4 Semantics of UML Models for Dynamic Behavior

A Survey of Different Approaches

Mass Soldal Lund[1], Atle Refsdal[1], and Ketil Stølen[1,2]

[1] SINTEF ICT, Norway
{Mass.S.Lund,Atle.Refsdal,Ketil.Stolen}@sintef.no
[2] Department of Informatics, University of Oslo, Norway

Abstract. Models are used for a number of different purposes, from the requirements capture and design of a new system, to the testing of an existing system. Many different modeling languages are available, and the semantics given for the languages vary from informal natural language descriptions to various kinds of mathematical or logical definitions. When choosing a modeling language and accompanying semantics, a number of things need to be taken into consideration, such as who are the users of the models, what is the purpose of the models, what kind of application is being modeled, and what are the essential features that must be captured.

When modeling embedded systems, an essential aspect is the interaction between hardware and software. Hence, we need to capture the behavior of the hardware and software components. For capturing the dynamic behavior of components, modeling languages like UML sequence diagrams, state machines and similar notations are often used. This paper surveys different approaches to formally capturing the semantics of models expressed using languages of this kind.

4.1 Introduction

In the context of development of embedded systems, a model is a description of a computer system, possibly including its human users, controlled process or environment, in some modeling language. Modeling plays an increasingly important role and is used for a number of purposes throughout the lifetime of a system, from initial requirements capture and design to testing and maintenance of the running system. Some models are intended to be processed automatically, for example by code generators or model checkers, while other models are used as an aid in communication between for example system developers and client representatives.

A large number of languages for modeling computer systems are available. The semantics given for the languages range from natural language explanations of modeling language constructs and examples to highly formal mathematical or logical definitions. When choosing a language and accompanying semantics, a

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 77–103, 2010.

number of issues need to be taken into consideration. One question is: Who will use the models, and what level of training do they have? Clearly, the language need to have a notation that is understandable by the users of the models, at least at an intuitive level. As an example, mathematical and logical formulas may be well understood by computer scientists and some developers, but will be incomprehensible for most client representatives.

Another question is: What is the purpose of the models? If the models are to be used for giving formal proofs of system properties, then the language must be supported by a formal semantics defined in clear mathematical or logical terms. If the models will be used for code generation or automatic model checking then we need to ensure that the semantics can also be processed by a computer. On the other hand, if the models are intended for communicating with client representatives then a natural language explanation of the language features may be appropriate.

A third question is: What kind of system is being modeled, and what are the essential features or properties that need to be captured by the models? For example, capturing real-time requirements may be essential when modeling an emergency communication network, but of little importance when designing a chocolate automaton.

For embedded systems, the interaction between software and hardware components is an essential feature. This means that we need to model the behavior of hardware as well as software components, and in particular their mutual interaction. For capturing dynamic component behavior, modeling languages like UML [1] sequence diagrams and state machines are currently the most highly profiled.

This paper surveys approaches to giving formal semantics to models expressed in UML sequence diagrams, state machines or similar notations, such as MSC [2], LSC [3], the Statecharts language [4], SDL [5], etc. An overview is given of different types of semantics and their strong and weak points. The survey is not exhaustive, but covers the most common variants. The survey does not address semantics for hybrid models which is a field in its own [6, 7].

The rest of this paper is organized as follows: Sect. 4.2 characterizes the scope of the survey and defines more carefully notions like “model”, “semantics”, and “embedded system”. Furthermore, the semantic challenges related to embedded systems are discussed and summarized as a set of criteria against which the different approaches should be evaluated. In Sect. 4.3, the different types of semantics and their strong and weak points are discussed. Sect. 4.4 presents a survey of semantics approaches for UML sequence diagrams and similar notations. Section 4.5 is similar to Sect. 4.4, except that we now consider semantic approaches for UML state machines and similar notations. In Sect. 4.6 we evaluate the semantic approaches surveyed in the previous two sections with respect to the evaluation criteria formulated in Sect. 4.2. Summary and conclusions are given in Sect. 4.7.

4.2 Characterization of Scope, Main Notions, and Criteria for Evaluation

Embedded systems can be defined as "combinations of computer hardware and software, and perhaps mechanical or other parts, designed to perform dedicated functions"[1] or "programmable, electronic (often in combination with mechanical) systems that control and determine the functioning of devices (machines, appliances, instruments, constructions)."[2] MP3 players, routers, sensors, copying machines, and cars are examples of embedded systems. The fact that embedded systems, unlike general purpose computers, are dedicated to specific tasks, means that they can be optimized with respect to for example performance or reliability.[3]

A model is a description of a system in some modeling language, such as the UML. The semantics of a model explains what the model means. More exactly, the semantics of a model is a function mapping the syntactically well-formed models of the modeling language into syntactically well-formed expressions in a language that is well understood. What is a well-understood language depends on the intended users of the semantics. It often makes sense to define several equivalent semantics for the same modeling language; for example, an axiomatic semantics for logical deduction, a denotational semantics for mathematical reasoning, an operational semantics for building tools, and a natural language semantics to explain the language to its end-users. If the expressions of the modeling language is mapped into a mathematical or logical domain so that the semantic representation can be manipulated and analyzed using well-established mathematical and logical techniques, we say that the language has a formal semantics.

When modeling and developing embedded systems, several considerations need to be taken into account. One is that the close interplay between dedicated hardware and software components means that there is less room for corrections and refactoring during the development process than for conventional computer systems. A formal approach to model analysis and incremental development is therefore highly desirable when developing embedded systems. Hence, modeling languages should be supported by formal semantics, as well as definitions of refinement characterizing what it means for a more concrete or detailed model to "implement" or fulfill the requirements of a more abstract model. This reduces ambiguity and facilitates rigorous, and possibly automated, mathematical or logical proof of system properties.

An essential requisite for an incremental development process is the ability to leave some decisions open for later development steps. Consequently, we need

[1] From Netrino embedded systems glossary:
`http://www.netrino.com/Embedded-Systems/Glossary`

[2] From Embedded Systems Institute:
`http://www.esi.nl/frames.html?/institute/research.html`

[3] Other examples are characteristics such as size and power usage, but this is outside the scope of this paper.

a modeling language that has the ability to express underspecification or implementation freedom. By this we mean that a model may explicitly provide alternative ways of fulfilling a task, so that the choice is left open to those responsible for implementing or further refining the specification. Moreover, in an incremental development process one cannot describe all the relevant system behavior in a single step. Thus we want to be able to produce models that are incomplete in the sense that not all system behavior has been considered and categorized as either positive (acceptable, desirable) or negative.

Finally, there is the issue of the kinds of features or properties that can be captured by the modeling language. Properties can be categorized according to the basis on which they are falsified: Properties that can be falsified on the basis of a single trace are called *trace properties*, while properties that are falsified on the basis of a set of traces are called *trace-set properties* [8].[4] Examples of trace properties are safety and liveness [9, 10], while permissions often used in relation to policies and many information flow properties are examples of trace set properties. Most modeling languages are well-suited to capture trace properties, but only some allow us to specify trace-set properties as something distinguishable from underspecification. Distinguishing trace-set properties from underspecification is necessary since trace-set properties should be preserved under refinement while this is not the case for underspecification.

Performance and reliability requirements are usually of high importance for embedded systems. This is for example the case for routers and sensors. Indeed, in many cases the motivation for building a dedicated embedded system is to achieve high performance and reliability. Performance and reliability requirements are typically expressed in terms of time and/or probability. Therefore, modeling languages for embedded systems should ideally have the ability to capture real-time requirements (a special kind of trace properties) and probabilistic requirements (a special kind of trace-set properties). These requirements should be fully integrated in the semantics of the models in order to ensure that they are taken into account when analyzing the models.

Based on the above considerations, we have identified the following questions that we will use to evaluate the surveyed semantic approaches:

- What kind of semantics is given?
- Can underspecification be represented?
- Can trace-set properties be represented?
- Can incomplete models be represented?
- Is the approach supported by definitions of refinement?
- Can real-time requirements be captured by the semantics?
- Can probabilistic requirements be captured by the semantics?

