Softw Syst Model (2008) 7:237–252
DOI 10.1007/s10270-007-0054-z

REGULAR PAPER

Assert and negate revisited: Modal semantics for UML sequence diagrams

David Harel · Shahar Maoz

Received: 26 August 2006 / Revised: 17 February 2007 / Accepted: 2 March 2007 / Published online: 15 May 2007

Abstract Live Sequence Charts (LSC) extend Message Sequence Charts (MSC), mainly by distinguishing possible from necessary behavior. They thus enable the specification of rich multi-modal scenario-based properties, such as mandatory, possible and forbidden scenarios. The sequence diagrams of UML 2.0 enrich those of previous versions of UML by two new operators, *assert* and *negate*, for specifying required and forbidden behaviors, which appear to have been inspired by LSC. The UML 2.0 semantics of sequence diagrams, however, being based on pairs of valid and invalid sets of traces, is inadequate, and prevents the new operators from being used effectively.

We propose an extension of, and a different semantics for this UML language—*Modal Sequence Diagrams* (*MSD*)—based on the universal/existential modal semantics of LSC. In particular, in MSD *assert* and *negate* are really modalities, not operators. We define MSD as a UML 2.0 profile, thus paving the way to apply formal verification, synthesis, and scenario-based execution techniques from LSC to the mainstream UML standard.

Communicated by Dr. Oystein Haugen.

Preliminary version appeared in SCESM '06: Proc. of the 2006 Int. workshop on Scenarios and State Machines, Shanghai, China (May 2006) [15]. This research was supported by the Israel Science Foundation (grant No.287/02-1), and by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

D. Harel (✉) · S. Maoz
The Weizmann Institute of Science, Rehovot, Israel
e-mail: dharel@weizmann.ac.il

S. Maoz
e-mail: shahar.maoz@weizmann.ac.il

Keywords UML Interactions · Sequence diagrams · Live sequence charts · Standardization · Formal semantics

1 Introduction

While the language of Message Sequence Charts (MSC), defined by ITU [18] is a popular means for specifying interactions between objects or processes, it is expressively weak, being based on a modest semantic notion of a partial ordering of events. This is true also for most proposed extensions of MSC, where despite enhancements with regular expression constructs, the underlying semantics of a partial order remains intact. Live Sequence Charts (LSC), introduced in [7], extend MSC by allowing a distinction between possible and necessary behavior, with, for example, a hot/cold modality for elements within the charts. It thus enables specification of rich multi-modal scenario-based properties, such as liveness and safety, using mandatory, possible and forbidden scenarios. Since its introduction in 1998, the LSC language with its formal (operational) semantics has triggered much research both on the requirements/specification level and as an executable programming language for reactive systems. Thus, formal methods and tools can take advantage of scenario-based specifications throughout the development cycle of systems, from requirements capture and analysis, to design and specification, implementation, testing, formal verification, and system execution (see, e.g.,[8,16,22,25,27,31]). Initial projects that use LSC have been carried out recently in the automotive, telecommunication, and hardware domains (see, e.g., [3,6]).

The Unified Modeling Language (UML) includes a section on sequence diagrams, which early versions of the standard adopted directly from MSC. In its most recent version

[34] [1], the UML 2.0 Sequence Diagrams specification is based on a revised metamodel and has adopted additional important and useful features from MSC and high-level MSC (HMSC) [18]. These include, for example, the concept of *InteractionUse* (originally called *MSC reference*), which allows referring to another interaction from within a diagram (sharing portions of an interaction between several other interactions), the notion of *InteractionFragment*, and its operators for choice, sequential, parallel, and iterative composition.

However, the most interesting addition to sequence diagrams in the UML 2.0 standard are two new and crucial operators, *assert* and *negate*, which were probably intended to bring the multi-modal hot/cold essence of LSC into the new standard, and to allow the specification of required and forbidden behaviors, respectively. These two new operators constitute the main mechanism adopted by the UML 2.0 authors for increasing the actual expressive power of the language. Unfortunately, in Sect. 2.3 we show that the semantics suggested in the UML 2.0 specification document, which is based on pairs of valid and invalid sets of traces, is inadequate, and does not lead to a well defined semantics of these new features of the language. The semantic confusion around the use of *assert* and *negate* prevents these features from being used effectively, since the ability to specify liveness and safety properties, e.g., via required and forbidden behaviors, is a fundamental prerequisite for the use of sequence diagrams in specification, testing and verification, and also, of course, when executing the language and using it for actual programming, as discussed in [16].

We believe that the root of the problem is that increasing the expressive power of sequence diagrams to allow liveness and safety specifications requires a universal semantic interpretation, similar to the one developed for LSC, which is not present in the standard's suggested semantics.

Thus, in this paper we propose *Modal Sequence Diagrams* (*MSD*)[2], which is an extension of UML 2.0 Sequence Diagrams based on the universal/existential distinction that is at the heart of live sequence charts. MSD allows denoting parts of an interaction, such as messages and constraints, or a complete interaction, as universal, i.e., mandatory, thus specifying that messages *have* to be sent, conditions *must* become true, etc. Technically, we do this by defining *modal* interaction fragments. These extend the *InteractionFragment* class of the UML 2.0 metamodel with an attribute *interactionMode*, which can be either *hot* (universal) or *cold* (existential). We argue that this yields a most natural and useful way to define the assert and negate operators of UML 2.0 Sequence Diagrams, exploiting the multi-modal spirit of LSCs and adapting it to the UML standard. Thus, *assert* and *negate*, we claim, are not to be viewed as operators but as modalities.

The similar, but not identical, formalizations to the semantics of UML 2.0 Interactions in, e.g., [5,17,33] and the standard document itself [34] (which unfortunately contains additional problems, some of which we hint to at the beginning of Sect. 6), make it difficult to refer to it as a baseline for the proposed extension. Still, we believe our version, MSD, is very close to UML 2.0 Sequence Diagrams as they are described in the standard documents, yet has stronger expressive power and far more robust semantics. The fact that MSD is defined as a UML profile makes it possible to carry over to mainstream UML most of the recent results and applications worked out for LSC, in formal verification, testing, synthesis, and scenario-based execution. As we shall see, this requires only minor modifications (e.g., ignoring LSC precharts).

The paper is organized as follows. In Sect. 2 we present preliminary background on LSC, on UML 2.0 Sequence Diagrams, and on the problematic definition of *assert* and *negate* in the semantics suggested in the standard, which is the primary motivation of our work. Section 3 presents the basics of Modal Sequence Diagrams. Section 4 discusses and demonstrates additional useful constructs in the language.

Following Sect. 5, which discusses related work, Sect. 6 contains a discussion and directions for future work.

Technical details related to the MSD profile are given in Appendix A. An outline of the formal semantics for MSD is given in Appendix B.

2 Preliminaries

In this section we give short background on Live Sequence Charts and on UML 2.0 Sequence Diagrams. Historically, both languages are descendants of Message Sequence Charts (MSC) [18]. We assume the reader is familiar with the basic syntax and partial order semantics of MSCs and concentrate on the differences between the two languages that are relevant to this paper.

2.1 Live sequence charts

A sequence chart consists of lifelines and events, such as sending a message, receiving a message, or evaluating a condition. Events are ordered along lifelines from top to bottom. The LSC language [7] defines two types of charts: *universal* (annotated by a solid borderline) and *existential* (annotated by a dashed borderline). Universal charts are used to specify restrictions over all possible system runs. A universal chart typically contains a *prechart* that specifies the scenario which, if successfully executed, forces the system to satisfy

[1] Throughout this paper, we refer to the latest available UML 2.0 superstructure specification document version of August 2005. Changes we have seen so far in the draft adopted specification for UML 2.1 superstructure of April 2006 are insignificant to the issues presented in the paper.

[2] In the preliminary version of the present paper [15], we used the term *Modal UML Sequence Diagrams* (*MUSD*).

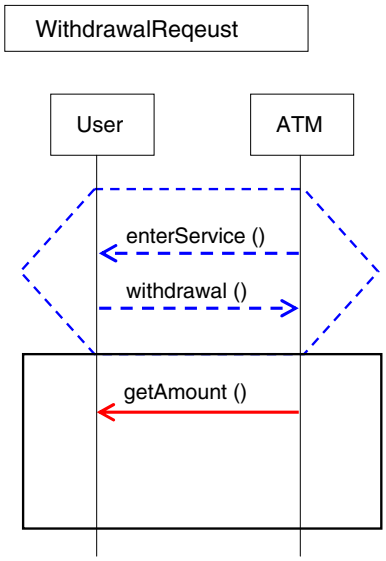

Fig. 1 A universal LSC: `WithdrawalRequest`

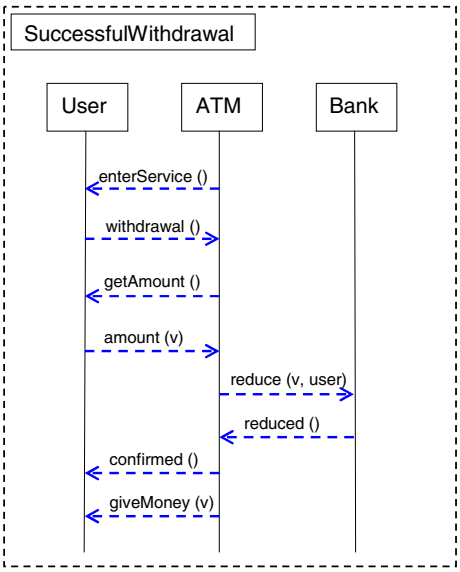

Fig. 2 An existential LSC: `WithdrawalSuccess`

the scenario given in the actual chart body. Existential charts specify sample interactions between the system and its environment, and must be satisfied by at least one system run. They do not force the application to behave in a certain way in all cases, but rather state that there is at least one set of circumstances under which a certain behavior occurs. Existential charts can be used to specify system tests, or simply to illustrate longer (non-restricting) scenarios that provide a broad picture of the behavioral possibilities to which the system gives rise.

