

Polymorphic scenario-based specification models: semantics and applications

Shahar Maoz

Received: 9 February 2010 / Revised: 20 June 2010 / Accepted: 28 June 2010
© Springer-Verlag 2010

Abstract We present *polymorphic scenarios*, a generalization of a UML2-compliant variant of Damm and Harel's live sequence charts (LSC) in the context of object-orientation. Polymorphic scenarios are visualized using (modal) sequence diagrams where lifelines may represent classes and interfaces rather than concrete objects. Their semantics takes advantage of inheritance and interface realization to allow the specification of most expressive, succinct, and reusable universal and existential inter-object scenarios for object-oriented system models. We motivate the use of polymorphic scenarios, formally define their trace-based semantics, and present their application for scenario-based testing and execution, as implemented in the S2A compiler developed at the Weizmann Institute of Science. We further discuss advanced semantic issues arising from the use of scenarios in a polymorphic setting, suggest possible extensions, present a UML profile to support polymorphic scenarios, consider the application of the polymorphic semantics to other variants of scenario-based specification languages, and position our work in the broader context of behavioral subtyping.

Communicated by Andy Schuerr and Bran Selic.

Preliminary version appeared in MoDELS'09: Proc. ACM/IEEE 12th Int. Conf. on Model Driven Engineering Languages and Systems (October 2009) [30]. This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science. In addition, part of this research was funded by an Advanced Research Grant awarded to David Harel of the Weizmann Institute from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007–2013). Part of this work was done while the author was a post-doctoral researcher at RWTH Aachen University, Germany.

S. Maoz (✉)
The Weizmann Institute of Science, Rehovot, Israel
e-mail: shahar.maoz@weizmann.ac.il

Keywords Live sequence charts · UML interactions · Sequence diagrams · Polymorphism · Scenario-based modeling · Behavioral subtyping

1 Introduction

Scenario-based modeling, where interactions between system objects are specified using variants of sequence diagrams, has been adapted to the UML2 standard and has attracted much research efforts in recent years (see e.g., [5, 11, 17, 23, 38, 41]). Specifically, we are interested in a UML2-compliant variant of Damm and Harel's live sequence charts (LSC) [6, 15], which extends classical sequence diagrams with universal/existential and must/may modalities.

Polymorphism—the ability of a type T_1 to appear and be used like another type T_2 —is a fundamental characteristic of object-oriented design, enabling important features such as modularity and reuse. While UML class diagrams syntax includes constructs that support a polymorphic interpretation, such as inheritance and interface realization relations, a polymorphic interpretation for UML inter-object behavioral diagrams, such as sequence diagrams, seems to be missing. This limits the applicability of these diagrams to object-oriented system models.

In this paper we address this limitation by presenting *polymorphic scenarios*, as a generalization of sequence diagrams in the context of object-oriented system models. In polymorphic scenarios, sequence diagram lifelines may represent classes or interfaces rather than specific objects. Semantically, they thus apply to all objects directly or indirectly instantiated from the represented classes, or all objects realizing the represented interfaces in the model.

Combined with the expressive power of LSC, the polymorphic extension results in a powerful modeling language. A polymorphic scenario-based specification made of a set of

universal and existential scenarios is a succinct specification that entails a rather strong notion of *behavioral subtyping*: liveness and safety properties of a superclass's interaction with its environment hold for all objects directly or indirectly instantiated from it in the model. Thus, *inter-object behavior common to all objects derived directly or indirectly from a certain type, can be formally specified at the most abstract level where it is applicable, instead of being repeated for each class (or worse, for each object)*.

The polymorphic extension is independent of other semantic concerns related to sequence diagrams that are supported by UML2 interactions, such as the existential versus universal interpretations, the use of negative scenarios, strict versus weak sequencing, synchronous versus asynchronous messages, etc. Similar to the lifeline composition extension of [3], the focus of the polymorphic extension is on the relations between the lifelines that appear in the interaction and the objects in the system.

The main technical contribution of our work is in defining a semantics for a polymorphic extension of the UML2-compliant variant of LSC. Specifically, we give a trace-based semantics that generalizes the definitions given in [15] from the concrete to the polymorphic case. Technically, this is done by adding to the automata defined in [15] a dynamic (ad hoc, late) binding mechanism that supports classical object-oriented polymorphism. Moreover, following LSC, the semantics is defined not only for single diagrams, but also for *scenario-based specifications*, which include several, possibly inter-dependent universal and existential interactions. When realized in a system model, the polymorphic interpretation may result in different concrete interpretations, based on the various inter-dependencies in the specification model.

The polymorphic interpretation has far-reaching consequences on the use of scenario-based models throughout the development cycle. Specifically, we discuss its application to scenario-based testing and execution. An implementation of scenario-based testing and execution that supports the polymorphic semantics has been carried out in S2A [12,33], a compiler that translates UML2-compliant LSCs into AspectJ code. We discuss S2A and its use for polymorphic scenario-based testing and execution (play-out) in Sect. 4. Moreover, in Sect. 6 we present a UML profile—a variant of which is used by S2A—that extends the *modal* profile of [15] with the features required to support the polymorphic interpretation.

A number of advanced semantic issues and possible variations and extensions arise from our definition of the polymorphic semantics. These include, e.g., multiple binding choices, the single binding constraint, several possibilities to define existential acceptance, the distinction between flat and deep scope, and the combination of static and dynamic binding. We discuss these advanced topics in Sect. 5.

Finally, our work on polymorphic scenarios extends and generalizes the notion of symbolic lifelines presented for LSC in [35]. While much previous research discusses the semantics of MSCs or UML2 interactions (see e.g., [5,17,23,38]), we are not aware of any other work that explicitly and formally considers a polymorphic interpretation for sequence diagrams. Moreover, while our work is tightly related to the idea of behavioral subtyping—which has been studied in the past and addressed in different ways (see e.g., [9,14,27])—our work's *inter-object scenario-based approach* to behavioral subtyping seems to be unique. See Sect. 7 for a discussion of related work.

Our earlier paper [30] has introduced polymorphic scenario-based specifications, formally defined their semantics, and outlined their application in testing and execution. This paper extends our previous work by presenting the applications with an example and in more detail, by discussing some advanced semantic issues and possible extensions in depth, by suggesting a UML profile to support the polymorphic extension, and by positioning our work with regard to related literature and in the broader context of behavioral subtyping.

1.1 Paper structure

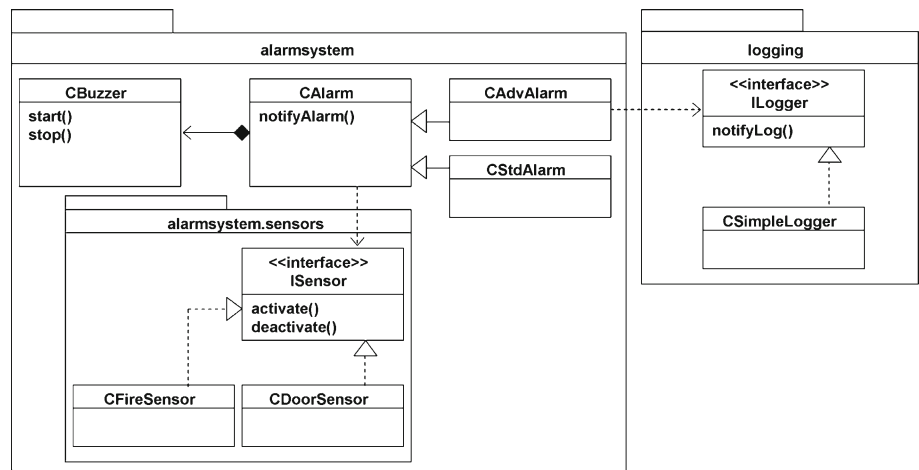
The remainder of this paper is organized as follows: Sect. 2 presents a motivating example, introducing and demonstrating the advantages and unique features of the polymorphic interpretation. Section 3 formally defines the syntax and semantics of polymorphic scenario-based specifications. A discussion of applications, specifically, polymorphic model-based testing and execution and their implementation, together with a concrete example, appears in Sect. 4. Section 5 presents some interesting, more advanced semantic issues and important extensions arising from our work. A UML profile defined to support the polymorphic extension appears in Sect. 6. Related work in the areas of scenario-based modeling and behavioral subtyping is presented in Sect. 7, together with a discussion of the challenges in applying the polymorphic semantics to other variants of scenario-based specification languages, and of the position of our work in the broader context of behavioral subtyping. Section 8 concludes and suggests future research directions.

2 Motivating example

We start off with a motivating example. The example is intentionally small and simple, to help us focus on the specific issue of interest.

Consider a model of an alarm system, made of an alarm controller, some sensors, and a buzzer. We consider a single simple use case where the alarm controller activates a sensor,

Fig. 1 The class diagram of the alarm system model



the sensor notifies the controller when it senses a movement, and the alarm controller starts the buzzer. More formally, see the class diagram shown in Fig. 1, which includes the class CBuzzer, a class CAlarm and its two sub classes CStdAlarm and CAdvAlarm, two classes CDoorSensor and CFireSensor realizing the ISensor interface, and a class CSimpleLogger realizing the interface ILogger. One difference between the standard alarm controller and the advanced one is that the latter maintains a log of alarm notifications, using a class realizing the ILogger interface.

Consider the following semi-formal behavioral specification:

- R1 Whenever an alarm controller (an object of type CAlarm) sends the message activate to a sensor (realizing the ISensor interface), and the sensor some time later sends the message notifyAlarm to the alarm controller, the controller must eventually send the message start to a buzzer.
- R2 Whenever a sensor sends the notifyAlarm message to an advanced alarm controller (an object of type CAdvAlarm), the advanced alarm controller must

eventually send the message notifyLog to a logger (implementing the ILogger interface).

- R3 The following sequence of events must be possible: an alarm controller sends the message activate to a sensor, the sensor sometime later sends the message notifyAlarm to the alarm controller, and the alarm controller sends the message activate to the sensor (again).

The above-mentioned specification is formalized in Fig. 2, which includes two universal diagrams D1 and D2 and an existential diagram D3. First, recall the universal/existential modality of LSC. Roughly, the universal diagrams specify a temporal invariant that must hold on all system runs, and from every point in a run: whenever the cold (dashed, blue) messages happen in the specified order, eventually the hot (solid, red) messages must happen in the specified order. The existential diagram specifies an example trace that must hold (that is, must happen in the specified order) in at least one point of some system run. Both diagrams, universal and existential, do not restrict any message not appearing in them to occur or not to occur during a system run.

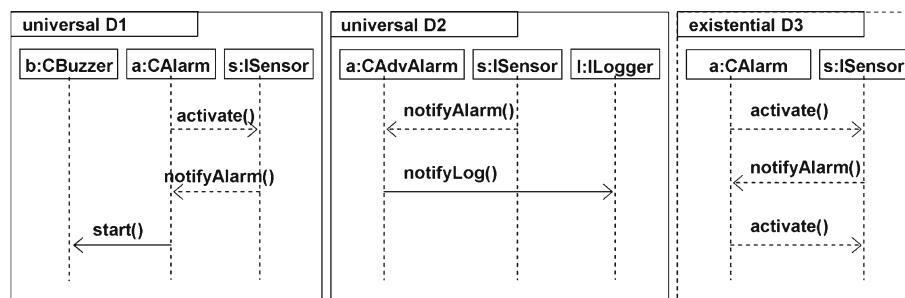


Fig. 2 A scenario-based specification model for the alarm system. The specification includes two universal diagrams, D1 and D2, and an existential diagram D3. In this example, cold messages are drawn using dashed lines and hot messages are drawn using solid lines (see [6, 15]). A lifeline is headed by its name and the type it represents. In this paper

we consider all lifelines to be polymorphic by default, so no special notation is used to mark the polymorphic features in this figure. For a formal definition of a profile and a suggested notation for polymorphic lifelines, see Sect. 6

```

tr1:
<cstdalarm,activate,fs1>
<cstdalarm,activate,ds1>
<fs1,notifyAlarm,cstdalarm>
<cstdalarm,start,cbuzzer>
<cstdalarm,activate,ds1>
...

tr2:
<cadvalarm,activate,fs3>
<cadvalarm,activate,ds1>
<fs3,notifyAlarm,cadvalarm>
<cadvalarm,notifyLog,simplelogger>
<cadvalarm,start,cbuzzer>
...