In the following we survey and evaluate a number of semantic approaches with respect to these questions. But first we give an overview of main categories of semantics of relevance.

[4] In [8], the term *possibilistic properties* is used instead of trace set properties.

4.3 Main Categories of Semantics

At an overall level, semantics of modeling languages can be categorized based on whether they are formal or not, i.e. whether the expressions of the modeling language are mapped into a mathematical or logical domain, or explained in natural language. An advantage with natural language explanations is that they can be understood by anyone, without requiring specialized training. However, natural language explanations tend to be ambiguous and often contain inconsistencies. For example, this is the case with the UML semantics provided by the Object Management Group (OMG) [11, 12, 13]. Formalizing the semantics of a language will help uncover ambiguities and inconsistencies. Moreover, formal semantics allows models to be analyzed with mathematical and logical tools and techniques, thus allowing system properties to be explored in a rigorous manner before the implemented system even exists. Being able to perform this kind of analysis as early as possible is particularly important when developing embedded systems, as the cost of redesigning dedicated components at a late stage typically will be high. Hence, a formal semantics is needed for the development process. There are, however, different styles of formalizing semantics, each with their strong and weak points. For the graphical modeling languages we are concerned with in this paper, denotational and operational semantics are the most relevant styles. We now look at these two styles of semantics and their strong and weak points.

David A. Schmidt [14] provides the following explanation for a denotational semantics:

> The *denotational semantics* method maps a program directly to its meaning, called its *denotation*. The denotation is usually a mathematical value, such as a number or function. No interpreters are used; a *valuation function* maps a program directly to its meaning.

This corresponds well with the explanation given by Andreas Prinz [12, p. 149]

> The basic idea is to give a denotation to every element of the language. This means to map the syntactical expressions of the language to a well-known domain.

Denotational semantics typically allows a fairly abstract system description. As they also build on known domains, they are well suited for mathematical reasoning and formal proof of properties. On the negative side, a denotational semantics provides little guidance for tool developers and will typically be too complex for users. Expressing states and operations is usually difficult with a denotational semantics.

For operational semantics, [14] suggests the following definition:

> The *operational semantics* method uses an interpreter to define a language. The meaning of a program in the language is the evaluation history that the interpreter produces when it interprets the program. The evaluation history is a sequence of internal configurations [...]

As a methodology for language development he suggests that "a denotational semantics is defined to give the meaning of the language" and that "the denotational definition is implemented using an operational definition" [14, p. 4].

Table 4.1. Different styles of semantics

Type of semantics	Advantages	Disadvantages
Informal	– Easy to communicate – Does not require specialized training	– Tends to be ambiguous – Often contains inconsistencies – Cannot be formally analyzed
Denotational	– Allows a fairly abstract system description – Builds on known domains – Well suited for mathematical reasoning and formal analysis of properties	– Provides little guidance for tool developers – Too complex for users – Expressing states and operations is usually difficult
Operational	– Provides good formalization of implementation – Well suited for building tools – Expressing states and operations is usually easy	– Tends to be very detailed – It is often difficult to derive formal proofs – Relies on the underlying semantics of the abstract computer

Hoare and He [15, p. 258] describe more explicitly the notion of an operational semantics:

> An *operational* semantics of a programming language is one that defines not the observable overall effect of a program but rather suggests a complete set of possible individual steps which may be taken in its execution. The observable effect can then be obtained by embedding the steps into an iterative loop [...]

Taken together, these two descriptions suggest that formalizing an operational semantics of a language is to define an interpreter for the language. The formal definition of the interpreter describes every step that can be made in the execution of the language in such a way that the executions are in conformance with the meaning of the language as defined by a denotational semantics.

Major advantages of operational semantics is that such semantics provides good formalization of implementation and is well suited for building tools. It is also typically well suited for state-based languages. On the other hand, operational

semantics tends to be very detailed, and it is often difficult to derive formal proof from operational semantics. Besides, an operational semantics relies on the underlying semantics of the abstract computer on which the interpreter is assumed to run [12]. Table 4.1 summarizes the strong and weak points of the different styles of semantics discussed above.

In the next three section we present and discuss a number of approaches to giving semantics to models. We concentrate on two categories of models, models expressed in a sequence diagram style and models expressed in a state machine style, and two main categories of semantics, denotational and operational. Making a complete and exhaustive presentation of every existing approach is an impossible task, and it has therefore been necessary to make a selection. With respect to sequence diagrams, we focus on UML sequence diagrams and Message Sequence Charts (MSC), and with respect to state machines, we focus on statecharts, UML state machines and SDL. Furthermore, we have aimed at making a representative selection of the approaches that exist.

In Sect. 4.4 we present and discuss approaches to giving semantics to sequence diagrams and similar notations, and in Sect. 4.5 we do the same with respect to state machines and similar notations. In Sect. 4.6 we evaluate and compare the approaches using the evaluation criteria identified in Sect. 4.2.

4.4 Sequence Diagrams and Similar Notations

In this section we present different approaches to defining formal semantics to models expressed in UML sequence diagrams and similar notations. This presentation cannot, however, be seen independently of the history of sequence diagrams. The various approaches of defining semantics have emerged at different points in this history, and are clearly influenced by the state of the language(s) at the time of their emergence.

Sequence diagrams is a graphical specification language defined in the Unified Modeling Language (UML) 2.x[5] standard [1]. Sequence diagrams as defined in the UML 2.x standard are the last of a sequence of languages that have evolved over the last 15 to 20 years. Both UML sequence diagrams and their predecessor Message Sequence Charts (MSC) [2] are specification languages that have proved themselves to be of great practical value in system development.

An early version called Time Sequence Diagrams was standardized in the 1980s (see [17, 18]). Better known are MSCs that were first standardized by ITU in 1993 (see e.g. [19]). This standard is usually referred to as MSC-92, and describes what is now called basic MSCs. This means that MSC-92 did not have high-level constructs such as choice, but merely consisted of lifelines

[5] The UML standard exists in versions 1.3, 1.4, 1.4.2, 1.5, 2.0 and 2.1.1. The for us relevant changes occurred in the transition from version 1.5 to version 2.0. Hence, in this paper we will operate with UML 1.x and UML 2.x with versions 1.4 [16] and 2.1.1 [1] as representatives.

and messages. MSC-92 had a lifeline-centric textual syntax[6], and was given a semantics formalized in process algebra.

In 1996, a new MSC standard was defined, called MSC-96 [20]. In this standard, high-level constructs and high-level MSCs were introduced, a kind of diagrams that show how control flows between basic MSCs. Further an event-centric textual syntax[7] and a new semantics were defined [21]. This semantics is also a kind of process algebra, but holds substantial differences from the MSC-92 semantics. Finally, the MSC-96 standard was revised in 1999 and became MSC-2000 [2], but kept the MSC-96 semantics. A further discussion on the MSC semantics is found below.

The first versions of the Unified Modeling Language (UML 1.x) [16] included a version of sequence diagrams similar to MSC-92, i.e., consisting of lifelines and messages but no high-level constructs. An important difference, however, was that the sequence diagrams of UML 1.x did not have the frame around the diagram, which in MSC-92 allowed messages to and from the environment of the specified system.

Sequence diagrams in UML 2.x may be seen as a successor of MSC-2000, since many of the MSC language constructs have been incorporated in the UML 2.x variant of sequence diagrams. UML 2.x sequence diagrams are, however, neither a subset nor a superset of MSC-2000; there are both similarities and differences between the languages [22]. Most notably MSCs do not have any notion of negative behavior.

The UML standard defines the semantics of sequence diagrams informally. Most notably, this is a trace-based semantics:

> **Basic trace model:** The semantics of an Interaction[8] is given by a pair $[P, I]$ where P is the set of valid traces and I is the set of invalid traces. $P \cup I$ need not be the whole universe of traces.
> A trace is a sequence of event occurrences denoted $\langle e1, e2, ..., en \rangle$. [1, pp. 479–480]

The UML standard [1] defines four timing concepts: Duration observation, duration constraint, time observation and time constraint. The timing concepts of the UML Testing Profile [23] are a combination of the timing concepts from the UML standard and the timers from MSC. In the UML Profile for Schedulability, Performance, and Time [24] timing is specified by timestamps on events. This UML

[6] Lifelines represent the time-lines of communicating parts or components in a sequence diagram. In "MSC-terminology", lifelines are called *instances* or *instance lines*. A lifeline-centric syntax means that each lifeline is characterized by itself and a diagram as a collection of lifelines.

