From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ [*]

(preliminary version)

Shahar Maoz
shahar.maoz@weizmann.ac.il

David Harel
dharel@weizmann.ac.il

The Weizmann Institute of Science, Rehovot, Israel

ABSTRACT

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 show how to compile multi-modal 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 specifications, we follow the ideas behind the LSC play-out algorithm to coordinate the simultaneous monitoring and direct execution of the specified scenarios. We demonstrate our compilation scheme using a small application whose inter-object behaviors are specified using LSCs.

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

General Terms: Design, Languages.

Keywords: Aspect Oriented Programming, Code Generation, Inter-object Approach, Live Sequence Charts, Scenario-based Programming, Scenarios, UML Sequence Diagrams, Visual Formalisms.

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; see [13, 23]. Also, the extremely popular concept of use cases [27] has an underlying inter-object flavor, and in a way, calls for formalization and instantiation using scenarios.

[*]This research was supported by the Israel Science Foundation (grant No.287/02-1), and by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

An important challenge of the inter-object, scenario-based approach to software specification is to find ways to construct executable systems based on it [18]. Many researchers have dealt with this challenge as a synthesis problem; see, e.g., [4, 19, 33, 50], where inter-object specifications, given in variants of Message Sequence Charts (MSC) [26], are translated into intra-object state-based executable specifications for each of the participating objects or components.

"Play-out" [24] is a recent example of a different approach. Instead of synthesizing intra-object state-based specifications for each of the components, 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 occur as dictated by the specified scenarios. No intra-object model for any of the participating components is built in the process.

Play-out is not really relevant to classical MSC or UML Sequence Diagrams [48], 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 multi-modal language of Live Sequence Charts (LSC) [13], 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 [36], and it has been used in the context of testing and formal verification (see, e.g., [12, 37]). However, the feature of LSC most relevant to the present paper 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.[1]

To date, the Play-Engine tool [23] contains the two implementations of play-out available, naïve play-out (or simply play-out) and smart play-out [20]. Both 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.

[1]In [21], we have shown how UML 2.0 Sequence Diagrams can be extended to encompass the multi-modal nature of LSCs, leading to executability for them too.

Aspect-Oriented Programming (AOP) has been proposed as a mechanism that enables the modular implementation of crosscutting concerns [30]. Separation of concerns provides better comprehensibility, reusability, traceability, and evolvability of software artifacts [15], and its realization is an important achievement of the software engineering community. Most relevant to our work, though, is that the aspect-oriented paradigm and the scenario-based approach to software specification are inherently similar: 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 paper 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 compile LSCs onto AspectJ, thus providing a non-interpreted means for executing multi-modal scenarios. As such, our compilation can also be viewed as full code generation from a visual formalism, similar to that carried out by tools like StateMate, Rhapsody [25], and RoseRT [43]. 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 (see [28]), and enabling links with standard tools and development environments.

In the rest of the paper, 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 [23] 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.

Our process of high-level compilation takes scenario-based specifications given in LSC and compiles them into what we shall be calling *Scenario Aspects*, implemented as AspectJ aspects.[2] The generated code is then compiled/linked with an existing or separately implemented Java program 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.

The paper is organized as follows. We start with a short review of the LSC language and the key concepts of Aspect-Oriented Programming. We then introduce the compilation scheme main idea. Section 4 gives a short overview of the compilation scheme implementation. In Section 5 we demonstrate the compilation scheme using an example application. We conclude with related work, discussion, and future work.

[2]We chose AspectJ because it is the most popular implementation of aspects to date and it suffices for our needs. However, our compilation scheme is general and can be adopted to compile to other AOP languages. A comparative discussion of other AOP languages with regard to our compilation scheme is outside the scope of this paper.

2. PRELIMINARIES

2.1 Live Sequence Charts

The language of LSCs [13] is a scenario-based visual formalism, which extends classical Message Sequence Charts (MSC [26]) with universal/existential modalities, thus allowing the specification of mandatory scenarios ("what must happen"), possible scenarios ("what may happen"), and negative scenarios ("what should never happen"). It has been extended with various rich features and the first executable semantics for it is described in [23].

2.1.1 The LSC Language

We use here a restricted subset of the LSC language, adopted to our needs, i.e., pragmatically geared towards code generation.[3] The presentation below is deliberately somewhat informal to make it accessible and to allow us to concentrate on the relevant parts to this paper, i.e., the execution mechanism. For a thorough definition of the LSC language and its operational semantics we refer the reader to [23].

We consider a system-model consisting of a set of classes C, where each class exposes a set of public attributes C_A and methods C_M (to keep the presentation below simple, we ignore types, and assume attributes are accessed by setter and getter methods). An LSC L is defined to be

$$L = \langle I_L, M_L, Cond_L \rangle$$

where I_L is a set of instance references over C (which also may include special references representing the environment, e.g., the user), M_L is a set of methods, and $Cond_L$ is a set of guards, each of which is a boolean expression over attributes and methods from (a subset of) the instances I_L (we assume expression evaluation to be side effect free). There are two types of instance references: concrete instance-level and symbolic class-level. Instance-level references refer (statically) to specific objects, while class-level references may stand for any instance of the associated class. Instance references are drawn as vertical lines and methods are drawn as horizontal lines. Time is assumed to go from top to bottom. An LSC *event* is either an actual method (for simplicity, we consider here a synchronous interpretation where calling a method and executing it are considered a single LSC event), or the act of evaluating a condition. That is:

$$E_L = M_L \cup Cond_L$$

Every instance line contains *locations*, each of which is an intersection of an instance line with an event from L. A method event defines two locations (one location 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). Naturally, we require that method events end on instance reference lines that refer to instances or classes that have them as public methods. Conditions' expressions are specified in Java syntax. We omit here obvious syntactic requirements.