```

Fig. 3 Two excerpts from event traces of the alarm system accepted by diagram D1. Events are represented as triplets of the form (caller object id, message signature, receiver object id), and traces are infinite sequences of events (events and traces are formally defined in Sect. 3). In the figure we use abbreviated lowercase class names as object ids for objects of the corresponding class (e.g., fs3 refers to an object of the class CFireSensor)

Second, and more importantly in the context of this paper, the sequence diagrams shown in this example have a *polymorphic interpretation*. That is, their semantics, when given as a set of system-model event traces (or ‘runs’), includes events occurring on all objects derived from the referenced classes or realizing the referenced interfaces. For example, the traces tr1 and tr2 shown in Fig. 3 are both in the trace language defined by diagram D1 shown in Fig. 2.¹

Moreover, the semantics of the entire specification consisting of the three diagrams together is polymorphic: the same concrete object instance may be referred to by more than one diagram using different (ad-hoc) polymorphic bindings. For example, the behavior of an instance of the class CAdvAlarm is constrained both by D1—where it is referenced as its superclass CAlarm using implicit up-casting—and by D2—where it is represented by its direct class.

Note how the polymorphic interpretation allows us to create *succinct* specifications; inter-object behavior common to all objects derived directly or indirectly from a certain type is specified at the most abstract level where it is applicable, instead of being repeated for each class (or worse, for each object). The resulting specifications may thus also be *reusable* and applicable to other systems, e.g., where we may have different realizations of some of the same interfaces. For example, if another type of sensor is added to the system, say a CFloodSensor, or if multiple instances of a class are added, say some CDoorSensors, the scenario-based specification model need not change.

The example demonstrates the significant consequences the polymorphic interpretation may have on the use of

¹ Note that while the alphabet of the events shown in the diagrams is made of triplets at the type level ((type, message, type)) (with additional lifeline-id reference), the events that make the traces are concrete (object-id, message, object-id). See the formal definitions in Sect. 3.

scenario-based models throughout the development lifecycle. Requirements can be formally specified in a succinct way, at the highest level of abstraction where they are applicable. Scenario-based tests may be succinctly defined and capture polymorphic behaviors. We return to these applications and to the alarm system example in Sect. 4.

The next section provides the required formal definitions for the syntax and semantics of polymorphic scenario-based specifications.

3 Polymorphic scenarios

We now give trace-based semantics for polymorphic scenarios. For simplicity we limit the presentation to synchronous messages. We assume the reader is familiar with well-known basic notions in the context of classical sequence diagrams such as the partial order on events induced by a diagram and the notion of a cut and thus concentrate on the features unique to the polymorphic extension. We use the terms interaction and scenario interchangeably.

A polymorphic interaction, represented by a sequence diagram, is made of a set of *lifelines*, each of which represents a class or an interface in a system-model. Trace-based semantics for a scenario-based specification is given by constructing an automaton for each diagram in the specification and relating the language accepted by the automaton to inter-object event traces of the system-model. We adopt the *modal* profile defined in [15] and the distinction of LSC between existential and universal diagrams (see [6, 15]).

The following generalizes the formal definitions given in [15] from concrete to polymorphic scenarios. We start off with formal definitions of the system-model and the syntax of polymorphic scenarios. We then give the semantics of universal and existential polymorphic scenarios, first informally and then formally. Finally, we relate a polymorphic scenario-based specification to a system-model. For simplicity, we consider here only Messages. Adding other constructs available for UML2 interactions such as StateInvariants (LSC conditions) and InteractionFragments with operators such as alt and loop, does not change the essence of the construction. We assume strict sequencing only.

3.1 The basics

3.1.1 System-model

We consider a *system-model* $Sys = \langle O, Ty, type, \leq_{Ty} \rangle$, which includes a (possibly infinite) set of objects $O = \{o_1, o_2, \dots\}$, a partially ordered set of types $Ty = \{ty_1, ty_2, \dots, ty_m\}$, and a mapping from each object in O to its type $type : O \rightarrow Ty$. The partial order \leq_{Ty} represents the inheritance / interface realization relation between the

elements of Ty . The mapping $type$ derives an *instanceof* Boolean function $instanceof : (O \times Ty) \rightarrow \{true, false\}$ such that $instanceof(o, ty) = true$ iff $type(o) \leq_{Ty} ty$.

A type $ty \in Ty$ has a finite set of signatures $m(ty) = \{m_1, m_2, \dots, m_s\}$. The subtyping partial order \leq_{Ty} over Ty implies signature set inclusion: $\forall ty_1, ty_2 \in Ty, ty_1 \leq_{Ty} ty_2$ implies $m(ty_2) \subseteq m(ty_1)$. We allow multiple inheritance with a disjoint signature restriction: $\forall ty_1, ty_2, ty_3 \in Ty$, if $ty_1 \leq_{Ty} ty_2$ and $ty_1 \leq_{Ty} ty_3$ and $ty_2 \not\leq_{Ty} ty_3$ and $ty_3 \not\leq_{Ty} ty_2$ then $m(ty_2) \cap m(ty_3) = \emptyset$. Note that we ignore the difference between class and interface types as it has no semantic significance in the trace-based semantics we present.

A *system-model event* e is a tuple $\langle o_{src}, m, o_{trg} \rangle$ where $o_{src}, o_{trg} \in O$ and $m \in m(type(o_{trg}))$, carrying the intuitive meaning of object o_{src} calling message m of object o_{trg} (we allow $o_{src} = o_{trg}$). A *system-model trace* is an infinite sequence of events e_1, e_2, e_3, \dots

3.1.2 Polymorphic scenario

A (modal) *polymorphic scenario* is a tuple $D = \langle mode, L, ltype, LPME, eventMode \rangle$ where

- $mode \in \{existential, universal\}$ is the mode of the scenario;
- $L = \{l_1, l_2, \dots, l_k\}$ is a finite set of lifelines; each lifeline l_i includes an ordered set of r_i event occurrence specifications (denoting message sent or received) on this lifeline: $\forall i, 1 \leq i \leq k, l_i = \{l_i^1, l_i^2, \dots, l_i^{r_i}\}$;
- $ltype : L \rightarrow Ty$ is a mapping from each lifeline to a type;
- $LPME$ is a set of lifeline-polymorphic-message-event triplets $\langle l_{src}^p, m, l_{trg}^q \rangle$ where l_{src}^p is a send message event occurrence specification on the source lifeline, l_{trg}^q is a receive message event occurrence specification on the target lifeline, and $m \in m(ltype(l_{trg}))$ is the signature of the message;
- and $eventMode : LPME \rightarrow \{cold, hot\}$ is a mapping giving a temperature to each message triplet in D (in existential diagrams we consider cold messages only).

The set of lifelines L and the mapping $ltype$ defines the set of possible bindings $Bind(L) \subseteq (O \cup \{\perp\})^k$ such that $\langle o_1, o_2, \dots, o_k \rangle \in Bind(L)$ iff $\forall i, 1 \leq i \leq k, o_i = \perp \vee instanceof(o_i, ltype(l_i))$. A given binding $\langle o_1, o_2, \dots, o_k \rangle \in Bind(L)$ defines a trivial projected function $bind : L \rightarrow (O \cup \{\perp\})$ from a lifeline to its bound object: $\forall i, 1 \leq i \leq k, bind(l_i) = o_i$. \perp is used to represent the value of a lifeline binding while it is unbound (free).

For a lifeline-polymorphic-message-event triplet $lpme = \langle l_{src}^p, m, l_{trg}^q \rangle \in LPME$ we use $src(lpme)$, $m(lpme)$,

$trg(lpme)$ to denote its source lifeline, message signature, and target lifeline, respectively. Based on $LPME$ we define the set of polymorphic-message-events in D : $PME = \{\langle ty_{src}, m, ty_{trg} \rangle \mid \exists lpme \in LPME \text{ s.t. } ty_{src} = ltype(src(lpme)) \wedge ty_{trg} = ltype(trg(lpme)) \wedge m = m(lpme)\}$. PME abstracts away lifeline locations and identities, but keeps their types. We use PME in the definition of the semantics below. Note that the same triplet $\langle ty_{src}, m, ty_{trg} \rangle \in PME$ may correspond to more than one message event occurrence in $LPME$ over identical lifelines in different locations or over different pairs of lifelines.

3.1.3 The semantics of a polymorphic scenario

The semantics of a polymorphic scenario D is given using an automaton A_D ; the trace-language of a scenario is the language accepted by its automaton $L(A_D)$. The construction of the automaton A_D is based on an *unwinding structure* S (see e.g., [20]). Intuitively, this structure is made of states representing cuts and includes paths for all possible linearizations of the partial order between events defined by the diagram; that is, where event occurrences on each lifeline are ordered from top to bottom, and message send event precedes the same message receive event.² For simplicity in this paper we treat message send and receive as a single event. We consider only well-formed diagrams, that is, diagrams that indeed induce a partial order (see [40]).

The unwinding structure is made of a set of cut-states S (with a designated minimal cut-state $s_{min} \in S$), and a partial (transition) function $R : S \times LPME \rightarrow S$.

The set of *enabled*-lifeline-polymorphic-message-event-occurrences in a cut $s \in S$ is defined by $EnLPME(s) = \{e \in LPME \mid \exists s' \in S : R(s, e) = s'\}$. The set of *enabled*-polymorphic-message-events in a cut s is defined by $EnPME(s) = \{\langle ty_{src}, m, ty_{trg} \rangle \in PME \mid \exists e \in EnLPME(s) : m(e) = m \wedge ty_{src} = ltype(source(e)) \wedge ty_{trg} = ltype(target(e))\}$.

The mapping $eventMode : LPME \rightarrow \{cold, hot\}$ of the diagram is extended in the unwinding structure S to cut-states as follows: $mode : S \rightarrow \{cold, hot\}$ is defined s.t. $mode(s) = hot$ if $\exists e : e \in EnLPME(s) \wedge eventMode(e) = hot$; otherwise $mode(s) = cold$. That is, a cut is hot iff at least one of its enabled message event occurrences is hot. The intended semantics of a hot cut is that of an unstable state; when the scenario is in a hot cut, there is at least one message that must eventually occur in order for the scenario to be satisfied (see [15]).

² This structure is common to most variants of sequence diagrams presented in the literature; we thus assume the reader is familiar with it and concentrate on the issues unique to the polymorphic extension we present here.

Note that while the ‘alphabet’ for messages appearing in a polymorphic scenario D is the ‘abstract’ type-level events alphabet $\Sigma_{abs} \subseteq Ty \times M \times Ty$ such that $\Sigma_{abs} = \{\langle ty_1, m, ty_2 \rangle \mid ty_1, ty_2 \in Ty \wedge m \in m(ty_2)\}$, the alphabet Σ for the automata defined below is the ‘concrete’ object-level message events alphabet $\Sigma \subseteq O \times M \times O$ such that $\Sigma = \{\langle o_1, m, o_2 \rangle \mid o_1, o_2 \in O \wedge m \in m(type(o_2))\}$.

We define the set of concrete object-level message events in Σ that may be unified with polymorphic message events in the set PME as follows: $CPME = \{\langle o_{src}, m, o_{trg} \rangle \mid \langle o_{src}, m, o_{trg} \rangle \in \Sigma \wedge \exists \langle ty_{src}, m, ty_{trg} \rangle \in PME \text{ s.t. } instanceof(o_{src}, ty_{src}) \wedge instanceof(o_{trg}, ty_{trg})\}$.