[7] In an event-centric syntax events, as opposed to lifelines, are the basic building blocks of a diagram. The event-centric syntax of MSCs is more general than the lifeline centric-syntax in that all diagrams expressed in the lifeline-centric syntax can be expressed in the event-centric syntax, but not the other way around.

[8] In the UML standard, *Interaction* is used as the common name for diagrams specifying interaction by sending and receiving of messages. Sequence diagrams are then one kind of Interaction [our note].

profile also has the notion of a timer, and a notion of a system clock that can produce interrupt events. The MSC standard defines three timing concepts: Timer, relative time constraints or relative time delays, and absolute measure or timing.

In the following we present briefly different denotational and operational semantics of sequence diagrams and similar notations. We start by presenting denotational semantics, then denotational semantics with time, and then denotational semantics with probabilities. Then we present operational semantics following the same structure.

4.4.1 Denotational Semantics

In [25] Katoen and Lambert define a denotational semantics for MSCs over sets of partially ordered multisets. They define two translations, one for basic MSCs and one for high-level MSCs. The former is defined over the instance oriented textual syntax of MSCs and therefore have one rule for strict sequencing of events on a single lifeline and one rule for co-region and parallel composition of lifelines. These two levels seem to be unnecessary. If the rules for lifeline composition is combined with the rules for sequential and parallel composition, the semantics can be defined directly over the HMSC syntax and we then get a more general approach. A similar denotational semantics for both basic MSCs and high-level MSCs is given in [26].

In [27], Krüger defines a variant of Message Sequence Charts that is supported by formal definitions of the semantics, as well as refinement relations. The semantics is defined in terms of streams, which consist of a sequence of system channel valuations and a sequence of state valuations. A system is represented semantically by a set of streams, and the existence of more than one stream indicates nondeterminism. The MSC variant proposed in [27] has some features that go beyond standard MSC. For example, a trigger composition operator allows us to specify that that the occurrence of an interaction sequence always causes the occurrence of another, thus providing a way of specifying liveness properties. In addition, [27] defines four different interpretations of MSCs: an existential interpretation, an universal interpretation, an exact interpretation and a negative interpretation. Four different refinement relations are defined: binding of references, which allows references to empty MSCs, property refinement, which reduces the set of possible behaviors of the system, message refinement which allows a single message to be replaced by a whole interaction sequence, and structural refinement, which allows a single lifeline to be replaced by a set of lifelines thus allowing decomposition.

The STAIRS semantics [28, 29] is a trace based formalization of sequence diagrams based on an extension of the semantic model of the UML standard, and hence distinguishes between positive, negative and inconclusive traces. But instead of a single pair (p, n) of positive and negative traces the semantic model of STAIRS is a set of pairs $\{(p_1, n_1), \ldots, (p_m, n_m)\}$. Such a pair of sets of traces (p_i, n_i) is referred to as an *interaction obligation*. The word "obligation" is used in order to emphasize that an implementation of a specification is required to fulfill every pair captured by the specification. This semantic model makes it

possible to define trace-set properties. Refinement is defined as refinement of each interaction obligation, and refinement of interaction obligations is defined as reducing the set of positive traces by making them negative and reducing the set of inconclusive traces by making them positive or negative.

Störrle [30, 31, 32] defines a denotational trace based semantics for UML 2.x sequence diagrams that is quite similar to the STAIRS semantics. Among the notable differences are that Störrle does not treat choices as underspecification. Further, Sörrle gives a different treatment of negative behavior where sequence diagrams are not allowed to be inconsistent and the negative operator can indirectly specify positive traces. Refinement is defined, but are more restricted as there is no treatment of underspecification in the semantics.

Cengarle and Knapp [33] defines denotational semantics for UML 2.x sequence diagrams. Their denotational semantics is trace based and similar to STAIRS and the semantics of Störrle with respect to the positive parts of sequence diagrams. In difference from STAIRS and Störrle, they make a prefix closure of negative traces, but does not allow inconsistent sequence diagrams. Their refinement relation differs from STAIRS in that the set of inconclusive traces may be increased, something which is a problem with respect to the monotonicity of the composition operators.

In [34], Küster-Filipe gives an LSC inspired denotational semantics of UML 2.x sequence diagrams based on partially ordered sets. The partially ordered sets of sequence diagrams is used to build event structures, and modal logic constraints over these event structures are used to express negative behavior, as well as must and may behavior.

4.4.2 Denotational Semantics with Time

In [35, 36], the semantics of basic MSCs given in [26] is presented in a timed version. A timing function assigns time stamps to the events of a MSC, and the MSC can be annotated with timing constraints in the form of minimum and maximum time intervals between events. In addition, algorithms are given for checking the realizability of MSCs and whether or not there exists a timing function that is consistent with the timing constraints of an MSC.

In [37], Zheng et al. give a semantics with time for MSC-2000. The semantics is based on labeled partially ordered sets and defines semantics for both basic MSCs, high-level operators of MCSs and high-level MSCs. Time is represented by a function mapping each event in a diagram to a set of time values, giving the absolute time interval in which the event should occur. Relative timing constraints are expressed by a function mapping pairs of events to intervals of time values. In [38], horizontal and vertical refinement of their timed MSCs are defined.

Timed STAIRS is an extension to STAIRS defined in [39, 40]. In timed STAIRS there is a distinction between syntactic and semantic events: in the syntax an event is a triple of a kind (transmit, receive, or consumption), message and time-stamp tag, while in the semantic events the time-stamp tags are mapped to timestamps represented by real numbers. A requirement is placed on traces to ensure that time increases monotonically in every trace. Time constraints are defined as Boolean

expressions over the time-stamp tags of the events of a diagram. If the mapping of timestamps to time-stamp tags in a trace satisfies the constraint, the trace is interpreted as positive, otherwise it is interpreted as negative.

4.4.3 Denotational Semantics with Probabilities

Performance Message Sequence Chart (PMSC) [41, 42] extends MSC with syntactic constructs for expressing performance requirements. The aim is to integrate performance characteristics, such as response time and throughput, in functional specifications. Of particular interest is the new operator altprob for probabilistic choice that is introduced in [42]. This operator allows exact probabilities to be assigned to the alternatives represented by its operands. This means that underspecification with respect to probability cannot be captured by this operator. Apart from mentioning instance decomposition, refinement is not discussed, and no definition is given of what it means for a system to comply with a PMSC specification. The semantics of PMSC is explained at a purely intuitive level.

Probabilistic STAIRS (pSTAIRS) [43, 44, 45] generalizes timed STAIRS in order to allow probabilistic requirements, including soft real-time requirements, to be captured. Sets of acceptable probabilities, rather than a single probability, can be assigned to alternatives. Hence, it is possible to express requirements such as "the probability of receiving a reply within 5 seconds after sending a request should be at least 0.9" or, for a machine simulating a coin toss, "the probability of getting a heads outcome should be between 0.4 and 0.6". Semantically, probabilistic STAIRS extends the semantic model of timed STAIRS by assigning probability sets to each interaction obligation, thus yielding so-called *p-obligations*. Refinement is defined in a similar way as for timed STAIRS, with the additional constraint that the probability set of the refined p-obligation must be a subset of the original p-obligation, thus narrowing the range of acceptable probabilities.

4.4.4 Operational Semantics

In 1995 a formal algebraic semantics for MSC-92 was standardized by ITU [46, 47]. MSC-92 has a lifeline-centric syntax and its semantics is based on characterizing each lifeline as a sequence (total order) of events. These sequences are composed in parallel and a set of algebraic rules transforms the parallel composition into a structure of (strict) sequential composition and choice. The causality of messages is obtained by a special function that removes from the structure all paths that violate the invariant. In a way this semantics is not a proper operational semantics since a diagram first has to be transformed into the event structure before runs can be obtained. This transformation replaces parallel composition with choice and hence creates an explosion in the size of the representation of the diagram. In addition, the lifeline-centric syntax is not suitable for defining nested high-level constructs. In [48], similar semantics for UML 1.x sequence diagrams is given.