[3]Expressions are specified directly in Java, and not in, e.g., OCL [41].

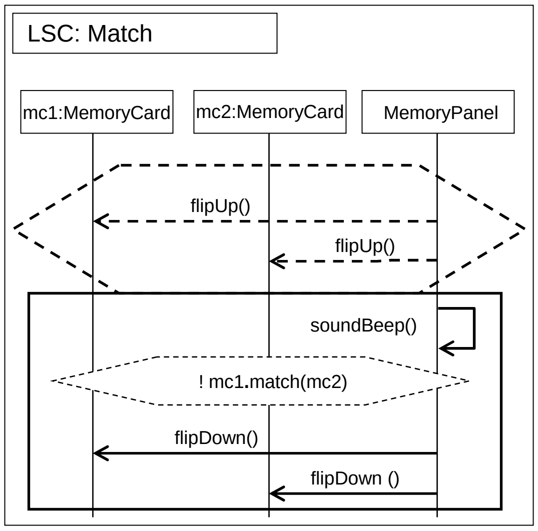

Figure 1: The LSC for `Match`.

Every LSC induces a partial order between locations: locations along a single instance line are ordered top-down, and locations that belong to the same LSC event (a method call and execution, or a condition evaluation) are at the same place in the partial order (the special case of a self method is regarded as a single location on the relevant lifeline). The partial order then, extends naturally from locations to events. Lastly, every location may be hot or cold; an event is hot if at least one of its locations is hot and is cold otherwise (see the use of the hot/cold distinction in the LSC execution mechanism described in the next subsection). Cold events are denoted by dashed lines and hot events by solid ones.

An LSC may be universal or existential. A universal LSC is associated with a pre-chart that specifies the scenario which, if successfully completed, forces the system to satisfy the scenario given in the actual chart body, the main chart. Thus, operationally, pre-chart scenarios are monitored while main chart scenarios, when reached, need to be executed. This distinction between monitoring and execution modes is key to our compilation and execution scheme as we shall show in the next section.

Existential LSCs specify sample scenarios. They can be used to specify system tests that should be monitored, but they do not affect the execution. We discuss the compilation of existential charts in Subsection 4.4.

Figure 1 shows an example universal LSC (borrowed from our initial case study, a Memory Game, described in detail in Section 5), which specifies the following requirement: *Whenever the MemoryPanel calls the `flipUP()` method of one card, and then calls the `flipUP()` method of another card, it should call its `soundBeep()` method, and the system must call the first card `match` function to check whether the two cards match. If they do not match, the memory panel must call the `flipDown()` method of the first card and the `flipDown()` method of the second card.*

2.1.2 The Play-Out Execution Mechanism

An *LSC cut* is a mapping of every instance to one of its possible locations in the LSC. An LSC cut is *hot* if one of its locations is hot and is cold otherwise. An event is *minimal* in a chart if no other event in the chart comes before it in the partial order induced by the chart. Minimal events are important in our execution mechanism: whenever an event e occurs, a new copy of each chart that features e as a minimal event is instantiated and start being monitored to see if its pre-chart completes successfully. An event e is *enabled* with respect to a cut Cut if the location in Cut of every instance participating in the event e is the one exactly prior to e. An event e *violates* a chart L in a cut Cut if e is in M_L but is not enabled with respect to Cut.

The execution mechanism reacts to the events that are (statically) referenced in one or more of the LSCs (see [23]). For each LSC instance copy, instantiated following the occurrence of a minimal event, the mechanism checks whether the event is enabled with regard to the current cut; if it is, it advances the cut accordingly; if it is violating and the current cut is cold, it discards this LSC copy; if it is violating and the current cut is hot, program execution aborts; if it does not appear in the LSC, it is ignored. Conditions are evaluated as soon as they are enabled in a cut; if a condition is evaluated to true, the cut advances accordingly; if it evaluates to false and the current cut is cold, the LSC copy is discarded; if it evaluates to false and the current cut it hot, program execution aborts. If the cut of an LSC copy reaches maximal locations on all instance reference lifelines, the LSC copy is discarded. We consider all events in precharts to be cold. Once all LSC's cuts have been updated, the execution mechanism chooses an event to execute from among the execution-enabled methods (main chart events) that are not violating any chart, if any.