The intended semantics for a universal polymorphic scenario is that of a temporal invariant that holds on all system-model traces and from any point on those traces. Thus, the semantics of a universal polymorphic scenario is given using an alternating automaton (see below). Roughly, for each run of the automaton, instantiated following the occurrence of a minimal event in the partial order induced by the diagram, the automaton checks whether the message of this event is enabled or violating with regard to the current cut. If it is enabled, the automaton checks for a binding: if there are free (yet unbound) lifelines that can bind to the event’s concrete source and target object (or there are lifelines that are already bound to the event’s source or target), the automaton binds the free lifeline(s) and advances the cut-state accordingly. Otherwise, it ignores the event. If the message in this event is violating, that is, it appears in the diagram but is not currently enabled, the automaton checks for binding too: if there are lifelines that are already bound to the event’s source and target, the event is indeed violating and the violation is handled according to the current cut-state mode: if the cut is hot, it is a hot violation, and the run moves to a rejecting sink state; if the cut is cold, it is a cold violation, and the run moves to an accepting sink state. If the automaton reaches the maximal cut-state it moves to its accepting sink too.

The intended semantics of an existential polymorphic scenario is that of an example; there must be at least one possible system-model run where the scenario ‘happens’ at least once. Thus, the semantics of an existential polymorphic scenario is given using a nondeterministic automaton whose first state needs to ‘guess’ as to when does an accepting sequence begin. A similar mechanism to the one described above for binding of enabled events is used in the existential case.

The above-intended semantics and informal automata constructions are formalized in the definitions of the two automata given in the following subsections.

3.2 Universal polymorphic scenarios: formally

The semantics of a universal polymorphic scenario is given using an alternating automaton; the trace-language of a diagram is the language accepted by its automaton. 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., [26]). Given a universal diagram D , we construct an alternating Büchi automaton $A_D = \langle \Sigma \cup \epsilon, Q, q_{in}, \delta, \alpha \rangle$, where

- $\Sigma = \{\langle o_1, m, o_2 \rangle \mid o_1, o_2 \in O \wedge m \in m(type(o_2))\}$;
- $Q = S \times Bind(L) \cup \{q_{rej}, q_{acc}\}$ is a set of states (we use $cut(q)$ to denote the cut-state s of a state $q = \langle s, \langle o_1, \dots, o_k \rangle \rangle$);
- $q_{in} = \langle s_{min}, \{\perp\}^k \rangle$ is the initial state;
- $\alpha = \{\langle s, \langle o_1, \dots, o_k \rangle \rangle \mid mode(s) = cold\} \cup \{q_{acc}\}$ is the accepting condition (that is, all cold states and q_{acc} are accepting);
- and $\delta : Q \times \Sigma \longrightarrow B^+(Q)$ is a transition function defined as follows:
 - Σ labeled self transitions on q_{acc} and q_{rej} :
 $\forall cme \in \Sigma : \delta(q_{acc}, cme) = q_{acc}, \delta(q_{rej}, cme) = q_{rej}$
 - $\Sigma \setminus CPME$ labeled self transitions on all cut-states:
 $\forall q \in Q \setminus \{q_{rej}, q_{acc}\}, \forall cme \in \Sigma \setminus CPME :$
 $\delta(q, cme) = q$
 - Handling message events in $CPME$:
 $\forall q = \langle s, \langle o_1, \dots, o_k \rangle \rangle \in Q \setminus \{q_{in}, q_{rej}, q_{acc}\},$
 $\forall cme \in CPME :$
 - (the source and target objects of cme are already bound) for l_{src}^i, l_{trg}^j s.t. $source(cme) = bind(l_{src}) \wedge target(cme) = bind(l_{trg}) \wedge \langle l_{src}^i, m(cme), l_{trg}^j \rangle \in EnLPME(cut(q))$:
 $\delta(q, cme) = \langle R(cut(q), e), \langle o_1, \dots, o_k \rangle \rangle$ where $e = \langle l_{src}^i, m(cme), l_{trg}^j \rangle$;
 - for l_{src}^i, l_{trg}^j s.t. $source(cme) = bind(l_{src}) \wedge target(cme) = bind(l_{trg}) \wedge \langle l_{src}^i, m(cme), l_{trg}^j \rangle \notin EnLPME(cut(q))$:
 - if $mode(cut(q)) = cold$, then $\delta(q, cme) = q_{acc}$,
 - if $mode(cut(q)) = hot$, then $\delta(q, cme) = q_{rej}$;
 - (otherwise, the source object of cme is already bound and the target can bind to a free lifeline) for l_{src}^i s.t. $source(cme) = bind(l_{src})$, for all l_{trg}^j s.t. $instanceof(target(cme), ltype(l_{trg})) \wedge bind(l_{trg}) = \perp \wedge \langle l_{src}^i, m(cme), l_{trg}^j \rangle \in EnLPME(cut(q))$:
 $\delta(q, cme) = \bigwedge_{l_{trg}^j} \langle R(cut(q), e), \langle \bar{o}_1, \dots, \bar{o}_{l_{trg}}, \dots, \bar{o}_k \rangle \rangle$
 where $e = \langle l_{src}^i, m(cme), l_{trg}^j \rangle \wedge \bar{o}_{l_{trg}} = target(cme) \wedge \forall h \neq l_{trg} : \bar{o}_h = o_h$;
 - (otherwise, symmetrically, the target object of cme is already bound and the source can bind to a free lifeline)
 Same as above only replace $source(cme)$ and $target(cme)$, l_{trg} and l_{src} .

- (otherwise, the source and the target objects of cme are not yet bound but each can bind to a free lifeline)
for all l_{src}^i, l_{trg}^j s.t. $bind(l_{src}) = \perp \wedge bind(l_{trg}) = \perp \wedge$
 $instanceof(target(cme), ltype(l_{trg})) \wedge$
 $instanceof(source(cme), ltype(l_{src})) \wedge$
 $\langle l_{src}^i, m(cme), l_{trg}^j \rangle \in EnLPME(cut(q))$:
 $\delta(q, cme) = \bigwedge_{l_{src}^i, l_{trg}^j}$
 $\langle R(cut(q), e), \langle \bar{o}_1, \dots, \bar{o}_{l_{src}}, \dots, \bar{o}_{l_{trg}}, \dots, \bar{o}_k \rangle \rangle$
where $e = \langle l_{src}^i, m(cme), l_{trg}^j \rangle \wedge$
 $\bar{o}_{l_{src}} = source(cme) \wedge \bar{o}_{l_{trg}} = target(cme) \wedge$
 $\forall h \neq l_{trg}, l_{src} : \bar{o}_h = o_h$;
- (otherwise, cme is ignored)
 $\delta(q, cme) = q$;

and for the initial state $q_{in} = \langle s_{min}, \{\{\perp\}^k\} \rangle, \forall cme \in CPME$:

$$\delta(q_{in}, cme) = q_{in} \wedge \bigwedge_{l_{src}^i, l_{trg}^j}$$

$$\langle R(s_{min}, e), \langle \bar{o}_1, \dots, \bar{o}_{l_{src}}, \dots, \bar{o}_{l_{trg}}, \dots, \bar{o}_k \rangle \rangle$$

for all l_{src}^i, l_{trg}^j s.t.
 $instanceof(target(cme), ltype(l_{trg})) \wedge$
 $instanceof(source(cme), ltype(l_{src})) \wedge$
 $\langle l_{src}^i, m(cme), l_{trg}^j \rangle \in EnLPME(cut(q_{in}))$
 where $e = \langle l_{src}^i, m(cme), l_{trg}^j \rangle \wedge \bar{o}_{l_{trg}} = target(cme) \wedge$
 $\bar{o}_{l_{src}} = source(cme) \wedge \forall h \neq l_{src}, l_{trg} : \bar{o}_h = o_h$.

Several advanced semantic issues and extensions related to the construction above are discussed in Sect. 5.

3.3 Existential polymorphic scenarios: formally

The semantics of an existential polymorphic scenario is given using a non-deterministic automaton; the trace-language of the diagram is the language accepted by the automaton.

Given an existential diagram D , we construct a non-deterministic Büchi automaton $A_D = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$, where

- $\Sigma = \{\langle o_1, m, o_2 \rangle \mid o_1, o_2 \in O \wedge m \in m(type(o_2))\}$;
- $Q = S \times Bind(L) \cup \{q_{rej}\}$ is a finite set of states (we use $cut(q)$ to denote the cut-state s of a state $q = \langle s, \langle o_1, \dots, o_k \rangle \rangle$);
- $q_{in} = \langle s_{min}, \{\{\perp\}^k\} \rangle$ is the initial state;
- $\alpha = \{q \in Q : cut(q) = s_{max}\}$ is the accepting condition;
- and $\delta : Q \times \Sigma \rightarrow 2^Q$ is defined as follows:
- Σ labeled self transitions on q_{max} and q_{rej} :
 $\forall cme \in \Sigma : \delta(q_{max}, cme) = \{q_{max}\}, \delta(q_{rej}, cme) = \{q_{rej}\}$;
- $\Sigma \setminus CPME$ labeled self transitions on all cut-states:
 $\forall q \in Q \setminus \{q_{rej}\}, \forall cme \in \Sigma \setminus CPME : \delta(q, cme) = \{q\}$;

- Handling message events from $CPME$:
Same as in the universal case, only replace the conjunction \wedge with set union \cup and have bounded $\langle l_{src}^i, m(cme), l_{trg}^j \rangle \notin EnLPME(cut(q))$ leading to $\{q_{rej}\}$. On the initial state q_{in} replace the conjunction \wedge with set union.

In Sect. 5 we discuss some possible variants of the semantics defined by the construction above.

3.4 Relating a polymorphic specification to a system-model

Recall that the *trace-language* of a polymorphic scenario D is the word language $L(D)$ accepted by its automaton. Following LSC, a *specification* is a set $Spec = Ex \cup Un$, where Ex and Un are sets of existential and universal diagrams, respectively (see [6, 15]). We denote the runs of a system-model Sys by L_{Sys} . We say that a system-model Sys satisfies a specification $Spec = Ex \cup Un$, in symbols, $Sys \models Spec$, iff

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

4 Applications

4.1 Polymorphic scenario-based testing

A common use of sequence diagrams in model-driven development is for testing purposes. That is, one may specify testing scenarios using sequence diagrams (see e.g., [4], for the use of sequence diagrams to specify the behavior of test cases in the UML Testing Profile [37]). Taking advantage of the polymorphic extension, testing scenarios can be defined at a rather high level of abstraction, i.e., at the interface or abstract classes level, and thus be applicable to, and reused across, all concrete system models realizing the generic behavior.

As an example, recall the alarm system described in Sect. 2. One may use diagram D1 as a test case, activating a sensor, generating a notification, and waiting for the alarm object to call the buzzer. The test is specified at the `ISensor` interface and `CAAlarm` class level; its definition need not change when applied to different system model implementations of the alarm system, e.g., with different sensors or an instance of a new class derived from `CAAlarm` (also, `CAAlarm` may be an abstract class).

