A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ

SHAHAR MAOZ, DAVID HAREL, and ASAF KLEINBORT, The Weizmann Institute of Science

We exploit the main similarity between the aspect-oriented programming paradigm and the inter-object, scenario-based approach to specification, in order to construct a new way of executing systems based on the latter. Specifically, we transform multimodal scenario-based specifications, given in the visual language of *live sequence charts* (LSC), into what we call *scenario aspects*, implemented in AspectJ. Unlike synthesis approaches, which attempt to take the inter-object scenarios and construct intra-object state-based per-object specifications or a single controller automaton, we follow the ideas behind the LSC play-out algorithm to coordinate the simultaneous monitoring and direct execution of the specified scenarios. Thus, the structure of the specification is reflected in the structure of the generated code; the high-level inter-object requirements and their structure are not lost in the translation.

The transformation/compilation scheme is fully implemented in a UML2-compliant tool we term the *S2A compiler* (for Scenarios to Aspects), which provides full code generation of reactive behavior from inter-object multimodal scenarios. S2A supports advanced scenario-based programming features, such as multiple instances and exact and symbolic parameters. We demonstrate our work with an application whose inter-object behaviors are specified using LSCs. We discuss advantages and challenges of the compilation scheme in the context of the more general vision of scenario-based programming.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms: Design, Languages

Additional Key Words and Phrases: Aspect oriented programming, code generation, inter-object approach, live sequence charts, scenario-based programming, scenarios, UML sequence diagrams, visual formalisms

ACM Reference Format:
Maoz, S., Harel, D., and Kleinbort, A. 2011. A compiler for multimodal scenarios: Transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. 20, 4, Article 18 (September 2011), 41 pages.
DOI = 10.1145/2000799.2000804 http://doi.acm.org/10.1145/2000799.2000804

1. INTRODUCTION

Interest in inter-object, scenario-based specifications has increased in recent years. The underlying idea is based on the belief that these provide an intuitive and natural way to think about and capture complex reactive behavior [Damm and Harel 2001; Harel and Marelly 2003a]. Also, the popular concept of use cases [Jacobson 1992] has

---

This work is a revised and extended version of Maoz and Harel [2006] and Harel et al. [2007].
This research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science and by a Grant from the G.I.F., the German-Israeli Foundation for Scientific Research and Development. In addition, part of this research was funded by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).
Author's address: S. Maoz; email: Shahar.maoz@weizmann.ac.il.

DOI 10.1145/2000799.2000804 http://doi.acm.org/10.1145/2000799.2000804

an underlying inter-object flavor, and in a way, calls for formalization and instantiation using scenarios.

An important challenge of the inter-object, scenario-based approach to software specification is to find ways to construct executable systems based on it [Harel 2001]. Many researchers have dealt with this challenge as a synthesis problem [Alur et al. 2003; Harel and Kugler 2002; Krüger et al. 1999; Whittle et al. 2005], where inter-object specifications, given in variants of *message sequence charts* (MSC) [ITU 1996], are translated into intra-object state-based executable specifications for each of the participating objects or components, or into a single global controller automaton.

The *play-out* technique [Harel and Marelly 2003b] is a recent example of a different approach. Instead of synthesizing intra-object state-based specifications for each of the components or building a global controller automaton, the play-out algorithm executes the scenarios directly, keeping track of all user and system events for all objects or components simultaneously, and causing other events and actions to be carried out as dictated by the specified scenarios.

Play-out is not really relevant to classical MSC or UML sequence diagrams [UML 2005], as these are expressively weak, merely specifying existential scenarios that may occur in runs of the real system (which is specified in a more standard, intra-object fashion). For example, they cannot specify what must occur, what may not occur, etc. Rather, play-out was developed for the multimodal language of *live sequence charts* (LSC) [Damm and Harel 2001], which extends classical MSC with a distinction between mandatory-universal behavior and provisional-existential behavior. As a specification language, LSC's expressive power is comparable to that of various temporal logics [Kugler et al. 2005], and it has been used in the context of testing and formal verification [Combes et al. 2008; Klose et al. 2006; Lettrari and Klose 2001]. However, the feature of LSC most relevant to the present article is the fact that its semantics and expressive power are rich enough to give rise to full executability. Thus, LSC can really be viewed as a high-level visual programming language for reactive systems, as discussed in length in Harel and Marelly [2003a]. In Harel and Maoz [2008], we showed how UML2 sequence diagrams can be extended to encompass the multimodal nature of LSCs, leading to executability for them too. In the context of execution, the multimodality of LSC refers not only to the may/must mode but also to the monitor/execute mode (see Section 2.2). The variant of LSC used in the current article is based on the UML2-compliant variant presented in Harel and Maoz [2008].

The *Play-Engine* tool [Harel and Marelly 2003a] contains the two main implementations of play-out available, naïve play-out (or simply play-out) and *smart play-out* [Harel et al. 2002].[1] Naïve play-out chooses the next method to execute based only on the current enabled and violating events, while smart play-out uses model-checking techniques to look ahead and compute a safe execution path, if one exists. Both implementations essentially work as LSC interpreters and can drive the simulation of an application execution, provided it implements certain custom interfaces. Hence, they do not integrate with a standard development environment nor can they produce a standard executable program. This limits the applicability of the play-out execution mechanism, and hence of scenario-based programming in general, in real-world software development.

The paradigm of *aspect-oriented programming* (AOP) has been proposed as a mechanism that enables the modular implementation of crosscutting concerns [Kiczales et al. 1997]. Separation of concerns provides better comprehensibility, reusability, traceability, and evolvability of software artifacts [Elrad et al. 2001], and

[1]Another implementation termed *planned play-out* implements smart play-out using AI planning algorithms, see [Harel and Segall 2007].

its realization is an important achievement of the software engineering community. Most relevant to our work, however, is that the aspect-oriented paradigm and the scenario-based approach to software specification posses an important similarity: in both, part of the system's behavior is specified in a way that explicitly crosses the boundaries between objects.

One may view the current article as taking advantage of this similarity in order to construct a new way of executing systems that are based on the inter-object, scenario-based paradigm. More specifically, we show how to transform LSCs onto AspectJ, thus providing a non-interpreted means for executing multimodal scenarios. Our transformation/compilation can also be viewed as full code generation from a visual formalism, similar to that carried out by tools like Statemate [Harel and Naamad 1996; Harel et al. 1990], Rhapsody [Harel and Gery 1997],[2] and RoseRT.[3] This is in contrast to the skeletal, template code generated by many other CASE tools, requiring the reactive behavior to be coded separately. In addition, compiling LSCs into runnable code in an accepted programming language has the advantage of making it possible to include the scenario-based approach in an overall system development effort, among other things, enabling the formalization of crosscutting use cases [Jacobson and Ng 2004], and enabling links with standard tools and development environments.

In the rest of the article, we assume a hybrid approach to system modeling and execution, where each component may have its own intra-object behavior specified and implemented, and where scenario-based inter-object specifications are intended to specify additional behaviors of the system. Thus, unlike the common approach to synthesis, where scenario-based specifications are translated into state-based specifications for each participating component *before* they are simulated or executed, we follow the ideas behind the play-out algorithm from Harel and Marelly [2003a] to coordinate the simultaneous direct execution of the scenarios together with the execution of the separately specified and implemented intra-object, possibly state-based, behaviors. Of course, a special case is one where none of the objects/components has been given its own intra-object, state-based behavior (except for internal methods), so that the entire interactive nature of the system is specified in an inter-object fashion.

Our process of high-level compilation takes scenario-based specifications given in LSC and transforms them into what we shall be calling *scenario aspects*, implemented as AspectJ aspects.[4] A special aspect, called the *coordinator*, is also generated, and is responsible for coordinating the execution of the different, possibly interdependent, scenarios, according to the play-out operational semantics. The generated aspect code, consisting of the scenario aspects and coordinator aspect, is then compiled/linked with an existing or separately implemented Java program code to create a single executable application.

The use of aspects allows us to carry the scenario-based specification over from the model to the code, while still eventually producing a single standard executable program. That said, an important benefit of our approach is that the structure of the specification is reflected in the structure of the generated code. Since each generated scenario aspect is responsible for a single inter-object scenario-based chart, which can

[2]Rhapsody. IBM Rational (previously I-Logix, Telelogic) Rhapsody.
http://www.telelogic.com/products/rhapsody/index.cfm.
[3]RoseRT. IBM Rational Rose Technical Developer (includes Rational Rose RealTime).
http://www-306.ibm.com/software/awdtools/developer/technical/.
[4]We chose AspectJ because it is the most popular implementation of aspects to date and it suffices for our needs. However, our transformation scheme is general and can be adopted to compile to other AOP languages (see Section 6). A comparative discussion of other AOP languages with regard to our compilation scheme is beyond the scope of this article.

be viewed as a single requirement, the high-level inter-object requirements and their structure are not lost in the translation. This may have important benefits in the later stages of the development cycle, during testing and maintenance. We touch upon this facet of our work in Section 7.

A scenario-based specification model typically consists of many interdependent scenarios. Thus, we note that our work does not require its input scenarios to be independent. Indeed, one of the fundamental ideas in LSCs, which is reflected in the play-out mechanisms and their implementations, is that the LSCs (or appropriate groups of LSCs; sometimes those will be formally clustered into use cases) form an integral whole, influencing and triggering each other in many ways. Thus, one of the roles of our coordination aspect is indeed to resolve (some of the) possible conflicts between the various requirements, if any. Play-out is explained in detail in Section 2.2. A discussion of the challenges in coordinating the execution of inter-dependent scenarios appears in Section 7.1.

It is important to note that play-out is neither superior nor inferior to synthesis. Each approach has its advantages and limitations. One of the advantages of play-out is the fact that the structure of the scenario-based specification is reflected in the generated code and is not lost in the translation. Another advantage is the low complexity of compilation relative to synthesis. Play-out's major limitation relative to synthesis is its poorer expressive power; it defines a weaker semantics, without look-ahead or with limited look-ahead, and thus, compilation does not guarantee deadlock-free execution. In Section 7.1 we discuss the various play-out semantics, their advantages and their limitations.

An implementation of play-out as defined in our work requires a mechanism to support four major features. First, unification, which is the ability to recognize events as they happen in the system and to identify them with corresponding model-level events. Second, dynamic binding, which is the ability to dynamically link objects in the system to lifelines representing them in the model. Third, direct execution, which is the ability to execute system events directly. And fourth, coordination, which is the ability to reason about the system's global state, taking the different interdependent requirements into consideration, in order to proactively decide on the next event to be executed. In Section 3 we describe our compilation scheme in detail and show how it provides these necessary features.

Our transformation scheme is fully implemented in a UML2-compliant tool we term the *S2A compiler* (standing for *Scenarios to Aspects*, and first presented and demonstrated in Harel et al. [2007]). S2A provides the full executable code for the generation of reactive behaviors from inter-object multimodal scenarios. It supports advanced scenario-based features, such as concrete and symbolic parameters, class inheritance, interface implementation, and dynamic creation of objects. We consider S2A to be a test bed for experiments in scenario-based programming. The code snippets throughout the article are all examples of aspect code generated by S2A. An overview of S2A's implementation appears in Section 4.

Finally, our work is part of a more general and long-term effort around the play-in/play-out approach [Harel and Marelly 2003a, 2003b] and the vision of *liberating programming* presented by Harel [2008]. These aim at bridging the gap between requirements and execution, blurring the distinction between users, designers, and programmers, and ultimately freeing programmers from the need to write down a program as a textual artifact, from the need to specify requirements (the what) separately from the program (the how) and to pit one against the other, and from the need to structure behavior according to the system's structure, providing each piece or object with its own behavior. We consider our current work to be a step towards this far-reaching dream.

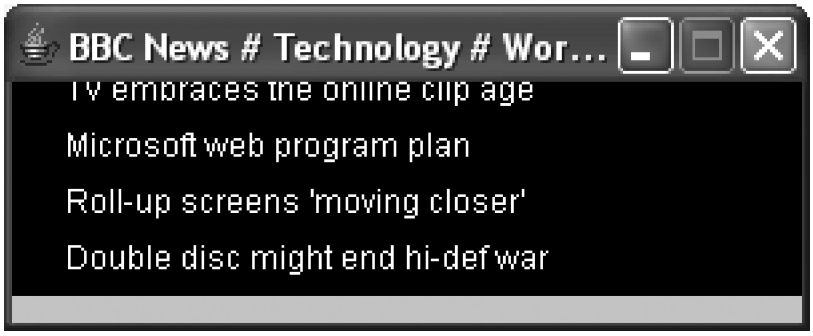

Fig. 1. The RSS News Ticker GUI.

1.1 Example Application: An RSS News Ticker

Throughout this article, we use a running example to illustrate the issues we discuss. The example is an RSS News Ticker, a small desktop application that reads RSS news feeds from selected news websites and presents news headlines in a continually running display. When the user clicks a headline, a web browser opens to show the corresponding article. In addition, the Ticker allows the user to set the URL addresses for the news feeds, to switch between horizontal and vertical scrolling displays, and to control the scrolling speed. It also checks periodically for news feed updates and displays a thin red/green blinking bar to indicate the application's state during updates. Figure 1 displays a screenshot of the Ticker's GUI.

The RSS News Ticker was constructed using the S2A compiler. The classes of the application were written manually in Java, including their inner behavior. However, the interaction between the classes was specified using the UML2-compliant variant of the LSC language, using the *modal* profile [Harel and Maoz 2008], and compiled into aspects using the S2A compiler. The application's code, including the generated code and its UML model, which includes the LSC specification, can all be downloaded from the S2A Website.[5]

We now briefly describe the Ticker's architecture, at a level sufficient for understanding the examples in the article. Part of the Ticker's class diagram appears in Figure 2. The central class of the application is the `RSSDisplayFrame` class, which uses several other classes and contains the GUI data and its related operations. Among these classes are three timers, the `RSSFeedTickerTimer`, the `BlinkerTickerTimer`, and the `ScrollTickerTimer`. The first is used to time and trigger the reading of RSS feeds from the user-configured websites. The second is used to control the color switch between red and green in the thin bar. Finally, the third timer is used to control the scrolling of the headlines displayed in the frame. All three timers extend an abstract timer class called `AbstractTickerTimer`.

The `RSSDisplay` is another class used by the `RSSDisplayFrame` class. It has two subclasses, `VerticalRSSDisplay` and `HorizontalRSSDisplay`. This structure allows dynamic switching between two possible views, one where the headlines move from the bottom of the frame upward, and the other where the headlines move from right to left.

The `PopupListener` and `RSSList` classes complete the list of classes used by the `RSSDisplayFrame` class. The first is responsible for the popup context menu's GUI and operations, and the second stores the parsed RSS items (the headlines).

[5]S2A Website. http://www.wisdom.weizmann.ac.il/~maoz/s2a/.