Note that play-out requires careful event unification and dynamic binding mechanism. Roughly, two methods are unifiable if their senders (receivers) are concrete instance-level (or already bound) and equal, or symbolic class-level of the same class and at least one is still unbound. When methods with arguments are considered, an additional condition requires that corresponding arguments have equal concrete values, or at least one of them is free. The formal definitions of unification for LSCs can be found in [23, 40]. Our compilation scheme exploits the similarity between the unification semantics of play-out and that of AOP.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has been proposed as a mechanism that enables the modular implementation of crosscutting concerns [30]. 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 [1, 29], an extension of Java. Since we are using AspectJ in our current work, we very briefly describe the key features of AOP used in our current work, using 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* is a program construct which designates a set of join points, plus, optionally, values from the execution context of those join points. For example, a pointcut can capture the execution of a certain method along with its arguments and a reference to its target object (the instance which executes it). Pointcuts can be combined using Boolean operators. Wildcard-based syntax is used in order to construct pointcuts that capture join points that share common characteristics.

To declare the code that should execute at a join point that has been selected by a pointcut, AspectJ supports method-like constructs for *before*, *after*, and *around* advice; before advice executes prior to the join point, after advice executes following the join point, and 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 *inter-type declaration* is a static crosscutting construct which enables the introduction of new methods or members to a class.

Finally, the *aspect* is the central unit of AspectJ. It is defined by an aspect declaration, similar to that of a class declaration. 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.

3. THE COMPILATION SCHEME

We are now ready to present our compilation scheme. The key to the compilation scheme is the translation of each LSC into a *Scenario Aspect* that simulates an abstract automaton whose states represent cuts along the LSC lifelines and whose transitions are triggered by pointcuts. Each scenario aspect is locally responsible for listening for relevant events and advancing its cut state accordingly. Most importantly, the compilation scheme generates a coordinator, implemented as a separate aspect, which observes the cut state changes of all active scenario aspects, chooses a method, and executes it.

In order to build the automaton representation, the LSC must be statically analyzed. The analysis involves simulating a 'run' over the LSC, which captures all 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 violations at this cut. An additional transition from each hot cut state to a designated 'error' state corresponds to all possible violations at this cut. During code construction, the sets of execution-enabled, monitoring enabled, cold violation, and hot violation events at each cut are computed and 'stored' in the state.

An LSC cut that includes an enabled condition is not represented as a state of the scenario aspect. 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.

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'. Note also that the distinction between monitoring (pre-chart) and executing (main chart) events is not represented in the structure of the automaton, only in the information stored in each state.

Analogous automaton representations of LSC for the use in the context of formal verification were given in, e.g., [11, 32]. In [21], we used a similar automaton construction to define the semantics of Modal UML Sequence Diagrams. 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.

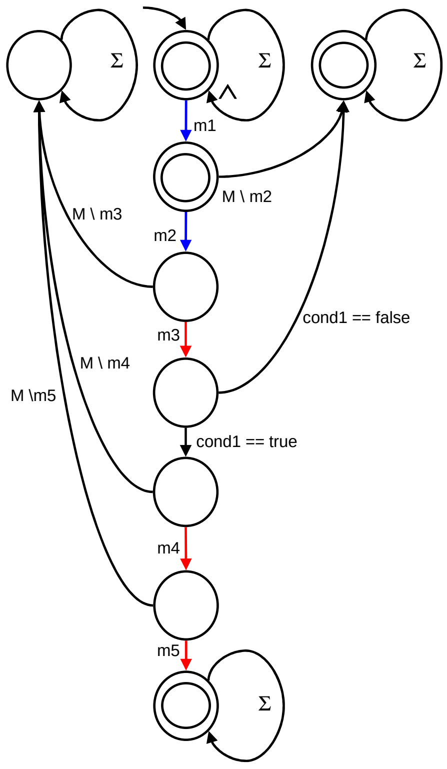

Figure 2: The automaton for LSC `Match`. `M` is the set of messages in the LSC. `m1` stands for `MemoryPanel:mc1.flipUp()`, `m2` stands for `MemoryPanel:mc2.flipUp()`, etc. `cond1` stands for `!mc1.match(mc2)`. Self transitions labeled $\Sigma \setminus M$ have been omitted.

Fig. 2 shows the automaton representing the universal LSC `Match` from Fig. 1. Note the universal quantification on the outgoing transitions of the initial state. This quantification reflects the fact that 'multiple copies' of the automaton may be active simultaneously.

As explained above, the automata are responsible for listening for system events and advancing their cut accordingly. However, they do not drive the execution. Rather, the coordinator that observes all cut state changes in active LSCs is the one responsible for choosing a method for execution and executing it.

4. IMPLEMENTATION OVERVIEW

We now give a short overview of the key features of the code generation scheme and related architecture. This conference version of the paper is limited in space hence we must leave out many implementation details.

4.1 From an LSC to a Scenario Aspect

Each LSC is compiled into a *Scenario Aspect*, implemented as an AspectJ aspect. LSC lifelines are translated into members of the relevant type. Each LSC method is translated into a `pointcut` that captures the execution of the corresponding method, together with the context of the calling object `this`, the receiving object `target`, and arguments (as applicable).

An `after` advice is associated with each pointcut. The advice binds the context of the pointcut to the corresponding members (of the respective active copies), when applicable, and calls a private method `changeCutState`. This method, built as a series of switch case statements, is responsible for advancing the scenario aspect cut state along the locations of each lifeline, and for identifying cold and hot violations as necessary. In case of a cold violation, or a successful LSC completion, all lifeline locations and bindings (of the respective active copy) are reset, and the active copy is discarded. In case of a hot violation, an appropriate exception is thrown.