MSC-96 got a standardized process algebra semantics in 1998 [21, 49, 50]. This semantics is event-centric and has semantic operators for all the syntactic operators

in MSC-96. Further, these operators are "generalized" to preserve the causality of messages by coding information about messages into the operators in the translation from syntactical diagrams to semantic expressions. Runs are characterized by inference rules over the semantic operators. Compared to UML semantics, the most notable thing about this semantics is that it has no notion of negative behavior, and therefore also makes no distinction between negative behavior and inconclusive behavior (behavior that is neither positive nor negative). This is no surprise since MSC does not have the negative operator of UML 2.x. The only available meta-level is a flat transition graph, and this does not give sufficient strength to extend the semantics with negative behavior. Nor is it possible to define trace-set properties over this transition graph. The semantics has no explicit communication medium; the communication model is "hard-coded" in the semantics by the "generalized operators" and does not allow for variation. Even though MSC has timing concepts, these are not given proper treatment in the semantics.

Another process algebra semantics for MSC is presented in [51]. This semantics may in some respects be seen as more general than both the MSC-92 and the MSC-96 semantics. A simple "core semantics" for MSCs is defined and this semantics is then inserted into an environment definition. Varying the definition of the environment allows for semantic variability and extendibility, e.g., with respect to the communication model. However, the semantics is heavily based on synchronization of lifelines on the entry of referenced diagrams and combined fragments and diverges in this respect from the intended semantics of MSCs and UML sequence diagrams. Further, the same strategy as for the MSC-92 semantics is applied; interleaving is defined by means of choice, and the message invariants obtained by removing deadlocks. This results in an unnecessary amount of computation, especially in the cases where we do not want to produce all traces but rather a selection of the traces that a diagram defines.

Realizability of MSCs is the focus of both [36, 52] and [53]. They define synthesis of MSC to concurrent automata and parallel composition of labeled transition systems (LTS), respectively. (Each lifeline is represented as an automaton or LTS; the lifelines are then composed in parallel.) Further they define high-level MSCs as graphs where the nodes are basic MSCs. In addition, [53] defines both syntax and semantics for negative behavior. In both approaches the translation of high-level MSCs to concurrent automata/LTSs removes the semi-global nature of choices in a specification, and the high-level MSC graphs are non-hierarchical, disallowing nesting of high-level operators. In [53] communication is synchronous.

Various attempts at defining Petri-net semantics for MSCs have been made [54, 55, 56, 57]. In [54, 56] only basic MSCs are considered. In [57], high-level MSCs are defined as graphs where each node is a basic MSC. As with the above mentioned semantics, it is then possible to express choices and loops, but the approach does not allow for nesting of high-level operators. In [55], a Petri-net translation of the choice operator is sketched, but no loop defined. In [58] a Petri-net semantics for UML 1.x sequence diagrams is presented, but as with the Petri-net semantics of basic MSCs it has major limitations.

Jonsson and Padilla [59] present a semantics for MSC which is based on syntactic expansion and projection of diagram fragments during execution. Each lifeline is represented by a thread of labels where the labels refer to events or diagram fragments. The threads are executed in parallel and when a label referring to a fragment is reached the fragment is projected and expanded into the threads. Expansions may happen at arbitrary points since there are no rules in the semantics itself for when to expand. This creates a need for execution strategies, and the approach may be seen as having an informal meta-level where ad hoc strategies are described. However, if completeness is to be ensured, or if the semantics is to be extended with negative behavior or trace-set properties, this meta-level must be formalized. The semantics requires explicit naming of all diagram fragments and this yields an unnecessary complicated syntax. It does not have an explicit communication medium; the communication model is "hard-coded" into the semantics and does not allow for variation.

In [60, 61] an operational semantics for UML 2.x sequence diagram is given. The semantics is defined as the combination of two transition systems, which are referred to as an *execution system* and a *projection system*. The projection system is used for finding enabled events at each stage of the execution and is defined recursively. These two systems work together in such a way that for each step in the execution, the execution system updates the projection system by passing on the current state of the communication medium, and the projection system updates the execution system by selecting the event to execute and returning the state of the diagram after the execution of the event. The execution system can be configured with different communication models, and the semantics also provides a formal meta-level for specifying execution strategies and for handling of negative behavior and trace-set properties. The semantics is proved to be sound and complete with respect to the denotational semantics of STAIRS (see above).

In [62, 63] an operational semantics for UML 2.x sequence diagrams that is equivalent to the denotational semantics defined in [33] (see above) is given. This operational semantics has some similarities to the operational semantics of [60, 61]; for every execution step an event is produced and at the same time the syntactical representation of the diagram is reduced by the removal of the event produced. Contrary to [60, 61], their semantics treats sequence diagrams as complete specifications (with no inconclusive behavior). The rules are defined so that a given diagram produces a set of positive and negative traces that together exhaust the trace universe. The negative operator is replaced by a "not" operator. This operator is defined so that the sets of positive and negative traces are swapped, with the result that specifying some behavior as negative means also specifying the complement of this behavior as positive. A variant of the (positive part) of the operational semantics where each lifeline is executed separately, and an extension with channels, are given in [63].

In [64], Cavarra and Küster-Filipe present an operational semantics for UML 2.x sequence diagrams inspired by Live Sequence Charts (LSC) (see below). The semantics is formalized in pseudo-code that works on diagrams represented as

locations in the diagram, but no translation from diagrams to this representation is provided. The arguments of choices have guards and there is nothing to prevent the guards of more arguments in a choice to evaluate to **true**. In this case the uppermost operand will be chosen, which means that the choices essentially are treated as nested **if-then-else** statements and may not be used for underspecification. Each lifeline is executed separately which means that synchronization at the entry of choices is necessary to ensure that all lifelines choose the same operand. They also make the same assumption about negative behavior as in LSCs, that if a negative fragment is executed, then execution aborts.

Grosu and Smolka [65] provide a semantics for UML 2.x sequence diagrams based on translating the diagrams to Büchi automata. The approach is based on composing simple sequence diagrams (no high-level operators) in high-level sequence diagrams (interaction overview diagrams), where a simple diagram may be a positive or a negative fragment of the high-level diagram it belongs to. Positive behavior is interpreted as liveness properties and negative behavior as safety properties. Hence, for a high-level diagram two Büchi automata are derived; a liveness automaton characterizing the positive behavior of the diagram and a safety automaton characterizing the negative behavior. The diagrams are composed by strict sequencing rather than weak sequencing, and hence has implicit synchronization of lifelines when entering or leaving a simple diagram. Refinement is defined as language inclusion.

Live Sequence Charts (LSC) [3, 66, 67] is a variant of MSC where diagrams may be tagged as universal or existential, and parts of diagrams as hot or cold. In addition, a diagram may have a triggering pre-chart. The semantics of LSC characterizes the execution of diagrams. It also evaluates the conditions imposed on diagrams by designating them as universal or existential, or by marking parts of diagrams as hot or cold. The semantics complies with neither the MSC nor the UML standard. Most importantly it requires synchronization between lifelines at every entry point of diagram fragments, e.g. when resolving a choice.

Harel and Maoz [68] use LSC semantics to define negative behavior of UML 2.x sequence diagrams. The operators are defined using already existing constructs of LSCs, and hence no changes or additions to the LSC semantics are needed in their approach.

In Triggered Message Sequence Charts (TMSC) [69, 70], an initial part of a diagram can be designated as a trigger diagram, with the interpretation that if the behavior described by the trigger diagram takes place, then the behavior described by the rest of the diagram must subsequently take place. Unlike the pre-charts of LSC, however, the trigger condition applies locally to each lifeline. This means that, for any given lifeline, if the events on that lifeline described by the trigger diagram take place, then the following events on that lifeline must subsequently take place. As the fulfillment of the trigger condition is determined locally on each lifeline, there is no need for synchronization between the lifelines. A refinement relation is defined, with the intuitive interpretation that a specification S_1 is refined by a specification S_2 if S_2 is more deterministic than S_1. TMSC contains two operators for choice. A delayed choice must be preserved in

a refinement step. An internal choice can be resolved at any point (including at design time). In addition, an internal choice may be refined by a delayed choice.