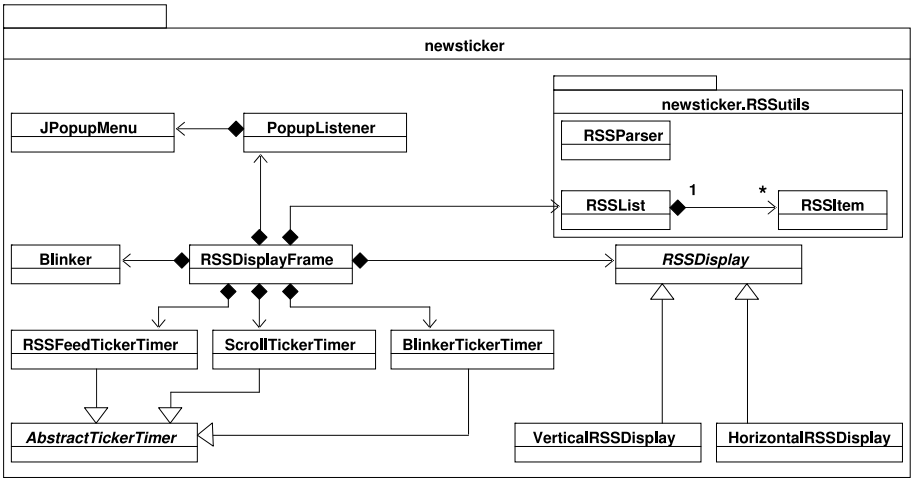

Fig. 2. Part of the RSS News Ticker class diagram.

1.2 Article Organization

The article is organized as follows. We start in Section 2 with a short review of the LSC language, the play-out operational semantics, and the key concepts of AOP relevant to our work. In Section 3 we present the transformation/compilation scheme's main ideas. Section 4 gives a short overview of the scheme's implementation in the S2A compiler. Section 5 elaborates on our support (or lack thereof) for some of the more advanced features of the LSC language. Section 6 describes some early applications and extensions, and Section 7 provides analysis and a critical evaluation of various facets of our work. Related work is discussed in Section 8. We conclude in Section 9 with a summary and directions for future work.

2. PRELIMINARIES

2.1 Live Sequence Charts

Live sequence charts (LSC) [Damm and Harel 2001; Harel and Marelly 2003a] constitute a visual formalism for inter-object scenario-based specification and programming, which extends the partial-order semantics of classical message sequence charts (MSC) [ITU 1996] mainly by adding universal and existential modalities. It thus allows the definition of inter-object scenarios that specify, among other things, possible, mandatory, and forbidden behavior.

LSC was introduced in Damm and Harel [2001]. A translation of LSC into various temporal logics appears in Kugler et al. [2005]. The language has been the subject of research in the areas of verification and testing (see e.g., Klose et al. 2006, Lettrari and Klose 2001), and in the areas of scenario-based execution (play-out) and synthesis [Harel and Kugler 2002; Harel and Marelly 2003a]. Initial projects that use LSC have been carried out recently in the automotive, telecommunication, and hardware domains [Bunker et al. 2005; Combes et al. 2008]. In the following, we start off with a very basic description of the language. We then continue to the specific variant used in our work.

Syntactically, as in MSCs, an LSC consists of a set of *lifelines*, representing instances of system objects, and a set of events, mainly *methods* and *conditions*. Lifelines are drawn using vertical lines; methods and conditions use horizontal arrows and hexagons, respectively. Every lifeline contains locations, each of which is an

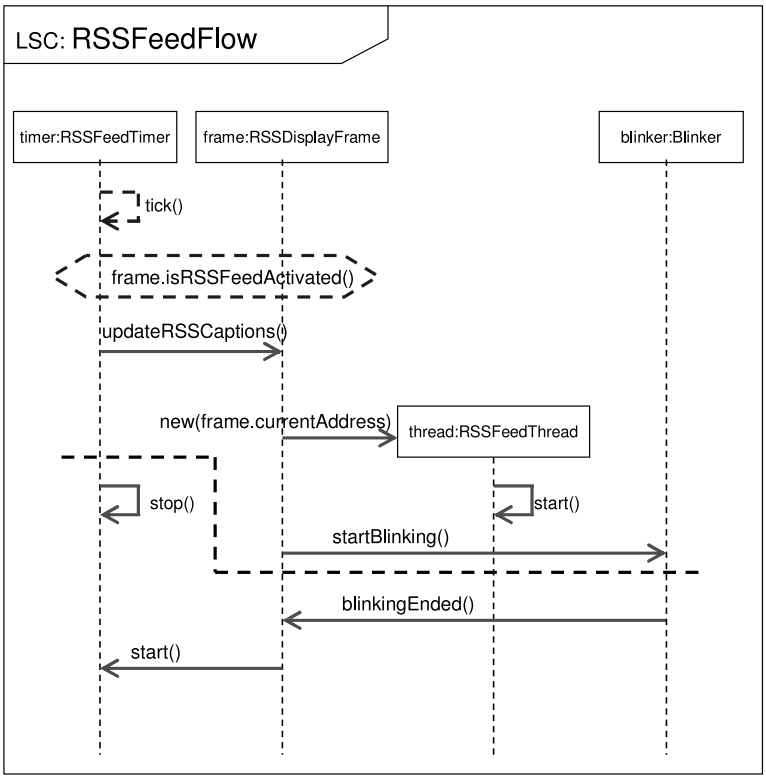

Fig. 3. The `RSSFeedFlow` LSC. One of the scenarios in the RSS News Ticker program. The dashed black line represents a cut.

intersection of lifeline with an event. A method event defines two locations (one in case of a self call). A condition event defines one or more locations. All instance references whose attributes or methods are used in the condition expression must synchronize on its evaluation; other instance references may synchronize too. Most importantly, in LSC, each event has a *mode*, which can be hot or cold. Cold events are drawn using a blue line while hot events use red lines.

An example LSC named `RSSFeedFlow`, taken from the RSS News Ticker specification, is shown in Figure 3. Its lifelines represent an `RSSFeedTimer` object, an `RSSDisplayFrame` object, an `RSSFeedThread` object, and a `Blinker` object. It contains many method events and one condition. Note the mode of the different events in the LSC.

Semantically, LSCs are interpreted over infinite traces of events. Like classical MSCs, an LSC defines a partial-order on its events. Events covering the same lifeline are fully ordered from top to bottom but the order between events covering different lifelines is otherwise not restricted. An important concept in the semantics of LSC is the *cut*, which is a mapping from each lifeline to one of the event locations covering it. A cut represents a state in the scenario's progress. The partial-order on events is naturally extended to a partial-order between cuts.

In Figure 3, a cut has been drawn using a dashed black line just before the `RSSFeedTimer` `stop()` method and after the `Blinker` `startBlinking()` method called by the `RSSDisplayFrame`. Note that the order between the `stop()` method and the `blinkingEnded()` method (as well as other methods before it) is not defined. That is,

according to the partial-order defined by the LSC, `stop()` may occur any time between the `updateRSSCaptions()` method and the `start()` method of the timer.

An LSC cut defines a set of *enabled events*; those immediately following it (i.e., directly below it) in the partial-order defined by the chart. All other events that appear in the chart and are not currently enabled are considered *violating events*. In addition, the mode of events is extended to cuts as follows: a cut is hot if at least one of its enabled events is hot, and is cold otherwise.

The enabled events induced by the cut shown in Figure 3 are `stop()` and `blinkingEnded()`. The cut is hot, since it has at least one hot enabled event; in fact, here both enabled events are hot.

Most importantly, the mode of an event (and a cut) carries a semantic meaning over and above the partial-order. In every run of the system, whenever cold events from a chart occur in the trace in the order specified by the chart (and cold conditions evaluate to true), subsequent hot events must eventually occur in the trace in the order specified by the chart (and subsequent hot conditions must evaluate to true). Thus, hot cuts may be viewed as unstable states, in which the LSC is not allowed to stay forever.

A trace-based semantics for LSC was defined in Harel and Maoz [2008] using alternating weak word automata [Kupferman and Vardi 2001]. Each LSC gives rise to an automaton. The language accepted by the automaton is the language defined by the LSC: a run satisfies an LSC iff it is accepted by its automaton. The semantics of an LSC specification, consisting of a set of (universal) LSCs, is defined as the intersection of the trace-languages defined by the LSCs in the set.

2.2 Play-Out

As described in Harel and Marelly [2003a, 2003b], the play-out execution mechanism aims at directly executing an LSC specification consisting of a set of LSCs. Basically, the execution mechanism reacts to system events that are statically referenced in one or more of the LSCs.

2.2.1 Execution and Monitoring. In the variant of LSCs used in the present work, an *execution mode* is added to methods in the language, which can be either *monitor* or *execute*. Monitored methods are drawn using dashed lines while execute methods use solid lines.

In the example `RSSFeedFlow` LSC of Figure 3, all cold methods have execution mode monitor and all hot methods have execution mode execute.

2.2.2 LSC Lifelines. System vs. environment, concrete vs. symbolic. An LSC lifeline has a name, a type, and two modes: *systemMode*, which can be either *system* or *environment*, and *symbolicMode*, which can be either *concrete* or *symbolic*.

Thus, an LSC lifeline may represent system or environment objects. System objects are the ones controlled by the play-out mechanism. Environment objects are external to the play-out execution mechanism. They can be monitored, but play-out cannot drive their execution (i.e., it cannot make them call a method).

In addition, LSC lifelines are either concrete or symbolic. Concrete lifelines represent concrete objects. Their binding is statically defined (in a configuration file) and is common to all instances of the LSC to which they belong. A symbolic lifeline is labeled with a class or an interface name; each of them may represent any object whose class directly or indirectly inherits from this class or implements this interface. These lifelines are dynamically bound to concrete objects participating in the scenario, during play-out. A symbolic lifeline may bind to different objects in different instances of the same LSC.

In the example of Figure 3, all lifelines are system lifelines; the frame and the blinker lifelines are concrete, while the timer and thread lifelines are symbolic.

2.2.3 LSC Methods. An LSC method covers two lifelines, representing its caller and callee objects. An LSC method may have parameters of three kinds: symbolic, opaque, and exact. In Section 3 we describe the compilation scheme while ignoring method parameters. Support for methods with different kinds of parameters is discussed in Section 5.1.

2.2.4 LSC Conditions. An LSC condition holds a Boolean expression. It is specified in our variant of LSC using a UML2 state-invariant. A condition may cover one or more lifelines. The covered lifelines are those that represent objects that participate in the evaluation of the expression or need to synchronize on it (note the special case where a constant true condition is used as a synchronization construct between a number of lifelines).

In the example of Figure 3, the (cold) condition expression is `frame.isRSSFeed-Activated()`. It synchronizes on both timer and frame lifelines.

2.2.5 Events Unification and Lifeline Binding. Play-out requires a careful mechanism for event unification and dynamic binding.

Roughly, an occurrence of a method in the system is *unifiable* with an LSC method with the same signature in an LSC instance if each of its (concrete or symbolic) lifelines is either (1) already bound to the caller (or callee) object of the method, or (2) symbolic and still unbound but representing a class (or an interface) that the caller (or callee) inherit from (or implements).

As part of play-out, once an LSC method in an LSC instance unifies with an executed method, its lifelines are dynamically bound to the caller and callee of the method. Once bound, lifeline binding remains so until the LSC instance is closed.

When methods with parameters are considered, an additional unification condition requires that corresponding parameters have equal values, or that the symbolic parameter is free (i.e., it is not yet bound or assigned a value).

2.2.6 Play-Out Algorithm. Whenever a method is invoked in the system, play-out checks whether it is unifiable with any LSC method in one of the LSCs and LSC instances, if any. If it is unifiable and is minimal in the partial order defined by one of the LSCs, a new instance of that LSC is created, and its cut advances from the minimal cut to the locations just after the event. If it is unifiable and is enabled in an LSC instance, play-out advances the cut of the LSC instance accordingly. If it is unifiable and is violating in an LSC instance, play-out checks the mode of the current cut of this LSC instance: if the cut is hot, the violation is a *hot violation* and an exception is thrown, as this should never happen; if the cut is cold, the violation is a *cold violation*, and the scenario instance is discarded. If it is not unifiable it is ignored.

Conditions are evaluated as soon as they are enabled in a cut; if a condition evaluates to true, the cut advances accordingly to the locations just after the condition; if it evaluates to false and the current cut is cold, the LSC instance is discarded; if it evaluates to false and the current cut is hot, an appropriate exception is thrown, as this should never happen.

If the cut of an LSC instance reaches maximal locations on all lifelines, the instance is discarded.

Most importantly, once all LSC cuts have been updated following an occurrence of an event (and the evaluation of enabled conditions, if any), the play-out execution mechanism chooses an event to execute from among the events that are currently enabled in at least one of the charts, have execution mode `execute`, and are not violating in any

of the charts whose cut is hot. The choice depends on a strategy. The naive play-out strategy (as implemented in our work) arbitrarily chooses one of these methods, that is, an execution-enabled method that is not violating any chart, if any. If the strategy returns a method, play-out executes it. Play-out continues iteratively, to listen and react to system events, ad infinitum.

2.2.7 Remarks. In our settings, we identify LSC methods with Java methods. We also consider only the synchronized semantics of LSC. That is, we assume that the caller of the method is blocked until the method is executed. Another general assumption made by the play-out mechanism is that it is infinitely faster than the system; that is, any series of operations that the play-out mechanism should execute as a reaction to an event in the system will be fully executed before another relevant system event occurs.

Finally, a note for the reader familiar with the original version of the language, as defined in Damm and Harel [2001] and Harel and Marelly [2003a]. The UML2-compliant version defined in Harel and Maoz [2008] and used here is a slightly more uniform and generalized variant of the original one. Specifically, cold monitor fragments are like pre-charts. Also, since the mode and the execution mode are orthogonal, we are able to specify cold-execution methods. This adds some flexibility that was not possible in Harel and Marelly [2003a].

Full definitions of the play-out algorithm for LSCs, including unification, can be found in Harel and Marelly [2003a]. Note that our definition of unification and lifeline binding has a polymorphic flavor not included in Harel and Marelly [2003a]. A full definition of the polymorphic binding semantics can be found in Maoz [2009b].

2.2.8 Example. In the context of play-out, the `RSSFeedFlow` LSC shown in Figure 3 details and formalizes the following behavior.

> *RSSFeedFlow*. Whenever the `RSSFeedTimer` ticks (its `tick()` method is executed), a condition checking whether RSS feed is currently activated is evaluated. If it is activated, the timer tells the frame to update the RSS captions. Then the frame creates a new `RSSFeedThread` (that will call its own `start()` method), and tells the blinker to blink. In the meantime, the timer calls its own `stop()` method. Eventually, the blinker tells the frame that blinking has ended (by calling its `blinkingEnded()` method), and the frame tells the timer to start ticking again (by calling its `start()` method).

Figure 4 shows another LSC from the Ticker's specification. This LSC details and formalizes the following behavior.

> *Blinking*. Whenever the `frame` tells the `blinker` to start blinking, the latter creates a `blinkerTickerTimer` and sets its interval to 500 milliseconds. Next, the `blinker` sets its own `blinking` attribute to `true` and tells the `frame` (which started this scenario) that blinking has started; meanwhile, the newly created `blinkerTickerTimer` calls its own `start` method. Next, for at most 5 times, whenever the timer ticks, it tells the `blinker` to switch its color. Finally, the `blinker` stops the timer it has created and informs the `frame` that blinking has ended.