Finally, each scenario aspect implements a public method `getCutState`, which assigns instances of LSC methods, including context, between four sets (monitoring enabled, execution-enabled, cold violation, and hot violation), according to the current cut state of the scenario. The `getCutState` method is called by the LSC Coordinator, described next.

4.2 The LSC Coordinator

The LSC Coordinator is implemented as another generated aspect, declared with top precedence. It is responsible for collecting cut state information from the active scenario aspects: this sums up to the four sets of LSC methods described above, including the dynamic context information of each method, when applicable. The coordinator uses a Strategy (see Subsection 4.3) to choose a method for execution. If and when a method is chosen, the coordinator executes it using (generated) inter-type declarations of implemented wrapper methods. Thus, the method is called by the actual object rather than by the coordinator aspect per se.

4.3 The Play-Out Strategy

To allow easy interchange between different play-out algorithms, we use an abstract play-out strategy class (using the Strategy design pattern [16]). Thus, a play-out algorithm needs to be encapsulated into a concrete strategy in order to be used by the LSC Coordinator.

The play-out algorithm currently implemented in our setup is the naïve play-out of [24]. Roughly, as explained above, this means making a nondeterministic choice between execution-enabled methods that are not violating any currently active LSC (see Section 7 for references to other play-out algorithms).

4.4 Compiling Existential Charts

The compilation scheme for existential charts is similar. Each existential LSC is translated into a monitoring scenario aspect. A monitoring scenario aspect does not affect execution and thus does not implement the `getCutState` method. It monitors the relevant system events, advances its cut state accordingly, and throws appropriate exceptions when completed or violated.

5. INITIAL CASE STUDY

We demonstrate our approach using a simple example of a Memory Game desktop application.[4] While the example is

[4]Writing a small memory game application is a popular programming exercise. Ours was inspired by a lab exercise given

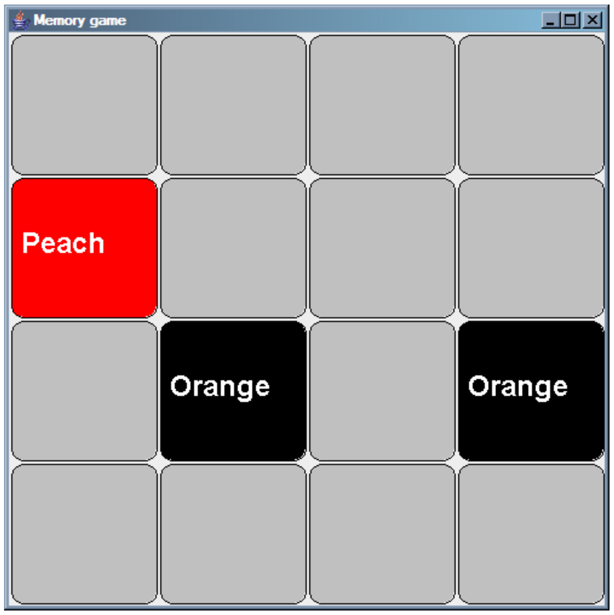

Figure 3: The GUI of the Memory Game application.

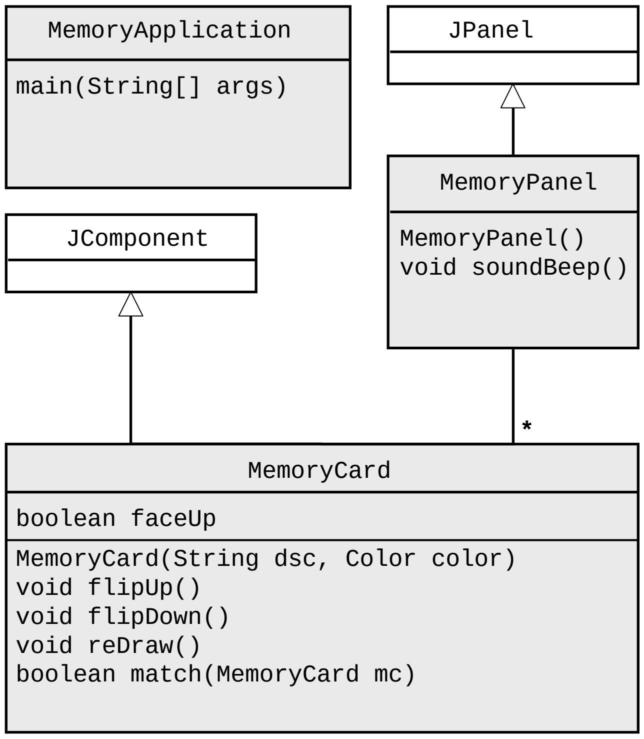

Figure 4: The class diagram for the Memory Game application.

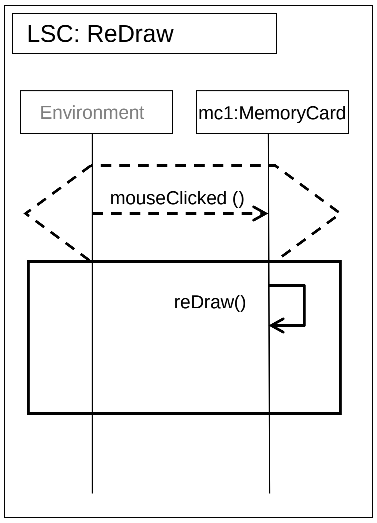

Figure 5: The LSC for `ReDraw`.

