let us assume a syntactic category Name of *names*. The abstract syntax of interactions is given by the grammar in Table 9.9. An *interaction environment* ν is a set of interactions $\mathsf{sd}(n, T)$ where all interactions in ν have a different name n.

Let $q \in$ Path. A function $f_q : \mathbb{E} \to \mathbb{E}$ is defined as follows: If $e \in \mathbb{E}$ is an event with information set $i = (\mathit{start}, \mathit{finish}, \mathit{region}, \mathit{loop}, \mathit{basic})$, then $f_q(e)$ is the event that is obtained from e by substituting $(\mathit{start}, \mathit{finish}, \mathit{region}', \mathit{loop}', \mathit{basic}')$ for i, with basic',

TABLE 9.9 Abstract Syntax of High-Level Interaction Terms

$n \in$ Name		
$S \in$ Interaction	$::=$	$\mathsf{sd}(n, T)$
$T \in$ IFragment	$::=$	$\ldots \mid \mathsf{ref}(n)$

TABLE 9.10 Intermediate Semantics of High-Level Interactions (Positive Fragment)[a]

$\mathcal{P}_\mathrm{i}\llbracket - \rrbracket : \text{Interaction} \to (\text{IEnvironment} \to \wp(\mathbb{T}))$
$\mathcal{P}_\mathrm{i}\llbracket \mathsf{ref}_q(n) \rrbracket \nu = (\mathcal{P}_\mathrm{i}\llbracket T \rrbracket \nu)_{q.}\ \textit{if}\ \mathsf{sd}(n, T) \in \nu$

[a] Only clauses differing from Table 9.4 are shown.

TABLE 9.11 Intermediate Semantics of High-Level Interactions (Negative Fragment)[a]

$\mathcal{N}_\mathrm{i}\llbracket - \rrbracket : \text{Interaction} \to (\text{IEnvironment} \to \wp(\mathbb{T}))$
$\mathcal{N}_\mathrm{i}\llbracket \mathsf{ref}_q(n) \rrbracket \nu = (\mathcal{N}_\mathrm{i}\llbracket T \rrbracket \nu)_{q.}\ \textit{if}\ \mathsf{sd}(n, T) \in \nu$

[a] Only clauses differing from Table 9.5 are shown.

$region'$, and $loop'$ being defined as $q.basic$, $\{q.q' \mid q' \in region\}$, and

$$loop'(q'') := \begin{cases} loop(q') & \text{if } q'' = q.q' \\ 0 & \text{otherwise} \end{cases}$$

respectively. The function $f'_q : \mathbb{E} \to \mathbb{E}$ is defined on stateful events (σ, e) by applying the function f_q to the second component e. We lift f'_q to pomsets (see Section 9.2.1). For each $p \in \mathbb{D}$ we define $(p)_{q.}$ by $f'_q(p)$.

The intermediate semantics of high-level interactions is given by a pair of semantic functions $\mathcal{P}_\mathrm{i}\llbracket - \rrbracket$ and $\mathcal{N}_\mathrm{i}\llbracket - \rrbracket$ which map interaction terms to sets of positive and negative traces, respectively (see Tables 9.10 and 9.11).

It appears that the specification treats interaction uses by macro expansion ("The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is" [45, p. 487]). If also (mutually) recursive interactions are to be handled, some notion of fixpoint in constructing the semantics has to be involved. The semantic functions $\mathcal{P}_\mathrm{i}\llbracket - \rrbracket$ and $\mathcal{N}_\mathrm{i}\llbracket - \rrbracket$ can be rendered as a monotonic $F : \wp(\mathbb{T}) \times \wp(\mathbb{T}) \to \wp(\mathbb{T}) \times \wp(\mathbb{T})$, where $\wp(\mathbb{T}) \times \wp(\mathbb{T})$ is equipped with the ordering $(X_1, X_2) \subseteq (X'_1, X'_2)$ if, and only if, $X_1 \subseteq X'_1$ and $X_2 \subseteq X'_2$; thus, we are assured that the least and the greatest fixpoint exist. If the least fixpoint is chosen, only finite (stateful) traces can be produced; if the greatest fixpoint is chosen, infinite (stateful) traces are also possible.

9.3 ALTERNATIVE SEMANTICS

9.3.1 Operational Semantics

An operational semantics for a part of the positive fragment of the term language of UML interactions is defined. The reduced syntax of the term language is given in Table 9.12. Therein, B ranges over Basic (as defined in Section 9.2.2) and p ranges over Path (see Section 9.2.4).

TABLE 9.12 Reduced Syntax of Interaction Terms

$T \in \mathbb{I}\text{Fragment}$	$::=$	B_q
	$\mid$	$\mathsf{strict}_q(T_1, T_2)$
	$\mid$	$\mathsf{seq}_q(T_1, T_2)$
	$\mid$	$\mathsf{par}_q(T_1, T_2)$
	$\mid$	$\mathsf{loop}_q(T)$
	$\mid$	$\mathsf{alt}_q(T_1, T_2)$