To illustrate the main concepts and constructs of the LSC language we use a simple example, an ATM (Automatic Teller Machine) that offers money withdrawals (the example is intentionally similar to other examples, given, e.g., in [17], in order to allow easy comparison between the different approaches). The chart appearing in Fig. 1 specifies that whenever the ATM asks the user to enter a service and the user chooses `withdrawal`, the ATM must send the user a message asking for the withdrawal amount. This message is depicted using a solid red arrow, to denote that it is mandatory, *hot* in LSC terminology; when reached, the message must be sent and must be received.

The chart `SuccessfulWithdrawal` appearing in Fig. 2 is an existential chart, as denoted by its dashed borderline. It describes a scenario in which a user asks to withdraw an amount v from her account. Since the chart is existential, it specifies a possible scenario, i.e., a scenario that should be satisfied by at least one system run.

The examples show only the most basic LSC features. The LSC language has been extended to support many additional features, such as symbolic lifelines, timing constructs, and forbidden elements. For a detailed description of these the reader is referred to [16].

2.2 UML 2.0 sequence diagrams

We briefly summarize the abstract syntax and semantics of UML 2.0 interactions as they are given in the standard [34]. A more thorough description can be found in the standard superstructure document itself and in, e.g., [5,33].

UML 2.0 Interactions can be displayed in several different types of diagrams but the differences between the types are not relevant to our discussion, so that we use Sequence Diagrams and Interactions interchangeably. A UML 2.0 Sequence Diagram has sets of *Lifelines*, *Messages*, and *InteractionFragments*. A lifeline represents *OccurrenceSpecifications* at one of the entities involved in the interaction. A message typically associates two *OccurrenceSpecifications*, corresponding to sending and receiving events. Since *Interaction* is a subclass of *InteractionFragment*, interactions may contain any number and kind of interactions. An important kind of interaction is *CombinedFragment*, consisting of an operator and a number of operands, which may be either plain interactions or, again, combined fragments.

The semantic domain for UML 2.0 Interactions consists of *OccurrenceSpecifications*, which represent moments in time (with no duration) associated with actions, such as the sending or receiving of a message. A trace is a sequence of occurrence specifications.

According to the standard, the semantics of an interaction is given as a pair of sets of traces, representing, respectively, the valid and invalid traces. These two sets need not be exhaustive, as the traces that are not included in the union of the two sets are not described and we cannot know whether they are valid or invalid; unspecified traces are thus contingent. According to the standard, invalid traces arise only out

of the operators *negate* and *assert*. Ignoring these, the semantics of combined fragments of the diagrams with operators such as *seq*, *par*, *loop*, and *alt* is quite intuitive. For example, the set of traces defined by applying the *alt* operator (with guards) to a set of interaction fragments is the union of the guarded traces of the operands, and the set defined by the *loop* operator with an integer constraint *n* is the result of iteratively applying *n* concatenations to the traces of the loop operand (body).

In addition to these high-level structural operators, the standard defines a macro-like mechanism for modularity, using the *ref* operator. If we forbid the use of recursive references (which, by the way, the standard does not refer to at all, so there is no way of knowing whether they are allowed or not. . .), the semantics of a fragment that includes *ref* is simply the semantics of the fragment after syntactically substituting the references with copies of the referenced interactions.

2.3 The problem with *assert* and *negate*

We now briefly illustrate the inadequacy and vagueness of *assert* and *negate* as they are defined in the standard specification document [34]. As to assert, the standard explains that *"the sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace"* [34, p. 456]. This suggests that the invalid set of traces for an asserted fragment is its complement, i.e., the set of all other possible traces. Later however, the standard declares that *"the invalid set of traces are associated only with the use of a Negative CombinedFragment"* [34, p. 468]. This seems contradictory. Also, it is not clear what is the scope of *"the only valid continuations"* [34, p. 456] (i.e., in LSC lingo, what is the prechart).

As to negate, according to the standard, it *"designates that the combined fragment represents traces that are defined to be invalid"* [34, p. 455], so *neg* seems a reasonable and a natural way to specify counterexamples. However, *"all interaction fragments that are different from Negative are considered positive meaning that they describe traces that are **valid** and **should be possible"*** [34, p. 455, emphasis added]. This leads to the basic problem: what does a 'valid' trace mean? Is it sometimes possible, initially possible, or always possible?

These semantic confusions are significant, in that they prevent these important features from being used effectively. Indeed, *assert* is defined in the standard specs but is not mentioned in recent works such as [11], which discuss liveness in sequence diagrams, nor does it appear in the popular UML user guide [2]. In fact, no tool seems to exist which adequately supports a reasonable interpretation of *assert* and *negate* as they are defined in the standard, nor have we seen any paper with a satisfactory proposal for such an interpretation. We believe that the root of the problem is that increasing the expressive power of sequence diagrams to allow liveness and safety specifications requires a carefully worked out distinction between existential and universal interpretations, similar to that made in LSCs [7,16]. In a nutshell, the way we do this, to be described next, calls for viewing *assert* and *negate* as modalities rather than as operators.

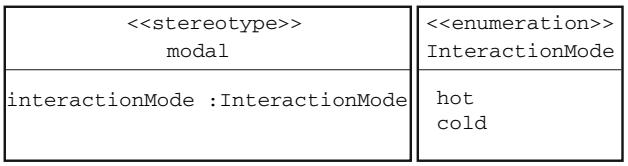

Fig. 3 The stereotype *modal*

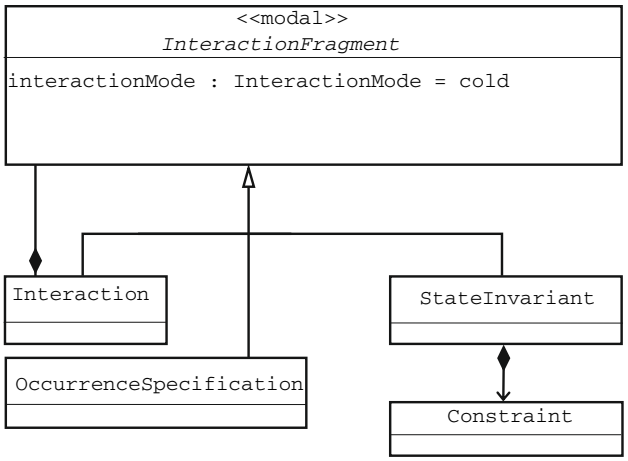

Fig. 4 Extending *InteractionFragment* with the *modal* stereotype

3 Modal sequence diagrams

We use the term *Modal Sequence Diagrams* (*MSD*) for our proposed modal extension of UML Sequence Diagrams. As a basis for the extension, we start off with the so called *existential* or *positive fragment* of UML 2.0 Interactions, i.e., without *assert* and *negate*. To allow the specification of modalities over interactions, we define a stereotype *modal* with a single attribute *interactionMode*, which, in the spirit of LSC, can be either *hot* (universal) or *cold* (existential) (see Fig. 3). The new stereotype is introduced as an extension of the abstract class *InteractionFragment*, and hence also of its subclasses: *Interaction*, *OccurrenceSpecification*, and *StateInvariant* (see Fig. 4).

We now explain the semantics of an MSD specification. We consider a system-model as consisting of objects, or instances, each of which has a set of associated actions and variables. In general, we consider infinite runs.

An *MSD specification* is a set of modal sequence diagrams. Roughly, the semantics of an MSD specification is just like that of LSCs [7,16]. It is given by identifying the

trace-language of each diagram and then using it to define when a system-model satisfies the specification: A system-model satisfies an MSD specification if (1) every one of its possible runs satisfies each universal diagram in the specification, and (2) every existential diagram is satisfied by at least one possible system run.

3.1 The basics

The semantics of an MSD interaction fragment depends both on the partial order induced by its occurrence specifications and on its mode. In order to support sequential composition of MSDs, we leave out the *prechart* construct from LSCs. Instead, we take a more general approach, where cold fragments inside universal interactions serve prechart-like purposes: a cold fragment does not have to be satisfied in all runs but if and when it is satisfied it necessitates the satisfaction of its subsequent hot fragment; and this is true in all runs.

As to notation, we can propose a number of alternative ways of distinguishing universal from existential elements. One is to adopt the notation from LSC: universal MSDs are annotated by a solid borderline and existential MSDs by a dashed borderline. The same method can be used to distinguish universal elements (messages, constraints, etc.) from existential ones. One can also use the LSC color coding: hot elements in red and cold elements in blue. In addition, we suggest to change the keyword on the top left corner of universal diagrams from `sd` to `usd`.

Consider the example `Withdrawal` MSD of Fig. 5. This MSD is universal, as can be seen by its solid border and the keyword `usd`. It specifies that whenever the ATM asks the user to enter a service and the user asks for a withdrawal, the ATM must ask the user for the withdrawal amount. Then, if the user has entered an amount v, the ATM must ask the bank to reduce the amount from the user's account. If and when the bank sends the `reduced` message to the ATM, the ATM must send the user a confirmation message and give her the money.

To conform to the standard's notation, we allow the use of *assert* in MSD, but we interpret it as syntactic sugar for assigning a hot mode to all the *OccurrenceSpecifications* inside the interaction fragment operand. Fig. 6 is thus equivalent to Fig. 5.