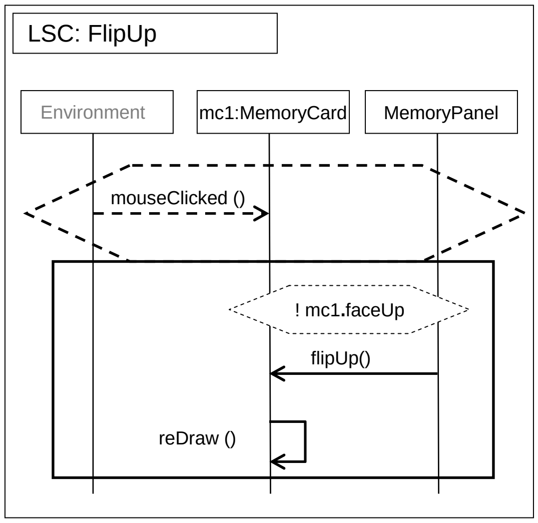

Figure 6: The LSC for `FlipUp`.

deliberately small to fit in a conference paper, it shows the key features of our approach, namely the scenario aspect generated code, the coordinated simultaneous execution of multiple scenario aspects, and the power of symbolic class-level (as opposed to instance-level) scenario-based specifications when used not only for requirements documentation but also for monitoring and direct execution. Overall, it shows the applicability of our approach to creating 'real', albeit small, actual applications, as opposed to 'simulations'.

Figure 3 shows the GUI of the Memory Game application, during a game session. It consists a panel and sixteen cards, each of which has a description and a color attached to it. The cards are instances of class `MemoryCard`. The UML class diagram in Figure 4 shows the architecture of the system. Each class defines a public interface and is internally implemented. The classes do not implement any inter-object behavior, however; i.e., the code in each class does not contain references to public members or calls to public methods of another class or between instances of the same class (except initialization during constructors execution). The three classes constitute the 'base program', on top of which the scenario aspects will be defined. We give partial informal requirements for the inter-object behavior of the Memory Game in a scenario-based fashion. These include three scenarios:

- `ReDraw`: Whenever a card is clicked, it should redraw itself.
- `FlipUp`: Whenever a card is clicked and its face is down, the panel should tell it to face up and then it should redraw itself.
- `Match`: Whenever the panel tells one card to turn face up and then tells another card to turn face up, the panel should play a beep sound, and the cards should be compared. If they do not match, the panel should tell both cards to turn face down.

Note that these scenario-based requirements are, to a large extent, overlapping: the event of clicking a card appears in

by instructor Audrey Lee in CSC112 at Smith College, available at http://maven.smith.edu/∼alee/classes/112a/labs/.

`ReDraw` and in `FlipUp`, and must be monitored by both; the event where a card redraws itself appears in both requirements too, so play-out must try to synchronize its execution between them; finally, the event where the panel tells a card to face up appears in `FlipUp` and in `Match`: it must be executed by the former and monitored by the latter, synchronically. Figures 5, 6, and 1 show the translation of the above requirements to LSCs, respectively. Note that the lifelines representing the cards are symbolic class-level instance references; i.e., they allow the same LSC to be activated with different card (or cards) each time. This is critical in making the scenario-based LSC specification concise and reusable.

The code generation process produces three scenario aspects, one for each specified LSC, and a single LSC Coordinator aspect. Fig. 7 shows a snippet of the code from `LSCMatchAspect.aj` scenario aspect including a pointcut, a corresponding advice, and excerpts from the `changeCutState` and `getCutState` methods.

Let us now trace the first few events in a possible execution of the Memory Game, to better illustrate the play-out algorithm, the event unification mechanism, and the concepts of minimal events, enabled events, and violations. A call to the method `mouseClicked()` on a card is a minimal event in both `ReDraw` and `FlipUp`. Thus, when the user clicks a card, a new copy of `ReDraw` is created, its class-level `MemoryCard` lifeline binds to the actual card instance that was clicked (target), and its cut advances to the point where the self method `reDraw()` is execution-enabled. At the same time, a copy of FlipUp is created, and its class-level `MemoryCard` lifeline binds to the actual card instance that was clicked. Its cut then advances to the point where the cold condition is enabled and the condition is immediately evaluated.