9.3.1.1 Domains and Restriction Functions

We define the domain $\mathbb{E}_\tau$ of events and the *silent event* τ as $\mathbb{E} \cup \{\tau\}$. The domain $\mathbb{D}_\tau$ comprises all finitary pomsets labeled with events from $\mathbb{E}_\tau$. We define $\alpha(\tau) = \emptyset$ (i.e., the silent event does not conflict with any event). The domains $\mathbb{P}_\tau$ and $\mathbb{T}_\tau$ comprise all pomsets $p \in \mathbb{D}_\tau$ such that p is locally linear and a trace, respectively. We extend the lifeline function α to pomsets $p = [(O, \leq_O, \lambda_O)] \in \mathbb{D}_\tau$ by $\alpha(p) = \bigcup_{o \in O} \alpha(\lambda_O(o))$. Given a process $P \subseteq \mathbb{D}_\tau$, we set $\alpha(P) = \bigcup_{p \in P} \alpha(p)$. On processes in $\wp(\mathbb{D}_\tau)$ and for a set of lifelines L, the *restriction* function $restr(L) : \wp(\mathbb{D}_\tau) \to \wp(\mathbb{D}_\tau)$ removes all those pomsets from a process which show an event that lies on a lifeline of L [i.e., $restr(L)(P) = \{p \in P \mid \alpha(p) \cap L = \emptyset\}$]. We also write $P[L]$ for $restr(L)(P)$.

Transition rules regarding the construct $\mathsf{seq}_q(T_1, T_2)$ can only be correct (with respect to the denotational semantics) if it is guaranteed that after execution of an event e of the term T_2, no event e' of T_1 is executed that conflicts with e. If a nondeterministic choice construct (alt or loop) occurs in term T_1, it may happen that both traces containing events conflicting with e (type 1) and traces not containing such events (type 2) occur in the positive evaluation set of T_1. The desired completeness of the transition rules necessitates retaining traces of type 2, even though traces of type 1 have to be discarded. One possible solution to this problem is based on a syntactic transformation $\mathrm{R}_L : \text{Term} \to \text{Term}$ such that $\mathcal{P}[\![\mathrm{R}_L(T)]\!] = \mathcal{P}[\![T]\!][L]$, where Term is an appropriate extension of the language $\mathbb{I}$Fragment and L is a set of lifelines. Typically, this "syntactic restriction function" is defined by $\mathrm{R}_L(T) = \mathsf{restr}(L, T)$, where $\mathsf{restr}(L, T)$ is a language extension whose denotational semantics is given by $\mathcal{P}[\![\mathsf{restr}(L, T)]\!] = \mathcal{P}[\![T]\!][L]$ and whose operational semantics is given by the following rule:

$$\text{(restr)}\ \frac{T \xrightarrow{\bar{e}} T'}{\mathsf{restr}(L, T) \xrightarrow{\bar{e}} \mathsf{restr}(L, T')}\quad \text{if } \alpha(\bar{e}) \cap L = \emptyset$$

However, we choose a slightly different approach using a language extension None whose denotational semantics is given by $\mathcal{D}[\![\mathsf{None}]\!] = \emptyset$. The operational semantics is given simply by the fact that there is no rule for None. The definition of the syntactic restriction function R_L is shown in Table 9.13. Therein, *uop* is a unary operator and *bop* is a binary operator. By induction on the structure of $T \in$ Term, it can easily be shown that $\mathcal{P}[\![\mathrm{R}_L(T)]\!] = \mathcal{P}[\![T]\!][L]$.

TABLE 9.13 Syntactic Restriction Function (for $L \subseteq \mathbb{I}$)

$\mathrm{R}_L : \mathrm{Term} \longrightarrow \mathrm{Term}$		
$\mathrm{R}_L(\mathsf{None}_q)$	$=$	None_q
$\mathrm{R}_L(B_q)$	$=$	$\begin{cases} B_q & \textit{if } \alpha(B) \cap L = \emptyset \\ \mathsf{None}_q & \text{otherwise} \end{cases}$
$\mathrm{R}_L(uop_q(T))$	$=$	$uop_q(\mathrm{R}_L(T))$
$\mathrm{R}_L(bop_q(T_1, T_2))$	$=$	$bop_q(\mathrm{R}_L(T_1), \mathrm{R}_L(T_2))$

Furthermore, we introduce a function ren : Path × $\{1,2\}$ × Term ⟶ Term, which is defined as follows:

$$\begin{aligned}
\mathrm{ren}(p, n, const) &= const \\
\mathrm{ren}(p, n, uop_q(T)) &= \begin{cases} uop_{p.n.q'}(\mathrm{ren}(p, n, T)) & \text{if } q = p.q' \\ uop_q(T) & \text{otherwise} \end{cases} \\
\mathrm{ren}(p, n, bop_q(T_1, T_2)) &= \begin{cases} bop_{p.n.q'}(\mathrm{ren}(p, n, T_1), & \\ \qquad\qquad \mathrm{ren}(p, n, T_2)) & \text{if } q = p.q' \\ bop_q(T_1, T_2) & \text{otherwise} \end{cases}
\end{aligned}$$

We also write ${}_{p.n}T$ for $\mathrm{ren}(p, n, T)$.

9.3.1.2 Transition System A configuration of our operational small-step semantics is a term $T \in \mathrm{Term}$. The only terminal configuration is Empty, which is defined by ε. Transitions are of the form $T \xrightarrow{\bar{e}} T'$ with $T, T' \in \mathrm{Term}$, $T \neq \mathsf{Empty}$, and $\bar{e} \in \mathbb{E}_\tau$. The rules for the transition relation are shown in Table 9.14. In these rules the variously decorated metavariables range as follows: T over Term, e over $\mathbb{E}$, and $\bar{e}$ over $\mathbb{E}_\tau$. Given a locally linear pomset B, Min (B) is defined as the set of all events of B that are minimal with respect to the ordering of B. $B \setminus \{e\}$ is obtained from B by removing the (unique) occurrence of e.