We define the trace-language of a universal MSD using an *alternating weak word automaton* (AWW) [23,29] (Similar constructions for LSC were given by Bontemps et al. in [1] and by Klose et al. in [20]). For a universal diagram U, the AWW consists of a state for each legal cut and a transition for each legal occurrence specification, representing the partial order induced by the diagram. All states have self transitions labeled $\Sigma \setminus M$, where Σ is the set of all occurrences in the system-model and M is the set of occurrence specifications mentioned in U. All the states that have no hot

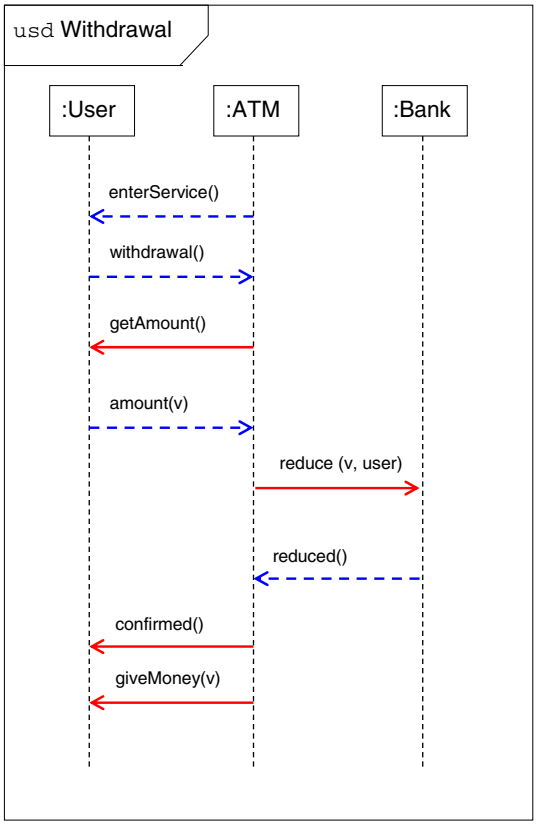

Fig. 5 A universal MSD for `Withdrawal`

outgoing transitions represent cold cuts and are accepting. These states also have an outgoing transition to an accepting sink state, labeled $M \setminus mc$, where mc is the set of cold enabled events in the state. All non-accepting states, i.e., states with outgoing hot transitions, have an additional outgoing transition to a rejecting sink state labeled $M \setminus mh$, where mh is the set of enabled events in the state.[3] Finally, the initial state, representing the minimal legal cut, has a self transition labeled Σ, quantified universally with all the minimal occurrence specifications. The partition of the state space is induced naturally from the partial order of legal cuts. We give an outline of this construction more formally in Appendix B.

Figure 7 shows part of the AWW corresponding to the `Withdrawal` MSD (we simplify the example by unifying a message sent and received into a single occurrence). A system-model satisfies the universal MSD `Withdrawal` if all its runs are in the trace-language of the diagram, i.e., if all its runs are accepted by the AWW of Fig. 7.

For an existential MSD the construction is similar. States represent cuts and transitions represent legal occurrences.

[3] Specifically, this means that when there are hot and cold enabled messages in the same (hot) cut, an occurrence of any one of the enabled messages, either hot or cold, is not considered a violation, so the trace remains in the trace-language of the diagram.

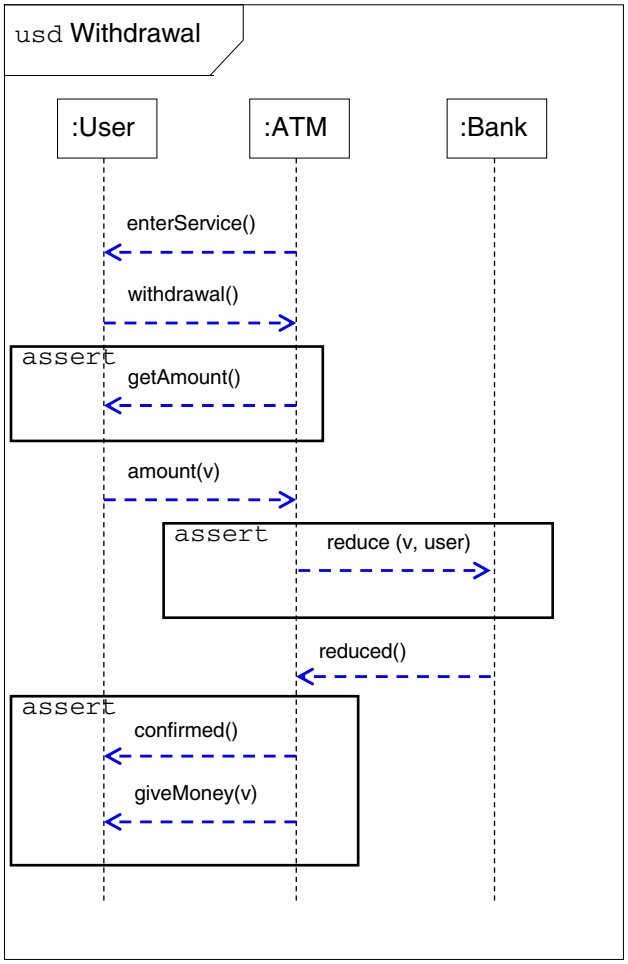

Fig. 6 *Assert* as syntactic sugar for assigning a *hot* mode to all the *OccurrenceSpecifications* inside the interaction fragment operand

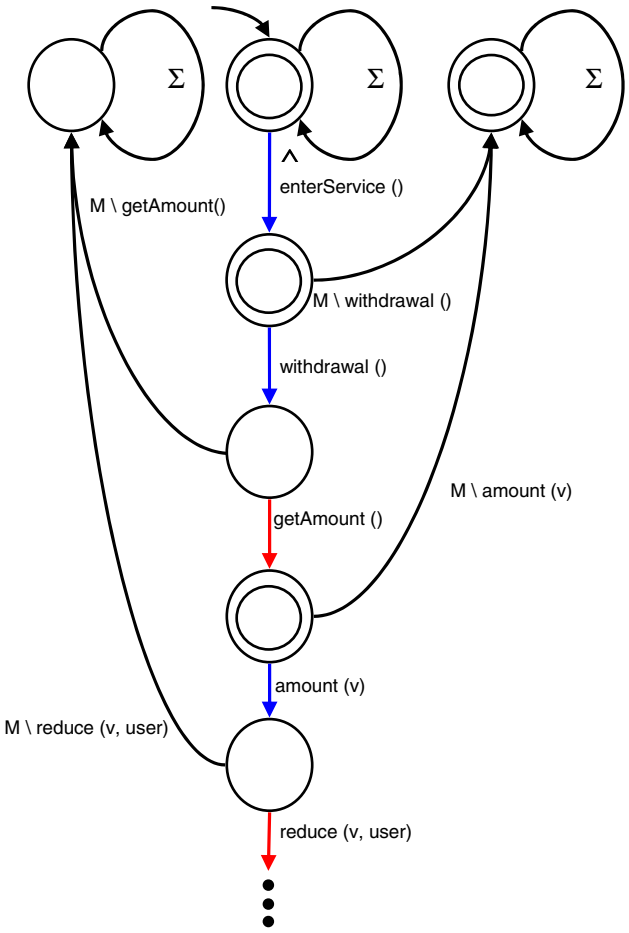

Fig. 7 Part of the automaton for `Withdrawal` of Fig. 6. Self transitions labeled $\Sigma \setminus M$ omitted

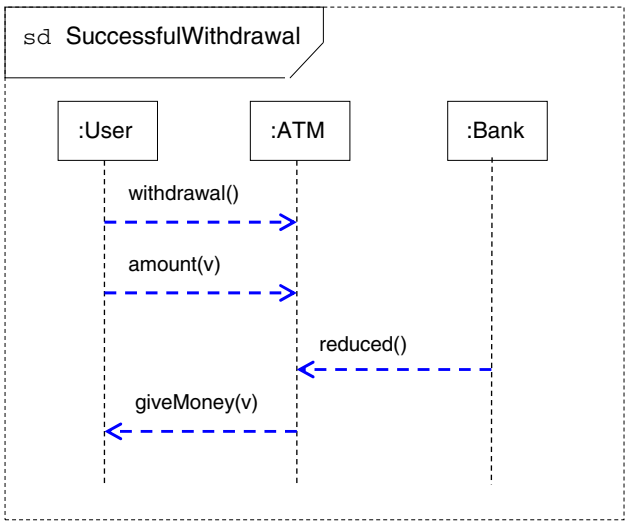

Fig. 8 An existential MSD describing successful withdrawal

The main differences are that the only accepting state is the one representing the maximal cut, and that the initial state's self transition labeled Σ is quantified existentially (as a non-deterministic choice) with all the minimal occurrence specifications, because the automaton has to guess when a satisfying run starts (see Appendix B). A system-model satisfies an existential diagram if it has at least one run that is accepted by the diagram's automaton. Figures 8 and 9 show an example existential MSD of a successful withdrawal scenario and its automaton, respectively.

3.2 Universal and existential constraints

A UML 2.0 *StateInvariant* specifies a runtime constraint on the participants of the interaction, and corresponds to the condition construct of MSC and LSC. Constraints are Boolean expressions over attributes associated with the participants in the enclosing interaction. In order to allow expressions that involve attributes from more than one participant, we propose to extend the abstract syntax definition given in the standard [34, p. 487] so that *StateInvariant* may cover more than one

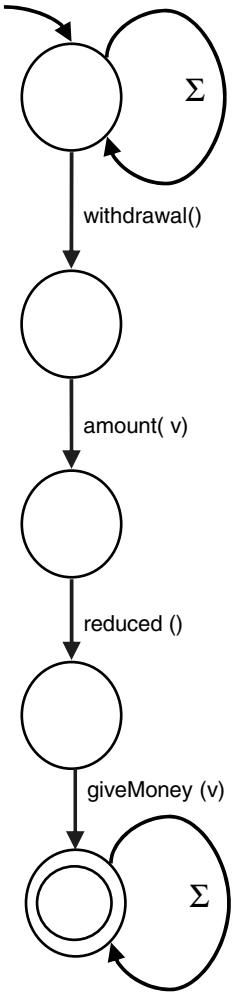

Fig. 9 The automaton for the existential interaction `SuccessfulWithdrawal` of Fig. 8. Self transitions labeled $\Sigma \setminus M$ omitted

lifeline from the interaction.[4] This modification may look minor but is rather important; it is necessary in order to synchronize the moment of evaluation of the constraint, specifically when it includes attributes from two or more instances, or when the moment of evaluation should depend on occurrences from two or more instances; see [16].