An implementation of polymorphic scenario-based testing, following the semantics presented in this paper, has been carried out in the context of Java within the *S2A compiler* [12, 33]. S2A (for Scenarios to Aspects) is a compiler that translates LSCs, given in their UML2-compliant variant using the *modal* profile of [15], into AspectJ code [1, 18], and

Fig. 4 A scenario-based trace of the alarm system with a CStdAlarm and two sensors

```

1 E: 1264348840484 1: void alarmSystem.sensors.ISensor.activate()
2 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 0 <- alarmSystem.CBuzzer@87816d
3 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 1 <- alarmSystem.CStdAlarm@112f614
4 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 2 <- alarmSystem.sensors.CFireSensor@422ede
5 B: alarmSystem.aspects.MSDAspectD3[0] lifeline 0 <- alarmSystem.CStdAlarm@112f614
6 B: alarmSystem.aspects.MSDAspectD3[0] lifeline 1 <- alarmSystem.sensors.CFireSensor@422ede
7 C: alarmSystem.aspects.MSDAspectD1[0] (0,1,1) Cold
8 C: alarmSystem.aspects.MSDAspectD3[0] (1,1) Cold
9 E: 1264348840484 2: void alarmSystem.sensors.ISensor.activate()
10 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 0 <- alarmSystem.CBuzzer@87816d
11 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 1 <- alarmSystem.CStdAlarm@112f614
12 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 2 <- alarmSystem.sensors.CDoorSensor@19efb05
13 B: alarmSystem.aspects.MSDAspectD3[1] lifeline 0 <- alarmSystem.CStdAlarm@112f614
14 B: alarmSystem.aspects.MSDAspectD3[1] lifeline 1 <- alarmSystem.sensors.CDoorSensor@19efb05
15 C: alarmSystem.aspects.MSDAspectD1[1] (0,1,1) Cold
16 C: alarmSystem.aspects.MSDAspectD3[1] (1,1) Cold
17 E: 1264348841656 3: void alarmSystem.CAlarm.notifyAlarm()
18 C: alarmSystem.aspects.MSDAspectD1[1] (0,2,2) Hot
19 C: alarmSystem.aspects.MSDAspectD3[1] (2,2) Cold
20 E: 1264348841656 4: void alarmSystem.CBuzzer.start()
21 F: alarmSystem.aspects.MSDAspectD1[0] Violation
22 C: alarmSystem.aspects.MSDAspectD1[1] (1,3,2) Cold
23 F: alarmSystem.aspects.MSDAspectD1[1] Completion

```

thus provides full code generation of reactive behavior from visual declarative scenario-based specifications. S2A implements a compilation scheme presented in [31]. Roughly, each sequence diagram is translated into a *scenario aspect*, implemented in AspectJ, which simulates an automaton whose states correspond to the scenario cuts; transitions are triggered by AspectJ pointcuts, and corresponding advice is responsible for advancing the automaton to the next cut state. In S2A, sequence diagrams messages are identified with method calls in Java.

Most important in the context of this paper, though, is that S2A supports polymorphic scenarios. Taking advantage of AspectJ and Java semantics, the generated code is able to monitor the activation and progress of all realizations of the polymorphic UML2-compliant LSCs as they come to life during an execution of a reference Java program. This includes the instantiation of multiple copies of each scenario aspect and the implementation of the late binding and unification mechanism of the trace-based semantics formally defined in Sect. 3. That is, the generated aspect advice code is responsible not only for advancing the automaton to the next cut state but also for checking and handling late binding and ‘new automata’ instantiation.

We have created a Java implementation of a simple simulation of an alarm system based on the design shown in Sect. 2, with two sensors and with a CStdAlarm. We used the diagrams shown in Fig. 2 as input for S2A and generated scenario aspects for them. Thus, when executing the (automatically instrumented) Java program, we were able to view how the polymorphic semantics is realized. For example, multiple instances of scenario D1 were created, each with a binding to a different sensor. Then, when one sensor notified the alarm, only the corresponding scenario instance, the one where the notifying sensor was bound, advanced to its next cut state.

Next, we modified the implementation: replaced the CStdAlarm with a CAdvAlarm and added another sensor.

Then, we were able to reuse exactly the same test case specifications for the modified system (in fact, we did not have to run S2A again but used the same generated aspects). Moreover, now we could also observe how diagram D2 is realized; after activation and notification, the alarm object was bound as a CAdvAlarm to an instance of diagram D2, and at the same time as a CAlarm to all instances of diagram D1, resulting in a truly polymorphic setting. These observations were made visible using *scenario-based traces* (see [29]), automatically generated by the runtime component of S2A, as we demonstrate below.

Figure 4 shows an excerpt from a scenario-based trace of a concrete run of the alarm system with a CStdAlarm alarm object. As we activated two different sensor objects, a CDoorSensor object and a CFireSensor object, we see two instances of D1, where in each the ISensor lifeline binds to a different sensor (lines 4 and 12) (in the traces, lifelines are numbered from left to right, starting with 0; cut states appear as tuples of integers; diagram instance numbers appear inside square brackets next to the name of the diagram). Later, when the door sensor notifies the alarm, we see the cut of its instance of diagram D1 advances from (0, 1, 1) to (0, 2, 2) (lines 15 and 18) and then to (1, 3, 2) where it is completed (lines 22–23). We also see the other instance of diagram D1 closing, with a cold violation (line 21), because the start call from the alarm to the buzzer was not enabled for this instance.

Figure 5 shows a similar excerpt from a scenario-based trace, this time of a concrete run of the modified alarm system implementation, with an advanced alarm instead of a standard alarm (that is, a CAdvAlarm alarm object instead of the CStdAlarm), and with another door sensor. When all three sensors are activated, we can see the three instances of D1, where in each the ISensor lifeline binds to a different sensor object (lines 4, 12, and 20). We can also see that in each of these instances, the second lifeline binds to the CAdvAlarm alarm object (lines 3, 11, and 19). Moreover, we see an instance of diagram D2 (lines 26–27, 29, and

```

1 E: 1264348976656 1: void alarmSystem.sensors.ISensor.activate()
2 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 0 <- alarmSystem.CBuzzer@1d9dc39
3 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 1 <- alarmSystem.CAdvAlarm@b89838
4 B: alarmSystem.aspects.MSDAspectD1[0] lifeline 2 <- alarmSystem.sensors.CFireSensor@93dcd
5 B: alarmSystem.aspects.MSDAspectD3[0] lifeline 0 <- alarmSystem.CAdvAlarm@b89838
6 B: alarmSystem.aspects.MSDAspectD3[0] lifeline 1 <- alarmSystem.sensors.CFireSensor@93dcd
7 C: alarmSystem.aspects.MSDAspectD1[0] (0,1,1) Cold
8 C: alarmSystem.aspects.MSDAspectD3[0] (1,1) Cold
9 E: 1264348976656 2: void alarmSystem.sensors.ISensor.activate()
10 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 0 <- alarmSystem.CBuzzer@1d9dc39
11 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 1 <- alarmSystem.CAdvAlarm@b89838
12 B: alarmSystem.aspects.MSDAspectD1[1] lifeline 2 <- alarmSystem.sensors.CDoorSensor@22c95b
13 B: alarmSystem.aspects.MSDAspectD3[1] lifeline 0 <- alarmSystem.CAdvAlarm@b89838
14 B: alarmSystem.aspects.MSDAspectD3[1] lifeline 1 <- alarmSystem.sensors.CDoorSensor@22c95b
15 C: alarmSystem.aspects.MSDAspectD1[1] (0,1,1) Cold
16 C: alarmSystem.aspects.MSDAspectD3[1] (1,1) Cold
17 E: 1264348976656 3: void alarmSystem.sensors.ISensor.activate()
18 B: alarmSystem.aspects.MSDAspectD1[2] lifeline 0 <- alarmSystem.CBuzzer@1d9dc39
19 B: alarmSystem.aspects.MSDAspectD1[2] lifeline 1 <- alarmSystem.CAdvAlarm@b89838
20 B: alarmSystem.aspects.MSDAspectD1[2] lifeline 2 <- alarmSystem.sensors.CDoorSensor@1d1acd3
21 B: alarmSystem.aspects.MSDAspectD3[2] lifeline 0 <- alarmSystem.CAdvAlarm@b89838
22 B: alarmSystem.aspects.MSDAspectD3[2] lifeline 1 <- alarmSystem.sensors.CDoorSensor@1d1acd3
23 C: alarmSystem.aspects.MSDAspectD1[2] (0,1,1) Cold
24 C: alarmSystem.aspects.MSDAspectD3[2] (1,1) Cold
25 E: 1264348980046 4: void alarmSystem.CAlarm.notifyAlarm()
26 B: alarmSystem.aspects.MSDAspectD2[0] lifeline 0 <- alarmSystem.CAdvAlarm@b89838
27 B: alarmSystem.aspects.MSDAspectD2[0] lifeline 1 <- alarmSystem.sensors.CDoorSensor@1d1acd3
28 C: alarmSystem.aspects.MSDAspectD1[2] (0,2,2) Hot
29 C: alarmSystem.aspects.MSDAspectD2[0] (1,1,0) Hot
30 C: alarmSystem.aspects.MSDAspectD3[2] (2,2) Cold
31 E: 1264348980046 5: void alarmSystem.CBuzzer.start()
32 F: alarmSystem.aspects.MSDAspectD1[0] Violation
33 F: alarmSystem.aspects.MSDAspectD1[1] Violation
34 C: alarmSystem.aspects.MSDAspectD1[2] (1,3,2) Cold
35 F: alarmSystem.aspects.MSDAspectD1[2] Completion
36 E: 1264348980046 6: void logging.ILogger.notifyLog()
37 B: alarmSystem.aspects.MSDAspectD2[0] lifeline 2 <- logging.CSimpleLogger@fd13b5
38 C: alarmSystem.aspects.MSDAspectD2[0] (2,1,1) Cold
39 F: alarmSystem.aspects.MSDAspectD2[0] Completion

```

Fig. 5 A scenario-based trace of the alarm system with a CAdvAlarm and three sensors

37–39), which was not present in the trace of the original implementation shown in Fig. 4 because in D2 the first lifeline represents CAdvAlarm, not any subtype of CAlarm.

Scenario-based traces, such as the ones discussed above, are difficult to analyze manually and in textual format. Instead, they can be visualized and interactively explored in a prototype tool called the Tracer [32] (first presented in [34]), developed at the Weizmann Institute of Science. It is outside the scope of this paper to present the Tracer in detail. However, for the interested reader, Fig. 6 shows a screen dump from the main view of the Tracer, after loading the alarm system scenario-based specification model (with some additional scenarios) and a trace similar to the one shown in Fig. 5. For additional details on the Tracer see [32] and the Tracer website [2].

S2A supports not only method calls but also conditions (defined using UML2 StateInvariants), alt and loop interaction fragments, and exact, symbolic, and opaque method parameters. It also supports scenarios with combined static and dynamic lifeline binding (which were used in our example, see Sect. 5.3), and flat scope lifelines (see Sect. 5.6). However, the current implementation of S2A does not support ‘and’ or ‘or’ multiple binding choices (see Sect. 5.2); in S2A, binding non-determinism is solved ad hoc by arbitrarily choosing an available binding if one exists (the

current implementation’s support for this kind of non-determinism is thus partial).

4.2 Polymorphic scenario-based execution (play-out)

In addition to providing a means for polymorphic scenario-based monitoring and testing, S2A supports scenario-based execution (play-out). Play-out, originally defined and implemented in the Play-Engine tool [16], is an operational (executable) semantics for LSC, that is, a method to simulate or execute an LSC specification. Recalling the details of play-out and describing its semantics for polymorphic scenarios is outside the scope of this paper. However, for readers familiar with play-out, we briefly present the following issues.

When using an LSC specification as a program for play-out, some methods are tagged not only with a hot or cold temperature but also with an execution mode (as opposed to the default monitoring mode). When a method with an execution mode is enabled in at least one scenario instance and is not violating in any other scenario instance, the play-out mechanism executes it. Thus, a key part of play-out semantics concerns the strategy for choosing the next method to execute in case there is more than one execution-enabled non-violating method.