4.4.5 Operational Semantics with Time

In [51], Letichevsky et al. claim they also have an extension to the semantics where timing concepts such as time intervals and timing of events are defined.

The operational semantics of [60, 61] has in [60] an extension with data, variables and time. Each lifeline has a set of local variables and a data state that assigns values to these variables. In order to model time a special variable ***now*** is introduced. Because the approach only has local variables, this variable is placed in the data state of every lifeline in a diagram. It can, however, be considered a global variable in the sense that all the local ***now*** variables are updated simultaneously and with equal increments, i.e. that the time of all lifelines are synchronized. Except for increments by a special tick rule, the ***now*** variables are read only, something that is ensured by syntactic constraints.

In [67], a time extension to LSCs is presented where a clock variable *Time* is added to the formalism. Time is then treated as data and time constraints can be expressed by means of ordinary variables.

Kosiuczenko and Wirsing [71] make a formalization of MSC-96 in a timed version of the term rewriting language Maude. Every lifeline in a diagram is translated into an object specified in Maude, and the behaviors of these objects are specified by the means of states and transition rules. This way of reducing diagrams to sets of communicating objects has the effect that all choices are made locally in the objects and the choice operator looses its semi-global nature. Hence, this formalization does not capture the intended understanding of the choice operator. With respect to time, their semantics only deals with timers, and their formalization makes restrictions on the MSC semantics.

4.4.6 Operational Semantics with Probabilities

We are not aware of any operational semantics with probabilities for sequence diagrams or similar notations.

4.5 State Machines and Similar Notations

In this section we present some of the approaches that have been taken for assigning formal semantics to models expressed in UML state machines and similar languages. UML state machines represent one of many variations that have emerged since Harel introduced the Statechart language in 1987 [4]. Over the years very much work has been dedicated to providing a satisfactory formal semantics. An extensive overview is beyond the scope of this article; our aim is to illustrate the variety of approaches that have been taken. An alternative overview from a different angle can be found in [72].

4.5.1 Denotational Semantics

Broy et al. [73, 74, 75] build a mathematical system model for UML in layers. Each layer builds an algebra consisting of a universe of elements with accompanying functions and laws for the functions. The third part, presented in [75], includes the "state machine part", which is given in terms of state transition systems. A state transition system consists of a state space (a set of states) and a state transition function. The theory of state transition systems is based on the theory of streams of FOCUS [76] for the I/O behavior, and thus inherits refinement from there. State transition systems can describe not only the behavior of a single object, but also a collaborating group of objects.

A set theoretic approach to defining a semantics is taken in [77]; object states, events, guards, and run-to-completion processing is described in set theoretic terms. The aim is to provide a compositional semantics that allows models to be subject to hierarchical and modular approaches to verification and testing.

4.5.2 Denotational Semantics with Time

In [75], the state transition systems are generalized into timed transition systems to account for time. The approach assumes a discrete global time. In each step/transition the system is provided with a finite set of input events and produces a finite set of output events; this takes a fixed amount of time corresponding to a clock tick.

Rossi et al. [78] provide a formalization of (fundamental aspects of) UML state machines in terms of a temporal logic over discrete time called LNint-e.[9] Time is represented by a discrete, linear and infinite set with a total ordering. LNint-e allows inclusion of interval expressions, and time can be treated both absolutely and relatively. The temporal primitives from which expressions can be built are instants, intervals and dates. A state machine diagram is represented by set of predicates, and the formalization can be generated automatically. States are formalized by means of expressions that can be affirmed over intervals ("hereditary interval expressions").

Hinkel, Holz and Stølen [79, 80] give a semantics for SDL specifications (whose behavioral descriptions are similar to UML state machines) based on streams and stream processing functions within the framework of FOCUS [76]. This allows properties of SDL specifications to be proved using techniques of classical higher order logic and of domain theory. Time is represented by a global clock which increases time and is accessible to all processes. Time is an orthogonal concept to system behavior, and time proceeds independently from the behavior. Timers set by processes will expire after a finite duration of time and are put in the

[9] Note that we have chosen to include [78] among the denotational semantics because the translation from a state machine diagram to a set of logical formulae can be viewed as a translation into a well-known domain. As there are very few approaches that give an axiomatic semantics for UML sequence diagrams and state machines, we have chosen not to have a separate category for axiomatic semantics.

input queue of the process. The refinement relations provided by FOCUS can be used also for the approach of [79, 80].

4.5.3 Denotational Semantics with Probabilities

We are not aware of any approaches that assigns a formal denotational semantics to state machines that also include probabilities.

4.5.4 Operational Semantics

In [4], Harel provides a brief discussion of how a formal semantics for statecharts could be provided, without giving definitions. The semantics is built around a function that provides the set of next possible configurations from a current configuration together with a set of conditions and a set of external simultaneous events. The set of possible next configurations represent nondeterminism. An updated and more thorough presentation of the semantics is provided in [81], which explains the executable semantics of the STATEMATE system [82].

One approach to assigning formal semantics to UML state machines is to use abstract state machines [83]. Following the description of [84], abstract state machines are transition systems whose states are multi-sorted first-order structures, i.e. sets with relations and functions. Relations can be considered as characteristic Boolean-valued functions. The transition relation is specified by rules that describe the modification of the functions from one state to the next. These update rules are of the form "if *Condition* then *Updates*", where *Updates* is a set of function updates (assigning new function values for arguments) which are simultaneously executed when *Condition* is true.

An example of an approach that uses abstract state machines is [85], which employs multi-agent abstract state machines to model the dynamic semantics of UML state machines. Their model is intended to define rigorously the UML event handling scheme so that semantic variation points become explicit, while reflecting the original structure of UML state machines. Furthermore, object interaction is formalized by combining control and data flow. This work is further extended by the authors in [86] to cover concurrent states, while [84] surveys their previous work in order to further discuss semantic variation points and unclarities of UML state machines from a formal point of view.

In [87], Jürjens extends the semantics given in [85, 86] by modeling actions, internal activities, and their operations and parameters explicitly, as well as providing message passing between different diagrams. This constitutes a further step toward formal modeling of complete UML specifications and the goal of executable UML specifications. A thorough presentation of Jürjens' work on formalization of UML is given in [88], which provides a formal semantics for UML state machines (as well as other UML languages such as sequence diagrams and static structure diagrams) in terms of so-called UML Machines and UML Machine Systems. UML Machines are inspired by abstract state machines; they are transition systems whose states are algebraic structures. In addition, UML Machines have built-in communication mechanisms similar to the corresponding

mechanisms in UML. UML Machines interact by exchanging messages which are dispatched from (or received in) multi-set buffers called output queues (or input queues). Based on UML Machines, [88] defines refinement relations, as well as security properties such as integrity and authenticity, and provides proofs of preservation of security properties under refinement.

van der Beeck [89] starts with a precise textual syntax definition for UML state machines. The terms of this textual syntax is designed to closely resemble the intuitive notion of state machines. From the textual syntax a structured operational semantics is developed in two phases. First an auxiliary semantic which only deals with processing single input events is defined. Then this auxiliary semantics is used to define a semantics that also handles processing of sequences of input events. Unlike many other approaches, [89] supports the history mechanism of UML state machines, as well as entry and exit actions.

4.5.5 Operational Semantics with Time

In [81] Harel and Naamad provide two models of time: one synchronous and one asynchronous. For the synchronous model it is assumed that the system executes a single step each time unit as a reaction to the external changes that have occurred in the single time unit since the completion of the previous step. For the asynchronous model it is assumed that the system reacts whenever an external change occurs. Several external changes may occur simultaneously, and several steps may take place within a single point in time.