If the cold condition evaluates to false, the copy of `FlipUp` is discarded. The coordinator receives the cut state information from the active scenarios (only `ReDraw` has an active copy), discovers that `reDraw()` is execution-enabled and is not violating in any other active chart, and executes it (on the bound card). As the `reDraw()` method is not minimal in any chart, no new copies are activated. The active copy of `ReDraw` then advances its cut and thus reaches completion and closes.

```
pointcut MemoryPanelMemoryCardFlipUp(MemoryPanel s, MemoryCard t):
        call(void MemoryCard.flipUp())
        && this(s) && target(t);

after (MemoryPanel s, MemoryCard t): MemoryPanelMemoryCardFlipUp(s, t) {
        changeCutState(LSCMethod.MEMORYPANEL_MEMORYCARD_FLIPUP, s, t);
}

private void changeCutState(int lscMethod, Object sourceObject, Object targetObject) {

        //...
        switch (lscMethod) {
                case LSCMethod.MEMORYPANEL_MEMORYCARD_FLIPUP:
                        if (isInCut(0,0,0)) {
                                mc1 = (MemoryCard)targetObject;
                                setCut(1,0,1);
                                return;
                        }
                        if (isInCut(1,0,1)) {
                                mc2 = (MemoryCard)targetObject;
                                setCut(1,1,2);
                                return;
                        }
                        break;

                case LSCMethod.MEMORYPANEL_MEMORYPANEL_SOUNDBEEP:
                        if (isInCut(1,1,2)) {
                                setCut(1,1,3);
                                if (!mc1.match(mc2)) {
                                        setCut(2,2,4);
                                        return;
                                }
                        }
                        break;

                case LSCMethod.MEMORYPANEL_MEMORYCARD_FLIPDOWN:
                        //...
        }
        LSCCompletion();
        //...
}

public static void getCutState(HashSet mE, HashSet eE, HashSet cV, HashSet hV) {

        //...
        if (isInCut(2,2,4)) {

                // LSCMethod.MEMORYPANEL_MEMORYCARD_FLIPUP from MemoryPanel to mc1 is cold violation
                cV.add(LSCMethod_MemoryPanelmc1flipUp);

                // LSCMethod.MEMORYPANEL_MEMORYCARD_FLIPDOWN from MemoryPanel to mc1 is execution enabled
                eE.add(LSCMethod_MemoryPanelmc1flipDown);

                //...
        }
        //...
}
```

Figure 7: Code snippet from the `LSCMatchAspect.aj` scenario aspect including a pointcut, a corresponding advice, and excerpts from the `changeCutState` and `getCutState` methods. The arguments for the `isInCut` and `setCut` methods refer to location numbers on the corresponding lifelines. Bookkeeping code to manage multiple active copies, handling arguments, and many other details have been omitted from the figure.

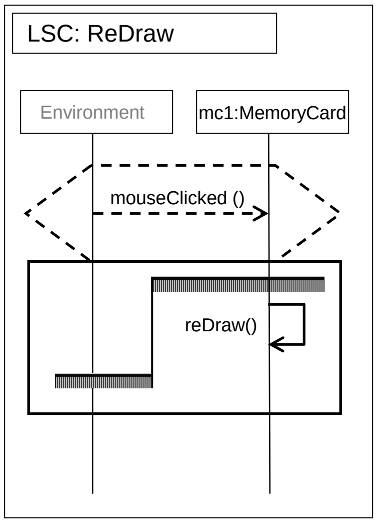

Figure 8: An active copy of the LSC for `ReDraw`. The cut is denoted by a comb-like thick line. The method `reDraw()` is execution-enabled.

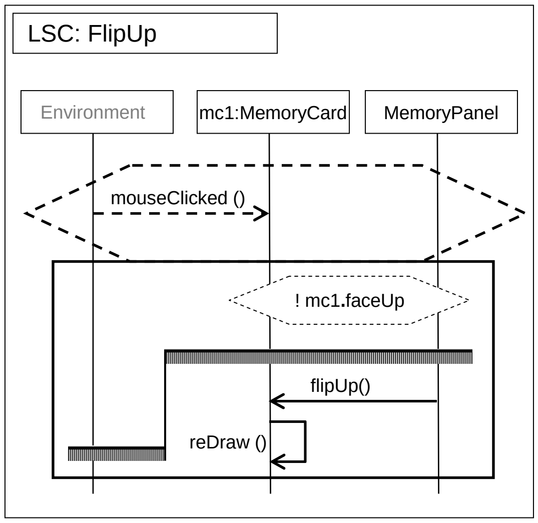

Figure 9: An active copy of the LSC for `FlipUp`. The cut is denoted by a comb-like thick line. The method `flipUp()` is execution-enabled while the method `reDraw()` constitutes a hot violation.

Otherwise, if the cold condition evaluates to true, the copy of `FlipUp` advances its cut further, to the point where the method `flipUp()` is enabled and `reDraw()` is a hot violation (see Figures 8,9). Observing this, the coordinator discovers that the method `flipUp()` is the only execution-enabled method which does not violate any active chart (the `reDraw()` method is enabled in the active copy of `ReDraw` but is violating in the active copy of `FlipUp`). Note that the two `reDraw()` methods are unified because they are bound to the same source and target object. So, the coordinator executes the `flipUp()` method (this is done using an inter-type declaration of a wrapper method; thus the actual call comes from the `MemoryPanel`, not from the coordinator per se).

A `flipUp()` method call from the `MemoryPanel` to a card is a minimal event in `Match`. Thus, as it is executed, a new copy of `Match` is created, its `mc1` class-level `MemoryCard` lifeline binds to the actual card that received the call, and its cut is advanced accordingly. At the same time, the active copy of `FlipUp` identifies the execution of its enabled event and advances its cut accordingly. Next, since `reDraw()` is execution-enabled in both `FlipUp` and `ReDraw` and is not violating in any other active copy, the coordinator executes it.