Note that the scenario-based requirements specified by these two LSCs are not independent. Specifically, in the `RSSFeedFlow` LSC, immediately after the call to `startBlinking()`, `blinkingEnded()` is execution-enabled (regardless of whether the `stop()` method of the `RSSFeedTimer` has been executed yet or not). Still, in the `Blinking()` LSC, after the call to `startBlinking()`, `blinkingEnded()` is violating for

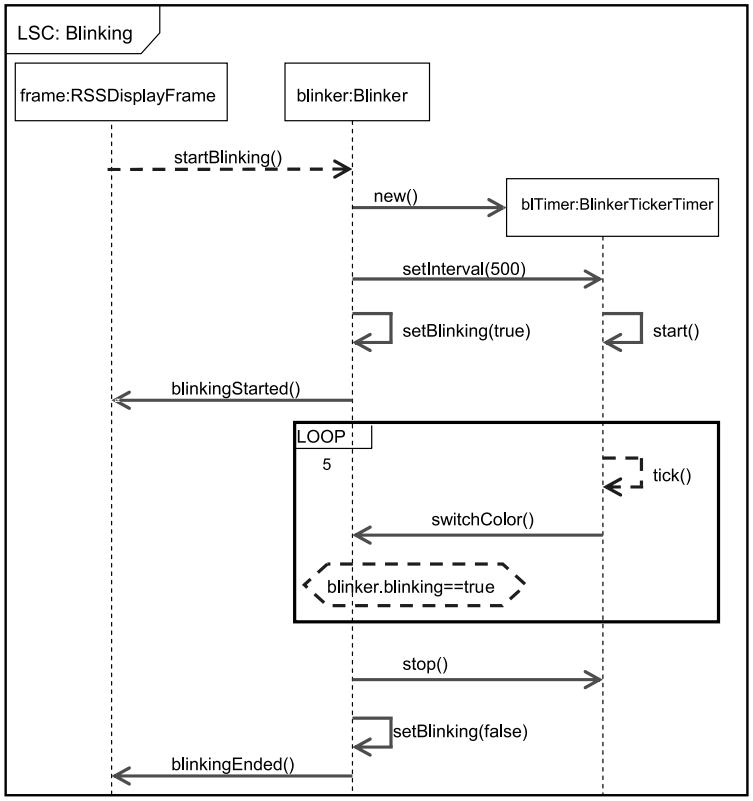

Fig. 4. The `Blinking` LSC. One of the scenarios in the RSS News Ticker program.

most of the scenario, until it is finally enabled just before the last cut. Thus, play-out coordination is required. The play-out strategy we use guarantees `blinkingEnded()` will not be executed until it is not violating in any of the LSCs.

The above is a rather simple example for the coordination required during execution, that is, play-out. Our compilation scheme, described in Section 3, indeed supports this kind of coordination.

2.3 Aspect Oriented Programming and AspectJ

Aspect-oriented programming (AOP) has been proposed as a mechanism that enables the modular implementation of crosscutting concerns [Kiczales et al. 1997]. An aspect can be thought of as a special kind of object that observes a base program and reacts to certain actions by running extra code of its own. The most popular implementation of AOP is AspectJ[6] [Kiczales et al. 2001], an extension of Java. Below we briefly describe the features of AOP, focusing on those used in our work. We use AspectJ terminology.

Dynamic crosscutting is the weaving of new behavior into the execution of a program. A *join point* is a certain well-defined point in the execution of a program, such as a call to a method or an assignment to a member of an object. A *pointcut*, defined using the keyword `pointcut`, is a program construct that designates a set

[6]Eclipse AspectJ. The AspectJ project at Eclipse.org. http://www.eclipse.org/aspectj.

```
public aspect VerySimpleAspect {

  //Pointcut
  pointcut methodCall() : call(void MyClass.m(..));

  //Advice
  after() : methodCall() {
        System.out.println("A method call to m occurred.");
      }
}
```

Fig. 5. A very simple aspect with a pointcut and an after advice.

of join points, plus, optionally, values from their execution context. For example, a pointcut can capture the execution of a certain method along with its arguments and a reference to its caller and callee, using the keywords `args`, `this`, and `target`. Pointcuts can be combined using Boolean operators. Wildcard-based syntax is used in order to construct pointcuts that capture join points sharing common characteristics.

To declare the code that should execute at a join point selected by a pointcut, AspectJ supports *advice* constructs: *before*, *after*, and *around*. A before advice executes prior to the join point, an after advice executes following the join point, and an around advice surrounds the join point's execution and allows to bypass execution, continue the original execution, or cause execution with an altered context. An advice may have access to the context captured by its pointcut.

Static crosscutting is the weaving of modifications to the static structure of the program. An *intertype declaration* is a static crosscutting construct that enables the introduction of new methods or members to a class.

Finally, an *aspect* is the central unit of AspectJ. It is defined by an aspect declaration, similar to that of a class declaration, using the `aspect` keyword. An aspect typically includes pointcuts, advice, and inter-type declarations, as well as other kinds of declarations such as members and methods permitted in class declarations. A `declare precedence` keyword can be used to specify an order of execution when two or more aspects apply to the same joinpoint. Our compilation scheme takes advantage of these features.

A very simple example of an aspect appears in Figure 5. The figure shows an aspect with a pointcut and an after advice. A thorough description of AspectJ syntax and semantics, with examples, can be found in Kiczales et al. [2001].

3. THE TRANSFORMATION/COMPILATION SCHEME

We are now ready to present the compilation scheme. We will illustrate it using another LSC from the Ticker's program: the `ChangeToHorizontalDisplay` LSC, shown in Figure 6, which specifies a scenario where a user action triggers the change of the Ticker application's display to be horizontal.

The key to the transformation/compilation scheme is the translation of each LSC into a *scenario aspect*. The scenario-aspect code simulates an abstract automaton whose states represent cuts along the LSC lifelines and whose transitions represent enabled events. Each scenario-aspect is locally responsible for listening out for relevant events and advancing its cut state accordingly. Most importantly, the compilation scheme generates a *coordinator*, implemented as another, separate aspect, which observes the cut-state changes of all active scenario aspects, uses a strategy to choose a method, and upon making a choice executes the method.

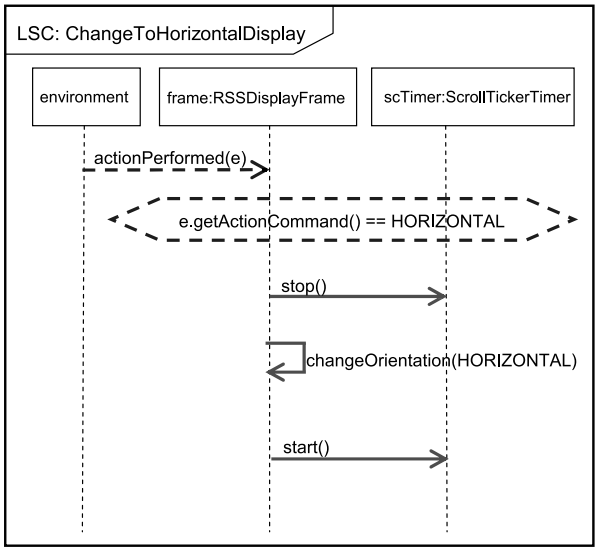

Fig. 6. The `ChangeToHorizontalDisplay` LSC. To make the figure more readable the expression within the condition uses pseudo code. The actual Java expression inside the cold condition is "`e.getActionCommand().equals(frame.HORIZONTAL)`."

3.1 Scenario Aspects

Each LSC is translated into a *scenario aspect*, which is responsible for monitoring the events relevant to the LSC instances, following their cut changes, reporting on violations and completions, and, as necessary, allowing the coordinator to query their current enabled and violating events. To address these responsibilities, the scenario-aspect code simulates an abstract automaton whose states represent cuts along the LSC lifeline and whose transitions represent enabled events. The scenario aspect listens out for relevant events, using pointcuts, and advances its cut state accordingly, using advice.

3.1.1 Building the Automaton. Building the automaton representation of an LSC requires a static analysis of the LSC. The analysis involves simulating a "run" over the LSC, to capture all its possible cuts. Each cut is represented by a state. Transitions between states correspond to enabled events. An additional transition from each cold cut state to a designated completion state corresponds to all possible (cold) violations at the cut. An additional transition from each hot cut state to a designated error state corresponds to all possible (hot) violations at the cut. During the automaton's construction, the sets of execution-enabled, monitoring-enabled, cold violation, and hot violation events at each cut are computed and stored in the state.

Note that, since the construction of the automaton does not require information from other LSCs, the compilation of each LSC is independent of the rest of the specification. Thus, scenario aspect code generation can be carried out "locally"; the coordination between LSCs is handled by the coordinator aspect.

Note also that the distinction between monitoring and executing methods is not represented in the structure of the automaton, but only in the information stored in each state.

Figure 7 shows the automaton representing the LSC from Figure 6, namely `ChangeToHorizontalDisplay`. Following Harel and Maoz [2008], this is an alternating weak word automaton [Kupferman and Vardi 2001]. Note the universal quantification

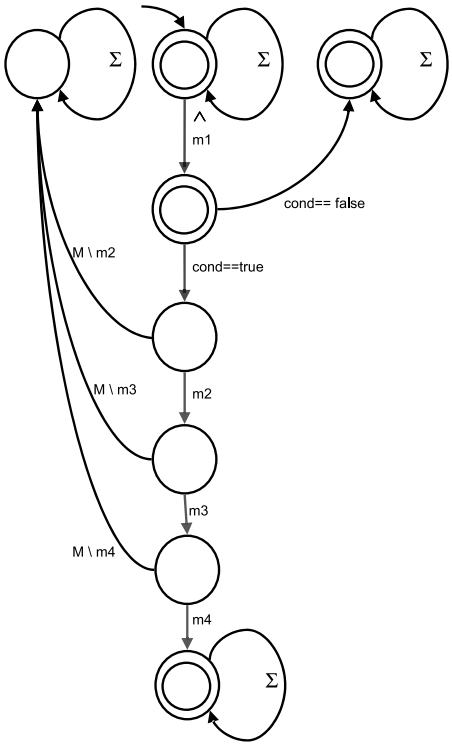

Fig. 7. The automaton for LSC `ChangeToHorizontalDisplay`. `M` is the set of messages in the LSC, `m1` stands for `actionPerformed(e)`, `m2` stands for `stop()`, etc., and `cond` stands for `e.getActionCommand()== HORIZONTAL`. Self transitions labeled $\Sigma \setminus M$ have been omitted. The $\wedge$ symbol attached to the edges emanating from the initial state indicates universal quantification.

on the outgoing transitions of the initial state (indicated using the $\wedge$ symbol), which reflects the fact that multiple copies of the automaton may be active simultaneously (although for this specific LSC, this is never the case). For the formal construction of the automaton see Harel and Maoz [2008].

Analogous automata representations of LSC for the use in the context of formal verification were given by, for instance, Bontemps and Heymans [2002] and Klose and Wittke [2001]. In Harel and Maoz [2008], we used a similar automaton construction to define the semantics of LSC. The construction yields an alternating weak automaton, where the partition of the states is induced by the partial-order of events specified in the LSC.

For each LSC in the specification, following the construction of the LSC's automaton, code for the scenario aspect is generated. Roughly, the structure of the code includes three main parts, as described in the following sections.

3.1.2 Pointcuts and Advice Code. Each method of the LSC is translated into a `pointcut` that captures its execution, together with the context of the calling object `this`, the callee object `target`, and its arguments `args` (as applicable). An `after` advice is associated with each pointcut. The advice calls a private method `changeCutState()` with the LSC method's context in its arguments (as follows). For example, the method `stop()` in the `ChangeToHorizontalDisplay` of Figure 6 is translated into the pointcut and advice shown in Figure 8.

```
pointcut RSSDisplayFrame_ScrollTickerTimer_stop(RSSDisplayFrame rSSDisplayFrame,
                ScrollTickerTimer scrollTickerTimer):
        call(void ScrollTickerTimer.stop(..))
        && target(scrollTickerTimer) && this(rSSDisplayFrame) ;

after(RSSDisplayFrame rSSDisplayFrame,ScrollTickerTimer scrollTickerTimer):
    RSSDisplayFrame_ScrollTickerTimer_stop(rSSDisplayFrame,scrollTickerTimer)
{
    changeCutState(LSCMethods.RSSDisplayFrame_ScrollTickerTimer_stop,
                rSSDisplayFrame,scrollTickerTimer,null);
}
```

Fig. 8. The generated pointcut and advice corresponding to the `stop()` method of the `ChangeToHorizontalDisplay` LSC shown in Figure 6.

Recall that according to LSC semantics, an LSC does not prescribe the occurrence or nonoccurrence during a run of methods not appearing in it, including between events that do appear explicitly in the LSC. The occurrence of such methods does not affect the LSC. Our use of specific pointcuts, one for each method in the LSC, reflects this semantics: the code of each scenario aspect is only aware of methods explicitly mentioned in its LSC.

Note that the advice must call the general method responsible for the automaton's transitions, `changeCutState()`, since the required behavior depends on the current cut states of the LSC's active instances. Each method is enabled in one or more cuts and is violating in others. Thus, we cannot localize the response and must call the general cut-state changing method where the automaton's transitions are encoded.

3.1.3 Cut-State Changing Code. The method `changeActiveLSCCutState()` is responsible for advancing the scenario-aspect cut state along the locations of each lifeline, and for identifying cold and hot violations as necessary. It is built as a series of switch-case statements. The method's arguments are an active LSC copy (see Section 5.3), an LSC method identifier, the LSC method's callee, the LSC method's caller, and the LSC method's actual parameter values.

Each case in the switch-case structure corresponds to an id of one LSC method. Inside each such case there are several blocks of code, one for each possible instance of this method in the chart (the same method may appear in the chart multiple times). Each block of code consists of two parts, the first contains tests for the current cut and possible unification, and the second contains the actions that take place if all the tests evaluate to true.

The cut test queries the current cut of the active LSC and compares it to the different possible cuts at which the method under consideration is enabled. The unification test validates the method's target and source types, checks whether the lifelines covered by the method are bound or not, and matches the method's parameters types and values.

Once all the tests have been carried out successfully, a series of actions occurs. This includes changing the cut-state of the active LSC instance, and, as necessary, binding the relevant objects (caller and callee, source and target) to the relevant lifelines, binding parameters, and evaluating conditions that became enabled after the corresponding LSC method was executed, if any.