In [90], timed UML state machines are compiled into timed UPPAAL automata [91], which are timed automata as originally defined by Alur and Dill [92], extended with primitives for synchronization. The passage of time is represented by increasing the value of a finite number of real-valued clocks by the same amount. [90] extends the UML notation ($\mathsf{after}(t)$) by allowing clocks to be explicitly declared in class diagrams. These clocks can be tested in transition guards and reset as the effect of a transition. Furthermore, clock invariants may be associated with states to model timeouts. Even though a formal semantics as such is not provided in [90], the translation of timed UML state machines has been implemented in a prototype tool called HUGO/RT. The resulting timed automata can then be analyzed by the UPPAAL model checker.

Building on ideas from timed process calculi, [93] suggests an approach to formalizing the Statechart language [4] semantics as flattened transition systems. Transition relations are defined via structured operational rules. The work is motivated by the desire to achieve a semantics that is compositional (in the sense that the semantics of a statechart can be determined from the semantics of its components), while obeying causality and synchrony. In this context, causality means the following: A statechart may respond to an event by engaging in an enabled transition, thus performing a *micro step*. This transition may generate new events which in turn may trigger additional transitions. Synchrony means that one execution step (a *macro step*) is complete as soon as this chain reaction comes to a halt. The semantics proposed in [93] represents macro steps as sequences of micro steps which begin and end with explicit global clock ticks. The

flat labeled transition systems thus have two kinds of transitions: those representing the execution of a statechart transition, and those representing global clock ticks. Clock transitions are only allowed if no additional action transitions can be executed.

4.5.6 Operational Semantics with Probabilities

Jansen et al. [94, 95] define StoCharts as an extension of UML state machines to deal with quality of service (QoS) aspects. Probability is handled by allowing state transitions to select probabilistically out of different effects. In addition, the "after" operator is given a stochastic interpretation allowing the time delay to be sampled from an arbitrary probability distribution. A formal semantics is provided in the form of a mapping to Stochastic Input/Output Automata (IOSA), which is an automata model based on timed, stochastic and probabilistic (I/O-)automata extending the UML state machine semantics of [96].

Motivated by the need for quantitative dependability and performance analysis of UML behavioral models of embedded systems, [97] presents patterns for translating UML state machines with timing and stochastic information and classification of model elements (such as fault states) into Stochastic Rewards Nets (SRN). SRNs are Petri-nets that are generalized to handle rewards (various measures) and by assigning guards and distributions of the firing time to transitions. The SRN resulting from the translation gives a precise mathematical model that can be analyzed by sophisticated tools. Standard UML mechanisms are employed to achieve the required expressiveness for the UML state machines; timing and stochastic information in captured by tagged values, while classification of model elements is achieved by stereotyped states and events.

4.6 Evaluation and Comparison

The evaluation of the semantic approaches surveyed in Sects. 4.4 and 4.5 is presented in Table 4.2 and Table 4.3, respectively. In these tables we indicate with check marks whether the properties, given as evaluation criteria in Sect. 4.2, are fulfilled. It should be noted that we have been somewhat liberal in the evaluation, and the evaluation is to some degree based on the claims of the authors of the evaluated papers. With respect to refinement, we have not assessed whether or not the provided definitions of refinement correspond to our view of refinement, but checked the refinement box if any refinement relation or similar notion is defined. In the following we give further comments on the two tables.

In Table 4.2, we see that most of the approaches are evaluated to support underspecification. The general rule is that an approach providing an explicit mechanism for specifying nondeterministic choice supports underspecification, unless such choices are interpreted as *must* behavior, as in [30, 32]. The approaches evaluated as supporting trace set properties are the approaches that explicitly distinguish between underspecification and inherent nondeterminism, as for example [28], the approaches distinguishing between universal and existential behavior, as for example [3, 66, 67, 68], and the approaches distinguishing

Table 4.2. Evaluation of semantics for sequence diagrams and similar notations

	Denotational semantics?	Operational semantics?	Underspecification?	Trace set properties?	Incomplete models?	Refinement?	Real-time?	Probabilities?
Katoen, Lambert [25]	√		√					
Krüger [27]	√		√	√	√	√		
Haugen, Husa, Runde, Seehusen, Solhaug, Stølen (STAIRS) [28, 29]	√		√	√	√	√		
Störrle [30, 31, 32]	√			√	√	√	√	
Cengarle, Knapp [33]	√		√			√		
Küster-Filipe [34]	√			√	√			
Alur, Etassami, Holzmann, Peled, Yannakakis [26, 35, 36]	√		√		√		√	
Zheng, Khendek, Hélouët, Parraux [37, 38]	√		√			√	√	
Haugen, Husa, Runde, Stølen (Timed STAIRS) [39, 40]	√		√	√	√	√	√	
Faltin, Lambert, Mitchele-Thiel, Slomka (PMSC) [42, 41]	√		√	√			√	√
Refsdal, Husa, Runde, Stølen (pSTAIRS) [43, 44, 45]	√		√	√	√	√	√	√
Mauw, Reniers (MSC-92) [46, 47]		√	√					
Mauw, Reniers (MSC-96) [21, 49, 50]		√	√				√	
Letichevsky, Kapitonova, Kotlyarov, Volkov, Letichevsky Jr., Weigert [51]		√	√				√	
Alur, Etassami, Yannakakis [36, 52]		√	√					
Uchitel, Kramer, Magee [53]		√	√		√			
Graubmann et al. [54, 55, 56, 57]		√						
Jonsson, Padilla [59]		√	√					
Lund, Stølen [60, 61]		√	√	√	√	√	√	
Cengarle, Knapp, Mühlberger [62, 63]		√	√					
Cavarra, Küster-Filipe [64]		√		√	√			
Grosu, Smolka [65]		√	√		√	√		
Harel, Damm, Maoz, Marelly, Thiagarajan (LSC) [3, 66, 67, 68]		√	√	√	√		√	
Sengupta, Cleveland (TMSC) [69, 70]		√	√	√	√	√		
Kosiuczenko, Wirsing [71]		√	√				√	

Table 4.3. Evaluation of semantics for state machines and similar notations

	Denotational semantics?	Operational semantics?	Underspecification?	Trace set properties?	Incomplete models?	Refinement?	Real-time?	Probabilities?
Broy, Cengarle, Rumpe [75]	√		√			√	√	
Simons [77]	√		√					
Rossi, Enciso, de Guzmán [78]	√						√	
Hinkel, Holz, Stølen [79, 80]	√		√			√	√	
Harel, Naamad [4, 81]		√	√				√	
Börger, Cavarra, Riccobene [85, 86]		√	√					
Jürjens [87, 88]		√	√	√		√		
von der Beeck [89]		√	√					
Knapp, Merz, Rauh [90]		√	√				√	
Lüttgen, von der Beeck, Cleaveland [93]		√	√				√	
Jansen, Hermanns, Katoen [94, 95]		√	√	√			√	√
Huszerl, Kosmidis, Cin, Majzik, Pataricza [97]		√	√	√			√	√

between *must* and *may* behavior, as for example [34]. The final evaluation criteria we want to comment upon is the support for incomplete models. This is difficult to assess, as we can always *choose* to interpret a sequence diagram as an incomplete model. The evaluation was therefore based on the approaches' treatment of negative behavior, their support for existential behavior, and their definitions of refinement.

A few comments to Table 4.3 are also needed. First, we notice that none of the approaches capture incomplete models. The reason is that state machines, unlike sequence diagrams, focus on describing a single component rather than an interaction scenario. All state machine variants we are aware of describe only the behavior that the component may exhibit; behavior not explicitly described is negative in the sense that it should not occur. There is, therefore, no explicit operator for expressing negative behavior, and all behavior is either positive or negative – there is no inconclusive behavior. Second, most approaches have received a check mark under "Underspecification", but only a few under "Trace set properties". The reason is that, for approaches with only one kind of transition, we have assumed that nondeterministic choices between transitions represent underspecification, rather than explicit nondeterminism. This decision

was made because a fairly standard notion of refinement is trace inclusion – the requirement that the traces of the refined specification is a subset of the traces of the original specification. Third, all approaches with probabilities have received a check mark in the "Trace set properties" column, as all alternatives with a certain (non-zero) probability are necessarily represented in a correct implementation. In this sense, probabilistic choices can be viewed as a kind of inherent nondeterminism, which means that trace set properties can be captured.