9.3.2 Event Structures

An alternative definition of the formal semantics of UML 2.0 interactions is given by Küster Filipe [39,40]. The language is enriched with OCL constraints (see also [10]) as well as locations and temperature as defined for LSCs (see [16]). These additions serve the needs of expressing liveness properties such as progress of a lifeline or the requirement that a sent message may or must be received. The approach concentrates on positive behavior; that is, it does not consider negative traces as in the semantics defined above. Then Küster Filipe [40] purposely disregards some interaction building operators: neg since forbidden behavior can be expressed by a false state invariant appended to the interaction modeling that behavior, and assert since mandatory behavior can be indicated by a hot interaction fragment. The presentation is, moreover, simplified by the omission of loop and strict, which can be integrated. In other words, only alt, par, and seq are considered. The operator alt can have more than

TABLE 9.14 Operational Semantics of Interactions (Part of Positive Fragment)

(basic)	$B_q \xrightarrow{e} (B\setminus\{e\})_q \quad \text{if } e \in \mathrm{Min}\,(B)$		
(strict^1)	$\dfrac{T_1 \xrightarrow{\bar{e}} T_1'}{\mathsf{strict}_q(T_1, T_2) \xrightarrow{\bar{e}} \mathsf{strict}_q(T_1', T_2)}$	(strict^2)	$\mathsf{strict}_q(\mathsf{Empty}_{q'}, T_2) \xrightarrow{\tau} T_2$
(seq^1)	$\dfrac{T_1 \xrightarrow{\bar{e}} T_1'}{\mathsf{seq}_q(T_1, T_2) \xrightarrow{\bar{e}} \mathsf{seq}_q(T_1', T_2)}$	(seq^2)	$\mathsf{seq}_q(\mathsf{Empty}_{q'}, T_2) \xrightarrow{\tau} T_2$
(seq^3)	$\dfrac{T_2 \xrightarrow{\bar{e}} T_2'}{\mathsf{seq}_q(T_1, T_2) \xrightarrow{\bar{e}} \mathsf{seq}_q(\mathrm{R}_{\alpha(\bar{e})}(T_1), T_2')}$		
(par^1)	$\dfrac{T_1 \xrightarrow{\bar{e}} T_1'}{\mathsf{par}_q(T_1, T_2) \xrightarrow{\bar{e}} \mathsf{par}_q(T_1', T_2)}$	(par^2)	$\dfrac{T_2 \xrightarrow{\bar{e}} T_2'}{\mathsf{par}_q(T_1, T_2) \xrightarrow{\bar{e}} \mathsf{par}_q(T_1, T_2')}$
(par^3)	$\mathsf{par}_q(\mathsf{Empty}_{q'}, T_2) \xrightarrow{\tau} T_2$	(par^4)	$\mathsf{par}_q(T_1, \mathsf{Empty}_{q'}) \xrightarrow{\tau} T_1$
(loop^1)	$\mathsf{loop}_q(T) \xrightarrow{\tau} \mathsf{Empty}_q$	(loop^2)	$\mathsf{loop}_q(T) \xrightarrow{\tau} \mathsf{seq}_q(T, \mathsf{loop}_{q.2}({}_{q.2}T))$
(alt^1)	$\mathsf{alt}_q(T_1, T_2) \xrightarrow{\tau} T_1$	(alt^2)	$\mathsf{alt}_q(T_1, T_2) \xrightarrow{\tau} T_2$

two operands, each of them must be accompanied by a precondition, and at most one of them is executed. The semantic domain is that of labeled event structures, a true-concurrent model that naturally captures alternative and parallel behavior (see [55]). Labeled event structures are nothing but labeled pomsets (see [48]) equipped with a binary conflict relation. The abstract syntax for interactions is considerably more involved than the one given above; in return, there is a relatively easy way to define, given an interaction term, the conditions on a labeled event structure that, on the one hand, satisfy the interaction and, on the other, may possibly lose cold messages. Besides this logic for interobject communication, Küster Filipe [40] defines a *home logic* for the description of intraobject behavior.

Küster Filipe expands her semantics by including the ref operator [39]. This operator references an interaction fragment which appears in a different diagram. This fragment is called an *interaction use*. By means of ref, interactions can be decomposed or, put the other way, defined hierarchically and reused. Furthermore, ref allows the decomposition of lifelines, whose messages can trespass the diagram boundaries through gates. Lifeline decomposition can be used for modeling components whose internals are hidden or unknown. Küster Filipe addresses refinement by means of a categorical construction over two categories of labeled event structures [39]. In this setting, refinement consists of solving references to interactions and gates. This definition aims at formal reasoning and verification of complex scenario-based interobject behavioral models; these matters have not been worked out yet.

This semantics over event structures, restricted to the simplest operators, is comparable to the positive semantics presented in the sections above. Beyond the core constructs, the proposals seem to diverge, as different language fragments and extensions are considered in each case.