As can be seen in the abstract syntax of MSD (Fig. 4), a hot/cold modality is attached not only to messages (through their send and receive message occurrence specification) but also to *StateInvariants*. Semantically, a trace where the constraint of the *StateInvariant* is evaluated to FALSE is not in the trace-language of the enclosing interaction (because it cannot be completed into an accepted run). The constraint of a universal (hot) *StateInvariant* must evaluate to TRUE, because all system runs that reach it have to be in the trace-language of the interaction.

Figure 10 shows a universal MSD with two constraints. The first is existential; it does not have to be true in all runs. If it is true, and the run continues with the `reduced` message sent from the Bank to the ATM, the universal constraint that specifies that the balance has to be positive when the end of this interaction is reached. Figure 11 shows the alternating automaton for the MSD of Fig. 10.

This example also shows the importance of including in the domain of event occurrences, and hence in the trace-language of an interaction, not only messages sent and received but also all the values of attributes of participating instances.

[4] This was defined for LSC in [16], was suggested for UML 2.0 in the preliminary version of the present paper [15], and was recently also proposed by Knapp and Wuttke in [21].

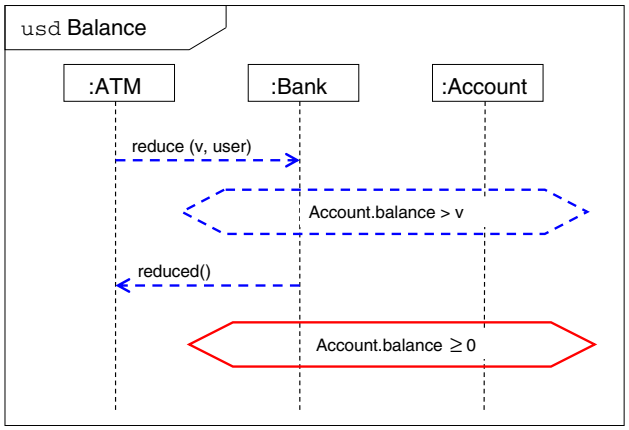

Fig. 10 A universal MSD with hot and cold conditions

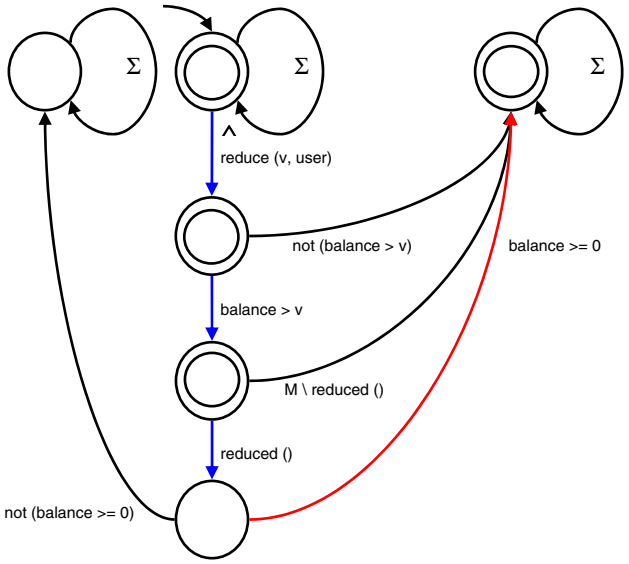

Fig. 11 The automaton for `Balance` of Fig. 10. Self transitions labeled $\Sigma \setminus M$ omitted

3.3 Specifying forbidden scenarios

Forbidden scenarios are system runs we do not want our system-model to exhibit. Using the universal modality, our definition of MSD has already allowed defining forbidden scenarios implicitly. For example, the universal MSD of Fig. 5 specifies that a trace where the ATM received the reduced message from the bank but never gave the money to the user

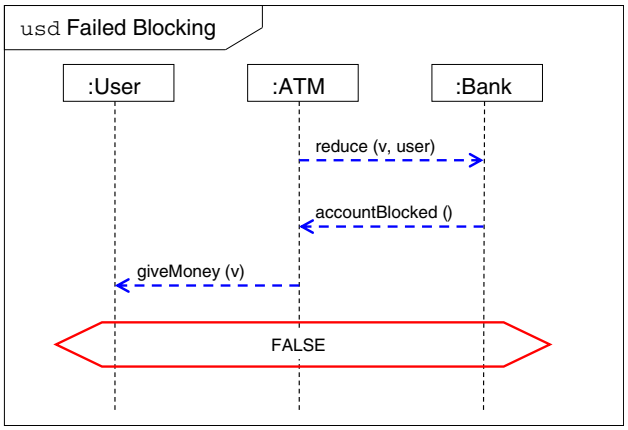

Fig. 12 Using a hot FALSE condition to specify a forbidden scenario

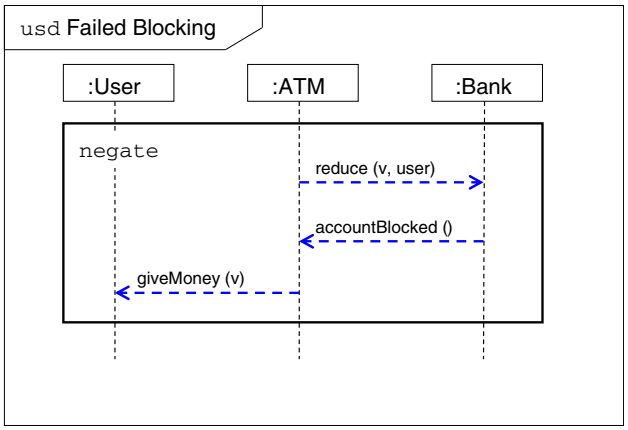

Fig. 13 Negate as syntactic sugar for a hot FALSE condition

is illegal; a system-model that can possibly exhibit it does not satisfy the MSD specification. Still, the ability to explicitly specify forbidden scenarios is very important in practice, since in many cases, specifically in formal verification, it is more natural and shorter to specify what should never happen (a safety property; a counterexample) than to specify all legal possibilities.

In MSD, specifying forbidden scenarios can be done using a universal constant FALSE constraint, just like in LSCs. Figure 12 shows an example of a universal MSD with a constant FALSE constraint; it specifies that a trace in which the bank answered the request for reduction with the message `account blocked`, but the ATM has delivered money to the user, should never happen.

To use the standard's notation, we allow the use of the *negate* operator. As in the case of *assert*, we interpret it as syntactic sugar for adding a hot constant FALSE constraint immediately after the last *OccurrenceSpecification* (actually, all the last ones) in the *InteractionFragment* operand; i.e., as a maximal element in the fragment. Figure 13 is thus equivalent to Fig. 12.

Placing the universal FALSE constraint as a maximal element following an interaction fragment means that the fragment specifies a forbidden scenario, i.e., one that the system-model is not allowed to satisfy under any set of circumstances. If the system-model satisfies the preceding fragment it must satisfy the hot FALSE constraint that follows it, which is impossible and leads to a contradiction.

3.4 Handling multiple MSDs

As with LSCs, an MSD specification typically includes many overlapping interactions, i.e., ones that have non-disjoint sets of occurrence specifications (more precisely, different occurrence specifications which reference the same events). The UML 2.0 specification document, however, does not clearly define the relationship between different sequence diagrams in creating a single specification. A naïve union of trace-languages may suffice if all the diagrams are existential, i.e., when no mandatory specifications are given, but it definitely does not suffice in the case of a general MSD specification, which may consist of overlapping existential and universal diagrams.

The trace-language of a set of universal diagrams is the intersection of the given trace-languages. This relates to the notion of specification consistency and system-model synthesis, i.e., to the issue of whether, given an MSD specification, there is a system-model that satisfies it and if there is one how can we construct it. These questions are easy to handle when all specifications are existential, but become a lot more complex (and interesting) for general universal MSDs. They were answered in detail for LSC specifications in [13,14], some of the results of which may be adapted to MSD too.

3.5 Using existential sequence diagrams

Existential MSDs that include no hot fragments have weak expressive power but are, of course, useful. First, given a set of system runs and an existential diagram, one can test whether any of the runs is in the trace-language of the diagram. Thus, existential diagrams with no hot fragments may be used for non-restrictive tests. Note that a single system run may suffice to ensure the satisfiability of an existential diagram with no hot fragments.

Second, the existential mode may be viewed as a form of under-specification of requirements, suitable for the early phases of the system's life cycle. As more information is gathered about the system, the existential specification can be refined by adding universal diagrams and hot fragments. Moving from existential to universal specifications is thus a form of refinement, orthogonal to other forms of interaction specifications refinement such as partial versus complete order and incomplete versus complete information protocols.

Finally, a specification with only existential diagrams is rather weak, as it defines only a set of example runs. On the other hand, a system-model that has no runs at all may satisfy any specification consisting of only universal diagrams (assuming no diagram contains hot minimal elements). Thus, like in LSC, the full expressive power of MSD comes from the combination of existential and universal diagrams in a single specification.

4 Advanced constructs

MSD is easily and naturally applied to interactions that include combined fragments with operators such as *loop* and *alt*. Consider for example the universal diagram shown in Fig. 14. It specifies that whenever the ATM calls the bank with a login request, the bank must try to log the user into the database. If the database is busy, it must inform the bank that it is busy, and the bank must retry to login. If the database is not busy, a cold violation occurs, and the scenario is exited (since the condition is cold, this is a legal trace; i.e., this trace is in the trace-language of the diagram; perhaps the other case, where the database is not busy, is handled in another diagram). If and when the loop is completed three times, the bank must inform the ATM to try again later. Note that to specify that a loop must be completed, one needs to assign a hot mode to all the interaction fragments inside the loop's operand. After the bank informs the ATM to try again later, the ATM may try to login again or call the bank with an offline request (the *alt* operator in this fragment does not have guards, so we do not know how the choice is made). In the latter case, the ATM must eventually also call its own method to display the offline message.