Recall that according to LSC semantics, conditions are evaluated as soon as they are enabled. Thus, An LSC cut that includes an enabled condition is not represented as a state of the scenario-aspect code. Instead, the generated code ensures that as soon as the condition is enabled it will be evaluated, and the next cut state will be set accordingly.

```
protected void  changeActiveLSCCutState(int LSCm, Object sourceObject,
        Object targetObject,ActiveLSCAspect activeLSC,ArrayList args)
{
    //...
    switch (LSCm)
    {
        case LSCMethods.env_RSSDisplayFrame_actionPerformed:
            if(activeLSC.instancesEquals(RSSDisplayFrame_INST_frame,null))
            {
                unification=true;
                if(activeLSC.isInCut(0,0,0))
                {
                    activeLSC.setLineInstance(
                            RSSDisplayFrame_INST_frame,targetObject);
                    activeLSC.setPrivateVariable(
                            ActionEvent_arg_symbolic_e,args.get(0));
                    activeLSC.setCut(1,1,0);
                    if(evaluateCondition(1,activeLSC))
                    {
                        activeLSC.setCut(1,2,0);
                        return;
                    }
                    break;
                }
            }
    }
    //...
    if(activeLSC.checkViolation())
            activeLSC.completion();
    //...
}
```

Fig. 9. A snippet from the generated `changeActiveLSCCutState()`, which is responsible for cut changes. It shows the part of the generated code that runs after the execution of the `actionPerformed(e)` method.

If one of the tests evaluates to false, the cut-state does not change. Instead, the code checks for a violation. In the case of a cold violation or a successful LSC completion, the active copy in the context is discarded. In the case of a hot violation, an appropriate exception is thrown.

The code snippet in Figure 9 shows part of the `changeActiveLSCCutState()` method. It checks whether an execution of the method `actionPerformed(e)` can be unified with the static instance of this method as specified in the LSC, and whether the method is enabled with respect to the current active instance's cut. If all tests are positive, the `RSSDisplayFrame` lifeline and the LSC variable *e* of type `ActionEvent` bind, the active instance's cut-state is advanced, and the next condition in the partial order defined by the LSC is evaluated. If one of the tests fails, it may lead to a violation, as indicated in the two last lines of the snippet.

3.1.4 Cut-Information Exposing Code. As explained earlier, one of the responsibilities of the scenario aspect is to expose cut-state information so that the coordinator can query the current enabled and violating events of the LSC instances. The generated method `getActiveLSCCutState()` handles this responsibility; it gets an active LSC instance as a parameter, and returns the required information.

The method assigns instances of LSC methods, including context, among four method sets: execution-enabled, monitoring-enabled, cold violations, and hot violations. The sets are defined based on the current cut of the LSC instance received as a parameter.

```
protected void getActiveLSCCutState(LSCMethodSet ME,LSCMethodSet EE,
    LSCMethodSet CV,LSCMethodSet HV,ActiveLSCAspect activeLSC)
{
    RSSDisplayFrame frame = (RSSDisplayFrame)activeLSC.
        getLineInstance(RSSDisplayFrame_INST_frame);
    ScrollTickerTimer scrollTickerTimer = (ScrollTickerTimer)activeLSC.
        getLineInstance(ScrollTickerTimer_INST_scrollTickerTimer);
    Integer VAR_exact_or0 = (Integer) activeLSC.
        getPrivateVariable(int_arg_exact_or0);
    ActionEvent VAR_symbolic_e = (ActionEvent) activeLSC.
        getPrivateVariable(ActionEvent_arg_symbolic_e);
    //...
    ArrayList<Object>  args37 = getArgsList(VAR_symbolic_e);
    LSCMethod LSCm37 = new LSCMethod(null,frame,
        LSCMethods.env_RSSDisplayFrame_actionPerformed,
            args37);// actionPerformed() Monitored
    LSCMethod LSCm38 = new LSCMethod(frame,scrollTickerTimer,
        LSCMethods.RSSDisplayFrame_ScrollTickerTimer_stop,
            null);// stop() Monitored
    LSCMethod LSCm39 = new LSCMethod(frame,scrollTickerTimer,
        LSCMethods.RSSDisplayFrame_ScrollTickerTimer_start,
            null);// start() Execute
    ArrayList<Object>  args40 = getArgsList(VAR_exact_or0);
    LSCMethod LSCm40 = new LSCMethod(frame,frame,
        LSCMethods.RSSDisplayFrame_RSSDisplayFrame_changeOrientation,
            args40);// changeOrientation() Execute
    //...
    if(activeLSC.isInCut(0,0,0))
    {
            ME.add(LSCm37);
            CV.add(LSCm38,LSCm39,LSCm40);
            return;
    }
    if(activeLSC.isInCut(1,2,0))
    //...
```

Fig. 10. A snippet from the generated `getActiveLSCCutState()` method.

As the coordinator is not aware of the active LSC instances, the call from the coordinator to the `getActiveLSCCutState()` method must be done indirectly, through a method defined in the super class aspect `LSCAspect`. For further details on multiple instances see Section 5.3.

Figure 10 shows a code snippet of the `getActiveLSCCutState()` method, taken from the generated `ChangeToHorizontalDisplay` scenario aspect.

3.2 The Coordinator

The scenario-aspect automata are responsible for listening out for system events and advancing their cut accordingly. However, they do not drive the execution. Rather, it is the coordinator aspect that observes all cut-state changes in LSC instances, and is responsible for querying the scenario aspects regarding their cut states, choosing a method for execution, and executing it.

The coordinator code is implemented as a separate generated aspect. As a trigger for cut-state information collection, it uses a single pointcut defined as the disjunction of all the joinpoints that appear in all the generated scenario aspects. Thus, the

coordinator is made aware of any possible change in any of the scenario-aspects' cut state; that is, of each possible advance/violation/completion of any active LSC instance.

The after advice code collects cut-state information (sets of enabled and violating events, including dynamic context, i.e., bound objects, arguments values) from all active scenario aspects, and, when necessary, uses a *strategy* to choose a method for execution. If a method has been selected, the coordinator executes it using an inter-type declaration inside a generated wrapper method. The use of inter-type declarations for method execution is indeed necessary; to continue playing-out with correct unification and dynamic binding, scenario-aspect pointcuts that listen out for these methods, must correctly interpret their caller and callee objects (using the `this` and `target` keywords). For example, in the `ChangeToHorizontalDisplay` LSC, the `stop()` method of the `scrollTickerTimer` object must be called by the `RSSDisdplayFrame` object.

The strategy is responsible for choosing the next method to execute, based on the information that the coordinator has collected. In our current implementation, S2A includes a default play-out strategy that implements the basic (naïve) play-out algorithm of Harel and Marelly [2003a, 2003b], choosing a method for execution from among the ones enabled for execution in at least one chart and not hot-violating in any chart (see Section 2.2). An advanced user may define and integrate a new strategy, by implementing the required interface (called `IPlayOutStrategy`) and pointing S2A to the implementation in the compiler's configuration file.

Figure 11 shows a code snippet taken from the generated coordinator aspect of the RSS News Ticker application. Note that the coordinator aspect is declared with top precedence. This means that its after advice will be executed only after all advices in all the scenario aspects have been executed. We thus take advantage of the precedence feature of AspectJ. This is necessary in order to ensure that all LSCs update their cut state before the coordinator queries it.

3.3 Conclusion

To conclude the presentation of the transformation/compilation scheme, recall the four major features required to support the play-out mechanism as defined in our work (and as mentioned earlier in Section 1): unification, dynamic binding, direct execution, and coordination. In the following, we show how each of these features is handled in our scheme.

Unification, which is the ability to recognize events as they happen in the system and to identify them with corresponding model-level events, is addressed by our scheme using the scenario-aspect pointcuts (see Section 3.1.2).

Dynamic binding, which is the ability to dynamically link objects in the system to lifelines representing them in the model, is addressed by our scheme using the `this` and `target` keywords, which provide access to the caller and callee objects of the methods captured by the pointcuts, and using corresponding scenario-aspects variables, representing lifelines and keeping the references to the bound objects throughout the lifetime of each scenario instance (see Section 3.1.2).

Direct execution, which is the ability to execute system events directly, is implemented using intertype declarations inside generated wrapper methods in the coordinator aspect (see Section 3.2).

Finally, coordination, which is the ability to reason about the system's global state, taking the different interdependent requirements into consideration, in order to proactively decide on the next event to be executed, is supported through the scenario-aspect cut-state changing and information exposing code (see Sections 3.1.3 and 3.1.4), and the coordinator aspect's use of a strategy (see Section 3.2).

```
public aspect LSCCoordinatorAspect
{
    declare precedence: LSCCoordinatorAspect, *;
    ICoordinatorStrategy strategy = new CoordinatorStrategyNaiveImpl();
    //..
    public void RSSDisplayFrame.Wrapperstop(ScrollTickerTimer scrollTickerTimer){
        scrollTickerTimer.stop();
    }
    //..
    public void RSSDisplayFrame.WrapperchangeOrientation(
                                RSSDisplayFrame rSSDisplayFrame,int exact_or0){
        rSSDisplayFrame.changeOrientation(exact_or0);
    }
    //...
    pointcut LSCMessage():
        call(void ScrollTickerTimer.start(..))
        || call(void ScrollTickerTimer.stop(..))
    //..

    after (): LSCMessage(){
        LSCMethodSet monitoringEnabled = new LSCMethodSet();
        //...
        LSCAspectFrameInit.aspectOf().getCutState(monitoringEnabled,
                                executingEnabled, coldViolation, hotViolation);
        //...
        LSCAspectChangeToHorizontalOrientation.aspectOf().
                getCutState(monitoringEnabled, executingEnabled,
                                                coldViolation, hotViolation);
        //..
        LSCMethod LSCm = strategy.chooseMethod(monitoringEnabled,
                                executingEnabled, coldViolation, hotViolation);
        if(LSCm != null){
            switch (LSCm.messageID){
                //...
                case LSCMethods.RSSDisplayFrame_ScrollTickerTimer_stop:
                 ((RSSDisplayFrame)LSCm.sourceInstance).Wrapperstop(
                    (ScrollTickerTimer)LSCm.targetInstance);
                break;
                case LSCMethods.RSSDisplayFrame_RSSDisplayFrame_changeOrientation:
                 ((RSSDisplayFrame)LSCm.sourceInstance).WrapperchangeOrientation(
                    (RSSDisplayFrame)LSCm.targetInstance,(Integer)LSCm.args.get(0));
                break;
                //...
```

Fig. 11. A snippet from the generated coordinator aspect.

4. THE S2A COMPILER: IMPLEMENTATION OVERVIEW

We briefly review the general architecture and implementation of the S2A compiler. The input for the compiler is a valid UML2 model extended with the *modal* profile defined in Harel and Maoz [2008]. S2A reads the UML model in its standard *XMI* representation [UML 2005], and parses it to an in-memory data structure using the open source UML2 Eclipse API.[7] Thus, S2A does not depend on the specific tool that created the model; any UML2-compliant editor can be used to create the specification.

Figure 12 shows a screenshot of the IBM Rational Software Architect (RSA) IDE,[8] which allows editing of UML models with user defined profiles. The resulting UML model can then be exported to the standard XMI format.

S2A's output is a set of generated AspectJ aspects; one scenario aspect for each LSC in the specification, and a single coordinator aspect. After analysis and code generation, the AspectJ compiler is used to weave the generated aspect code into

[7]Eclipse UML2. http://www.eclipse.org/modeling/mdt/?project=uml2.
[8]IBM RSA. IBM Rational Software Architect. http://www306.ibm.com/software/awdtools/developer/technical/.

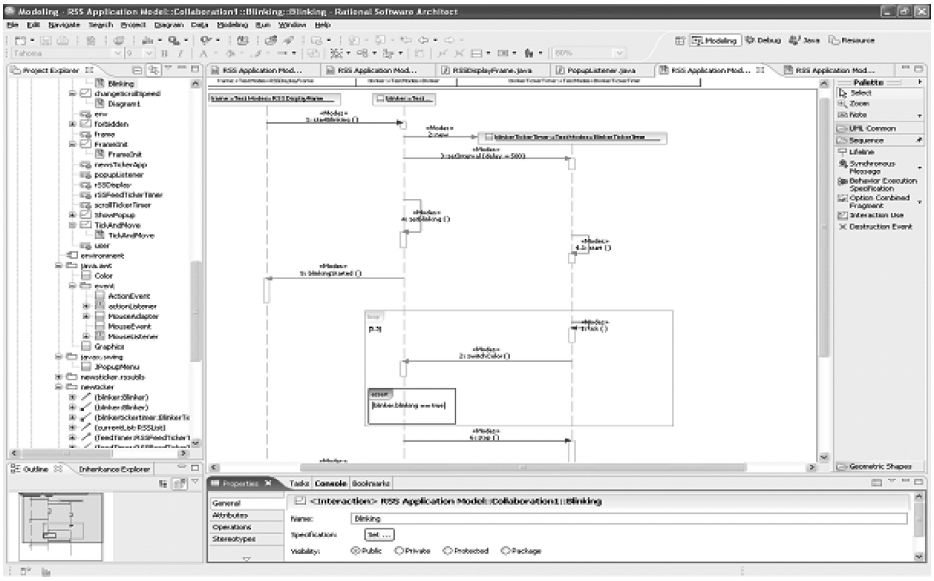

Fig. 12. A snapshot from the IBM RSA development environment, which allows the editing of a UML model with user defined profiles and thus makes it possible to create UML2-compliant LSC specifications. An LSC from the RSS News Ticker specification is shown in the main window.

existing application classes that were either coded manually or generated by some other tool.

To keep the generated code as compact and as readable as possible, the generated scenario aspects extend predefined aspects and classes residing in the S2A *runtime library*. The application's classes and the generated aspects must be compiled together with this library. This allows encoding the common generic behavior of an abstract scenario aspect in the superclass `LSCAspect` (which resides in the runtime library), and generating only the relevant behavior induced by the specific LSCs. See Section 7.2 for a discussion on the readability and usability of the generated code. The runtime library contains also the `IPlayOutStrategy` interface, its default naïve implementation, and some helper classes.

5. ADVANCED FEATURES

We now discuss some of the more advanced features of our version of LSCs, with their realization in the compilation scheme and their implementation in the S2A compiler. These include handling methods with different kinds of parameters, support for symbolic lifelines, multiple scenario instances, conditions, and hidden events. We use code snippets and example scenarios taken from the RSS News Ticker application to demonstrate these features.

5.1 Methods with Parameters

Our version of play-out, as realized in the compilation scheme, supports three kinds of method parameters: symbolic, opaque, and exact. We explain and demonstrate each of these below.

5.1.1 Symbolic Parameters. A symbolic parameter is a method parameter that is not given a value in the LSC, but only a name and a type. At runtime, when a method of the same signature occurs, the symbolic parameter binds to the actual value with which

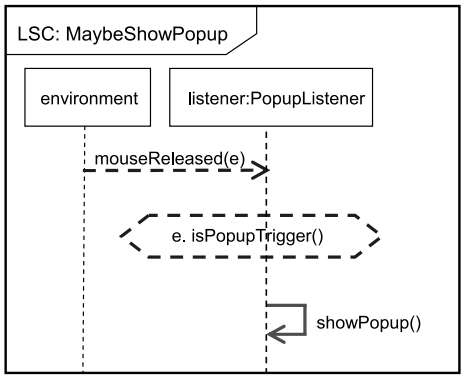

Fig. 13. The `MaybeShowPopup` LSC.

the method was executed in the program. Once bound to a value, it remains bound for the remainder of the LSC, for instance, when it appears as a parameter in another method parameter or within a boolean expression in a condition or a loop/alternative fragment guard. The use of methods with symbolic parameters enables the specification of succinct and generic scenarios [Marelly et al. 2002]. The exact unification and binding rules can be found in Harel and Marelly [2003a].