9.3.3 Other Formalisms: MSCs and LSCs

The language of message sequence charts (MSCs [30]) is designed to describe the interaction between a number of independent message-passing instances. MSC is a graphical scenario language, equipped with a formal semantics (see [24,25,32,41], to name a few) and nevertheless of practical use. MSC captures interobject communication patterns typically emerging from use cases, and is easily used in conjunction with other methods and notations. An MSC basic diagram usually contains an MSC heading, a representation for one or more instances, possibly a condition, input and output events including perhaps messages to the environment, and in some cases instance terminations. An MSC diagram may refer to another one, and messages arising from a referred diagram exit this diagram through a gate. The MSC language became more sophisticated, allowing, besides higher-order diagrams, alternatives and restrictive conditions, general ordering, inline expressions, data, time, object orientation, remote method calls, and so on.

Although widely used in industry, MSCs are expressively weak, as they permit only the specification of sample scenarios that are based semantically simply on the notion of partial order of events. MSCs allegedly turn from existential into universal specification of behavior as the requirements evolve to a more formal and/or specific design (see [7]). In particular, the language of MSCs leaves a number of questions open like, such as specification of mandatory behavior, safety and liveness properties, and activation time.

Live sequence charts (LSCs, [16]) increase the expressive power of MSCs by the addition of constructs that allow the specification of liveness properties. LSCs impose a clear distinction between possible and mandatory behavior and at both the global and local levels.

As with basic MSCs, the elementary building blocks of an LSC are instances and messages. Instances are depicted by an instance head, a lifeline, and possibly an instance end. Messages are represented by arrows connecting lifelines. There are two types of messages, synchronous and asynchronous. The former are associated with horizontal arrows (→), the latter with slanted arrows with half stick heads (↘). LSCs allow the specification of time constraints either in the form of an MSC-style timer or in interval notation. Mandatory behavior is specified by universal charts, possible scenarios by an existential chart. Along lifelines a number of locations are identified: for example, the point depicting the arrival of a message. Locations, messages, and conditions have a temperature, hot or cold. Hot locations enforce progress (i.e., the instance must move beyond the location), whereas at cold locations the instance need not move farther. Hot messages imply that the message, if sent, will be received, whereas cold messages may be lost. Hot conditions must be met; cold conditions that fail to hold imply that the chart is to be exited.

An LSC consists, in general, of a pre-chart and a chart. The live interpretation of such an LSC requires that the behavior specified by the chart *must* be exhibited by a system whenever the system has shown the behavior specified by the pre-chart. Live elements, called *hot* (indicating that progress is enforced), make it possible to define forbidden scenarios. Mandatory and possible conditions, invariants, and other finesses, such as simultaneous regions and coregions, activation, and quantification may also be specified.

The formal semantics of LSCs is based on the concept of timed Büchi automata (see [7]). The acceptance criterion for Büchi automata takes the infiniteness of the words into account. Timed Büchi automata also take the occurrence times of the letters of words into account.

The language of LSCs is thus much more expressive than that of MSCs or of UML 2.0 interactions. Consequently, LSCs require a more involved domain for the definition of a formal semantics. Complexity and expressive power of LSCs have been studied by a number of people (see, e.g., [6,17,28]). Additionally, Harel and Maoz [27] treat the constructs assert and neg not as operators but as modalities, give an interpretation of them into LSCs, and define a UML 2.0 profile for the positive fragment of the language of interactions that includes those modalities; the resulting language is called modal sequence diagrams (MSDs).

9.4 IMPLEMENTATION AND REFINEMENT

The trace-based semantics of the preceding section assigns a pair of sets of traces to each interaction: positive and negative traces. This semantics has been developed by following the UML 2 specification as closely as possible. However, the specification does not tell under which circumstances a given system can be said to be complying with an interaction, or, put differently, when a system is an implementation of an interaction. Moreover, it would be rather useful to also have a notion of refinement for interactions.

9.4.1 Implementation

For discussing possible notions of implementation we take a system abstractly to be a set of traces over stateful events $\underline{\mathbb{E}}$ (see Section 9.2.5.4); that is, we assume that implementations and the interpretation of interactions are grounded in a common semantic domain. For concrete systems this representation in terms of traces may be, at least partially, achieved by appropriate instrumentation in order to monitor their particular stateful event occurrences.

As a first possible notion of implementation [12], we say that a system $I \subseteq \underline{\mathbb{T}}$ *implements* an interaction S, written as $I \models S$, if $I \cap \mathcal{P}[\![S]\!] \neq \emptyset$ and $I \cap \mathcal{N}[\![S]\!] = \emptyset$, (i.e., if I shows at least one positive trace and does not show any negative trace). The definition is sensible, since it can easily be verified by induction that in the trace-based semantics each interaction shows at least one positive trace.[24] Another possibility [36] is to require a system simply not to show any negative traces but

[24] When an interaction operator such as "refuse" [36] is introduced, which does not show positive traces, the implementation relation $\models$ may be weakened as follows: $I \models S$ if $\mathcal{P}[\![S]\!] \neq \emptyset$, $I \cap \mathcal{P}[\![S]\!] \neq \emptyset$, and $I \cap \mathcal{N}[\![S]\!] = \emptyset$.

to be indifferent to positive and inconclusive traces. This notion of implementation assumes that an assert is used to rule out inconclusive traces.