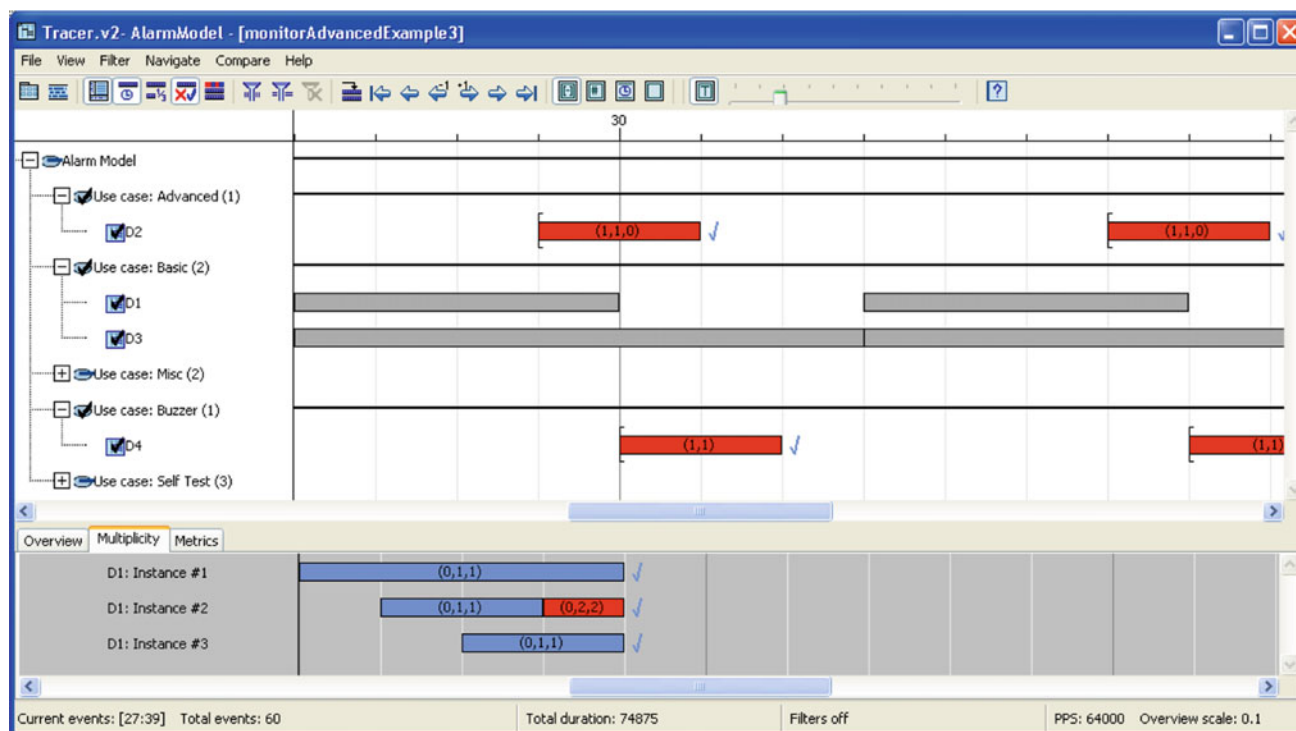


Fig. 6 A screen dump from the Tracer showing a trace similar to the one shown in Fig. 5. Time goes from left to right; the tree on the left shows the hierarchy of use cases and scenarios in the specification model; bars show the cut states of the scenarios as they progress over time; the \checkmark symbols represent scenario completions (a \times symbol

The original (so called naïve) play-out arbitrarily chooses one enabled method that is not violating in any chart and executes it. In a polymorphic settings, however, this becomes more complicated: the ‘same’ method may be simultaneously enabled (or violating) in different scenarios where it is referenced between lifelines at different levels of the type hierarchy. Thus, the need arises to consider the enabled and violating messages at the concrete level, that is, with their dynamically bound source and target objects, and not at the more abstract level of the static types of the lifelines they cover.

In addition, in some cases, a message may be enabled for execution (not just monitoring) while one of (or both of) its lifelines are not yet bound. These problems need to be addressed when defining an operational play-out semantics for polymorphic scenarios. Note that this issue is caused by the dynamic-binding semantics; in a static-binding setting all lifelines are always bound.

The code generated by S2A for play-out supports a polymorphic setting. Information about enabled and violating messages from all scenario instances, including their current dynamic binding, is collected by a (generated) coordinator and transferred into a strategy method. The strategy returns one execution-enabled message that is not violating in any scenario instance (if there is such a message) and the

may appear in case of hot violations). Note the gray bar representing the instances of diagram D1 from location 26 to location 30 of the trace, and the three corresponding instances shown in the Multiplicity view at the lower pane of the main application window

coordinator executes it using inter-type declarations (additional details about the coordinator and strategy appear in [31,33].) However, the case where an execution-enabled message has its source or target not yet bound is not addressed by the current implementation of S2A and so may result in a runtime exception.

5 Advanced semantic issues and extensions

We discuss several advanced semantic issues that arise from our work and present some important variations and extensions.

5.1 Multiple scenario copies

The automaton construction for a universal scenario induces two types of ‘multiple scenario copies’. First, multiple copies of the same scenario where lifelines bind to different concrete objects. These are ‘instantiated’ whenever the automaton reads a minimal event that has a new binding (that is, its source or target objects (or both) are not bound in another instance of the scenario). Second, multiple copies of the same scenario where lifelines bind to the same objects. These are ‘instantiated’ whenever the automaton reads a minimal event

that has an existing binding that is also currently enabled in another copy (this kind of instantiation is independent of the polymorphic extension and applies to universal LSC semantics in general, see [15]). Both ‘instantiations’ are formalized in the universal ‘and’ transition defined on the initial state. For a trace to be accepted by a universal scenario, that is, to be in the language it defines, all ‘runs’ must be accepting (all scenario instances must accept).

In the existential case no ‘multiple copies’ are induced as there is no universal ‘and’ transition in the automaton. Still, in implementing the testing of an existential scenario (e.g., in S2A), multiple copies are created, so as to monitor all possible acceptances. In contrast to the universal case, for a trace to be accepted by an existential scenario (that is, for the run of S2A to pass the test), at least one of its created instances must accept.

5.2 Multiple binding choices

When two or more lifelines in a single diagram represent the same type (or different types related by \leq_{Ty}), two or more transitions for a single concrete event but with different bindings may be simultaneously enabled at some state. In the automaton construction this case is represented by the ‘and’ choices over source and target lifeline selection (recall that the automaton is alternating, hence allowing both ‘and’ and ‘or’ transitions).

As an example, consider a diagram whose lifelines include one representing `CAlarm`, another one representing `CAdvAlarm`, and two lifelines representing `ISensor`. Assume a cut state where all these four lifelines are free (yet unbound) and two notification events are enabled, one from the first sensor to the `CAlarm` lifeline and one from the second sensor to the `CAdvAlarm` lifeline (recall the partial-order semantics, which indeed makes this situation possible). When considering the occurrence of a concrete notification event from a sensor to an advanced alarm object, there are two ways to bind it and advance the cut state accordingly. Choosing the “correct” binding may affect eventual acceptance or rejection of the trace.

The example is illustrated in Fig. 7. It shows how the polymorphic interpretation and the dynamic binding mechanism add a dimension of non-determinism to the scenario-based specification, on top of the ‘classical’ non-determinism inherent in the partial-order semantics.

In the automaton construction for the universal scenario case given in Sect. 3.2 we used an ‘and’ transition to handle these cases. Thus, for the trace to be accepted, all possible transitions resulting from the different binding choices must be extended to an accepting trace.

That said, one may consider the above to be too strict and strong a requirement, and instead suggest a more permissive semantics, with non-deterministic ‘or’ selection between

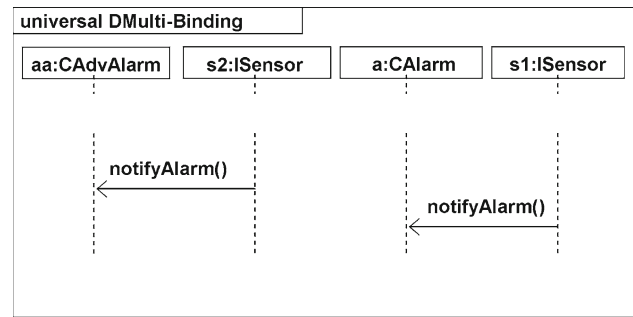


Fig. 7 Assume a cut state where the four lifelines are free (yet unbound) and the two notification events are enabled (there may be additional lifelines in this diagram). When considering the occurrence of a concrete notification event from a sensor to an advanced alarm object, there are two ways to bind it and advance the cut state accordingly. See the discussion in Sect. 5.2

binding choices (in the formal automaton construction this means replacing \wedge with \vee , except for the ‘and’ transition on the initial state). For the trace to be accepted in this case, it would suffice to have at least one of the binding choices extended to an accepting trace. Indeed, our implementation in S2A does not handle this kind of multiple binding choices in a universal way but only in an existential non-deterministic way.

It is important to note that cases that admit multiple binding choices are not expected to be common in practice. If necessary, a tool may be developed to warn the specifier of such a case, either statically (with an over-approximating safety warning) or dynamically. Finally, the choice between the two semantic possibilities—the universal and the existential solution to the multiple choices binding issue—may depend on the specific application domain.

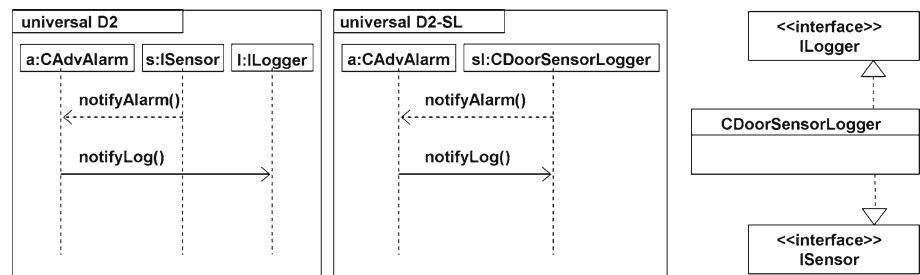
5.3 Combining static and dynamic binding

Our automaton construction is a conservative generalization of the non-symbolic case defined in [15], where lifelines are statically bound to concrete objects. To formally handle non-symbolic lifelines, just define an initial state q_{in} where lifelines are already bound (that is, where not all lifelines are bound to \perp).

It is important to note that this small extension enables the definition of scenarios where some lifelines are statically bound while others are dynamically bound. For example, consider a system where there are two buzzers of the same type `CBuzzer`, `buzzer1` and `buzzer2`, but only one should be used by the alarm (the other one is, say, for backup purposes, and participates in some specific backup scenario). In this case, the first lifeline of diagram D1 should be statically bound to `buzzer1`, regardless of the dynamic bindings of the other lifelines.

Our implementation in S2A indeed supports this extension. Lifelines’ binding may be defined as `static`, instead

Fig. 8 In the presence of the class `CDoorSensorLogger`, which realizes both the `ISensor` interface and the `ILogger` interface, should the two sequence diagrams above be considered equivalent? See the discussion in Sect. 5.4



of dynamic (see the profile definition in Sect. 6), and the compiler looks for the statically bound objects, by name, using a predefined interface. That is, for statically bound objects the engineer is required to write the Java code that returns a reference to the correct object and put this code in a special static class that the generated aspect code would use at runtime.

Indeed, in our implementation of the alarm system presented in Sect. 4, the `CBuzzer` lifeline is statically bound to a buzzer (the only buzzer in the system). We do so since this is the only buzzer in our application anyway (`CBuzzer` is a singleton), and in order to be able to catch related violations (see Sect. 5.5).

5.4 Single binding constraint

One may be interested to ensure that no two lifelines bind to the same object in a single instance of a scenario. That is, although we allow (as we should allow) two or more lifelines in the same scenario to represent the same type (or different types that are related by \leq_{T_y} or that have a common type below them in \leq_{T_y}), the construction of the transition function δ could ensure that such lifelines will never bind to the same concrete object in a single path in the automaton runs tree. Our construction partly addresses this constraint (to make the formal presentation simple, we did not handle self messages in this regard and also ignored some special cases).