The compilation scheme supports the use of methods with symbolic parameters. To capture the dynamic context of a method's parameter we use the AspectJ keyword `args`. After doing so, the parameters values are stored in a list, which is part of the data sent to the `changeActiveLSCCutState()` method. The parameter values are considered in the unification test. If the cut tests and the unification tests are successful, these values bind to the corresponding statically defined fields of the appropriate `ActiveLSC` instance (see Section 5.3).

Consider, for example, the `MaybeShowPopup` LSC shown in Figure 13. The LSC specifies a scenario that is triggered when a user releases a mouse button over the application's frame. If the button is the right-hand one, then the popup menu is displayed. The method `mouseReleased(e)` contains a symbolic parameter *e* of type `MouseEvent`. The type of the parameter is not shown in the visual syntax of the diagram, but is part of the UML2 model behind it. When the scenario is triggered by the execution of a unifiable method, the LSC variable *e* binds to the actual value with which the method `mouseReleased(e)` was executed. In the condition that follows, *e* is already bound and can be used in the evaluation of the Boolean expression.

The code snippet in Figure 14 illustrates how S2A handles symbolic parameters. The parameter context is captured in the pointcut using the AspectJ keyword `args`. Later, in the `changeActiveLSCCutState()` method, the parameter binds to the active LSC field representing the corresponding LSC variable. The code also shows how the already bound variable can be (re)used in the remainder of the scenario. In this example, it is being used within the condition that follows (see the `evaluateConditions()` method).

5.1.2 Opaque Parameters. Another kind of parameter supported by the compilation scheme is the `OpaqueExpression`, which appears in the UML standard. When a method's parameter is defined as opaque, S2A copies the body of the expression to the appropriate place in the generated code without making any attempt to parse or to evaluate it. This allows the use of any Java expression as a method's parameter; reference to static constants, enumeration types, lifeline references or any other

```
public aspect LSCAspectMaybeShowPopup extends LSCAspect
{
    // ...
    pointcut env_PopupListener_mouseReleased(PopupListener popupListener,
            MouseEvent symbolic_e):
        execution(void PopupListener.mouseReleased(..))
        && target(popupListener)  && args(symbolic_e);

    after(PopupListener popupListener,MouseEvent symbolic_e):
        env_PopupListener_mouseReleased(popupListener, symbolic_e)
    {
        ArrayList<Object> args = getArgsList(symbolic_e);
        changeCutState(LSCMethods.env_PopupListener_mouseReleased,
                    null,popupListener,args);
    }
    //  ...
    protected void  changeActiveLSCCutState(int LSCm, Object sourceObject,
            Object targetObject,ActiveLSCAspect activeLSC,ArrayList args)
    {
        //  ...
        switch (LSCm)
        {
            //  ...
            case LSCMethods.env_PopupListener_mouseReleased:
                if(activeLSC.instancesEquals(PopupListener_INST_popupListener,
                        null))
                {
                    unification=true;
                    if(activeLSC.isInCut(0,0))
                    {
                        activeLSC.setLineInstance(
                                PopupListener_INST_popupListener,targetObject);
                        activeLSC.setPrivateVariable(
                                MouseEvent_arg_symbolic_e,args.get(0));
                        activeLSC.setCut(1,1);
                        if(evaluateCondition(2,activeLSC))
                        {
                            activeLSC.setCut(1,2);
                            //  ...
    // ...
    private boolean evaluateCondition(int conditionNumber,
            ActiveLSCAspect activeLSC)
    {
        PopupListener popupListener = (PopupListener)
                activeLSC.getLineInstance(PopupListener_INST_popupListener);
        MouseEvent e = (MouseEvent)
                activeLSC.getPrivateVariable(MouseEvent_arg_symbolic_e);
        switch (conditionNumber)
        {
            case 2:
                return e.isPopupTriggered();
        }
        return false;
    }
    // ...
```

Fig. 14. A snippet from the `MaybeShowPopup` scenario aspect code.

methods. The opaque expression is parsed only when the Java compiler and the AspectJ compiler compile the generated aspects. The content of any Boolean expression, inside conditions and guards, is also treated as an opaque expression (see Section 5.4 on conditions).

5.1.3 Exact Parameters. In addition to symbolic and opaque parameters, S2A supports exact parameters, which are constant values entered at design time. The call

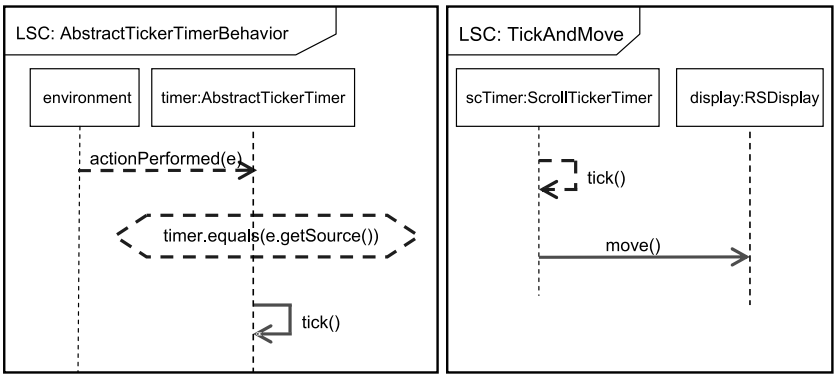

Fig. 15. The `AbstractTickerTimerBehavior` and the `TickAndMove` LSCs.

to `setInterval(500)` in the `Blinking` LSC shown in Figure 4 is an example of an LSC method with a constant parameter.

Note that exact parameters are not opaque. Their type must be specified, and their value is taken into consideration in unification tests.

5.2 Symbolic Lifelines

Symbolic lifelines and their play-out were defined in Marelly et al. [2002] and were originally implemented in the Play-Engine [Harel and Marelly 2003a]. The use of symbolic lifelines, that is, lifelines that represent sets of objects rather than specific objects, makes the language more expressive and succinct.

In the present work, we extended this feature of the language to support class inheritance (and interface implementation) in Java. That is, given an LSC with a symbolic lifeline l labeled with the class C, the set of objects that may bind to l contains not only objects of type C but also objects of any subtype of C.

To support symbolic lifelines with inheritance, S2A relies on Java and AspectJ dynamic binding mechanisms. Notice that by using these mechanisms we get "for free" also the ability to specify scenarios with lifelines that represent abstract classes and interfaces.

Consider for example the LSC `AbstractTickerTimerBehavior` of Figure 15 (left). This LSC has a symbolic lifeline representing an abstract class: the class `AbstractTickerTimer`. It thus specifies a behavior common to all objects deriving from a class inherited directly or indirectly from `AbstractTickerTimer`. During execution of the RSS Ticker application, there are three concrete objects that may be bound to this lifeline, that is, the three timers used in the application (see the class diagram in Figure 2). In fact, there can even be two or more instances of this LSC active simultaneously, each with a different timer object bound to the `AbstractTickerTimer` symbolic lifeline (see Section 5.3).

Note the polymorphic interpretation of the symbolic lifeline in its binding to different concrete objects. Just like concrete lifelines, events attached to symbolic lifelines are unified with executed methods and affect play-out. For example, consider the (simple) LSC `TickAndMove` that appears in Figure 15 (right), which specifies a scenario that drives the scrolling of the RSS captions. Roughly, it states that whenever a `ScrollTickerTimer` ticks, it calls the `move()` method of the `RSSDisplay` object. As part of play-out support for symbolic lifelines, the two `tick()` methods,

in the AbstractTickerTimerBehavior and the TickAndMove LSCs, may be unified at runtime, if the instance lifeline representing the AbstractTickerTimer in the AbstractTickerTimerBehavior LSC is bound at runtime to an instance of the ScrollTickerTimer, despite their appearance in the specification at different levels of the class hierarchy.

As another example, recall the Blinking LSC of Figure 4, which has a lifeline of type BlinkerTickerTimer. At runtime, the tick() methods in the Blinking and AbstractTickerTimerBehavior LSCs may unify, in an instance of the latter, whose AbstractTickerTimer lifeline binds to a BlinkerTickerTimer. In this case the coordinator makes sure tick() will not be executed unless it is enabled in the instances of both LSCs.

Allowing the specification of such generic scenarios, which may be instantiated with different types of objects, makes the LSC specification more expressive and succinct, and thus renders play-out more powerful. It also seems to be a natural feature in the context of Java and for engineers who are familiar with object-oriented design.

Some delicate semantic issues and technical implementation details arise in the context of this feature of the language and its implementation in S2A. They include, for example, static vs. dynamic resolving of references, as well as issues related to the semantics of call and execute pointcuts in AspectJ [Barzilay et al. 2004]. The details are outside the scope of this article. We hope to cover them in a future paper dedicated to this topic.

5.3 Multiple Scenario Instances

Recall that, according to play-out semantics, a given LSC may have multiple instances active simultaneously, each in a different cut and/or with different lifeline bindings.

As already mentioned, the LSC AbstractTickerTimerBehavior of Figure 15 (left) specifies a generic behavior in which the AbstractTickerTimer class participates. During the RSS News Ticker application run, there may be a point in time where several instances of this LSC are active simultaneously. For example, when two different timers, a ScrollTickerTimer and a BlinkerTimer, are triggered by the environment, two different instances of the AbstractTickerTimerBehavior LSC are activated, each with a different concrete object bound to the AbstractTickerTimer lifeline. The LSC variable *e* is also bound to a different value in each instance.

To support the multiple-instances semantics, the scenario aspect manages an array of active LSCs, representing instances of this LSC. The scenario aspect itself is used as a static object containing all the operations and data that are common to all instances of this LSC. These include the operations getActiveLSCCutState() and changeActiveLSCCutState(), and static data such as the number of lifelines, the list of hot cuts, etc. Moreover, the generated pointcuts are also part of the scenario aspect, since the events that need to be monitored and checked for possible unification are common to all the instances of a given LSC.

Figure 16 displays a snippet of the scenario aspect, which was generated from the AbstractTickerTimerBehavior LSC. The snippet contains an example of the static members and the operations belonging to a scenario aspect.

Bookkeeping of LSC instances is done using an array of activeLSC objects, each of which represents an active LSC instance and holds the data of this LSC instance, that is, lifeline binding references, the current cut of this instance, parameter values etc. Roughly, whenever the scenario aspect method changeCutState() is called, a test of whether the event that triggered the call is minimal takes place. If the test is positive, a new activeLSC object is created and is added to the active instances

```
public aspect LSCAspectAbstractTickerBehavior extends LSCAspect
{
    //Constants for instances, locations, variables
    static final int AbstractTickerTimer_INST_abstractTicker = 0;
    static final int ActionEvent_arg_symbolic_e = 0;
    MSDAspectAbstractTickerBehavior()
    {
        addMinimalEvent(MSDMethods.env_AbstractTickerTimer_actionPerformed);
        setHotCut(1,2);
        setLastCut(1,3);
        numberOfLifeLines = 2;
        numberOfInstances = 1;
        numberOfVariables = 1;
    }
    //Pointcuts and advice
    pointcut AbstractTickerTimer_AbstractTickerTimer_tick(
            AbstractTickerTimer abstractTickerTimer):
    //...
    //Automaton logic
    protected void  changeActiveLSCCutState //..
    //...
    protected void getActiveLSCCutState //..
    //...
```

Fig. 16. A snippet from the `AbstractTickerTimerBehavior` generated scenario aspect. Note that the generated scenario aspect is a subclass of `LSCAspect`. Note also the constants and the settings of various 'static' (i.e., common to all instances) properties of this scenario inside the constructor.

```
//...
protected void changeCutState(int LSCMethodID, Object sourceObject,
        Object targetObject,ArrayList args)
{
    completedActiveLSCs.clear();
    if (minimalEvents.contains(LSCMethodID))
    {
        ActiveLSCAspect newCopy = new ActiveLSCAspect(numberOfLifeLines,
                numberOfInstances,numberOfVariables,idCounter++,this);
        activeLSCArray.add(newCopy);
    }
    for(ActiveLSCAspect curCopy: activeLSCArray)
    {
        doBindings(curCopy);
        changeActiveLSCCutState(LSCMethodID,sourceObject,targetObject,
                curCopy,args);
    }
    for(ActiveLSCAspect deadCopy:completedActiveLSCs)
        activeLSCArray.remove(deadCopy);
}
//...
```

Fig. 17. A snippet from the `LSCAspect` code, the scenario-aspects super class. The snippet displays the `changeCutState()` method and how it handles the LSC's multiple active instances.

bookkeeping array. Each call to the `changeCutState()` method, triggers one call to the `changeActiveLSCCutState()` method, for each active LSC instance in the array. A violation (cold or hot) or a completion of an active LSC instance causes the removal of the corresponding `activeLSC` from the array.

Figure 17 contains a snippet from the code of `LSCAspect`, which serves as a super class for all the generated scenario aspects. The snippet displays the `changeCutState()` method and shows how it handles the LSC's multiple active instances. The `getCutState()` method has similar structure.

5.4 Conditions and Hidden Events

LSC conditions are formally specified in our work using UML `StateInvariants` with Boolean expression values. As explained above, we consider expressions inside conditions to be opaque. Thus, S2A does not parse or otherwise try to evaluate them, but, rather, only copies them into an appropriate place in the generated code.

Since S2A neither parses nor evaluates opaque expressions, they must be used carefully. For example, in general, S2A assumes the programmer uses only side-effect-free expressions inside conditions (and opaque parameters). It also assumes that the expression code will indeed compile. Our approach to handling expressions is thus similar to the way tools such as Rhapsody and Rational Rose Real-Time handle guards and code snippets added on state entrance and exit. In Section 7.1 we briefly evaluate the tradeoff involved in this design decision.

Boolean expressions appear not only within `StateInvariants` but also as guards in various combined interaction fragments (e.g., an LSC switch-case construct). In addition, the entrance and exit of any fragment are considered to be synchronization points; all lifelines participating in the fragment synchronize on enter and exit (this kind of synchronization is called a *hidden event* in Harel and Marelly [2003a]).

In our work, conditions and hidden events are evaluated as soon as they are enabled.[9] Thus, the code that calls for their evaluation is located in the method `changeActiveLSCCutState()`. Conditions appear immediately after the cut state of an active LSC was set to a cut where they are enabled. If a condition evaluates to true, the cut is advanced. If it evaluates to false, a cold or hot violation occurs, based on the current cut's mode. Guard expressions are treated similarly, with the difference that a false evaluation of a guard expression does not lead to a (cold or hot) violation. Instead, false evaluation of a guard results in evaluating the next guard, or, if no such guard exists, it results in a transition to the next cut state outside the interaction fragment in context.