In the remainder of this section we briefly define and demonstrate how the modal extension, MSD, relates to and extends several additional language constructs from UML 2.0 sequence diagrams.

4.1 The *break* operator and nested fragments

The interaction operator *break*, defined in the standard [34, p. 454], designates that the trace of its (possibly guarded) interaction operand should be considered, instead of the remainder of the enclosing interaction fragment: A break operator with a guard is chosen when the guard is true and the rest of the enclosing interaction fragment is ignored. When the guard of the *break* operand is false, the operand is ignored and the rest of the enclosing interaction fragment is chosen.

In MSD, following the definition of LSC *subcharts* in Chapter 10 of [16], a similar specification can be defined using cold violations in nested fragments. When a cold violation occurs, either because of a violation of the partial order when the cut state is cold, or as a result of a false evaluation

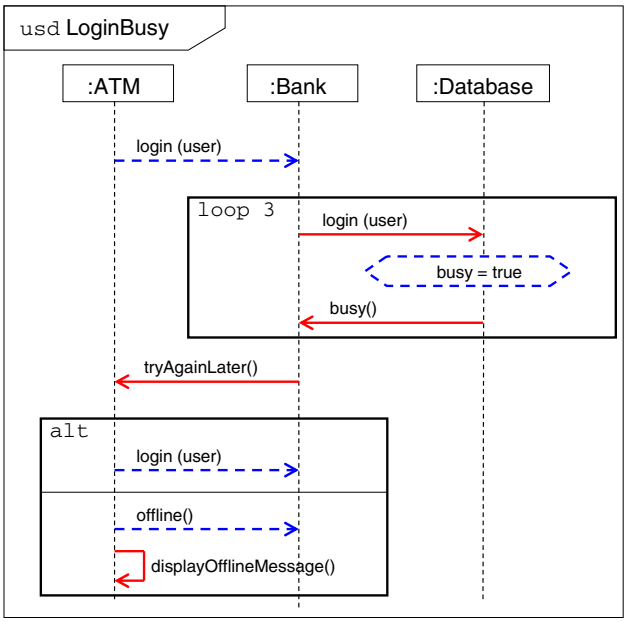

Fig. 14 Using the *loop* and *alt* operators

of an enabled cold condition, the (local) trace is accepted, the rest of the fragment is ignored, and the trace continues in the enclosing interaction fragment.

In UML, however, combined interaction fragments may be used with different operators and for different purposes. Hence, syntactic nesting of one fragment within another, e.g., in order to specify a number of guarded alternatives, does not necessarily mean that the inner fragment's trace-language is semantically nested; i.e., the scope of events occurrences considered as violating in the inner fragment may include the enclosing fragment. Therefore, we propose to explicitly designate nested fragments with a new operator *nested*. The trace-language of a nested fragment is independent of its enclosing fragment; i.e., the set of messages to consider is limited to the nested fragment and does not include messages from its enclosing fragment. Thus, as suggested above, when a cold violation occurs within a nested fragment, the (local) trace is accepted, the rest of the fragment is ignored, and the trace continues in the enclosing interaction fragment. If a hot violation occurs, however, the local trace is rejected, hence, the trace cannot be completed into a trace in the language of the enclosing fragment.

Figure 15 shows a simple example of the use of a nested fragment. Since the nested fragment does not begin with a hot element, it is essentially optional (because technically, the empty trace is in its trace-language). If the `log` message between the bank and the database does occur, and logging is enabled, the action must be logged and the database must send the message `done` to the bank. If the `log` message between the bank and the database occurs, but logging is not

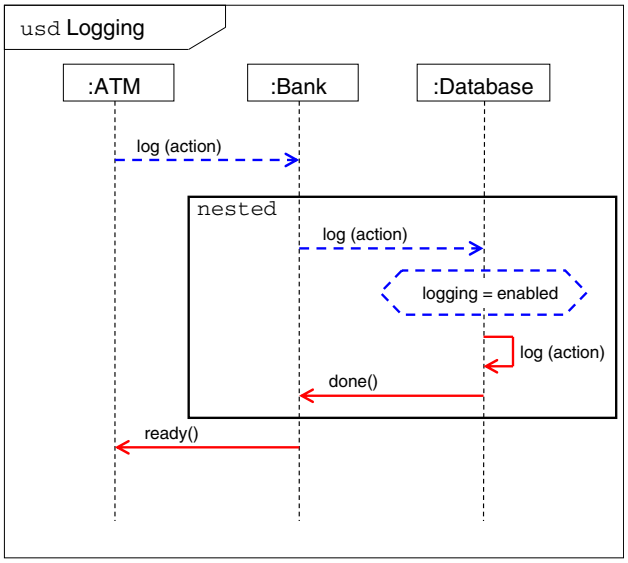

Fig. 15 Using a nested fragment

enabled, the nested fragment is exited, and the trace continues in the enclosing interaction.

4.2 The *consider* and *ignore* operators

By default, when interpreting an interaction, we consider exactly those occurrence specifications that are given in it explicitly. An interaction specification constrains the partial order only between occurrences that it contains, because, intuitively, a message or a constraint that does not appear explicitly in the specification is irrelevant to its trace-language and its satisfaction relation. In some cases, however, it is useful to be able to consider messages that do not appear in an interaction. An occurrence of such a message in a system-model run affects the trace-language of the interaction, because it is considered a violation of the partial order induced by the specified interaction.

The operator *consider* (and, respectively, its dual *ignore*), allows one to explicitly specify messages that should be considered (resp., ignored) although they do not appear (resp., do appear) in the interaction operand. The two operators are defined in the UML 2.0 standard specification [34, p. 458], where they require the designer to explicitly list all the messages that need to be considered (resp., ignored) within the interaction fragment operand. Adding more occurrences to the considered set results in more restrictive specifications: to be included in the trace-language of the diagram, a run not only needs to exhibit the occurrence specified in a matching order but also needs to not exhibit any of the additional occurrences in between. The dual holds for *ignore*: adding occurrences to the ignore set results in more permissive specifications.

The idea of adding a designated set of messages that are not allowed to occur anywhere except if specified explicitly in the chart, even if they do not appear in the chart, was considered for LSC in [13, p. 17]. It was later generalized in the notion of LSC *forbidden elements*, in Chapter 17 of [16].

In MSD, we generalize the use of the *consider* and *ignore* operators, taking advantage of its modal semantics, ideas from LSC forbidden elements [16], and the use of interaction fragments in the standard.

MSD extends the abstract syntax of *consider* and *ignore* given in the standard in two ways. First, we allow the use of the wild-card '$*$'. Thus, *consider* $*$ specifies that all system-model messages between the participants of this interaction fragment should be considered, i.e., that the traces explicitly specified in the fragment are complete.

Second, and more interestingly, instead of specifying a set of messages to be considered or ignored, we allow one to specify a set of modal interaction fragments. A successful completion of a considered fragment from this set, affects the trace-language of the enclosing combined fragment: completion of a hot considered fragment prevents the resulting trace from being included in the trace-language of the enclosing combined fragment; completion of a cold considered fragment does not prevent the resulting trace from being included in the trace-language of the enclosing combined fragment; instead, it results in a cold violation and the trace is allowed to continue in the enclosing combined fragment.

As to notation, one can use the notation suggested in the standard, optionally replacing the message names in the list with interaction names (as in *InteractionUse*). Alternatively, the considered fragments can appear explicitly as additional interaction operands after the first (main) operand of the combined fragment.

Note that our extension does not require a change in the abstract syntax of the standard, as *consider* and *ignore* are already operators of a combined fragment, which has a set of fragments as operands. We do, however, restrict each of the considered fragments to be either hot or cold; i.e., we do not allow alternation of modes.

Figure 16 demonstrates the use of the extended *consider* operator of MSD. It specifies that an occurrence of the message `cancel` between the user and the ATM during the execution of the first (main) operand, i.e., between the call to the message `amount(v)` and the receipt of the message `reduced`, is a cold violation; that the card status must equal 'inside' throughout the execution of the main operand; and that the bank is allowed to send the message `startBackup` to the ATM during the execution of the main operand, but then the ATM should not reply with the message `backupStarted`. The reason for the latter is that (parallel, interleaving) successful completion of this trace will constitute a hot violation and will prevent the resulting trace from being included in the trace-language of the interaction.

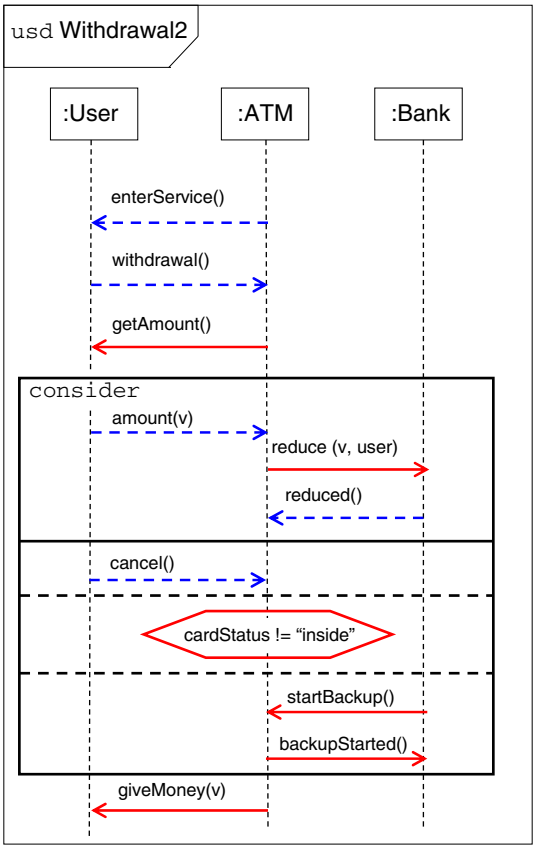

Fig. 16 Using the modal *consider* operator

Note that in this example the considered combined fragment is *not* explicitly nested. Therefore, if a cold violation occurs inside it, the resulting trace is accepted by the enclosing interaction and the rest of it is ignored. Note also that we still allow the use of the original *consider* and *ignore* operators; messages listed under *consider* (resp. *ignore*) are removed from (resp. added to) the self transition labeled $\Sigma \setminus M$ on the cut-states within the *consider* (resp. *ignore*) main operand.