Either definition allows us to handle *interaction formulas*, introducing boolean connectives for interactions. Writing $S_1 \wedge S_2$ for "interaction S_1 *and* interaction S_2 must hold," then $I \models S_1 \wedge S_2$ amounts to $I \models S_1$ and $I \models S_2$ in the classical sense [i.e., $I \cap (\mathcal{P}[\![S_1]\!] \cap \mathcal{P}[\![S_2]\!]) \neq \emptyset$ and $I \cap (\mathcal{N}[\![S_1]\!] \cup \mathcal{N}[\![S_2]\!]) = \emptyset$]. Similarly, writing $\neg S$ for "interaction S must *not* hold," then $I \models \neg S$, again interpreted classically, amounts to $I \cap \mathcal{P}[\![S]\!] = \emptyset$ or $I \cap \mathcal{N}[\![S]\!] \neq \emptyset$. Note that $I \models \neg S$ and $I \models \mathsf{neg}(T)$ [with $S = \mathsf{sd}(n, T)$] are quite different. We can also introduce an *or* connective $S_1 \vee S_2$ as an abbreviation for $\neg((\neg S_1) \wedge (\neg S_2))$, and again $I \models S_1 \vee S_2$ is quite different from $I \models \mathsf{alt}(T_1, T_2)$ [with $S_1 = \mathsf{sd}(n_1, T_1)$, $S_2 = \mathsf{sd}(n_2, T_2)$].

However, a single interaction or a set of interactions to be interpreted conjunctively rarely are used to describe an entire system. Generally, interactions are employed for describing particular situations of communication and interaction, and these situations may come up only once in awhile in a system and need not cover its complete behavior. This can be expressed in interactions by surrounding the interaction fragment describing such a partial behavior by ignore or consider. But what is left open is the possibility of identifying when a given interaction has to be obeyed during system execution and when it is not relevant, that is, being able to define a *precondition* under which an interaction takes effect (see the discussion on LSCs in Section 9.3.3). Let us write $S_1 \rhd S_2$ to mean informally: If interaction S_1 occurs in an implementation, then S_2 has to occur afterward. This amounts formally to defining $I \models S_1 \rhd S_2$ to hold if for all $t_1 \in \underline{\mathbb{T}}$ and $t_2 \in \underline{\mathbb{T}}$ with $t_1 ;_{\bowtie} t_2 \in I$, if $\{t_1\} \models S_1$, then $\{t_2\} \models S_2$.

9.4.2 Refinement

Refinement is a well-known concept in computer science. Given any specification formalism, be it a model or a program, *refinement* refers to the verifiable transformation of an abstract (high-level) word into a concrete (low-level) word of that language. Refinement can also cross language boundaries and relate a specification with a program; in this case the relation is sometimes called *implementation*. The emphasis here is put on the verifiability of the transformation. For this purpose, a formal semantics is indispensable.

An implementation relation between interaction diagrams (or interaction terms) and sets of traces supplies the natural basis for the definition of refinement: An interaction S' refines an interaction S, denoted by $S \rightsquigarrow S'$ if any implementation of S' is also an implementation of S (see [12]). Obviously, this refinement relation is reflexive, transitive, and antisymmetric (i.e., a partial order). This definition is the classical, model-theoretic notion; other possibilities are conceivable, like syntactical transformation of terms such that some conditions hold (e.g., the transformation rules are semantics preserving or semantics narrowing).

On the one hand, $S \rightsquigarrow S'$ if $I \models S'$ implies that $I \models S$. On the other, $I \models S$ if $I \cap \mathcal{P}[\![S]\!] \neq \emptyset$ and $I \cap \mathcal{N}[\![S]\!] = \emptyset$. Therefore, refinement is verifiable.

Notice that if associated with S' there are more positive traces and fewer negative traces than with S (i.e., if $\mathcal{P}[\![S]\!] \subseteq \mathcal{P}[\![S']\!]$ and $\mathcal{N}[\![S']\!] \subseteq \mathcal{N}[\![S]\!]$), then $S \rightsquigarrow S'$. This is not necessarily the only possibility. The verification of refinement via computing the sets of positive and negative traces can become very cumbersome. More interesting than this type of mathematical gymnastics with pairs of pairs of arbitrarily big trace sets is an inference system that allows the derivation of pairs of interaction terms in the refinement relation. Unfortunately, the interaction-building operators do not possess very useful properties in combination with a notion of refinement based on model inclusion. For instance, in general they are not monotonic: that is $S_1 \rightsquigarrow S_1'$ does not imply $op(S_1, \dots) \rightsquigarrow op(S_1', \dots)$ for every operator op. In some cases an inference is possible; a number of rules is given by Calegari García et al. [9,12].

A different notion of refinement in terms of reduction of uncertainty is given by Störrle: Two interactions are in the refinement relation when the sets of positive and negative traces of the abstract interaction, respectively, are included in the sets of positive and negative traces of the concrete interaction (see [49,50]). This definition requires disjointness of the sets of positive and negative traces associated with an arbitrary interaction.

In contrast, Kobro Runde et al. [36] require disambiguation of inconclusive traces and/or narrowing of the set of positive traces, thus reducing underspecification. This work includes an enlightening discussion on the differences between underspecification, object of disambiguation by refinement, and inherent nondeterminism, which is not to be removed from the abstract specification. The approach is part of STAIRS [37], a framework for stepwise development based on refinement of interaction specifications. Some interaction-building operators are not monotonic with respect to this notion of refinement, as discussed by Oldevik and Haugen [46]. Lund [43] presents a trace generation algorithm which to a great extent conforms[25] with the denotational semantics for interactions defined in STAIRS, as well as algorithms for test generation and test execution. Therein, trace generation and refinement à la STAIRS are used to devise a method for refinement verification.