The snippet of the generated code in Figure 14 demonstrates condition evaluations within scenario aspects. Note the `evaluateCondition()` method and the call for condition evaluation from within the `changeActiveLSCCutState()` method.

5.5 Additional Features

S2A supports additional important features of LSCs such as dynamic object creation, and control-flow constructs like switch-case (using the UML `ALT` alternative interaction fragment), and bounded and unbounded loops. The `Blinking` LSC presented in Figure 4 demonstrates the use of dynamic object creation and bounded loops. Note that LSC switch-case and loop constructs are not implemented using Java switches and loops. Rather, these are high-level control-flow constructs, which are reflected in the structure of the LSC automaton. For example, an LSC switch-case (a UML `ALT` alternative interaction fragment) results in branching in the LSC automaton, choosing between several next cut states using guards. We omit the technical details.

6. APPLICATIONS AND EXTENSIONS

A number of early case study applications created using S2A are available for download from the S2A Website. In addition, the transformation scheme and its accompanying S2A compiler have already been extended and used in some related research work, now briefly discussed.

[9]Other possible evaluation strategies for conditions and hidden events are considered in Harel and Marelly [2003a], together with a discussion of their advantages and disadvantages. These are outside the scope for this article.

In Lo et al. [2007] and Lo and Maoz [2008a, 2008b], data mining methods are used to extract statistically significant modal scenario-based specifications from program execution traces. The mined scenarios are visualized using the UML2-compliant variant of LSC. The work uses the S2A compiler to translate the mined scenarios into monitoring scenario aspects, which are then compiled and used as (monitoring) scenario-based tests over subsequent executions of the application under investigation.

Maoz et al. [2007] describe a framework for the visualization and exploration of execution traces of reactive systems. The technique links the static and dynamic aspects of the system, and supports synchronic and diachronic trace exploration, multiplicities, and event-based and real-time-based tracing; it uses overviews, filters, details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods. The work uses the S2A compiler to generate *scenario-based traces* [Maoz 2009a]. The ideas are demonstrated using a prototype tool called the Tracer.[10]

It is noteworthy that the Tracer can be used to visualize and explore general scenario-based traces. It is not limited to programs that are executed by play-out. In general, we believe that even when one follows a synthesis approach or just manually implements the requirements in code, viewing the execution in terms of the scenario-based specification defining it can be very useful, specifically for analysis, testing, and comprehension purposes. The S2A compiler and the Tracer indeed support this kind of view.

Atir et al. [2008] investigate the classical notion of object composition in the framework of scenario-based specification and programming. In order to support and take advantage of object composition, the LSC language is extended with appropriate syntax and semantics that allow the specification and interpretation of scenario hierarchies – trees of scenarios – based on the object composition hierarchy of the underlying model. This work has been recently implemented as an optional extension to the S2A compiler.

Poupko [2008] presents preliminary work investigating the use of the LSC language in specifying web service choreography. It uses the S2A compiler to compile each participant's view of the choreography into a separate scenario aspect, and then distributes the generated aspects code between the implemented services. Thus the code generated by S2A is integrated with the web service environment.

Finally, Maoz et al. [2009] present a version of the S2A compiler whose target language is AspectC++ [Spinczyk et al. 2002].[11] In this work, test cases specified visually using LSCs are automatically translated into scenario aspects written in AspectC++, for the purpose of test execution within an application running on Symbian OS, inside a Nokia smartphone. The work demonstrates the use of LSC and the compilation scheme presented here for scenario-based testing. It also shows the applicability of the scheme, with the required technical modifications, to aspect languages other than AspectJ.

7. ANALYSIS AND EVALUATION

In this section we provide a critical analysis of various aspects of our work, evaluate its advantages and limitations, and suggest related challenges. These include a discussion of play-out, smart play-out, and controller synthesis; a discussion of the readability and usability of the code; a complexity analysis and performance

[10]Tracer Website. http://www.wisdom.weizmann.ac.il/~maozs/tracer/.
[11]AspectC++. http://www.aspectc.org.

discussion for the compilation; a discussion on runtime performance; and suggestions for optimizations of the generated code.

7.1 Play-Out, Smart Play-Out, and Controller Synthesis

7.1.1 The Limitations of Naïve Play-Out. As discussed in Harel et al. [2002], the original play-out process of Harel and Marelly [2003a] implemented in our present scheme, is rather naïve. Specifically, some of the possible sequences of events may eventually lead to deadlocks. That is, to states where one active scenario requires the eventual execution of a first method before a second one, while another scenario requires the eventual execution of the second before the first; or, similarly, where one active scenario requires the eventual execution of a certain method while another forbids it. Moreover, the partial-order semantics among events in each chart and the ability to specify scenarios in different charts without providing explicit runtime dependencies are very useful in early requirement stages, but can cause underspecification and nondeterminism when one attempts to execute them. In naïve play-out, nondeterminism is solved ad-hoc, taking the current global state into consideration but not considering the future effects of the different choices. Even when a deadlock-free execution path is available, naïve play-out is not guaranteed to find it. As a result, naïve play-out will lead to a safe execution only if dependent scenarios agree on the order of shared events. One way to address this in the context of compilation (that is, not at runtime like the smart play-out technique that we discuss next), may be the development of a precompilation analysis that checks whether naïve play-out is safe for a given LSC specification. We believe this is possible and leave it for future work.

7.1.2 Smart Play-Out. Smart play-out [Harel et al. 2002] addresses the limitations of naïve play-out using model-checking techniques, which, for instance, can be used to look ahead at runtime and compute a safe execution path, if one exists. The details of smart play-out are outside the scope of this article.

Packaging smart play-out as a play-out strategy that can be used by the coordinator in our scheme involves no major technical difficulty. Changes to the compilation scheme would be relatively minor; the generated scenario aspects and coordinator would collect the global state of the LSCs, and smart play-out reasoning would be embedded in the strategy used by the coordinator.

We note, however, that the correctness (soundness and completeness) of smart play-out depends (among other things) on the assumption that the play-out mechanism has full control over the execution. Since in our scheme the generated scenario aspects are integrated with other code, whose behavior may not be fully modeled in the LSC specification, the applicability of smart play-out to our work may be limited. Stronger assumptions about the relationships between the LSC specification and the core code of the application, or, alternatively, a stronger notion of smart play-out that allows reasoning with incomplete information, may be required in order to integrate smart play-out (or a variant thereof) into our compilation scheme in a useful and effective way.

In addition, recall that our current compilation scheme handles expressions inside conditions and guards as opaque (see Section 5.4). Applying smart play-out, however, requires expressions to be parsed and evaluated by the execution mechanism for the purpose of reasoning. While this is possible, it limits the kind of expressions one can put in the program. This trade-off needs to be considered when applying smart play-out (and synthesis; see below) to our work.

Another barrier to the integration of smart play-out into our compilation scheme is its complexity. In Harel et al. [2009b] we show that a most simple variant of smart

play-out is PSPACE-hard. Nevertheless, this worst case complexity is rarely actually met; recent work done in our group presents heuristics to accelerate smart play-out [Harel et al. 2010], which might be used in our work in the future.

7.1.3 Controller Synthesis. Section 7.1.2 notwithstanding, one should realize that smart play-out is limited too. Specifically, in Harel et al. [2009a] we show that smart play-out, no matter how often repeated, is strictly weaker than full synthesis from LSCs. We thus believe that full controller synthesis from an LSC specification, following Harel and Kugler [2002], and as more recently investigated and implemented in Kugler et al. [2009] and Kugler and Segall [2009], is an alternative worth considering.

Thus, an important direction for future work involves the integration of the compilation scheme presented in this article with a strategy that is based on a synthesized controller. This would still lead to compiling LSCs into scenario aspects that monitor the progress of scenario instances, and generate a coordinator aspect that collects the global state from the scenarios and is able to inject method executions. However, the execution strategy itself, would be separately generated from a synthesized controller. In other words, even if a synthesized controller does become available, we would still need a mechanism to collect the relevant information from the program as it executes. In addition, we would still be interested in examining the execution of the resulting program from the point of view of the different scenario-based requirements (e.g., for model-level debugging using model-based traces [Maoz 2009a]).

In addition to the barriers mentioned above regarding smart play-out, two challenges can be seen in integrating our work with a strategy that is based on a synthesized controller. First, the complexity of synthesis and the size of the controller. Second, the significant gaps between the subset of the LSC language handled by synthesis solutions to date, and the much richer subset handled by play-out (and by our compilation scheme). We specifically have in mind multiple instances and symbolic (polymorphic) specifications. We pose these too as directions for future work.

7.1.4 Play-Out Semantics. To sum up the discussion above on naïve play-out, smart play-out, and synthesis, one may view these three different approaches to play-out as *three different operational semantics* for the same language. Naïve play-out defines one semantics. Smart play-out defines another, stronger semantics, and synthesis defines an even stronger one.

Thus, the limitations of naïve play-out discussed above do not render it useless. The fact that in the general case naïve play-out (and hence the code produced by our compilation scheme) may result in deadlocks, does not render it incorrect. No software development tool we are aware of that generates code from higher-level models (e.g., Rhapsody, RoseRT) guarantees a deadlock-free execution, and one should emphasize that the compilation scheme and implementation presented in this article are sound with regard to this semantics of naïve play-out.

7.2 Readability and Usability of the Generated Code

The evolution and maintenance of software constructed from a mix of manually created source code and automatically generated code is a well-known challenge. Indeed, in many other approaches and their accompanying tools, which transform visual formalisms into code (e.g., Statemate [Harel et al. 1990], IBM Rational RoseRT, Rhapsody), the generated code is inseparable from the manually created code, and thus creates difficulties in code evolution and maintenance (a problem that is partly addressed by complicated code synchronization utilities).

In contrast, since S2A generates aspect code, the generated code and the rest of the application's source code are strictly separated into different files throughout the development process. That is, the structure of the specification is reflected in the structure of the generated code. Moreover, this separation allows local changes in the program's code; for instance, changes to the implementation of methods that are independent of the methods appearing in the LSC specification, to be manually made in the separate files without worrying of accidentally changing the code responsible for the LSC execution, and without worrying that future LSC compilation would override manually made changes. While this does not provide guarantees for seamless maintenance, we consider it a nice advantage of our approach.

The readability of the generated code itself is another important concern for code generation tools. To improve readability of our generated code, most of the common functionality of all generated aspects is encapsulated in an abstract (nongenerated) aspect, `LSCAspect` (residing in the S2A runtime library), and all generated aspects are defined as its subclasses; see, for instance, line 1 of Figure 16. This helps ensure a relatively high level of abstraction in the generated code wherever possible. As an example, consider the violation checks and the completion operations of an active LSC copy, appearing in the last two lines of Figure 9. We thus take advantage of aspect inheritance in order to keep the generated code for scenario aspects as readable and as concise as possible.

We have made an additional effort to ensure that the generated code is readable and informative, for instance, by using clear naming conventions and avoiding redundant code cloning within each scenario aspect. For example, the scenario aspect name is based on the name of its source LSC (e.g., line 1 of Figure 16); lifelines and variables are translated into constants in the scenario aspect (Figure 16, lines 4-5); lifeline and variable bindings of an active LSC copy are abstracted into "set" and "get" methods (Figure 14); a pointcut's name is composed of the name of its source method in the specification, the method's caller, and the method's callee (the name convention for LSC methods identifiers is similar); cuts are represented as tuples of integers; all operations related to cuts, such as advancing the cut, querying about it, or comparing between cuts, are abstracted to operate on these tuples and are implemented in the generic `activeLSC` class (see Figure 14).

Moreover, due to the partial-order semantics, LSC conditions and guards may have to be evaluated in several different cuts, that is, in different transitions of the automaton. In order to keep the code compact and to avoid redundant clones of condition's/guard's expressions, we aggregate all the Boolean expressions appearing in the LSC into a single place in the scenario aspect code, the `evaluateConditions()` method (see Figure 14).

Finally, as another readability aid, the LSC method objects, which are constructed inside `getActiveLSCCutState()` and are sent to the coordinator, are followed by a documentation comment. The comment helps the reader identify which LSC method is represented by each object (see Figure 10).

7.3 Complexity Analysis

We briefly discuss the worst-case complexity of the compilation scheme, which is dominated by the size of the automaton constructed for each LSC.

Given an LSC with n events and k lifelines, due to the partial-order semantics, a naïve DFS-like traversal of the LSC generates an automaton of size $O(k^n)$. We use, however, a rather simple optimization, based on identifying equal cut states that are reachable from multiple sources and unifying them into a single state. Identical cuts are discovered and unified as soon as they are found, thus saving multiple traversals

of equal subtrees. This reduces the worst case size of the constructed automaton to $O(n^k)$ (since $n > k$, this approach is very effective).

Note that since the construction of the automaton does not require information from other LSCs, the compilation of each LSC is independent of the rest of the specification. The total complexity of a specification of m LSCs is the product of these n^k quantities for each of the LSCs. If n_0 and k_0 are the maximal number of events and lifelines in all the m LSCs, the total complexity is $O(mn_0^{k_0})$. Generating the coordinator aspect requires $O(l)$ time, where l is the total number of unique events appearing in the LSCs in the specification (the size of the alphabet).

7.3.1 Compilation Optimization. The following ideas for compilation optimization may be considered. The first is static analysis of the specification as a whole, to identify, for instance, unreachable cut states or opportunities for partial-order reduction. As a simple example: the execution of methods that appear only in a single LSC can be performed locally, whenever enabled, eliminating the need for coordination (for these methods). This example is inspired by the work in Harel et al. [2010]. The complexity of such global preprocessing analysis will in general be high, similar to that of smart play-out previously mentioned, but it may result in much smaller (and efficient) generated code.

Second, and perhaps more practical, is a local analysis of each LSC, which will not construct an automaton but will only represent the order of event occurrences on each lifeline, leaving the next cut state of the LSC to be computed at runtime. For an LSC with n events and k lifelines, compilation complexity would be $O(nk)$ (instead of the $O(n^k)$ mentioned above). Thus, this will be most effective when LSCs with many lifelines (and "a lot" of partial-order) are considered (as in Atir et al. [2008]). There will be a trade-off, however, between the time invested in compilation and the overhead incurred in runtime performance, since computing the set of enabled events for a given state, at runtime, will not be constant (as it is now) but linear in the number of lifelines in the LSC.

We leave the details and the implementation of these possible optimizations for future work.

7.4 Early Experience with Compilation Performance

Despite the $O(mn^k)$ upper bound on compilation complexity discussed previously, actual compilation time in most example applications we have tested to date is negligible. Specifically, for specifications of up to 20 LSCs, each with no more than 6 lifelines, S2A compilation takes less than a second on a very basic computer (Pentium 4, 2.8GHz, 1G RAM). This is because most real specifications do not make intensive use of explicit partial order and the number of lifelines per LSC is typically bounded by a small constant (even if the specification as a whole involves many different objects).

Some specifications do take significant time to compile. For example, some of the LSCs created in the process of compiling the trees of scenarios used in Atir et al. [2008] in the context of object composition have 16 lifelines. Our experience shows that the compilation of these specifications indeed takes much longer and generates very large code. Thus, the optimization ideas suggested in the previous subsection appear to be important for the successful application of our approach in general, and of S2A in particular, to real world software engineering.