Thus, using the MSD extended *consider* operator with *StateInvariants* (conditions), one can specify invariants that may/must hold during the entire execution of a fragment. The use of interaction fragments as operands for the *consider CombinedFragment*, together with the general modal semantics we provide for MSD, significantly increases the expressive power of the language and allows one more compact and intuitive specification of complex behavior.

5 Related work

In [32,33], Störrle presents the semantic problems around *assert* and *negate* as they are defined in the standard specifications. He discusses alternative interpretations for *assert* and *negate*, and shows the difficulty of finding one that is consistent both with the explanations given in the standard and with the intuitive intended meaning of required and forbidden behaviors.

Cavarra and Filipe [4] compare LSC and UML 2.0 Sequence Diagrams and nicely identify the limitations of the latter in adequately distinguishing possible from mandatory behavior. They suggest a solution in the form of an *after-eventuality* OCL template which can be applied at the local level of a message or at the more global level of an interaction. The after-eventuality template can be viewed essentially as a textual representation of a universal LSC. This solution, however, keeps the important liveness requirements within textual logical formulas outside the visual syntax of the modeling language. If the extensive use of textual logical formulas outside the diagrams is necessary, then one could use other appropriately expressive formalisms (e.g., temporal logic) to specify a system's behavior. The challenge we try to meet in our own work is to formally define a scenario-based specification language that is sufficiently expressive yet intuitive and visually appealing.

In [11], it is shown how to derive liveness and safety automata from UML 2.0 Sequence Diagrams, but that paper employs a different interpretation of the standard semantics, with no reference to the *assert* operator. Validity is interpreted as "liveness", as in *"each finite execution should be extendible to an execution where the positive trace eventually happens"* [11, p. 5]. This interpretation ignores the standard's *assert* operator, and does not allow one to specify possible traces, but only ones that must always be possible. This would appear to be a rather restrictive interpretation of the UML 2.0 standard.

STAIRS [17,30] is a requirements specification methodology based on UML 2.0, where the semantics of interactions is given using *interaction obligations*, which are pairs of sets of traces categorized as *positive* and *negative*. Traces not defined as positive or negative are called *inconclusive*. STAIRS deals with mandatory behavior using an external mandatory choice operator *xalt*, which does not appear in the UML 2.0 standard. Roughly, *xalt* means "for each operand one of the alternatives must be possible". Although *xalt* deals with mandatory behavior, it should not be confused with the universal modality adapted in MSD from LSC. The *xalt* operator specifies alternative traces that must be present in an implementation, i.e., must be possible, while assigning a universal modality to a trace specifies that it must eventually occur. In addition, and more relevant to the present paper, in STAIRS, the *assert* construct makes all inconclusive traces negative. This, too, is different from the universal modality of MSD. First, as defined in [30], STAIRS' *assert* makes *all* inconclusive traces negative, i.e., including all well-formed traces. In contrast, in MSD, the scope of *assert* is deliberately limited to the set of messages that appear in the diagram; i.e., a diagram does not

restrict the order between messages not appearing in it (to change this, one can use the *consider* and *ignore* operators). Second, and more importantly, STAIRS' notion of correct implementation of a specification is different from the MSD definition of when a system-model satisfies a specification. In STAIRS, a correct implementation *"may only produce traces belonging to the positive and inconclusive trace sets of the obligation, i.e. no negative trace must be produced by the implementation"* [30, p. 31]. Thus, STAIRS rules out behavior that is actually forbidden, but does not require the implementation of the positive traces. Third, the universal modality of MSD applies not only to *all* system-model traces, but also *globally* along each trace; i.e., *whenever* a cold prefix is successfully completed the following hot fragment must be successfully completed. As a very basic example, the requirement that "every 'request issuance' message is eventually followed by a 'grant' message" is very easily and naturally specified in MSD; it seems, however, that it is either impossible, or at least not easy, to specify this in STAIRS.

Finally, STAIRS includes an incremental development methodology for interactions, based on three types of specification refinement: supplementing, narrowing, and detailing. In Sect. 3.5 we briefly mentioned a possible incremental development process, which suggests to start with existential interactions and gradually add universal diagrams and hot fragments. This process can be viewed as a form of narrowing as it is defined in STAIRS.

In [17], it is suggested that the language of LSC could become in the future a profile of UML 2.0, and our definition of MSD indeed goes in that direction.

6 Discussion

Many authors have pointed to ambiguities and other semantic issues in the UML standard, not only in its previous versions but also in its most recent version UML 2.0 [34]. Many of these occur in the semantics of state machines; see, e.g., [9]. In Sect. 2.3, we discussed what we consider to be the most significant problem with the standard's section on sequence diagrams, i.e., the definitions of *assert* and *negate* and the semantics of valid traces. We designed MSD to address this. We have also encountered several additional problems in the standard's section on sequence diagrams, but they are outside the main topic of the present paper. We plan to describe these in detail in a follow-up paper or technical report. We note, however, that the problems we have found include, among others, missing constraints on the way the partial order is defined, and a problematic definition of what constitutes a basic element of a trace (are *StateInvariants* or *InteractionConstraints* part of a trace or not, and when should they be evaluated?). We acknowledge that the UML standard is in general not intended to be a formal specification in the full sense of the term. However, we consider this fact to add to the responsibility of people working on implementing the standard to bring such issues to the surface, to resolve them and to help make the standard precise, as we have tried to do in the present paper.

We have presented MSD, a modal extension to UML 2.0 Sequence Diagrams, based on the language of Live Sequence Charts. Roughly, MSD and the standard semantics for UML 2.0 agree on the trace-language of the existential (so called 'positive') fragment, without *assert* and *negate*. The difference stems from the addition of universal diagrams and hot interaction fragments, and how these change the semantics of the interaction when relating it to a system-model. Thus, we claim that increasing the expressive power of sequence diagrams to allow liveness and safety specifications requires a universal interpretation: *assert* and *negate* are thus to be considered as modalities, not as operators.

Historically, the LSC language and UML 2.0 Sequence Diagrams are both descendants of MSC [18]. MSDs extend UML Sequence Diagrams roughly in the same way that LSCs extended MSCs in [7]. We have designed MSD to bring the essence of LSC into the UML, but we have not included in this paper's presentation of MSD several important additions to the LSC language, most notably symbolic messages and instances [16,27]. A treatment of symbolic messages and instances in the context of object oriented inheritance and interface implementation has been done for MSD during the work on the S2A compiler [12,26] (see below) and will be formally reported in a future paper.

In this paper we mainly considered synchronous messages and strict sequencing, and did not explicitly address the semantics and use of MSD in asynchronous settings. This may also be a topic for future work.

Another topic not included here is the operational semantics of LSCs; i.e., the play-out execution mechanism of [16] and its implementation in the Play-Engine tool. Applying the play-out execution mechanism to MSD is an important topic we are currently pursuing; among other issues, it requires the definition of an additional pair of modalities: monitoring vs. executing. Indeed, recently, in [12,26], we presented a compilation scheme from MSD, extended with the monitoring and executing modalities, into AspectJ, whereby the resulting code implements the play-out execution mechanism. There, most of the ideas and techniques developed in [16] for LSCs are adapted and used in a relatively obvious way for MSDs. The same can be applied to the play-in method.

Our definition of MSD in the present paper precludes nesting of hot fragments inside existential diagrams. Indeed, following the stated justification given in [7], we judge that such nested quantification is too complex for real world usage of sequence diagrams (in particular, if hot fragments inside existential diagrams are allowed, a single system run may no longer suffice to ensure the satisfiability of an existential

diagram). Still, one could extend MSD to allow such nesting, change the semantics accordingly, and thus obtain a more expressive formalism.

We suggest that the MSD profile be used effectively, together with the standard UML Testing Profile [35], in specifying behaviors for test cases. We believe there is much benefit and no real difficulty in such integration. Formalizing and implementing this is a topic for future work.

Based on the modal semantics of MSDs, we feel that *assert* and *negate* can be used effectively, not only in a strict formal design context, but also in early requirements specifications, as powerful yet convenient and intuitive features of UML 2.0 Sequence Diagrams. In addition, given the fact that MSD is defined as a UML profile, our work paves the way to apply recent work on LSC in formal verification [8,19,31,36] to the UML. As a first step, one can formalize and implement a translation from MSD to Temporal Logic, similar to the one suggested for LSC in [22].

Finally, we claim that the problem with UML 2.0 Sequence Diagrams with regard to the semantics of sets of valid and invalid traces and the use of *assert* and *negate*, as it is currently defined in the standard, is indeed significant, and renders the use of this UML language in formal contexts highly problematic. Thus, we propose that MSD and its semantics (or some similar variant that solves these problems) may be adopted and integrated into the standard itself in a future version. We encourage systems designers to download the MSD profile (see Appendix A), apply it to their models, and take advantage of its expressiveness and robustness.

Acknowledgments We would like to thank Orna Kupferman for her comments, and Eran Gery for commenting on an earlier draft of the paper. We thank Asaf Kleinbort for implementing the MSD profile in the Eclipse UML2 environment and for proofreading the Appendices. We also thank the anonymous reviewers for helpful comments.