The above should help to better understand the play-out execution mechanism and its implementation in our scheme. We leave the analysis of other possible executions to the interested reader.

6. RELATED WORK

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

The Play-Engine [23] is an experimental tool for requirements capture and direct execution of LSCs, based on the play-in/play-out approach [24]. 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.[5] InterPlay [8] coordinates the simulation of the Play-Engine and a separately executed program, such as another Play-Engine, or a tool like Rhapsody [25] or RoseRT [43], given 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, so the result is a single executable program, despite the use of different multiple modeling methodologies in the requirements specification and coding phase.

In [34], Krueger et al. propose a translation of 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 [33], 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, bypasses the need for this kind of synthesis. In more recent work [35], Krueger et al. translate scenarios into aspects and let the weaver resolve some of the resulting non-determinism in a random fashion. Coordination between non-disjoint scenarios, i.e., synchronizing the interactions around common messages, is done statically and only between overlapping scenarios that the designer has explicitly specified to be ‘joined’.

Deubler et al.[14] 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 [34].

[5]Play-in is a user-friendly high-level way of specifying behavior and automatically generating the specification formally in LSCs [18, 23]. While it is outside the scope of our work, it should be clear that play-in can be combined with our present work too.

Whittle et el.[50] 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 [6], the authors present a requirements level aspect-oriented modeling approach. Both papers discuss the aspectual nature of crosscutting requirements, but use a synthesis algorithm, which results in a state-machine for each of the participating components.

Uchitel et al. in [46, 47] 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, e.g., detecting implied scenarios, or by the use of architectural information to synthesize the behavior of components types rather than that of instances. It seems that we and they agree on the intuitive nature and usefulness of scenario-based notations. 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 we would add a process that starts from basic scenarios and incrementally elaborates them with modalities.[6]

Stolz and Bodden [44] present a nicely built runtime verification framework for Java programs, where properties specified in LTL 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 by Kiviluoma et al.[31], who use generated aspects to simulate a small state machine that monitors behavioral requirements given as UML Sequence Diagrams. Since LSCs can be translated into LTL formulas (see [36]), these two papers have some similarities with ours, specifically in the possibility of using our code generation scheme to monitor existential charts. 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 is irrelevant to these two papers.

In [17], Groher et al. 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 the UML to model aspects, or suggest to use UML like notations or new profiles specifically for this purpose (see, e.g., [5, 9, 10, 39]). 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 et al. [28] 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 [28] as 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 [23]).

[6]A complete methodology for the use of our compilation scheme in a development process is, however, outside the scope of this paper and will be addressed separately.

Finally, some advanced approaches to AOP are compared to our work. Tracematches [3] is an extension of the AspectJ abc compiler [7], 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 class-level LSC lifelines. Another relevant approach is that of Stateful Aspects [49], implemented in the JasCo language [45], where pointcuts can declaratively specify protocol fragments equivalent to a finite state machine, 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 to simplify our code generation scheme and improve the efficiency of the final executable program. However, the two approaches are limited to regular protocols. Since non-regular protocols can be specified in LSC (using variables and unbounded loops), this limits their applicability to our work. Moreover, both approaches apply to 'local' traces, while play-out, and hence our compilation scheme, requires, in addition, simultaneous coordination between the traces, which is not addressed by these approaches.

7. DISCUSSION 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.

The main contribution of our work is that it translates 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 to software specification with mainstream software engineering.

Still, some possible drawbacks of our approach should be mentioned and addressed. The use of a high-level programming language necessarily entails some level of suboptimal performance. Using an inter-object scenario-based language makes the situation worse, since it seems to require central coordination at runtime, or alternatively, the construction of a very large state machine. This may require additional ideas to our approach when applied to large scale systems or where performance is of high importance.

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 execution, not necessarily between objects but between concurrently active coordinators.

The original play-out process, as described in [23, 24] and implemented in our present scheme, is rather naïve. Specifically, some of the sequences of events possible as a response to a user event may eventually lead to violations, and these cannot be avoided by the play-out process. Moreover, the partial order semantics among events in each chart and the ability to separate scenarios in different charts without having to say explicitly how they should be composed are very useful in early requirement stages, but can cause under-specification and nondeterminism when one attempts to execute them. Smart play-out [20] partially addresses these limitations using model-checking techniques, which, e.g., can be used to avoid violations within a super-step. Packaging smart play-out as a play-out strategy that can be used by the coordinator in our scheme should not be a problem.

Tool support for our compilation scheme is crucial. Specifically, this includes the full automation of the compilation process, integration with a visual editor, and some more advanced features such as a run-time animation option, as is available for example in Rhapsody and the Play-Engine. We have started this implementation, and plan to package our tool as an Eclipse [2] plug-in. A detailed presentation of our tool will be given in a future paper.

Our current compilation scheme does not cover the full LSC language. For some constructs, such as if-then-else and switch-case subcharts, various types of loops, and internal variables, extending the compilation scheme is straightforward. LSC's support of symbolic instances and parameterized methods, as defined and implemented in the Play-Engine, does not cover class inheritance. Our compilation scheme already supports symbolic class-level instances. Moreover, since we use Java as the target language, a unification mechanism that supports class inheritance is very natural to our compilation scheme and its implementation is very easy.