That said, due to the use of play-out, rather than smart play-out or synthesis, compilation is indeed local; each LSC is independently analyzed and each scenario aspect is independently generated. Thus, worst-case compilation complexity, as well as its actual running time, grows only linearly with the number of LSCs in the specification. Smart play-out and synthesis have greater expressive power, but require

a global analysis, that is exponential in the number of LSCs in the specification (see Section 7.1).

7.5 Early Experience with Runtime Performance

The runtime performance of the generated code is a well-known concern for code generation approaches and tools. In our context, runtime performance is affected mainly by the following parameters: the number of join points in each LSC and the extent to which they are generic, the number of active instances of each LSC, and the implementation of the strategy.

It is important to note that optimal performance is not the main goal of our work. Thus, for example, we chose to implement the compiler in Java, and selected Java and AspectJ as target languages, mainly due to the popularity of these languages and the availability of their excellent tool support. This choice was not made with the goal of optimal performance in mind. Towards the end of Section 8.3 we discuss related work on aspects that is geared towards optimal performance, and which may be applicable to our work in the future.

Another design decision that was not made with the goal of optimal performance in mind relates to code readability. Many of the readability related features discussed in Section 7.2 may hinder performance, for instance, the elimination of LSC condition clones.

We have experimented with a number of case study applications, including desktop applications with real-time user experience, such as the RSS News Ticker presented here and a Space Invaders game. The latter consists of more than 20 LSCs, some of which may have more than 30 instances simultaneously active at runtime. We used only a very basic 2.8GHz Pentium 4 machine with 1G RAM.

At the level of user experience we have not noticed any effect on performance, compared to an implementation of the same application using manually written standard Java. This result was expected: not all the behaviors of the applications at hand were modeled and executed using LSCs. Most low-level behaviors, such as writing to the screen, were implemented in Java and were abstracted away from the specification model. The LSCs included only higher-level method calls between application objects and conditions over parameters and observable objects properties, similar to the ones present in the RSS News Ticker example specification.

At a more detailed level, in specific experiments, an overhead was noticed. As expected, this increased linearly with the number of LSC instances simultaneously active in the application.

We acknowledge that the performance analysis discussed here is partial. As mentioned previously, optimal performance is not the main goal of our work. Thus, a thorough performance analysis remains to be done.

7.6 Possible Optimizations for the Generated Code

We consider the following suggestions for generated code optimization.

First, as long as a minimal event of an LSC has not occurred, one need not listen to any of the other methods mentioned in this LSC. The same is true whenever a minimal event occurs but all active LSC instances have already closed. That is, when there is no active LSC instance, the only events play-out needs to monitor are the minimal ones (due to the partial-order semantics, there may be more than one minimal event).

Thus, one may consider adding a simple condition to all advice of pointcuts not representing minimal events in that LSC. If no active LSC instance exists, the `changeCutState()` method should not be called, and the advice will complete instantly. Note that this cannot be generalized to other situations, because whenever an LSC

instance does exist, every method event captured by one of the pointcuts is potentially enabled or violating, depending on the current cut state of the LSC instance (and its bindings); hence, it cannot be handled without consulting the automaton.

Second, recall that whenever a method that may change the cut state of any of the LSCs occurs, the coordinator aspect collects cut-state information (sets of enabled and violating methods) from all LSCs, and each LSC collects this information from its active LSC instances, if any.

Thus, one may consider an optimized version of the generated coordinator that will statically compute, in advance, which events may change which of the LSCs cut states. Computing such an over-approximated mapping at compile time is not difficult. Using this approach, the coordinator would keep the results returned by each of the LSCs. Whenever an event occurs, it would call the `getCutState()` method of only the LSCs which may be affected by this event and combine the results with the cached results of the other LSCs. Alternatively, each LSC can manage this cache on its own, and update it only when the state of one of its active instances changes.

This may improve the generated code's performance without affecting compilation performance. Additional possible improvements may be achieved using advanced approaches to AOP; see the end of Section 8.3.

8. RELATED WORK

We now discuss related work on LSC play-out, synthesis, aspect modeling, aspect code generation, and other approaches to AOP.

8.1 The Play-Engine and InterPlay

The Play-Engine [Harel and Marelly 2003a] is an experimental tool for requirements capture and direct execution of LSCs, based on the play-in/play-out approach [Harel and Marelly 2003b].[12]

By compiling LSC specifications into AspectJ code we advance the ideas behind play-out from a tool dependent interpreter to having the potential of becoming the central part of a standard development and execution environment.

InterPlay [Barak et al. 2006] coordinates the simulation of the Play-Engine and a separately executed program, such as another Play-Engine, or an intra-object statechart tool like Rhapsody or RoseRT, given an appropriate custom interface implementation. It relies on a user-defined bidirectional mapping between LSC events and observable program events. In contrast, our compilation process integrates the LSC specification into the program code, the result thus being a single executable program, despite the use of different multiple modeling methodologies in the requirements specification and coding phase.

8.2 Synthesis and Aspects

Krüger et al. [2005] propose a translation of conventional MSCs into AspectJ in the context of exploring alternative service-oriented architectures. This work is related to ours, but it suggests using a synthesis algorithm, adopted from Krüger et al. [1999], to project specified behaviors onto each role, ultimately providing a state machine for each participating object. As explained earlier, play-out, and thus our compilation scheme too, attempts to bypass the need for this kind of synthesis. More recent work by Krüger et al. [2006] translates MSCs into aspects and lets the weaver resolve some

[12]Play-in is a user-friendly high-level way of specifying behavior and automatically generating the specification formally in LSCs [Harel 2001; Harel and Marelly 2003a]. The present work on LSC compilation and execution is independent of the method used to create the LSCs, using play-in, direct visual editing, or any other method.

of the resulting nondeterminism in a random fashion. Coordination between non-disjoint scenarios, that is, synchronizing the interactions around common messages, is done statically and only between overlapping scenarios that the designer has explicitly specified to be "joined." In contrast, in our work, coordination is carried out dynamically, at runtime.

Deubler et al. [2005] suggest modeling crosscutting services with UML sequence diagrams enhanced by aspect-oriented concepts. This work is related to ours in that it uses an interaction-centric development approach and concentrates on the behavioral part of aspects. Code generation, however, is considered in the context of synthesis, as in Krüger et al. [2005].

Whittle et al. [2005] translate requirements given in the form of MSCs and IPS (interaction pattern specification) into automata, where inter-dependencies between the scenarios are handled through the identification of common local states, explicitly specified by the designer as part of the requirements specifications, and by a unification of states with common incoming and outgoing transitions. In related work, Araujo et al. [2004] present a requirements level aspect-oriented modeling approach. Both works discuss the aspectual nature of crosscutting requirements but use a synthesis algorithm, which results in a state-machine for each of the participating components. In contrast to these synthesis-based work, in our work, the scenario-based structure of the specification is not "lost in the translation"; the structure of the code reflects the structure of the specification. We consider it an advantage.

Uchitel et al. [2004a, 2004b] promote the use of scenario-based languages for requirements elicitation and specification. They discuss the limitations of existing MSC synthesis approaches and propose to address them by, for instance, detecting implied scenarios, or by the use of architectural information to synthesize the behavior of component types rather than that of instances. It seems that ours and theirs share the positive view of the intuitive nature and usefulness of scenario-based notations. Their goal is mainly requirements elicitation, while our goal is execution. Also, our approach differs in that we use a more expressive formalism and show how to carry over the scenario-based behavioral specification from requirements to implementation. From a methodological point of view, they suggest incremental elaboration using implied scenarios, where, in the context of LSC, we would add a process that starts from basic scenarios and incrementally elaborates them with modalities. A complete methodology for the use of our compilation scheme in a development process is, however, outside the scope of this article and will be addressed separately.

8.3 Generated Aspects and Advanced Approaches to AOP

Stolz and Bodden [2006] present a nicely built runtime verification framework for Java programs, where properties specified in LTL (linear temporal logic) formulas over AspectJ pointcuts are checked during program execution by an alternating finite automaton, whose transitions are triggered through generated aspects. Another runtime verification framework is suggested in Kiviluoma et al. [2005], where generated aspects are used to simulate a small state machine that monitors behavioral requirements given as UML sequence diagrams. Since LSCs can be translated into LTL formulas [Kugler et al. 2005], these two works have some similarities with ours, specifically in the possibility of using our code generation scheme to monitor LSCs. Since our main motivation is execution, however, we use the LSC language distinction between monitoring and execution modes, and adopt the mechanism for simultaneous coordination between the automata from the play-out algorithm. Coordination between the generated aspect automata, as is done in our work, is irrelevant to these two works.

Groher and Schulze [2003] discuss the generation of AspectJ code skeletons from a UML model. Their approach offers a mapping between the structure of the model and the structure of the resulting program. The skeletons, however, cannot be executed, as the actual behavior is not modeled.

Many researchers consider the interesting question of using UML to model aspects, or suggest to use UML-like notations or new profiles specifically for this purpose [Araújo et al. 2002; Barra et al. 2004; Basch and Sanchez 2003; Mahoney and Elrad 2005]. Our present work differs in that it is not intended to answer this question; instead, we use a specific class of aspects in order to execute scenarios. Thus, we do not aim to create models that cover the expressive power of aspects.

Jacobson and Ng [2004] discuss a methodology for aspect-oriented software development with use-cases, and attempt to achieve use-case modularity through aspect technologies. The use-case abstraction level is not detailed enough to allow formal semantics nor expressive enough for actual code generation. Indeed, scenarios are viewed in Jacobson and Ng [2004] as a means to explicate and formalize use-cases. In contrast, we show how to actually compile scenarios to code via aspects, and in so doing provide a new possibility for executable use-cases (in addition to the play-out of Harel and Marelly [2003a]).

Finally, some advanced approaches within the AOP community itself are relevant to our work. Tracematches [Allan et al. 2005] is an extension of the AspectJ abc compiler [Avgustinov et al. 2005], which allows the programmer to trigger advice execution by specifying a regular pattern of events in a computation trace. The ability to use free variables in the matching pattern and the corresponding unification semantics is close to our use of parameterized methods and symbolic LSC lifelines. Recent work [Avgustinov et al. 2007] suggests optimizations for tracematches and addresses space leak problems in its implementation to increase scalability. It shows the feasibility of using tracematches for trace monitoring.

Another relevant approach is that of stateful aspects [Vanderperren et al. 2005], implemented in the JasCo language [Suvée et al. 2003], where pointcuts can declaratively specify protocol fragments expressible by finite state machines, and separate advice can be attached to every transition specified in the pointcut protocol.

Our pointcuts are single points, and we manage the finite state machine explicitly in the aspect's body. Thus, one could consider using tracematches or stateful aspects, both to simplify our generated code, by taking advantage of the more expressive constructs available in these approaches, and to improve the performance of the final executable program.

Moreover, association aspects, suggested in Sakurai et al. [2004] as an extension to AspectJ, allow one to associate an aspect instance with a group of objects and to specify aspect instances as execution contexts of advice. They thus enable one to use a natural way to specify stateful behavior related to a particular group of objects. More recently, Bodden et al. [2008] defined a generalization of association aspects called relational aspects, as an extension of tracematches. These may be useful as an alternative implementation of scenario aspects, in particular in supporting multiple scenario instances.

We should note however, that the two approaches, tracematches and stateful aspects, are limited to regular protocols. Since nonregular protocols can be specified in LSC (using variables and unbounded loops), this, to some extent, limits their applicability to our work.

More generally, while there are close similarities between the compilation scheme presented here and the advanced aspect-based runtime verification work cited above, there are also fundamental differences, related to the active nature of our work and to its need for runtime coordination between different scenarios. Specifically, the active

characteristics of play-out, whose main goal is *execution* and not just monitoring, implies that our generated code cannot be as passive and cannot run with as low an overhead as possible because it must actually do things, directing and affecting the behavior of the application. This is in contrast to runtime verification techniques, where the goal is indeed monitoring with lowest overhead, ideally causing no effect on the program's behavior (unless some safety violation occurs). Finally, runtime verification optimization approaches apply to "local" traces, or to each scenario alone, while play-out, and hence the compilation scheme presented here requires, in addition, coordination between the different automata—in our context, coordination between the generated aspects—which is irrelevant for runtime monitoring and hence is not addressed by these approaches. As a result, even if these approaches are implemented, in order to reduce monitoring overhead, and perhaps to produce more elegant code, we expect coordination between the scenarios, specifically, computing the next method for execution according to the strategy, to remain a performance challenge.

9. SUMMARY AND FUTURE WORK

One way of viewing our work is as an attempt to carry over a significant idea from the aspect-oriented world to the scenario-based one, exploiting one of the main achievements of research on aspects, which is the ability to execute aspect programs by compilation, in order to compile and execute inter-object scenario-based specifications.

Our main contribution is in translating the inter-object scenario-based requirements to code that can integrate seamlessly with existing programs and is compiled and executed in a standard manner. Thus, it constitutes a crucial step towards integrating the scenario-based approach into mainstream software engineering.

Still, some possible drawbacks of our work should be mentioned and addressed. A number of limitations, with corresponding future work challenges, were mentioned in Sections 7 and 8. These include the limitations of naïve play-out and the challenges in extending our work to smart play-out or controller synthesis (see Section 7.1), the complexity of the compilation and the opportunities for its optimization (Sections 7.3 and 7.6), the performance of the translation scheme (see Section 7.4), and runtime performance of the generated code (see Section 7.5 and the end of Section 8).

In addition to these, we consider the following issues.

Play-out in general, and our current compilation scheme in particular, require central coordination at runtime. The need for a centralized coordinator is a limitation not only from a performance point of view but also from an architectural point of view. Thus, an important research topic, which our group is pursuing at present, is to find ways to (partially) distribute the play-out effort, not necessarily between objects but between concurrently active coordinators.

Adding support for explicit time and real time, as was partly done for the Play-Engine [Harel and Marelly 2002], is challenging. Specifically, our current compilation scheme, and play-out in general, does not support multithreaded programs and assumes the system to be infinitely faster than its environment. Handling coordination in realtime and with multithreaded programs is thus an interesting topic for future research.

Another issue in our current work is the interpretation of LSC messages as method calls, rather than assuming the definition and implementation of a full-featured event model in the applications we aim to model and execute. As a result, our current implementation suffers from some synchronization problems (partly addressed by a queue implemented as part of the coordinator aspect). A possible solution is to require the model and the code to use separate alphabets, that is, to require that events appearing in the LSC specification are never independently initiated by the original code. This

requirement can be enforced at runtime, using aspects. We believe that this kind of layered architecture may be useful and leave the issue for future work.

From a methodological point of view the compilation scheme can be used during a multistep hybrid development process, even if we prefer not to use it to produce the final program. At first, many system components are not fully implemented and thus play-out is responsible for "executing" them. As the implementation progresses, execution responsibilities gradually move from the LSCs to the core program code; LSC methods marked for execution change into monitoring mode. Eventually, the system becomes fully implemented in the core code and the LSCs that were originally used for executing the partially implemented system remain to serve as optional monitors. We believe this kind of process may be useful and leave its details and evaluation for future work.