A Appendix: The MSD profile

We give technical details regarding the MSD profile. An XMI file for the MSD profile, compliant with UML 2.1 and most tools, and some related resources, can be downloaded from the second listed author's website.[5] In its most basic form, the MSD profile contains a single *enumeration InteractionMode* with two *enumeration literals*: *hot* and *cold*, and a single *stereotype modal*, with an attribute *interactionMode* of type *InteractionMode* (see Fig. 3). The *modal* stereotype is introduced as an extension of the abstract class *InteractionFragment*, and thus, of its subclasses *Interaction*, *OccurrenceSpecification*, *CombinedFragment*, and *StateInvariant* (see Fig. 4). *Hot* and *cold Interactions* are called *Universal* and *Existential* respectively.

[5] http://www.wisdom.weizmann.ac.il/~maozs.

Technically, the modal stereotype is applied to messages too. The *interactionMode* of a *message* is derived from the modes of the message's send and receive *MessageOccurrenceSpecifications*. In general, a *message* is *hot* if at least one of its ends is *hot*, and is *cold* otherwise.

We add a constraint to forbid nesting of *hot* elements inside *existential* interactions (see Sect. 6). Also, depending on the application, one may consider additional syntactic constraints. For example, consider a constraint that requires both the send and receive *MessageEnds* of any message to have the same mode. Such constraints decrease the expressive power of the language but may be appropriate for certain applications.

Finally, the profile's notation is adopted from LSC. *Universal* interactions are annotated by a solid borderline and *existential* interactions by a dashed borderline (the same method can be used to distinguish *hot* elements (messages, constraints, etc.) from *cold* ones). *Hot* elements are colored in red and *cold* elements in blue. In addition, we suggest to change the keyword on the top left corner of universal diagrams from `sd` to `usd`.

B Appendix: Outline of MSD formal semantics

In the following we give a technical outline of MSD formal semantics: constructing the automata for existential and universal MSDs and relating an MSD specification to a system-model. For simplicity, we consider here only *Messages* and *StateInvariants* (we also treat here message send and receive as a single event). Adding *InteractionFragments* with *InteractionOperators* such as *alt* and *loop*, does not change the essence of the construction. We consider strict sequencing only.

The *trace-language* of an MSD D is the word language $L(D)$ accepted by its automaton. We let Σ be the alphabet of the system-model messages. Given an MSD D, the construction of its automaton is based on an *unwinding structure* (see, e.g., [20]) that includes a set of event occurrences $E = E_m \cup E_c$ where E_m are message events and E_c are condition events, a set of cut-states S, a partial function $R : S \times E \longrightarrow S$, minimal and maximal cut-states s_{min} and s_{max}, a labeling function $l : E_m \longrightarrow M$, where $M \subseteq \Sigma$ is the set of messages appearing in D, and a labeling function $c : E_c \longrightarrow C$, where C is the set of conditions appearing in D. The set of *enabled message events* in a cut s is defined by $EME(s) = \{e \in E_m | \exists s' \in S : R(s, e) = s'\}$. The set of *enabled messages* in a cut s is defined by $EM(s) = \{m \in M | \exists e \in EME(s) : l(e) = m\}$. The set of *violating messages* in a cut s is defined by $VM(s) = \{m | m \in M \setminus EM(s)\}$. The set of *enabled condition events* in a cut s is defined by $ECE(s) = \{e \in E_c | \exists s' \in S : R(s, e) = s'\}$. The set of

enabled conditions in a cut s is defined by $EC(s) = \{cond \in C | \exists e \in ECE(s) : c(e) = cond\}$.

In addition, the mapping $mode : E \longrightarrow \{cold, hot\}$ is extended to cut-states: $mode : S \longrightarrow \{cold, hot\}$ where $mode(s) = hot$ if $\exists e : e \in EME(s) \cup ECE(s) \wedge mode(e) = hot$; otherwise $mode(s) = cold$.

The automata are defined over the alphabet $\Sigma \cup \epsilon$. We use ϵ to denote the no-op message. When annotated with a guard from $\{cond | cond \in C\} \cup \{not(cond) | cond \in C\}$, it represents conditions appearing in the traces.

B.1 Existential MSD

For an existential MSD, we construct a non-deterministic Büchi automaton $A = \langle \Sigma \cup \epsilon, Q, q_{in}, \delta, \alpha \rangle$, where $Q = S \cup \{q_{rej}\}$ is a finite set of states, $q_{in} = s_{min}$ is the initial state, the accepting condition is $\alpha = q_{max} (= s_{max})$, and the transition function $\delta : Q \times \Sigma \cup \epsilon \longrightarrow 2^Q$ is defined as follows:

- Σ labeled self transitions on q_{max} and q_{rej}:
  - $\forall m \in \Sigma : \delta(q_{max}, m) = \{q_{max}\}, \delta(q_{rej}, m) = \{q_{rej}\}$
- $\Sigma \setminus M$ labeled self transitions on all cut-states:
  - $\forall q \in S, \forall m \in \Sigma \setminus M : \delta(q, m) = \{q\}$
- Handling enabled messages:
  - $\forall q \in S \setminus \{q_{in}\}, \forall m \in EM(q) : \delta(q, m) = \{R(q, e) | e \in EME(q) \wedge l(e) = m\}$
  - $\forall m \in EM(q_{in}) : \delta(q_{in}, m) = \{R(q_{in}, e) | e \in EME(q_{in}) \wedge l(e) = m\} \cup \{q_{in}\}$
- Handling violating messages:
  - $\forall q \in S \setminus \{q_{in}\}, \forall m \in VM(q) : \delta(q, m) = \{q_{rej}\}$
  - $\forall m \in VM(q_{in}) : \delta(q_{in}, m) = \{q_{in}\}$
- Handling conditions:
  - $\forall q \in S \setminus \{q_{in}\}, \forall cond \in EC(q) :$ $\delta(q, \epsilon[cond]) = \{R(q, e) | e \in ECE(q) \wedge c(e) = cond\}$, $\delta(q, \epsilon[not(cond)]) = \{q_{rej}\}$
  - $\forall cond \in EC(q_{in}) : \delta(q_{in}, \epsilon[cond]) =$ $\{R(q_{in}, e) | e \in ECE(q_{in}) \wedge c(e) = cond\} \cup \{q_{in}\}$, $\delta(q_{in}, \epsilon[not(cond)]) = \{q_{in}\}$

B.2 Universal MSD

For a universal MSD, the same unwinding structure is used. Recall that in an alternating automaton the transition function is defined as $\delta : Q \times \Sigma \longrightarrow B^+(Q)$ where $B^+(Q)$ is the set of positive Boolean formulas over Q (see, e.g.,[23]). We construct an alternating Büchi automaton $A = \langle \Sigma \cup \epsilon, Q, q_{in}, \delta, \alpha \rangle$, where $Q = S \cup \{q_{rej}, q_{acc}\}$ is a finite set of states, $q_{in} = s_{min}$ is the initial state, the accepting condition is $\alpha = \{s | mode(s) = cold\} \cup \{q_{acc}\}$, and the transition function δ is defined as follows:

- Σ labeled self transitions on q_{acc} and q_{rej}:
  - $\forall m \in \Sigma : \delta(q_{acc}, m) = q_{acc}, \delta(q_{rej}, m) = q_{rej}$
- $\Sigma \setminus M$ labeled self transitions on all cut-states:
  - $\forall q \in S, \forall m \in \Sigma \setminus M : \delta(q, m) = q$
- Handling enabled messages:
  - $\forall q \in S \setminus \{q_{in}\}, \forall m \in EM(q) : \delta(q, m) = R(q, e)$, where $e \in EME(q) \wedge l(e) = m$
  - $\forall m \in EM(q_{in}) : \delta(q_{in}, m) = q_{in} \wedge R(q_{in}, e)$, where $e \in EME(q_{in}) \wedge l(e) = m$
- Handling violating messages:
  - $\forall q \in S \setminus \{q_{in}\}, \forall m \in VM(q) :$ if $mode(q) = cold$ then $\delta(q, m) = q_{acc}$, if $mode(q) = hot$ then $\delta(q, m) = q_{rej}$
  - $\forall m \in VM(q_{in}) :$ if $mode(q_{in}) = cold$ then $\delta(q_{in}, m) = q_{acc} \wedge q_{in}$, if $mode(q_{in}) = hot$ then $\delta(q_{in}, m) = q_{rej}$
- Handling conditions:
  - $\forall q \in S \setminus \{q_{in}\}, \forall cond \in EC(q) : \delta(q, \epsilon[cond]) = R(q, e)$, where $e \in ECE(q) \wedge c(e) = cond$; if $mode(q) = cold$ then $\delta(q, \epsilon[not(cond)]) = q_{acc}$; if $mode(q) = hot$ then $\delta(q, \epsilon[not(cond)]) = q_{rej}$
  - $\forall cond \in EC(q_{in}) : \delta(q_{in}, \epsilon[cond]) =$ $R(q_{in}, e) \wedge q_{in}$, where $e \in ECE(q_{in}) \wedge c(e) = cond$; if $mode(q_{in}) = cold$ then $\delta(q_{in}, \epsilon[not(cond)]) = q_{acc} \wedge q_{in}$; if $mode(q_{in}) = hot$ then $\delta(q_{in}, \epsilon[not(cond)]) = q_{rej}$

B.2.1 Improving the construction

The above construction of alternating Büchi automaton suffices to define the trace-language accepted by a universal MSD. In general, the use of alternation is beneficial as it allows easy complementation (see [28]). However, one may easily further improve the construction so that the resulting automaton is a weak alternating automaton. This has advantages in the context of formal model-checking (see [24]). Below we give an outline of the weak construction.

Recall that in a weak alternating automaton, the set of states Q is partitioned into partially ordered sets, each of which is classified as accepting or rejecting. The transition function is restricted so that in each transition the automaton either stays at the same set or moves to a set smaller in the partial order. Thus, each run eventually gets trapped in some set in the partition. Acceptance is then determined by the classification of the set.