9.5 VERIFICATION AND VALIDATION

As descriptions of emergent behaviors, interactions lend themselves to be seen as properties of a system that have to be verified. This view is also reflected by the notions of implementation in Section 9.4.1. On the other hand, interactions may also be interpreted as executable, high-level specifications, which should be used for validation in system development.

9.5.1 Model Checking

To verify that a particular interaction is indeed satisfied by a given system, research has concentrated mostly on the fully automatic technique of model checking. Interactions

[25] The operational semantics for seq does not take into account possible interleavings of events in the two operands [see rule (seq_G^3) in Table 9.14].

are turned into logical, temporal formulas or directly to some kind of automata that then can be run against the system.

Model checking of MSCs and LSCs has been studied in great detail (see [5,7, 33,42]). For UML interactions based on this previous work, a translation of a fragment of interactions into interaction automata has been developed [35]. The language fragment handles basic interactions and the operators seq, par, strict, ignore, as well as state invariants; loop is restricted to containing only basic interactions, as otherwise the model-checking problem becomes undecidable [1]. The interaction automata, interpreted as Büchi automata, are checked against instrumented UML state machines using the model translation tool Hugo/RT and the model checker Spin.

A similar goal is followed by Charmy [3]. The focus of Charmy is on architectural descriptions and verification of their consistency. The semantics of interactions, given by translation rules, however, deviates from the one presented here; currently, combined fragments are not supported.

9.5.2 Animation

The most outstanding example of interaction animation is the Play Engine (see [29]). The tool implements an extension of LSCs and supports two techniques. The first, called *play-in*, allows the intended system to be supplied with scenario-based behavior specificatons using a graphical user interface. The second, called *play-out*, permits execution or animation of the behavior specified. These two techniques combined constitute the play-in/play-out methodology.

The Play Engine can be used in more than one phase of system development: for instance, for requirements elicitation and for prototyping and testing. The authors also propose use of the Play Engine to program reactivity, which is based on interobject communication and thus closer to the way in which systems and their behavior are conceived. This is only possible because the language of LSCs was extended with symbolic instances and allows a message to cause a change of state in the destination instance. In this way, an instance can react to incoming messages also according to its internal state.

The animation, or play-out, is highly nontrivial. The scenarios specified may be very sophisticated, including the above-mentioned symbolic instances and state changes caused by message processing as well as the entire paraphernalia of LSCs, such as time and forbidden elements, may and must conditions, hot and cold messages, and universal and existential charts.

The semantics of the LSCs extension is not given just in the form of a tool. The Play Engine is accompanied by an operational semantics given as a transition system whose definition is based on the concepts of object-oriented system modeling and cut of a chart. There are two types of transitions, steps and supersteps. A superstep is a sequence of steps, and a step is an event carried out by the system in response to the input by the user. Once a stimulus has arrived, and due to underspecification and/or nondeterminism, more than one superstep may be enabled. Some of these enabled supersteps may, however, lead to an inconsistent system state that violates a constraint. The smart play-out mechanism of the Play Engine uses formal analysis methods, mainly model checking, to find a correct superstep if one exists, or to prove that a correct superstep does not exist. Complex case studies have been carried out using the Play Engine, such as the one reported by Combes et al. [14].

Apart from the Play Engine there are other tools, like Rhapsody (see [4]) and Unistep (see [52]), that support interaction animation. Because these tools are only commercially available, details on the respective realizations are not public.

A further animation of interactions was reported by Burd et al. [8], focusing on comprehensibility of interactions. An experiment was carried out which from a pedagogical point of view, showed that the number of misinterpretations considerably declines when users are in front of an animated interaction instead of a static representation. Animated interactions are also employed for requirements testing in the spirit of control flow analysis (see [23]).

Some other approaches translate interactions into a formalism susceptible to animation. Fernandes et al. [20] translate interactions, possibly including the operators opt, alt, par, and loop, and the interaction fragment ref, into colored Petri nets, which are then animated. In a similar manner, a set of MSCs can be translated into a statechart (see [26,38]) which is susceptible to animation (see, e.g., [4,18,53]).

REFERENCES

1. R. Alur and M. Yannakakis. Model checking of message sequence charts. In J. C. M. Baeten and S. Mauw, eds., *10th International Conference on Concurrency Theory* (*CONCUR'99, Proceedings*), Lecture Notes in Computer Science, vol. 1664, pp. 114–129. Springer-Verlag, New York, 1999.
2. E. P. Andersen. Conceptual modeling of objects: a role modeling approach. Ph.D. dissertation, Universitetet i Oslo, Oslo, Norway, 1997.
3. M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools* (*SCESM'06, Proceedings*), pp. 21–27. ACM Press, New York, 2006.
4. R. Boldt. Model-driven architecture, embedded developers and Telelogic Rhapsody. http://www.telelogic.com/download/get_file.cfm?id=4890, Oct. 2008. Accessed Nov. 28, 2008.
5. Y. Bontemps and P. Heymans. Turning high-level live sequence charts into automata. In *ICSE Workshop on Scenarios and State-Machines: Models, Algorithms and Tools* (*SCESM'02, Proceedings*), Orlando, FL, 2002.
6. Y. Bontemps and P.Y. Schobbens. The computational complexity of scenario-based agent verification and design. *Journal of Applied Logic*, 5(2):252–276, 2007.
7. M. Brill, W. Damm, J. Klose, B. Westphal, and H. Wittke. Live sequence charts: an introduction to lines, arrows, and strange boxes in the context of formal verification. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, eds., *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation* (*DFG*), *Final Report*. Lecture Notes in Computer Science, vol. 3147, pp. 374–399. Springer-Verlag, New York, 2004.