4.7 Summary and Conclusions

In this paper we have defined a set of evaluation criteria for semantics of models for embedded systems. We claim that our criteria represent an important set of the properties that semantics of models for embedded systems should support.

These evaluation criteria have been applied in an evaluation of formal semantics for models expressed in UML sequence diagrams, state machines, and similar notations. In the paper we have presented and evaluated in all more that 30 approaches, divided into four main categories: denotational semantics of sequence diagrams, operational semantics of sequence diagrams, denotational semantics of state machines and operational semantics of state machines. Our selection of approaches to evaluate is not exhaustive, but we believe that it gives a representative picture of the various approaches available.

As the evaluation reveals there is no lack of approaches to formal semantics for UML sequence diagrams and state machines, and many of these have desirable properties. We do not proclaim a winner, but we have established that formal semantics of relevant modeling languages are readily available for the developers of embedded systems. We have not evaluated to what degree the approaches presented in this paper are supported by suitable tools, nor to what degree they have been put to practical application. Still, judging from our evaluation, there should be a large potential for applying UML models supported by formal semantics in the development of embedded systems. It is up to developers to choose a suitable approach based on the nature of the system to be developed, and the background and experience of the development team.

Acknowledgements

The work on which this paper reports has partly been funded by the Research Council of Norway through the projects SARDAS (15295/431) and ENFORCE (164382/V30), and partly by the European Commission through the MODELPLEX project (Contract no. 034081) under the IST Sixth Framework Programme.

References

[1] Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1 (non-change bar). OMG Document: formal/2007-02-05 (2005)
[2] International Telecommunication Union: Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (1999)

[3] Damm, W., Harel, D.: LSCs: Breathing life into Message Sequence Charts. Formal Methods in System Design 19, 45–80 (2001)
[4] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
[5] International Telecommunication Union: Specification and description language (SDL), ITU-T Recommendation Z.100 (2000)
[6] Labinaz, G., Bayoumi, M.M., Rudie, K.: A survey of modeling and control of hybrid systems. Annual Reviews of Control 21, 79–92 (1997)
[7] Giese, H., Henkler, S.: A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages and Computing 17(6), 528–550 (2006)
[8] McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 79–93. IEEE Computer Society, Los Alamitos (1994)
[9] Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21(4), 181–185 (1985)
[10] Schneider, F.B.: Enforceable security policies. ACM Transactions on Information System Security 3(1), 30–50 (2000)
[11] Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? Computer 37(10), 64–72 (2004)
[12] Prinz, A.: Formal semantics of specification languages. Telektronikk (4), 146–155 (2000)
[13] Fecher, H., Schönborn, J., Kyas, M., de Roever, W.P.: 29 new unclarities in the semantics of UML 2.0 state machines. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 52–65. Springer, Heidelberg (2005)
[14] Schmidt, D.A.: Denotational semantics. A methodology for language development. William C. Brown (1988)
[15] Hoare, C.A.R., Jifeng, H.: Unifying theories of programming. Prentice-Hall, Englewood Cliffs (1998)
[16] Object Management Group: Unified Modeling Language Specification, version 1.4. OMG Document: formal/2001-09-67 (2001)
[17] Facchi, C.: Formal semantics of Time Sequence Diagrams. Technical report TUM-I9540, Technische Universität München (1995)
[18] International Telecommunication Union: Information technology – Open Systems Interconnection – Basic reference model: Conventions for the definition of OSI services, ITU-T Recommendation X.210 (1993)
[19] Bræk, R., Gorman, J., Haugen, Ø., Møller-Pedersen, B., Melby, G., Sanders, R., Stålhane, T.: TIMe: The Integrated Method. Electronic Textbook v4.0. SINTEF (1999)
[20] International Telecommunication Union: Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (1996)
[21] International Telecommunication Union: Message Sequence Chart (MSC), ITU-T Recommendation Z.120, Annex B: Formal semantics of Message Sequence Charts (1998)
[22] Haugen, Ø.: Comparing UML 2.0 Interactions and MSC-2000. In: Amyot, D., Williams, A.W. (eds.) SAM 2004. LNCS, vol. 3319, pp. 65–79. Springer, Heidelberg (2005)
[23] Object Management Group: UML Testing Profile, version 1.0. OMG Document: formal/2005-07-07 (2005)
[24] Object Management Group: UML Profile for Schedulability, Performance, and Time Specification, version 1.1. OMG Document: formal/2005-01-02 (2005)

[25] Katoen, J.P., Lambert, L.: Pomsets for Message Sequence Charts. In: Formale Beschreibungstechniken für Verteilte Systeme, pp. 197–208. Shaker (1998)
[26] Alur, J., Yannakakis, M.: Model checking of Message Sequence Charts. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 98–113. Springer, Heidelberg (1999)
[27] Krüger, I.H.: Distributed system design with Message Sequence Charts. PhD thesis, Technische Universität München (2000)
[28] Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software and Systems Modeling 4(4), 355–367 (2005)
[29] Seehusen, F., Solhaug, B., Stølen, K.: Adherence preserving refinement of trace-set properties in STAIRS: Exemplified for information flow properties and policies. Software and Systems Modeling 8(1), 45–65 (2009)
[30] Störrle, H.: Assert, negate and refinement in UML 2 interactions. In: 2nd International Workshop on Critical Systems Development with UML (CSD-UML 2003), Technische Universität München, pp. 79–93 (2003)
[31] Störrle, H.: Semantics of interaction in UML 2.0. In: IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), pp. 129–136. IEEE Computer Society, Los Alamitos (2003)
[32] Störrle, H.: Trace semantics of interactions in UML 2.0. Technical report TR 0403, Institut für Informatik, der Ludwig-Maximilians-Universität München (2004)
[33] Cengarle, M.V., Knapp, A.: UML 2.0 interactions: Semantics and refinement. In: 3rd International Workshop on Critical Systems Development with UML (CSD-UML 2004), Technische Universität München, pp. 85–99 (2004)
[34] Küster-Filipe, J.: Modelling concurrent interactions. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 304–318. Springer, Heidelberg (2004)
[35] Alur, R., Holzmann, G.J., Peled, D.: An analyzer for Message Sequence Charts. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 35–48. Springer, Heidelberg (1996)
[36] Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. IEEE Transactions on Software Engineering 29(7), 623–633 (2003)
[37] Zheng, T., Khendek, F., Hélouët, L.: A semantics for timed MSC. Electronic Notes in Theoretical Computer Science 65(7), 85–99 (2002)
[38] Zheng, T., Khendek, F., Parreaux, B.: Refining timed MSCs. In: Reed, R., Reed, J. (eds.) SDL 2003. LNCS, vol. 2708, pp. 234–250. Springer, Heidelberg (2003)
[39] Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: Why timed sequence diagrams require three-event semantics. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 1–25. Springer, Heidelberg (2005)
[40] Runde, R.K.: STAIRS - Understanding and developing specifications expressed as UML interaction diagrams. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo (2007)
[41] Faltin, N., Lambert, L., Mitschele-Thiel, A., Slomka, F.: An annotational extension of Message Sequence Charts to support performance engineering. In: 8th International SDL Forum: Time for Testing, SDL, MSC and Trends (SDL 1997), pp. 307–322. Elsevier, Amsterdam (1997)
[42] Lambert, L.: PMSC for performance evaluation. In: 1st Workshop on Performance and Time in SDL/MSC, pp. 70–80 (1998)
[43] Refsdal, A., Husa, K.E., Stølen, K.: Specification and refinement of soft real-time requirements using sequence diagrams. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 32–48. Springer, Heidelberg (2005)