Note that the partial order semantics of sequence diagrams suffices to ensure that when no loops are allowed the simple construction defined above is already weak (this is true also for the existential case). To keep the constructed automaton weak in the presence of loops (constant and $*$), we make the following changes:

- For each cold state q_{cold}, a new accepting state q'_{cold} is added to Q (we denote the set of new states PC):
  - $Q = S \cup \{q_{rej}, q_{acc}\} \cup PC$

- $\alpha = \{q_{acc}\} \cup PC$ (i.e., the original cold states are no longer accepting)

- The transition function for each new accepting state $q' \in PC$ is defined as follows:
  - $\forall m \in \Sigma \setminus M : \delta(q', m) = q'$
  - $\forall m \in M : \delta(q', m) = q_{rej}$
  - $\forall cond \in EC(q_{cold})$, where q_{cold} is the original cold cut-state corresponding to q', $\delta(q', \epsilon[cond]) = q_{rej}$, $\delta(q', \epsilon[not(cond)]) = q'$
- The transition function for $\Sigma \setminus M$ on all cut-states is redefined as follows:
  - $\forall q \in S, \forall m \in \Sigma \setminus M$: if $mode(q) = cold$ then $\delta(q, m) = q \vee q'$, where $q' \in PC$ is the new accepting state corresponding to q; if $mode(q) = hot$ then $\delta(q, m) = q$

Note that the constructed alternating weak word automaton is very simple. Specifically, its accepting sets are singletons. This may be an advantage in any application that needs to translate the constructed alternating automaton into a non-deterministic automaton (see, e.g., [10]).

B.3 Relating an MSD specification to a system-model

Recall that the *trace-language* of an MSD D is the word language $L(D)$ accepted by its automaton. An *MSD specification* is a set $Spec = Existential \cup Universal$, where $Existential$ and $Universal$ are sets of existential and universal diagrams, respectively. We denote the runs of a system-model Sys by L_{Sys}. We assume system-model runs include values of participating objects' attributes so that conditions can be evaluated.

Finally, a system-model Sys *satisfies* an MSD specification $Spec = Existential \cup Universal$ iff

- $\forall D \in Universal, \forall r \in L_{Sys} : r \in L(D)$
- $\forall D \in Existential, \exists r \in L_{Sys} : r \in L(D)$

References

1. Bontemps, Y., Heymans, P.: Turning high-level sequence charts into automata. In: Proceedings of the 1st international workshop on scenarios and state machines (SCESM'02), at the 24th international conference on software engineering (ICSE'02) (2002)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The unified modeling language user guide, 2nd edn. Addison Wesley, Reading (2005)
3. Bunker, A., Gopalakrishnan, G., Slind, K.: Live sequence charts applied to hardware requirements specification and verification: a VCI bus interface model. Software Tools Technol. Trans. **7**(4), 341–350 (2005)
4. Cavarra, A., Filipe, J.K.: Combining sequence diagrams and OCL for liveness. Electr. Notes Theor. Comput. Sci. **115**, 19–38 (2005)
5. Cengarle, M., Knapp, A.: UML 2.0 Interactions: semantics and refinement. In: Jürjens, J., Fernández, E.B., France, R., Rumpe, B. (eds.) 3rd International workshop on critical systems development with UML (CSDUML'04), pp. 85–99 (2004)
6. Combes, P., Harel, D., Kugler, H.: Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool, LNCS, vol. 3707, pp. 414–428 (2005)
7. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. J. Formal Methods Syst. Des. **19**(1), 45–80 (2001). Preliminary version in: Ciancarini, P., Fantechi, A., Gorrieri, R. (eds.) Proceedings of the 3rd IFIP international conference on formal methods for open object-based distributed systems (FMOODS'99), pp. 293–312, Kluwer Academic Publishers, Dordrecht (1999)
8. Damm, W., Westphal, B.: Live and let die: LSC-based verification of UML-models. Sci. Comput. Program. **55**(1–3), 117–159 (2005)
9. 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.) Proceedings of the 7th international conference on formal engineering methods (ICFEM'05), LNCS, vol. 3785, pp. 52–65 (2005)
10. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings 13th International Conference on computer aided verification (CAV'01), LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
11. 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'05), pp. 6–14. IEEE Computer Society (2005)
12. Harel, D., Kleinbrot, A., Maoz, S.: S2A: A compiler for multimodal UML sequence diagrams. In: Proceedings of the 10th international conference on fundamental approaches to software engineering (FASE'07), LNCS, vol. 4422, pp. 121–124, Springer, Heidelberg (2007)
13. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Int. J. Foundations Comput. Sci. **13**(1), 5–51 (2002)
14. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenario-based requirements. LNCS, vol. 3393, pp. 309–324 (2005)
15. Harel, D., Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. In: Proceedings of the 5th international workshop on scenarios and state machines (SCESM'06), at the 28th international conference on software engineering (ICSE'06), pp. 13–20, ACM Press, New York (2006)
16. Harel, D., Marelly R.: Come, let's play: scenario-based programming using LSCs and the play-engine. Springer, Heidelberg (2003)
17. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software Syst. Model. (SoSyM) **4**(4), 355–367 (2005)
18. ITU: International telecommunication union recommendation z.120: Message sequence charts. Tech. rep. (1996)
19. Klose, J., Toben T., Westphal, B., Wittke, H.: Check it out: on the efficient formal verification of live sequence charts. In: Ball, T., Jones, R.B. (eds.) Proceedings 18th international conference on computer aided verification (CAV'06), LNCS, vol. 4144, pp. 219–233, Springer, Heidelberg (2006)
20. Klose, J., Wittke, H.: An automata based interpretation of live sequence chart. In: Margaria, T., Yi, W. (eds.) Proceedings 7th international conference on tools and algorithms for the construction and analysis of systems (TACAS'01), LNCS, vol. 2031. Springer, Heidelberg (2001)
21. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Houmb, S.H., Georg, G., France, R., Petriu, D.C., Jürjens, J. (eds.) Proceedings of the international workshop on critical systems development using modeling languages (CSDUML'06), pp. 52–67 (2006)

22. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems (TACAS'05), LNCS, vol. 3440, pp. 445–460. Springer, Heidelberg (2005)
23. Kupferman, O., Vardi, M.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001)
24. Kupferman, O., Vardi, M., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
25. Lettrari, M., Klose, J.: Scenario-based monitoring and testing of real-time UML models. In: Gogolla, M., Kobryn, C. (eds.) UML, LNCS, vol. 2185, pp. 317–328. Springer, Heidelberg (2001)
26. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering (SIGSOFT'06/FSE-14), pp. 219–230. ACM Press, New York (2006)
27. Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Proceedings of the international conference on object-oriented programming, languages, and applications (OOPSLA'02)
28. Miyano, S., Hayashi, T.: Alternating finite automata on ω-Words. Theor. Comp. Sci. **32**, 321–330 (1984)
29. Muller, D.E., Saoudi, A., Schupp, P.E.: Alternating automata, the weak monadic theory of trees and its complexity. Theor. Comput. Sci. **97**(2), 233–244 (1992)
30. Runde, R.K., Haugen, Ø., Stølen, K.: Refining UML interactions with underspecification and nondeterminism. Nordic J. Comput **12**(2), 157–188 (2005)
31. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The rhapsody UML verification environment. In: Cuellar, J.R., Liu, Z. (eds.) Proceedings of the 2nd international conference on software engineering and formal methods (SEFM'04), Beijing, China, pp. 174–183. IEEE (2004)
32. Störrle, H.: Assert, negate and refinement in UML-2.0 interactions. In: Jürjens, J., Fernández, E.B., France, R., Rumpe, B. (eds.) Proceedings of the 3rd international workshop on critical systems development with the UML (CSDUML'04), pp. 79–94 (2004)
33. Störrle, H.: Trace semantics of UML 2.0 interactions. Tech. rep., University of Munich (2004)
34. UML: Unified modeling language superstructure specification, v2.0, formal/05-07-04. OMG specification, OMG (August 2005)
35. UML: unified modeling language testing profile, v1.0. OMG specification, OMG (July 2005)
36. Westphal, B.: LSC verification for UML models with unbounded creation and destruction. In: Cook, B., Stoller, S., Visser, W. (eds.) Proceedins workshop on software model checking (SoftMC'05), ENTCS, vol. 144, pp. 133–145. Elsevier, Dordrecht (2005)

Author's Biography

David Harel has been at the Weizmann Institute of Science in Israel since 1980. He was Department Head from 1989 to 1995, and was Dean of the Faculty of Mathematics and Computer Science between 1998 and 2004. He was also co-founder of I-Logix, Inc. He received his PhD from MIT in 1978, and has spent time at IBM Yorktown Heights, and sabbaticals at Carnegie-Mellon University, Cornell University and the University of Edinburgh. In the past he worked mainly in theoretical computer science (logic, computability, automata, database theory), and now he works in software and systems engineering, modeling biological systems, and the synthesis and communication of smell. He is the inventor of statecharts and co-inventor of live sequence charts, and co-designed Statemate, Rhapsody and the Play-Engine. He received the ACM Karlstrom Outstanding Educator Award (1992), the Israel Prize in Computer Science (2004), the ACM SIGSOFT Outstanding Research Award (2006), and three honorary degrees. He is a Fellow of the ACM and of the IEEE.

Shahar Maoz is studying towards a Ph.D. in Computer Science at the Weizmann Institute of Science in Israel. He received his M.Sc. in Computer Science from Tel-Aviv University. His M.Sc. thesis was in the area of Temporal Logics. Over the years he worked in industry as a programmer, a team leader and a program manager in various kinds of systems, including educational multi-media games and web-based knowledge management systems. His current research interests are in visual formalisms for software and systems modeling, semantics, and aspect-oriented software development.