Finally, once a compiler is available for scenario-based specifications, other tools for use in the development of scenario-based programs become necessary. For example, imagine a debugger for programs generated by S2A, which would allow setting breakpoints at the level of the model (visually, inside the sequence charts themselves); these will stop the execution at the correct moment and point the user back both to the generated aspect code and to the correct diagram (i.e., LSC instance) itself, drawn with its dynamic cut state. This too is left for future work.

ACKNOWLEDGMENTS

We would like to thank Mark Mahoney and Tzilla Elrad for their early communication with us, where they pointed us to their work on modeling aspects for reactive systems using LSCs [Mahoney 2005; Mahoney and Elrad 2005]. This inspired us to address the converse problem of compilation. In addition, we would like to thank Gera Weiss for early discussions on LSC compilation, and Itai Segall for initiating and helping with the Memory Game example presented in Maoz and Harel [2006]. Finally, we are grateful to Yishai Feldman for his useful comments on a draft of Maoz and Harel [2006], to Assaf Marron for comments on a draft of this article, to Eli Singerman for comments on our work, and to the anonymous reviewers for their constructive comments, which helped us in improving this article.

REFERENCES

ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L. J., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. R. Johnson and R. P. Gabriel Eds., ACM, New York, NY, 345–364.

ALUR, R., ETESSAMI, K., AND YANNAKAKIS, M. 2003. Inference of message sequence charts. *IEEE Trans. Softw. Engin. 29*, 7, 623–633.

ARAÚJO, J., MOREIRA, A., BRITO, I., AND RASHID, A. 2002. Aspect-oriented requirements with UML. In *Proceedings of the Workshop on Aspect-Oriented Modeling with UML*. M. Kande, O. Aldawud, G. Booch, and B. Harrison Eds.

ARAUJO, J., WHITTLE, J., AND KIM, D.-K. 2004. Modeling and composing scenario-based requirements with Aspects. In *Proceedings of the 12th IEEE International Conference on Requirements Engineering (RE'04)*. IEEE Computer Society, 58–67.

ATIR, Y., HAREL, D., KLEINBORT, A., AND MAOZ, S. 2008. Object composition in scenario-based programming. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08)*. J. L. Fiadeiro and P. Inverardi Eds., Lecture Notes in Computer Science, vol. 4961, Springer, 301–316.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, 87–98.

AVGUSTINOV, P., TIBBLE, J., AND DE MOOR, O. 2007. Making trace monitors feasible. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and*

Applications (OOPSLA'07). R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr. Eds., ACM, New York, NY, 589–608.

BARAK, D., HAREL, D., AND MARELLY, R. 2006. InterPlay: Horizontal scale-up and transition to design in scenario-based programming. *IEEE Trans. Softw. Engin. 32*, 7, 467–485.

BARRA, E., GÉNOVA, G., AND LLORENS, J. 2004. An approach to aspect modeling with UML 2.0. In *Proceedings of the 5th International Workshop on Aspect-Oriented Modeling (AOM'04)*.

BARZILAY, O., FELDMAN, Y., TYSZBEROWICZ, S., AND YEHUDAI, A. 2004. Call and execution semantics in AspectJ. In *Proceedings of the Workshop on Foundations of Aspect Oriented Languages (FOAL'04)*, C. Clifton, R. Lammel, and G. T. Leavens, Eds., 19–23.

BASCH, M. AND SANCHEZ, A. 2003. Incorporating aspects into the UML. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling*.

BODDEN, E., SHAIKH, R., AND HENDREN, L. J. 2008. Relational aspects as tracematches. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*. T. D'Hondt Ed., ACM, 84–95.

BONTEMPS, Y. AND HEYMANS, P. 2002. Turning high-level live sequence charts into automata. In *Proceedings of the 1st International Workshop on Scenarios and State-Machines (SCESM'02) at the 24th International Conference on Software Engineering (ICSE'02)*.

BUNKER, A., GOPALAKRISHNAN, G., AND SLIND, K. 2005. Live sequence charts applied to hardware requirements specification and verification. *Softw. Tools Techn. Transfer 7*, 4, 341–350.

COMBES, P., HAREL, D., AND KUGLER, H. 2008. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Softw. Syst. Model. 7*, 2, 157–175.

DAMM, W. AND HAREL, D. 2001. LSCs: Breathing life into message sequence charts. *Formal Meth. Syst. Des. 19*, 1, 45–80.

DEUBLER, M., MEISINGER, M., RITTMANN, S., AND KRÜGER, I. 2005. Modeling crosscutting services with uml sequence diagrams. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*. L. C. Briand and C. Williams, Eds., Lecture Notes in Computer Science, vol. 3713, Springer, 522–536.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect–oriented programming: Introduction. *Comm. ACM 44*, 10, 29–32.

GROHER, I. AND SCHULZE, S. 2003. Generating aspect code from UML models. In *Proceedings of the 4th Aspect-Oriented Software Development Modeling With UML Workshop*. F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kande, D. Stein, P. Tarr, and A. Zakaria Eds.

HAREL, D. 2001. From play-in scenarios to code: An achievable dream. *IEEE Comput. 34*, 1, 53–60.

HAREL, D. 2008. Can programming be liberated, period? *IEEE Comput. 41*, 1, 28–37.

HAREL, D. AND GERY, E. 1997. Executable object modeling with statecharts. *IEEE Comput. 30*, 7, 31–42.

HAREL, D. AND KUGLER, H. 2002. Synthesizing state-based object systems from LSC specifications. *Int. J. Found. Comput. Sci. 13*, 1, 5–51.

HAREL, D. AND MARELLY, R. 2002. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*. IEEE Computer Society, 193–202.

HAREL, D. AND MARELLY, R. 2003a. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer.

HAREL, D. AND MARELLY, R. 2003b. Specifying and executing behavioral requirements: the play-in/play-out approach. *Softw. Syst. Model. 2*, 2, 82–107.

HAREL, D. AND NAAMAD, A. 1996. The statemate semantics of statecharts. *ACM Trans. Softw. Engin. Methodol. 5*, 4, 293–333.

HAREL, D. AND MAOZ, S. 2008. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Softw. Syst. Model. 7*, 2, 237–252.

HAREL, D. AND SEGALL, I. 2007. Planned and traversable play-out: A flexible method for executing scenario-based programs. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. O. Grumberg and M. Huth Eds., Lecture Notes in Computer Science, vol. 4424, Springer, 485–499.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Engin. 16*, 403–414.

HAREL, D., KUGLER, H., MARELLY, R., AND PNUELI, A. 2002. Smart play-out of behavioral requirements. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*. M. Aagaard and J. W. O'Leary Eds., Lecture Notes in Computer Science, Springer, 378–398.

HAREL, D., KLEINBORT, A., AND MAOZ, S. 2007. S2A: A compiler for multi-modal UML sequence diagrams. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07)*. M. B. Dwyer and A. Lopes Eds., Lecture Notes in Computer Science, vol. 4422, Springer, 121–124.

HAREL, D., KANTOR, A., AND MAOZ, S. 2009a. On the power of play-out for scenario-based programs. In *Concurrency, Compositionality, and Correctness, Festschrift in Honor of Willem-Paul de Roever*. D. Darns, U. Hannemann, and M. Steffen Eds., Lecture Notes in Computer Science, vol. 5930, Springer, 207–220.

HAREL, D., KUGLER, H., MAOZ, S., AND SEGALL, I. 2009b. How hard is smart play-out? On the complexity of verification-driven execution. In *Perspectives in Concurrency Theory, A Festschrift for P. S. Thiagarajan*. K. Lodaya, M. Mukultfl, and R. Ramanujam Eds., Universities Press, 208–230.

HAREL, D., KUGLER, H., MAOZ, S., AND SEGALL, I. 2010. Accelerating smart play-out. In *Proceedings of the 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'10)*. Lecture Notes in Computer Science, vol. 5901, Springer, 477–488.

ITU. 1996. International Telecommunication Union Recommendation Z.120: Message sequence charts. Tech. rep., ITU.

JACOBSON, I. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison- Wesley.

JACOBSON, I. AND NG, P.-W. 2004. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. M. Akşit and S. Matsuoka Eds., Lecture Notes in Computer Science, vol. 1241, Springer, 220–242.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'0l)*. J. L. Knudsen Ed., Lecture Notes in Computer Science, vol. 2072, Springer, 327–354.

KIVILUOMA, K., KOSKINEN, J., AND MIKKONEN, T. 2005. Run-time monitoring of behavioral profiles with aspects. In *Proceedings of the 3rd Nordic Workshop on UML and Software Modeling*. 62–76.

KLOSE, J. AND WITTKE, H. 2001. An automata based interpretation of live sequence charts. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*. T. Margaria and W. Yi Eds., Lecture Notes in Computer Science, vol. 2031, Springer, 512–527.

KLOSE, J., TOBEN, T., WESTPHAL, B., AND WITTKE, H. 2006. Check it out: On the efficient formal verification of live sequence charts. In *Proceedings of the 18th International Conference on Computers Aided Verification (CAV'06)*. T. Ball and R. B. Jones Eds., Lecture Notes in Computer Science, vol. 4144., Springer, 219–233.

KRÜGER, I., GROSU, R., SCHOLZ, P., AND BROY, M. 1999. From MSCs to statecharts. In *Proceedings of the International Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*. F. J. Rammig Ed., Kluwer, 61–72.

KRÜGER, I., MATHEW, R., AND MEISINGER, M. 2005. From scenarios to aspects: Exploring product lines. In *Proceedings of the 4th International Workshop on Scenarios and State-Machines (SCESM'05) at the 27th International Conference on Software Engineering (ICSE'05)*. ACM Press, 1–6.

KRÜGER, I., MATHEW, R., AND MEISINGER, M. 2006. Efficient exploration of service-oriented architectures using aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM Press, 62–7l.

KUGLER, H. AND SEGALL, I. 2009. Compositional synthesis of reactive systems from live sequence chart specifications. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*. S. Kowalewski and A. Philippou Eds., Lecture Notes in Computer Science, vol. 5505, Springer, 77–91.

KUGLER, H., HAREL, D., PNUELI, A., LU, Y., AND BONTEMPS, Y. 2005. Temporal logic for scenario-based specifications. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. N. Halbwachs and L. D. Zuck Eds., Lecture Notes in Computer Science, vol. 3440, Springer, 445–460.

KUGLER, H., PLOCK, C., AND PNUELI, A. 2009. Controller synthesis from LSC requirements. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*. M. Chechik and M. Wirsing Eds., Lecture Notes in Computer Science, Springer, 79–93.

KUPFERMAN, O. AND VARDI, M. Y. 2001. Weak alternating automata are not that weak. *ACM Trans. Comput. Log. 2*, 3, 408–429.

LETTRARI, M. AND KLOSE, J. 2001. Scenario-based monitoring and testing of real-time UML models. In *Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools*. M. Gogolla and C. Kobryn Eds., Lecture Notes in Computer Science, vol. 2185, Springer, 317–328.

LO, D. AND MAOZ, S. 2008a. Mining scenario-based triggers and effects. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE, 109–118.

LO, D. AND MAOZ, S. 2008b. Specification mining of symbolic scenario-based models. In *Proceedings of the 8th ACM SIGPLAN SIGSOFT International Workshop on Program Analysis for Software Tools and Engineering (PASTE'08)*. S. Krishnamurthi and M. Young Eds., ACM, 29–35.

LO, D., MAOZ, S., AND KHOO, S.-C. 2007. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'01)*. R. E. K. Stirewalt, A. Egyed, and B. Fischer Eds., ACM, 465–468.

MAHONEY, M. 2005. In *Proceedings of MoDELS International Workshops, Doctoral Symposium, Educators Symposium* (Revised Selected Papers), J.-M. Bruel Ed., Lecture Notes in Computer Science, vol. 3844, Springer, 345–346.

MAHONEY, M. AND ELRAD, T. 2005. Weaving crosscutting concerns into live sequence charts using the play-engine. In *Proceedings of the 7th International Workshop on Aspect-Oriented Modeling*.

MAOZ, S. 2009a. Model-based traces. In *Proceedings of the 3rd International Workshop on Models at Runtime, MoDELS International Workshops, Doctoral Symposium, Educators Symposium*. M.R.V. Chaudron Ed. Lecture Notes in Computer Science, vol. 5421, Springer, 109–119.

MAOZ, S. 2009b. Polymorphic scenario-based specification models: Semantics and applications. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*. A. Schurr and B. Selic Eds., Lecture Notes in Computer Science, vol. 5795, Springer, 499–513.

MAOZ, S. AND HAREL, D. 2006. From multi-modal scenarios to code: Compiling LSCs into AspectJ. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. M. Young and P. T. Devanbu Eds., ACM, 219–229.

MAOZ, S., KLEINBORT, A., AND HAREL, D. 2007. Towards trace visualization and exploration for reactive systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)*. IEEE Computer Society, 153–156.

MAOZ, S., METSÄ, J., AND KATARA, M. 2009. Model-based testing using LSCs and S2A. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*. A. Schlirr and B. Selic Eds., Lecture Notes in Computer Science, vol. 5795, Springer, 301–306.

MARELLY, R., HAREL, D., AND KUGLER, H. 2002. Multiple instances and symbolic variables in executable sequence charts. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA'02)*. ACM, 83–100.

POUPKO, O. 2008. Specifying and executing web service choreography using live sequence charts. M.Sc. thesis, Weizmann Institute of Science.

SAKURAI, K., MASUHARA, H., UBAYASHI, N., MATSUURA, S., AND KOMIYA, S. 2004. Association aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*. G. C. Murphy and K. J. Lieberherr Eds., ACM, 16–25.

STOLZ, V. AND BODDEN, E. 2006. Temporal assertions using AspectJ. *Electr. Notes Theor. Comput. Sci. 144*, 4, 109–124.

SUVÉE, D., VANDERPERREN, W., AND JONCKERS, V. 2003. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM, 21–29.

UCHITEL, S., CHATLEY, R., KRAMER, J., AND MAGEE, J. 2004a. System architecture: The context for scenario-based model synthesis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'04)*. ACM Press, New York, NY, 33–42.

UCHITEL, S., KRAMER, J., AND MAGEE, J. 2004b. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Engin. Method. 13*, 1, 37–85.

UML. 2005. Unified modeling language superstructure specification, v2.0. OMG specification, OMG.

VANDERPERREN, W., SUVÉE, D., CIBRÁN, M. A., AND FRAINE, B. D. 2005. Stateful aspects in JAsCo. In *Software Composition*. T. Gschwind, U. Aßmann, and O. Nierstrasz Eds., Lecture Notes in Computer Science, vol. 3628, Springer, 167–181.

WHITTLE, J., KWAN, R., AND SABOO, J. 2005. From scenarios to code: An air traffic control case study. *Softw. Syst. Model. 4*, 1, 71–93.

Received May 2008; revised June 2009, September 2009; accepted November 2009