In the static non-symbolic case, this constraint is not an issue as it trivially holds if lifelines are required to have unique names (as they are indeed required in all variants of sequence diagrams used in the literature). In the symbolic case (like the one defined in [35]), and more so in the polymorphic case defined in the present paper, this constraint needs to be considered and addressed. Our construction handles it by always checking for possible bindings to already bound lifelines before checking possible bindings to free lifelines.

While the single binding constraint seems reasonable, it is important to understand its consequences, in particular in the polymorphic case, as the following example shows.

Consider an implementation of an alarm system similar to our example in Sect. 2, but where a new class,

`CDoorSensorLogger`, specializing `CDoorSensor`, is defined, which also realizes the `ILogger` interface. Then, according to the single binding constraint, after an object of this class notifies an instance of `CAAdvAlarm`, and an instance of diagram `D2` is created where the `ISensor` lifeline is now bound to the `CDoorSensorLogger` object, the latter part of this instance of `D2` would *not* apply to it, as despite the correct use of types, when the advanced alarm notifies it (as a logger), the scenario cannot bind the `ILogger` lifeline (the same object is already bound to the `ISensor` lifeline). Instead, a new diagram `D2-SL` would be required, which has only two lifelines, one representing a `CAAdvAlarm` and one representing a `CDoorSensorLogger`. This example is illustrated in Fig. 8. It is debatable whether the semantics should be defined such that `D2` and `D2-SL` are equivalent or not in this kind of setting.

5.5 Predefined all-instances semantics

One may suggest a semantics for polymorphic scenarios defined by a ‘static instantiation’ of all diagrams with all possible combinations of object bindings (assuming a finite set of objects). The resulting semantics, which we may call ‘predefined all-instances semantics’, however, may not be identical to the dynamic binding semantics definition we give in the present paper. In the following, we explain why.

In our semantics, an event is considered violating only if it is not enabled and there are lifelines in the scenario that are already bound to its source and target objects. If its source and target objects are not yet bound, the event is ignored and is not considered violating. In the predefined all-instances semantics, lifelines would never be free. Therefore, such an event would be considered violating for all scenario instances where the source and target objects appear.

As a concrete example consider diagram `D1` shown in Fig. 2. After the first activation event, the `CAAlarm` and `ISensor` lifelines are bound. Assuming the `CBuzzer` lifeline is not statically bound (see Sect. 5.3), this lifeline is yet unbound. Thus, although it is not yet enabled, an occurrence of `start` before alarm notification will not be considered a violation.

The above example demonstrates the difference between the predefined all-instances semantics and the dynamic

binding semantics regarding violations. It also shows the expressive power of the combined static and dynamic binding solution described in Sect. 5.3.

We believe that in some cases, the predefined all-instances semantics may be more appropriate (and somewhat simpler) than the dynamic binding semantics definition that we gave in Sect. 3. When this is the case, changing our formal definitions to fit this semantics is technically simple. It only requires the initialization of each scenario with multiple copies having statically bound lifelines covering all possible binding combinations, modulo the set of all system objects (note, though, that the single binding constraint discussed in Sect. 5.4 would still need to be addressed in one way or another).

5.6 Flat semantics extension

The semantics we have defined takes advantage of inheritance and interface realization to allow the specification of most expressive, succinct, and reusable inter-object scenarios for object-oriented system models. As we discuss in Sect. 7, this results in what seems to be a unique and rather strong form of behavioral subtyping.

Yet, in some cases, one may consider our sweeping polymorphic interpretation to be too strong and to result in constraints that are too strict to be useful in practice. In many systems, inheritance is limited to structural subtyping that requires signature matchings but entails no behavioral constraints. While this may not represent a best practice or state-of-the-art design, it is indeed common for subclasses not to comply with their superclasses' complete input/output dependencies or behavior over time. This situation is evident in the motivation behind the various works in the literature on behavioral subtyping discussed in Sect. 7.

Therefore, we looked for ways to handle this issue in our work, for example, in case one is interested in using a scenario-based specification to test an existing system, which was not designed with a behavioral subtyping approach in mind.

To address the above and as a means to adding flexibility to our definition of polymorphic scenario-based specifications, we suggest an extension that distinguishes between deep and flat polymorphic lifelines. Lifelines tagged with deep polymorphic scope are interpreted according to the polymorphic semantics as usual. Lifelines tagged with flat polymorphic scope apply to objects instantiated from a lifeline's type, but not to objects instantiated from its subtypes.³

As an example, consider adding two classes to the class diagram shown in Fig. 1: `CSilentAlarm` as a sub-

class of `CAAlarm`, and `CFlashlight`, with a `startFlash` message. The silent alarm does not use the buzzer so the `CAAlarm` lifeline in diagram D1 should now be tagged `flat`, because it should not apply to `CSilentAlarm` (while it still may apply to `CAAlarm`). The reason to have `CSilentAlarm` as a subclass of `CAAlarm` in this design may be rooted in their common signatures and in some other behavior that is common between the two classes (and is not covered in our example). Obviously, additional diagrams may now be needed to cover the behavior of `CStdAlarm` and `CAdvAlarm`, as these are no longer in the scope of the modified D1 diagram.

The distinction between flat and deep polymorphic scope allows us to take advantage of inheritance and interface realization whenever possible, while keeping the flexibility not to do so when it does not fit our needs. As the example above shows, a specification may include both deep and flat polymorphic lifelines, even within the same diagram: although the `CAAlarm` lifeline of the modified D1 diagram is now flat, the `ISensor` lifeline in this diagram remains deep.

As an alternative we could have considered a different solution: keeping the deep polymorphic interpretation as is while allowing to attach to each lifeline a set of names of types that should not be bound. Another solution could have been formed by allowing to specify a set of names of types that should be bound, per lifeline, instead of the sweeping polymorphic interpretation. We consider these solutions to be possible but less attractive as they are too specific: they depend on the specific types in the system and preclude reuse across implementations. The solution we have chosen, which combines flat and deep interpretations, enables reuse when possible, while still supporting specific idiosyncratic classes' behaviors, if required.

The UML profile we use for the polymorphic extension supports the distinction between flat and deep polymorphic scope (see Sect. 6), with `deep` as the default value. Updating the technical details of the formal definitions given in Sect. 3 to handle this distinction is straightforward. Moreover, we have extended S2A to support flat lifelines so indeed the example described above, with the silent alarm, could be simulated.

5.7 Existential acceptance

According to the construction given in Sect. 3.3, a single possible completion of an existential scenario at whatever level of abstraction in the type hierarchy is a sufficient condition for trace acceptance.

We could have suggested a number of other, different semantics, following different notions of *polymorphic coverage*: at least one possible completion for each possible

³ As a consequence, flat semantics cannot be applied to lifelines representing interfaces and abstract classes because this would result in an inconsistent scenario: one whose trace language is empty.

combination of objects at all derived levels (which may be too strong); at least one possible completion for each derived object at least once (very strong requirement but may be useful); and at least one possible completion for each type at least once (that is, at least one object per type has to participate in a completed scenario instance).

These variations in semantics reflect alternative quantifications over the objects in the system and its runs. We leave the formal definitions of these variants and the evaluation of their usefulness for future work.

6 A UML profile for polymorphic scenarios

We give technical details regarding a UML profile that supports polymorphic scenarios. The profile is built on top of the *modal* profile (MSD) defined in [15]. A variant of the profile described here is used by S2A.

First, we recall the modalities part adapted from [15]. The profile contains an enumeration *InteractionMode* with two enumeration literals: *hot* and *cold*, and a stereotype *modal*, with an attribute *InteractionMode* of type *InteractionMode*. The *modal* stereotype is introduced as an extension of the abstract class *InteractionFragment*, and thus, of its subclasses *Interaction*, *OccurrenceSpecification*, *CombinedFragment*, and *StateInvariant* (see Figs. 3 and 4 in [15]). *Hot* and *cold* interactions are called *Universal* and *Existential*, respectively.

Second, and more importantly in the context of this paper, to support the polymorphic features, including the extensions suggested in Sects. 5.3 and 5.6, the profile includes the following two enumerations and stereotype (see Fig. 9):

- enumeration *LifelineScope*, with two enumeration literals *deep* and *flat*;
- enumeration *LifelineBinding*, with two enumeration literals *static* and *dynamic*;
- and a stereotype *poly* with an attribute *lifelineScope* of type *LifelineScope* and an attribute *lifelineBinding* of type *LifelineBinding*.

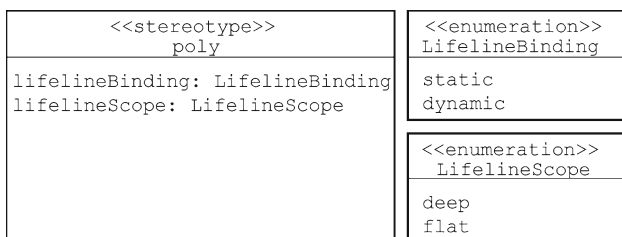


Fig. 9 The stereotype *poly*

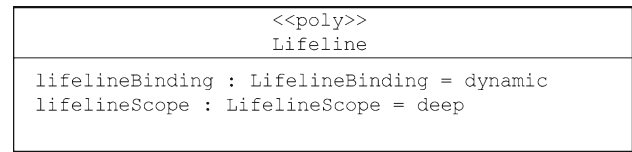


Fig. 10 Extending *Lifeline* with the *poly* stereotype

The stereotype *poly* is introduced as an extension to the class *Lifeline* (see Fig. 10). By default, the polymorphic scope is *deep* and the binding is *dynamic*.

Note that the part of the profile that supports the polymorphic features is independent of the part that supports the modalities. Indeed, a similar profile may be used to add the required polymorphic features to UML standard interactions without modalities (see our discussion of applying the polymorphic semantics to variants of sequence diagrams other than LSC in Sect. 7.1.1).

Finally, the notation we use for the polymorphic part of the profile is as follows: lifeline boxes are annotated with the stereotype *poly* and the values of its two attributes. An editor developed at the Weizmann Institute of Science supports this notation (see Fig. 11).

Note that for the purpose of simplicity and clarity in the example we presented in this paper, as shown in Fig. 2, we did not use the profile's notation for polymorphic lifelines. Rather, we preferred cleaner diagrams and explained the polymorphic nature of the lifelines in the text. Indeed, it seems reasonable to us *not* to use the extra notation if the meaning can be understood from the context. Obviously, different textual or graphical notations may be suggested to mark and distinguish the polymorphic features of the lifelines. In particular, we do not consider the notation shown in Fig. 11 to be significantly better than other alternatives.

7 Related work and discussion

We discuss related literature in the area of scenario-based specifications and consider the application of the polymorphic semantics to other variants of sequence diagrams. We then discuss related work in the area of behavioral subtyping and position our work in this context.