[44] Refsdal, A., Runde, R.K., Stølen, K.: Underspecification, inherent nondeterminism and probability in sequence diagrams. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 138–155. Springer, Heidelberg (2006)
[45] Refsdal, A.: Specifying computer systems with probabilistic sequence diagrams. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo (2008)
[46] Mauw, S.: The formalization of Message Sequence Charts. Computer Networks and ISDN Systems 28(1), 1643–1657 (1996)
[47] Mauw, S., Reniers, M.A.: An algebraic semantics of Basic Message Sequence Charts. The Computer Journal 37(4), 269–278 (1994)
[48] Okazaki, M., Aoki, T., Katayama, T.: Formalizing sequence diagrams and state machines using Concurrent Regular Expression. In: 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, SCESM 2003 (2003)
[49] Mauw, S., Reniers, M.A.: Operational semantics for MSC'96. Computer Networks 31(17), 1785–1799 (1999)
[50] Mauw, S., Reniers, M.A.: High-level Message Sequence Charts. In: 8th International SDL Forum: Time for Testing, SDL, MSC and Trends (SDL 1997), pp. 291–306. Elsevier, Amsterdam (1997)
[51] Letichevsky, A.A., Kapitonova, J.V., Kotlyarov, V.P., Volkov, V.A., Letichevsky Jr., A.A., Weigert, T.: Semantics of Message Sequence Charts. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 117–132. Springer, Heidelberg (2005)
[52] Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theoretical Computer Science 331(1), 97–114 (2005)
[53] Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specification and behavior models using implied scenarios. ACM Transactions on Software Engineering and Methodology 13(1), 37–85 (2004)
[54] Graubmann, P., Rudolph, E., Grabowski, J.: Towards a Petri net based semantics for Message Sequence Charts. In: 6th International SDL Forum: Using objects (SDL 1993), pp. 179–190. Elsevier, Amsterdam (1993)
[55] Heymer, S.: A semantics for MSC based on Petri net components. In: 4th International SDL and MSC Workshop (SAM 2000), pp. 262–275 (2000)
[56] Sgroi, M., Kondratyev, A., Watanabe, Y., Lavagno, L., Sangiovanni-Vincentelli, A.: Synthesis of Petri nets from Message Sequence Charts specifications for protocol design. In: Design, Analysis and Simulation of Distributed Systems Symposium (DASD 2004), pp. 193–199 (2004)
[57] Gunter, E.L., Muscholl, A., Peled, D.: Compositional Message Sequence Charts. International Journal on Software Tools for Technology Transfer 5(1), 78–89 (2003)
[58] Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable Petri net models. In: 3rd International Workshop on Software and Performance (WOSP 2002), pp. 35–45. ACM Press, New York (2002)
[59] Jonsson, B., Padilla, G.: An execution semantics for MSC-2000. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 365–378. Springer, Heidelberg (2001)
[60] Lund, M.S.: Operational analysis of sequence diagram specifications. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo (2008)

[61] Lund, M.S., Stølen, K.: A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 380–395. Springer, Heidelberg (2006)
[62] Cengarle, M.V., Knapp, A.: Operational semantics of UML 2.0 interactions. Technical report TUM-I0505, Technische Universität München (2005)
[63] Mühlberger, H.: Eine verteile operationale Semantik für UML 2.0-Interaktionen. Diplomarbeit, Institut für Informatik, der Ludwig-Maximilians-Universität München (2007)
[64] Cavarra, A., Küster-Filipe, J.: Formalizing liveness-enriched sequence diagrams using ASMs. In: Zimmermann, W., Thalheim, B. (eds.) ASM 2004. LNCS, vol. 3052, pp. 67–77. Springer, Heidelberg (2004)
[65] Grosu, R., Smolka, S.A.: Safety-liveness semantics for UML 2.0 sequence diagrams. In: 5th International Conference on Application of Concurrency to System Design (ACSD 2005), pp. 6–14. IEEE Computer Society, Los Alamitos (2005)
[66] Harel, D., Marelly, R.: Come, let's play: Scenario-based programming using LSCs and the Play-Engine. Springer, Heidelberg (2003)
[67] Harel, D., Thiagarajan, P.S.: Message Sequence Charts. In: Lavagano, L., Martin, G., Selic, B. (eds.) UML for real. Design of embedded real-time systems, pp. 77–105. Kluwer, Dordrecht (2003)
[68] Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. In: 5th International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2006), pp. 13–19. ACM Press, New York (2006)
[69] Sengupta, B., Cleaveland, R.: Triggered Message Sequence Charts. SIGSOFT Software Engineering Notes 27(6), 167–176 (2002)
[70] Sengupta, B., Cleaveland, R.: Triggered Message Sequence Carts. IEEE Transactions on Software Engineering 32(8) (2006)
[71] Kosiuczenko, P., Wirsing, M.: Towards an integration of Message Sequence Charts and Timed Maude. Journal of Integrated Design & Process Science 5(1), 23–44 (2001)
[72] Crane, M.L., Dingel, J.: On the semantics of UML state machines: Categorization and comparison. Technical report 2005-501, School of Computing, Queens's University, Kingston (2005)
[73] Broy, M., Cengarle, M.V., Rumpe, B.: Towards a system model for UML, the structural data model. Technical report TUM-I0612, Technische Universität München (2006)
[74] Broy, M., Cengarle, M.V., Rumpe, B.: Towards a system model for UML, part 2: The control model. Technical report TUM-I0710, Technische Universität München (2007)
[75] Broy, M., Cengarle, M.V., Rumpe, B.: Towards a system model for UML, part 3: The state machine model. Technical report TUM-I0711, Technische Universität München (2007)
[76] Broy, M., Stølen, K.: Specification and development of interactive systems. In: FOCUS on streams, interface, and refinement. Springer, Heidelberg (2001)
[77] Simons, A.J.H.: On the compositional properties of UML statechart diagrams. In: Rigorous Object-Oriented Methods (ROOM 2000), Workshops in Computing, BCS (2000) (2000)
[78] Rossi, C., Enciso, M., de Guzmán, I.P.: Formalization of UML state machines using temporal logic. Software and Systems Modeling 3(1), 31–54 (2004)

[79] Hinkel, U.: Verification of SDL specifications on the basis of stream semantics. In: 1st Workshop of the SDL Forum Society on SDL and MSC (SAM 1998), pp. 241–250 (1998)
[80] Holz, E., Stølen, K.: An attempt to embed a restricted version of SDL as a target language in Focus. In: Formal Description Techniques VII (FORTE 1994), pp. 324–339. Chapman and Hall, Boca Raton (1994)
[81] Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (1996)
[82] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: STATEMATE: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16(4), 403–414 (1990)
[83] Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
[84] Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASMs. Information and Software Technology 46(5), 287–292 (2004)
[85] Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: International Workshop on Abstract State Machines, Theory and Applications, pp. 223–241. Springer, Heidelberg (2000)
[86] Börger, E., Cavarra, A., Riccobene, E.: Modeling the meaning of transitions from and to concurrent states in UML state machines. In: 2003 ACM Symposium on Applied Computing, pp. 1086–1091. ACM Press, New York (2003)
[87] Jürjens, J.: A UML statecharts semantics with message-passing. In: 2002 ACM Symposium on Applied Computing, pp. 1009–1013. ACM Press, New York (2002)
[88] Jürjens, J.: Secure systems development with UML. Springer, Heidelberg (2005)
[89] von der Beeck, M.: A structured operational semantics for UML-statecharts. Software and Systems Modeling 1(2), 130–141 (2002)
[90] Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 395–416. Springer, Heidelberg (2002)
[91] Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer 1, 134–152 (1997)
[92] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
[93] Lüttgen, G., von der Beeck, M., Cleaveland, R.: A compositional approach to statecharts semantics. Technical report, Institute for Computer Applications in Science and Engineering (2000)
[94] Jansen, D.N., Hermanns, H., Katoen, J.P.: A QoS-oriented extension of UML statecharts. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 76–91. Springer, Heidelberg (2003)
[95] Jansen, D.N., Hermanns, H.: Qos modelling and analysis with UML-statecharts: the Stocharts approach. SIGMETRICS Performance Evaluation Review 32(4), 28–33 (2005)
[96] Eshuis, R., Wieringa, R.: Requirements-level semantics for UML statecharts. In: 4th International Conference on Formal Methods for Open Object-Based Distributed Systems IV, pp. 121–140. Kluwer, Dordrecht (2000)
[97] Huszerl, G., Kosmidis, K., Cin, M.D., Majzik, I., Pataricza, A.: Quantitative analysis of UML statechart models of dependable systems. The Computer Journal 45(3), 260–277 (2002)