8. E. Burd, D. Overy, and A. Wheetman. Evaluating using animation to improve understanding of sequence diagrams. In *10th International Workshop on Program Comprehension (IWPC'02, Proceedings)*, pp. 107–113. IEEE Computer Society, Los Alamitos, CA, 2002.
9. D. Calegari García, M. V. Cengarle, and N. Szasz. UML 2.0 interactions with OCL/RT constraints. In *Forum on Specification and Design Languages (FDL'08, Proceedings)*, pp. 167–172. IEEE, New York, 2008.
10. A. Cavarra and J. Küster Filipe. Combining sequence diagrams and OCL for liveness. *Electronic Notes in Theoretical Computer Science*, 115:19–38, 2005.
11. M. V. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, eds., *International Symposium of Formal Methods Europe (FME'02, Proceedings)*, Lecture Notes in Computer Science, vol. 2391, pp. 390–409. Springer-Verlag, New York, 2002.
12. M. V. Cengarle and A. Knapp. UML 2.0 interactions: semantics and refinement. In J. Jürjens, E. B. Fernandez, R. France, and B. Rumpe, eds., *3rd International Workshop on Critical Systems Development with UML (CSDUML'04, Proceedings)*, Technical Report TUM-I0415, pp. 85–99. Institut für Informatik, Technische Universität München, Munich, Germany, 2004.
13. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Object-Oriented Series. Prentice Hall, Upper Saddle River, NJ, 1994.
14. P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Software and Systems Modeling*, 7(2):157–175, 2008.
15. S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Object-Oriented Series. Prentice Hall, Upper Saddle River, NJ, 1994.
16. W. Damm and D. Harel. LSCs: breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
17. W. Damm, T. Toben, and B. Westphal. On the expressive power of live sequence charts. In T. W. Reps, M. Sagiv, and J. Bauer, eds., *Program Analysis and Compilation: Theory and Practice—Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, vol. 4444, pp. 225–246. Springer-Verlag, New York, 2007.
18. B. P. Douglass, D. Harel, and M. Trakhtenbrot. Statecharts in use: structured analysis and object-orientation. In F. Vaandrager and G. Rozenberg, eds., *Lectures on Embedded Systems*, Lecture Notes in Computer Science, vol. 1494, pp. 368–394. Springer-Verlag, New York, 1998.
19. G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In France and Rumpe [22, pp. 473–488].
20. J. Fernandes, S. Tjell, J. B. Jørgensen, and Ó. Ribeiro. Designing tool support for translating use cases and UML 2.0 sequence diagrams into a coloured Petri net. In *6th International Workshop on Scenarios and State Machines (SCESM'07, Proceedings)*, p. 2, Washington, DC, 2007. IEEE Computer Society, Los Alamitos, CA, 2007.
21. T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz. Timed sequence diagrams and tool-based analysis: a case study. In France and Rumpe [22, pp. 645–660].
22. R. France and B. Rumpe, eds. *2nd International Conference on the Unified Modeling Language (UML'99, Proceedings)*, Lecture Notes in Computer Science, vol. 1723. Springer-Verlag, New York, 1999.

23. V. Garousi, L. Briand, and Y. Labiche. Control flow analysis of UML 2.0 sequence diagrams. In A. Hartman and D. Kreische, eds., *Model Driven Architecture—Foundations and Applications* (*ECMDA-FA'05, Proceedings*), Lecture Notes in Computer Science, vol. 3748, pp. 160–174. Springer-Verlag, New York, 2005.

24. T. Gehrke. An algebraic semantics for an abstract language with intra-object-concurrency. In D. Pritchard and J. Reeve, eds., *4th International Conference on Parallel Processing* (*Euro-Par'98, Proceedings*), Lecture Notes in Computer Science, vol. 1470, pp. 733–737. Springer-Verlag, New York, 1998.

25. P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri net based semantics definition for message sequence charts. In O. Færgemand and A. Sarma, eds., *6th SDL Forum: Using Objects* (*SDL'93, Proceedings*). North-Holland, Amsterdam, 1993.

26. D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: generating statechart models from scenario-based requirements. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, eds., *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science, vol. 3393, pp. 309–324. Springer-Verlag, New York, 2005.

27. D. Harel and S. Maoz. Assert and negate revisited: modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

28. D. Harel, S. Maoz, and I. Segall. Some results on the expressive power and complexity of LSCs. In A. Avron, N. Dershowitz, and A. Rabinovich, eds., *Pillars of Computer Science: Essays Dedicated to Boris* (*Boaz*) *Trakhtenbrot on the Occasion of His 85th Birthday*. Lecture Notes in Computer Science, vol. 4800, pp. 351–366. Springer-Verlag, New York, 2008.

29. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, New York, 2003.

30. ITU. *Message Sequence Chart* (*MSC*). Recommendation Z.120, ITU-TS. International Telecommunication Union, Geneva, 1996 (revised 2003).

31. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*, 4th ed., Addison-Wesley, Wokingham, UK, 1993.

32. J.-P. Katoen and L. Lambert. Pomsets for MSC. In H. König and P. Langendörfer, eds., *Formale Beschreibungstechniken für verteilte Systeme* (*8th GI/ITG-Fachgespräch, Proceedings*), pp. 197–207. Shaker, Aachen, 1998.

33. J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In T. Margaria and W. Yi, eds., *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'01, Proceedings*). Lecture Notes in Computer Science, vol. 2031, pp. 512–527. Springer-Verlag, New York, 2001.

34. A. Knapp. A formal semantics for UML interactions. In France and Rumpe [22, pp. 116–130].

35. A. Knapp and J. Wuttke. Model checking of UML 2.0 interactions. In T. Kühne, ed., *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*, Lecture Notes in Computer Science, vol. 4364, pp. 42–51. Springer-Verlag, New York, 2007.

36. R. Kobro Runde, Ø. Haugen, and K. Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 12(2):157–188, 2005.

37. R. Kobro Runde, Ø. Haugen, and K. Stølen. The pragmatics of STAIRS. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds., *5th International Symposium on Formal Methods for Components and Objects (FMCO'06, Proceedings)*. Lecture Notes in Computer Science, vol. 4111, pp. 88–114. Springer-Verlag, New York, 2006.

38. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In F. Rammig, ed., *International Workshop on Distributed and Parallel Embedded Systems (DIPES'98, Proceedings)*. IFIP Conference Proceedings, vol. 155, pp. 61–71. Kluwer Academic, Norwell, MA, 1999.

39. J. Küster Filipe. Decomposing interactions. In M. Johnson and V. Vene, eds., *11th International Conference on Algebraic Methodology and Software Technology (AMAST'06, Proceedings)*, Lecture Notes in Computer Science, vol. 4019, pp. 189–203. Springer-Verlag, New York, 2006.

40. J. Küster Filipe. Modelling concurrent interactions. *Theoretical Computer Science*, 351(2):203–220, 2006.

41. A. Letichevsky, J. Kapitonova, V. Kotlyarov, V. Volkov, A. Letichevsky, Jr., and T. Weigert. Semantics of message sequence charts. In A. Prinz, R. Reed, and J. Reed, eds., *12th International SDL Forum (SDL'05: Model Driven Systems Design, Proceedings)*. Lecture Notes in Computer Science, vol. 3530, pp. 117–132. Springer-Verlag, New York, 2005.

42. S. Leue and P. Ladkin. Implementing and verifying MSC specifications using Promela/XSpin. In J.-C. Gregoire, G. J. Holzmann, and D. Peled, eds., *2nd International Workshop on SPIN Verification System (SPIN'96, Proceedings)*. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 65–89. American Mathematical Society, Providence, RI, 1997.

43. M. S. Lund. Operational analysis of sequence diagram specifications. Ph.D. dissertation, Universitetet i Oslo, Oslo, Norway, 2007.

44. OMG. Object constraint language, version 2.0. http://www.omg.org/spec/OCL/2.0/, May 2006. Accessed Dec. 11, 2008.

45. OMG. Unified modeling language: superstructure, version 2.1.2. http://www.omg.org/docs/formal/07-11-02,pdf, Nov. 2007. Accessed Nov. 11, 2008.

46. J. Oldevik and Ø. Haugen. Semantics preservation of sequence diagram aspects. In I. Schieferdecker and A. Hartman, eds., *4th European Conference on Model Driven Architecture, Foundations and Applications (ECMDA-FA'08, Proceedings)*. Lecture Notes in Computer Science, vol. 5095, pp. 215–230. Springer-Verlag, New York, 2008.

47. G. Övergaard. A formal approach to collaborations in the unified modeling language. In France and Rumpe [22, pp. 99–115].

48. V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

49. H. Störrle. Assert, negate and refinement in UML-2 interactions. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, eds., *2nd International Workshop on Critical Systems Development with UML (CSDUML'03, Proceedings)*. Technical Report TUM-I0323, pp. 79–93. Institut für Informatik, Technische Universität München, Munich, Germany, 2003.

50. H. Störrle. *Trace Semantics of UML 2.0 Interactions*. Technical Report 0409. Institut für Informatik, Ludwig-Maximilians-Universität München, Munich, Germany, 2004.

51. H. Störrle. *UML 2.0 für Studenten*. Pearson, Munich, Germany, 2005.

52. Sysoft. Unistep: animation of UML sequence diagram. Technical Report, 2008. http://www.sysoft-fr.com/en/unistep/unistep_webDemoUmlSeq.asp. Accessed Nov. 28, 2008.

53. Sysoft. Unistep: animation of UML statechart. Technical Report, 2008. http://www.sysoft-fr.com/en/unistep/unistep_webDemoUmlState.asp. Accessed Dec. 4, 2008.

54. C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, Hoboken, NJ, 2005.

55. G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science, vol. 4, Semantic Modelling*, pp. 1–148. Oxford Science, Oxford, UK, 1995.