7.1 Scenario-based specifications

Our work extends and generalizes the notion of symbolic lifelines originally presented for LSC in [35] and implemented in the Play-Engine [16]. There, an extension of play-out is defined for LSCs with symbolic lifelines, such that a lifeline representing a type may apply to any object of this type. However, the type system of the Play-Engine is flat, and a generalization to support class hierarchies and interfaces in

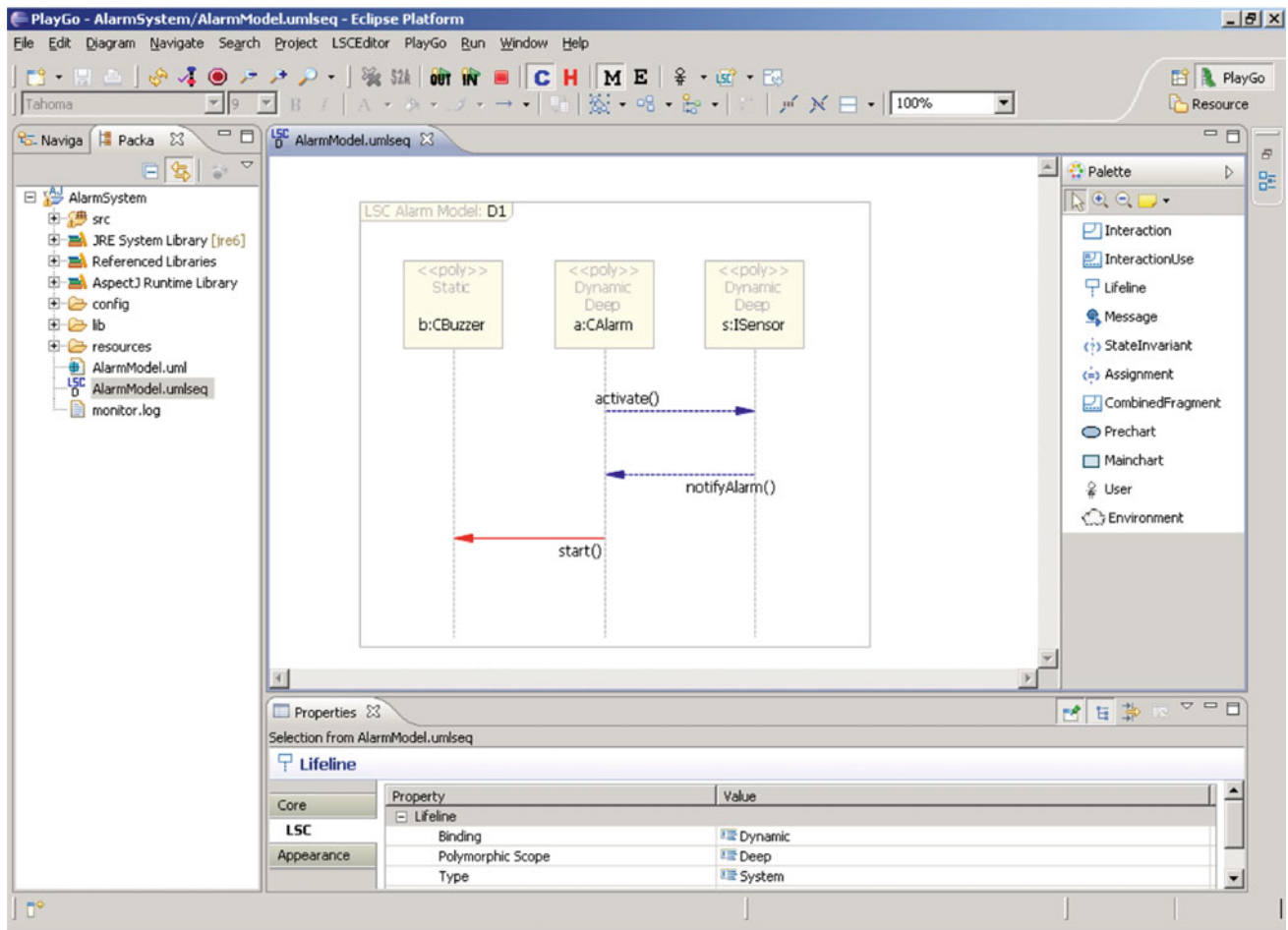


Fig. 11 A screen dump from an eclipse-based UML2-compliant editor that supports our notation for polymorphic lifelines, showing diagram D1 from the alarm system example. Note the application of the `poly`

stereotype to the lifelines in the diagram. A properties pane at the lower part of the screen shows the profile-related attributes of the `CAAlarm` lifeline

the context of object-orientation is not defined. Moreover, a trace-based semantics is not given. Thus, also, some of the issues discussed above in Sect. 5 do not appear in this previous work.

France et al. [10] present a UML-based technique for pattern specification, including interaction pattern specifications (IPs), where lifelines are labeled with role names. Conformance rules are defined between a pattern and its concretization. However, a polymorphic interpretation is not explicitly and formally considered in this work.

Haugen et al. present STAIRS [17], as an approach for the compositional development of UML interactions. STAIRS includes a trace-based three-valued semantics and a number of refinement mechanisms. An operational semantics for STAIRS was defined in [28]. To the best of our knowledge, STAIRS does not consider polymorphism. However, extending STAIRS to support polymorphism may be possible (see the discussion below). In this case, it may also be interesting to adapt the refinement mechanisms of STAIRS

to the polymorphic setting and define additional refinement mechanisms that are based on the polymorphic scope of the scenarios at hand.

Roychoudhury et al. [38] consider MSCs with symbolic lifelines. In this symbolic extension, each lifeline can denote some or all objects from a collection, using per event or per lifeline existential and universal quantification. The work defines an abstract execution semantics, allowing to validate models capturing interactions between large or even unbounded number of objects without state explosion. The semantics presented in [38] is different than LSC semantics. Also, unlike in our work, object-oriented hierarchies are not considered. Extending LSC with universal and existential quantification on lifelines was suggested already in [35]. In the future, it may be interesting to combine this extension with our work. Following [38], it would be interesting to define an abstract execution semantics for the extended polymorphic scenarios, specifically in the context of smart play-out and synthesis from LSC.

We are aware of a number of additional research efforts towards a semantics for MSCs or UML2 interactions (see e.g., [5,21,23]). It seems that none of these considers the relationship between interactions and a polymorphic object-oriented system-model.

7.1.1 Polymorphic scenario-based specifications beyond modal sequence diagrams

Given the above, it is important to examine whether and how can the polymorphic semantics for modal sequence diagrams (UML2-compliant LSC) we defined in the present paper, or a variant thereof, may be applied to other variants of sequence diagrams, in particular because the semantics of LSC is in many ways very different than the semantics of most other variants. We suggest the following observations.

First, some limitations of our work need to be addressed. The formal definitions we give in this paper are limited to messages only, as this suffices to give the essence of the polymorphic extension while keeping the presentation as simple as possible. Thus, obviously, a formal definition that covers state invariants and other constructs needs to be presented. As an example, the expressions inside state invariants, which typically refer to the properties of one or more objects referenced by the construct's covered lifelines, need to be evaluated with a polymorphic view too. For the most part other constructs are not affected by the polymorphic nature of the lifelines and so this should not be a problem.

A more challenging issue concerns the present paper's definition regarding send and receive of a message as one event, which does not necessarily hold in many of the variants of sequence diagrams suggested in the literature. Thus, an extension that considers dynamic binding for asynchronous messages needs to be defined. Specifically, one needs to decide when should the binding of the target lifeline occur; at the moment of sending the event or only when it is received.

Another challenge may be related to the definition of the `par` operator of UML2 interactions, which specifies interleaving sets of behaviors that may involve a common set of lifelines. In a dynamic binding setting, the occurrence of an event may have a binding side effect, in the sense that it may influence the binding of one or more lifelines. Such binding side effects may impose some constraints on the possible interleaving of the behaviors defined between the `par` operands.

Second, one needs to decide whether to define a dynamic binding mechanism or to assume all-instances combinations static binding that respects a polymorphic scope (see Sect. 5.5). Of the two alternatives, it seems that dynamic binding would be more difficult to define, e.g., in the context of the operational rules given for STAIRS in [28] or for UML interactions in [5]. Binding pre-conditions and

post-conditions may need to be added to the rules in a generic symbolic way that enables their dynamic application.

Finally, we believe that the profile we have defined, and its per-lifeline support for flat versus deep scope and static versus dynamic binding, could be used as a basis for adapting the polymorphic extension we have presented to the UML standard notation.

7.2 Behavioral subtyping

Our work is tightly related to the idea of behavioral subtyping, which has been studied in the past and addressed in different ways. In the following, we review some of the literature in this area and position our work in this context.

Liskov and Wing present behavioral subtyping in what is often referred to as the *Liskov Principle* [27]. This includes constraints on methods pre- and post-conditions, invariants, and a history rule applied to observable state changes, specifically in the presence of new methods in the subtype.

The paradigm of design by contract [36], which extends the ordinary definition of abstract data types with preconditions, postconditions, and invariants with precise and formal interface specifications for components, deals with inheritance too. Specifically, subclasses in an inheritance hierarchy are allowed to weaken preconditions and strengthen postconditions and invariants. Thus, design by contract approximates the notion of behavioral subtyping defined in [27] (the history rule is not present).

Polymorphic scenario-based specifications may be seen as inter-object behavioral contracts between components, which, like the contracts of [36], propagate from a superclass to its subclasses. In contrast to [27,36], polymorphic scenario-based specifications focus not on pre- and post-conditions but on the sequences of messages sent and received between the objects over time. Moreover, we did not consider explicit specification strengthening or weakening. Instead, implicitly, by definition, a class's behavior is constrained not only by the scenarios that apply to its superclasses but also by the scenarios that directly apply to it (and hence do not apply to its superclasses), if any.

The works by Ebert and Engels [7–9] distinguish between observable and invocable behavioral substitutability of state-based objects life cycles. Observable substitutability requires that the projection (restriction) of each sequence of method calls possible according to a subclass, to the set of calls of the superclass (its alphabet), would result in a sequence of method calls that is possible according to the superclass. Invocable substitutability requires that each sequence of method calls invocable on a class is also invocable on all its subclasses. In other words, a class description is interpreted to be an upper bound of its set of all possible behaviors in the former, and a lower bound of its set of all possible behaviors in the latter.

The semantics of polymorphic scenario-based specifications may be seen as similar to what Ebert and Engels call invocable behavioral substitutability: all sequences of method calls accepted by a scenario that references a superclass must also hold for the subclasses. Yet, our definitions are significantly different as they present an inter-object and not an intra-object point of view, consider not only one-way method invocations but more general messages sent and received between the participants, and, in addition, enable a rather high-level of abstraction: events not explicitly appearing in the diagram, even if they involve some of the objects that participate in the diagram, are not restricted by the diagram in any way, to occur or not to occur during the runs of the system. This intended under-specification applies to superclasses and subclasses alike.

Finally, Harel and Kupferman [14] consider the notion of inheritance in state-based object-based behavioral specifications too. Their work investigates substitutability in the context of OOAD and suggests the identification of inheritance with a trace containment or a simulation relation, depending on a linear-time or a branching-time semantics.

Since the liveness and safety properties defined by modal scenarios (both universal and existential) can be formulated using various temporal logics (see [25]), our work may be seen as closely related to Harel and Kupferman's work. Yet, as mentioned earlier, we take a scenario-based inter-object view rather than an intra-object view. This difference has far-reaching consequences on how behavioral subtyping may be defined, used, and verified, as we discuss next.

7.2.1 Polymorphic scenarios and behavioral subtyping: aspectual behavioral subtyping

Given the above discussion of related literature, it is clear that the semantics of polymorphic scenario-based specification models presented in this paper is in many ways influenced by and similar to other notions of behavioral subtyping and substitutability. Yet, significantly, while our work considers behavioral subtyping, it does so through a scenario-based inter-object view rather than a pure state-based intra-object view.

Each scenario represents and formalizes a system requirement; typically a combination of liveness and safety properties that involve a number of objects. Then, since behavior is broken by scenarios and not (only) by the structure of the system under development or investigation, behavioral inheritance is also done at the scenario-level and is achieved through the polymorphic interpretation of the scenarios, while referring to an ad-hoc set of participating objects, collectively, and abstracting away the complete state-machines and internal states of the individual objects involved.

As a consequence, in our approach, a superclass does not have to be substitutable with its subclasses in general. We

require that it is substitutable only with regard to the scenarios in the specification model (that apply to the superclass); substitution is intentionally limited to the specified scenarios. For example, referring to the alarm system presented in Sect. 2 again, it may be the case that $CADvAlarm$ includes behaviors that are incompatible with some of the behaviors of its superclass $CAAlarm$, even when they involve the same alphabet. Yet, modulo the scenarios in the specification model that apply to $CAAlarm$, and the liveness and safety constraints they induce, $CAAlarm$ is substitutable with $CADvAlarm$.

In other words, because we abstract away and do not specify the complete behavior of each object (or class) using its internal state-based specification, in general there is no guarantee of trace containment or simulation relation between the superclass and its subclasses. Not all liveness and safety properties that hold for a superclass propagate to its subclasses, but only those specified in the scenarios. As a result, trace containment or simulation between a superclass and its subclasses does not hold in general but only modulo the scenarios in the model. In this sense, our work suggests a notion of what we may call *aspectual behavioral subtyping*, where the 'aspects' are defined by the scenarios in the specification model.

Aspectual behavioral subtyping is defined between a superclass and a subclass, modulo a set of polymorphic aspects or concerns. Verifying aspectual behavioral subtyping in our context amounts to checking that in every system where the scenarios hold, substituting the superclass for the subclass maintains the validity of the scenarios.

Thanks to the division of the behavioral specification into scenarios, our notion of aspectual behavioral subtyping may be seen as a refined version of Harel and Kupferman's behavioral subtyping discussed earlier. Instead of checking trace containment or simulation between a superclass and a subclass, which would result in a single Boolean answer, checking aspectual behavioral subtyping modulo a set of scenarios may result in a more fine-grained answer, revealing that a superclass is substitutable with a subclass modulo some scenarios or concerns but not modulo others. For example, the subclass may replace the superclass modulo all 'basic functionality concerns' and 'security related concerns' but not modulo 'advanced functionality concerns' or 'error-handling concerns'. We consider this fine-grained notion of behavioral substitutability to be an important contribution and advantage of our work.

8 Conclusion and future work

The main contribution of this paper is in extending sequence diagrams with symbolic polymorphic lifelines that support object-oriented inheritance and interface realization, and

providing the extension with formal trace-based semantics. The work extends the expressive power of UML interactions in the context of object-oriented modeling and presents its application to scenario-based testing and execution. A number of advanced semantic issues that arise from the polymorphic interpretation are discussed. Some variations and further extensions are defined, to improve the applicability and the flexibility of the polymorphic interpretation in practice. Finally, the work is positioned in the broader context of behavioral subtyping.

We now suggest future research directions. A distinction between existential and universal bindings for symbolic lifelines, and a notion of a lifeline's binding rule (which appears also, albeit differently, in the UML2 standard as a `selector` property for lifelines), are suggested in [35]. Similar existential and universal lifeline quantification for MSC appears in [38]. We did not consider these in the present work. Our binding semantics for polymorphic lifelines may be viewed as 'existential binding'. A 'universal binding' would have resulted in the ability to specify 'broadcasting'. Binding rules allow to limit the possible lifeline bindings beyond the constraint defined by its type, and we consider them to be very useful. Adding these features or a variant thereof to the polymorphic scenarios presented in this paper is a possible future work direction.

The alternative semantics related to polymorphic coverage for existential scenarios, which we have discussed in Sect. 5.7, may be formalized and discussed in greater detail. Depending on the semantics chosen, one may be interested in generating code for the execution of specific per-class tests. We leave this topic for future work.

Another future work direction deals with defining a polymorphic extension to some of the other variants of sequence diagrams discussed in the literature, e.g., in [17] or in [5]. This is important because the semantics of LSC is in many ways very different than the semantics of most other variants. We have discussed the challenges related to this direction in Sect. 7.1.1. We note that since the polymorphic extension is rather general, we believe it can be added in this specific form, or in a similar form, to a future version of the UML standard section on interactions. The addition may be based on the profile we presented in Sect. 6.

As another possible direction for future work, we consider additional applications for polymorphic scenario-based specification models. Specifically, these include the extension of recent work in the area of model-checking sequence diagrams (e.g., [19,22]) and various kinds of synthesis from sequence diagrams (e.g., [13,24,39,41]) to support a polymorphic semantics.

Finally, in Sect. 7.2 we positioned our present work on polymorphic scenario-based specifications in the more general context of behavioral subtyping and suggested the notion of *aspectual behavioral subtyping*. The idea of aspectual

behavioral subtyping as a refined, fine-grained crosscutting variant of behavioral subtyping seems powerful and novel, and calls for further formalization and investigation.

Acknowledgments I would like to thank Yoram Atir, David Harel, Amir Kantor, Assaf Marron, Itai Segall, and the anonymous reviewers of the MoDELS'09 conference for comments on drafts of [30]. I would like to thank Gregor Engels for pointers to related work. Thanks also to Evyatar Shoshen and Smadar Szekely for their help in the implementation of the alarm system example and the integration of the polymorphic extension of S2A in Eclipse.

References

1. AspectJ. <http://www.eclipse.org/aspectj/>
2. Tracer Website. <http://www.wisdom.weizmann.ac.il/~maoz/tracer/>
3. Atir, Y., Harel, D., Kleinbort, A., Maoz, S.: Object composition in scenario-based programming. In: Fiadeiro, J.L., Inverardi, P. (eds.) Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08). Lecture Notes in Computer Science, vol. 4961, pp. 301–316. Springer, Berlin (2008)
4. Baker, P., Dai, Z., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Berlin (2008)
5. Cengarle, M.V., Knapp, A., Mühlberger, H.: Interactions. In: Lano, K. (ed.) UML 2 Semantics and Applications, pp. 205–248. Wiley, New York (2009)
6. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. *J. Formal Methods Syst. Des.* **19**(1), 45–80 (2001)
7. Ebert, J., Engels, G.: Observable or invocable behaviour—you have to choose! Tech. rep., Universität Koblenz, Koblenz, Germany (1994)
8. Ebert, J., Engels, G.: Structural and behavioural views on OMT-classes. In: Bertino, E., Urban, S.D. (eds.) Proceedings of the International Symposium on Object-Oriented Methodologies and Systems (ISOOMS'94). Lecture Notes in Computer Science, vol. 858, pp. 142–157. Springer, Berlin (1994)
9. Ebert, J., Engels, G.: Specialization of object life cycle definitions. Tech. Rep. 19–95, Koblenz University (1997)
10. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-based pattern specification technique. *IEEE Trans. Softw. Eng.* **30**(3), 193–206 (2004)
11. Harel, D.: From play-in scenarios to code: an achievable dream. *IEEE Comput.* **34**(1), 53–60 (2001)
12. Harel, D., Kleinbort, A., Maoz, S.: S2A: a compiler for multimodal UML sequence diagrams. In: Dwyer, M.B., Lopes, A. (eds.) Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07). Lecture Notes in Computer Science, vol. 4422, pp. 121–124. Springer, Berlin (2007)
13. Harel, D., Kugler, H.: Synthesizing state-based object systems from lsc specifications. *Int. J. Found. Comput. Sci.* **13**(1), 5–51 (2002)
14. Harel, D., Kupferman, O.: On object systems and behavioral inheritance. *IEEE Trans. Softw. Eng.* **28**(9), 889–903 (2002)
15. Harel, D., Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. *Softw. Syst. Modeling (SoSyM)* **7**(2), 237–252 (2008)
16. Harel, D., Marely, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Berlin (2003)
17. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. *Softw. Syst. Modeling (SoSyM)* **4**(4), 355–367 (2005)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.)

- Proceedings of the 15th European Conf. on Object-Oriented Programming (ECOOP'01). Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer, Berlin (2001)
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 of the 18th International Conference on Computer Aided Verification (CAV'06). Lecture Notes in Computer Science, vol. 4144, pp. 219–233. Springer, Berlin (2006)
 20. Klose, J., Wittke, H.: An automata based interpretation of live sequence charts. In: Margaria, T., Yi, W. (eds.) Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01). Lecture Notes in Computer Science, vol. 2031, pp. 512–527. Springer, Berlin (2001)
 21. Knapp, A.: A formal semantics for UML interactions. In: France, R.B., Rumpe, B. (eds.) Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99). Lecture Notes in Computer Science, vol. 1723, pp. 116–130. Springer, Berlin (1999)
 22. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Kühne, T. (ed.) Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers. Lecture Notes in Computer Science, vol. 4364, pp. 42–51. Springer, Berlin (2007)
 23. Krüger, I.: Capturing overlapping, triggered, and preemptive collaborations using MSCs. In: Pezzè, M. (ed.) Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 2621, pp. 387–402. Springer, Berlin (2003)
 24. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: Rammig, F.J. (ed.) International Workshop on Distributed and Parallel Embedded Systems (DIPES'98), IFIP Conf. Proc., vol. 155, pp. 61–72. Kluwer, Dordrecht (1998)
 25. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Halbwachs, N., Zuck, L.D. (eds.) Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05). Lecture Notes in Computer Science, vol. 3440, pp. 445–460. Springer, Berlin (2005)
 26. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. *ACM Trans. Comput. Log.* **2**(3), 408–429 (2001)
 27. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* **16**(6), 1811–1841 (1994)
 28. 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.) Proceedings of the 14th International Symposium on Formal Methods (FM'06). Lecture Notes in Computer Science, vol. 4085, pp. 380–395. Springer, Berlin (2006)
 29. Maoz, S.: Model-based traces. In: Chaudron, M.R.V. (ed.) Models in Software Engineering, Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers. Lecture Notes in Computer Science, vol. 5421, pp. 109–119. Springer, Berlin (2009)
 30. Maoz, S.: Polymorphic scenario-based specification models: Semantics and applications. In: Schürr, A., Selic, B. (eds.) Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09). Lecture Notes in Computer Science, vol. 5795, pp. 499–513. Springer, Berlin (2009)
 31. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Young, M., Devanbu, P.T. (eds.) Proceedings of the 14th International ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'06), pp. 219–230. ACM, New York (2006)
 32. Maoz, S., Harel, D.: On tracing reactive systems. *Softw. Syst. Modeling (SoSyM)* (2010). doi:[10.1007/s10270-010-0151-2](https://doi.org/10.1007/s10270-010-0151-2)
 33. Maoz, S., Harel, D., Kleinbort, A.: A compiler for multi-modal scenarios: transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.* (2009, accepted)
 34. Maoz, S., Kleinbort, A., Harel, D.: Towards trace visualization and exploration for reactive systems. In: Cox, P., Hosking, J. (eds.) Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07), pp. 153–156. IEEE Computer Society, USA (2007)
 35. Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), pp. 83–100. ACM, New York (2002)
 36. Meyer, B.: Applying “Design by Contract”. *IEEE Comput.* **25**(10), 40–51 (1992)
 37. OMG: UML Testing Profile. <http://www.omg.org/docs/formal/05-07-07.pdf> (2005)
 38. Roychoudhury, A., Goel, A., Sengupta, B.: Symbolic message sequence charts. In: Crnkovic, I., Bertolino, A. (eds.) Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC-FSE'07), pp. 275–284. ACM, New York (2007)
 39. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.* **29**(2), 99–115 (2003)
 40. Westphal, B., Toben, T.: The good, the bad and the ugly: well-formedness of live sequence charts. In: Baresi, L., Heckel, R. (eds.) Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE'06). Lecture Notes in Computer Science, vol. 3922, pp. 230–246. Springer, Berlin (2006)
 41. Whittle, J., Kwan, R., Saboo, J.: From scenarios to code: an air traffic control case study. *Softw. Syst. Modeling* **4**(1), 71–93 (2005)

Author Biography



Shahar Maoz is a postdoctoral research fellow with a Wolfgang-Gentner Minerva Fellowship at the Department of Computer Science 3 (Software Engineering), RWTH Aachen University, Germany. His research interests include software and systems modeling, static and dynamic analysis, software visualization, and aspect-oriented software development. Maoz received a PhD in Computer Science from the Weizmann Institute of Science, Israel (2009).