Adding support for explicit time and real-time, as was partly done for the Play-Engine in [22], however, is more challenging. Specifically, our current compilation scheme does not support multi-threaded programs and in general, assumes the system to be infinitely faster than its environment. Handling coordination in real-time and multi-threaded programs is thus an interesting topic for future research.

Another path for future work involves the compilation scheme itself. First, further, more global, static analysis may be performed during compilation to optimize the generated code. For example, the execution of methods that appear only in a single LSC can be performed locally, eliminating the need for coordination. Second, it is important to investigate the time and space complexity of the compilation phase as a function of the number of lifelines and events in each LSC and the number of classes and LSCs in the specification. We note, though, that as the construction of the abstract automaton that simulates the runs of the LSC does not require information from other LSCs, the compilation of each LSC is independent of the rest of the specification. Thus, the code generation process can be carried out 'locally' and its complexity should not be a major concern.

As discussed in the previous section, some aspect languages other than AspectJ, e.g., [42, 49], or extensions of AspectJ, e.g., [3], include advanced features that could be exploited by our compilation scheme in order to produce code that might be more succinct, accessible to a human programmer, or allow for more efficient execution using various methods of runtime weaving. Considering other target languages is thus a possible direction for future research and implementation work.

With regard to the hybrid approach to system modeling and execution we experiment with in this paper, we should carefully consider the question of what is the best way to specify the distinction between monitoring and execution modes when using scenario-based specifications. In the simplest case, a clear separation between intra-object and inter-object specification (disjoint alphabets a la InterPlay [8]) is assumed. In practice, however, it might be important to relax this assumption, in which case a new pair of modalities, execute and monitor, would be required. These would generalize the notion of pre-chart/main-chart, since each method would have a mode: monitor/execute. The new modalities would be orthogonal to the may vs. must (cold/hot) modalities.

Finally, in [21], we discussed the integration of the key features of the LSC language, namely the universal/existential distinction and hot/cold modalities, into the UML standard, and proposed Modal UML Sequence Diagrams (MUSD) as a UML 2.0 profile whose semantics is based on LSCs, and thus allows the specification of required and forbidden behaviors. We are currently working on formalizing the addition of the monitoring/execution modes suggested above to MUSD, so as to allow the definition of operational semantics for MUSD based on the operational semantics of LSC. This generalizes the notion of pre-chart/main-chart to the monitoring/execution modes at the message and interaction fragment level. Note that the compilation scheme presented here already supports this generalization [7]. Thus, the result is a compilation scheme from Multi-modal UML Sequence Diagrams to AspectJ code.

Acknowledgements

We would like to thank Mark Mahoney and Tzilla Elrad for pointing us to their work on modeling aspects for reactive systems using LSCs [38, 39] and for subsequent personal correspondence, which 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.

Finally, we thank Yishai Feldman and the anonymous referees for their useful comments.

8. REFERENCES

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

[2] Eclipse.org. http://www.eclipse.org/.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 345–364. ACM, 2005.

[4] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. 22nd Int. Conf. on Soft. Eng. (ICSE'00)*, pages 304–313, New York, NY, 2000. ACM Press.

[5] J. Araújo, A. Moreira, I. Brito, and A. Rashid. Aspect-oriented requirements with UML. In M. Kandé, O. Aldawud, G. Booch, and B. Harrison, editors, *Workshop on Aspect-Oriented Modeling with UML*, 2002.

[7] Indeed, the tool we are working on compiles the more general and standard executable MUSDs rather than LSCs.

[6] J. Araujo, J. Whittle, and D.-K. Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *Proc. Requirements Engineering Conf. (RE'04) at the 12th IEEE International*, pages 58–67, Washington, DC, USA, 2004. IEEE Computer Society.

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proc. 4th Int. Conf. on Aspect-Oriented Software Development (AOSD'05)*, pages 87–98. ACM Press, 2005.

[8] D. Barak, D. Harel, and R. Marelly. InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 66–86. Springer-Verlag, 2004.

[9] E. Barra, G. Génova, and J. Llorens. An approach to aspect modeling with UML 2.0. In *Proc. 5th Int. Workshop on Aspect-Oriented Modeling*, October 2004.

[10] M. Basch and A. Sanchez. Incorporating aspects into the UML. In *Proc. 3rd Int. Workshop on Aspect-Oriented Modeling*, 2003.

[11] Y. Bontemps and P. Heymans. Turning high-level live sequence charts into automata. In *Proc. 1st Int. Workshop on Scenarios and State-machines (SCESM'02) at the 24th Int. Conf. on Soft. Eng. (ICSE'02)*, Orlando, FL, May 2002.

[12] P. Combes, D. Harel, and H. Kugler. *Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-Engine Tool*, volume 3707 of *LNCS*, pages 414–428. Springer-Verlag, 2005.

[13] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).

[14] M. Deubler, M. Meisinger, S. Rittmann, and I. Krüger. Modeling crosscutting services with UML sequence diagrams. In L. C. Briand and C. Williams, editors, *MoDELS*, volume 3713 of *LNCS*, pages 522–536. Springer, October 2005.

[15] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Comminications of the ACM*, 44, 2001.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] I. Groher and S. Schulze. Generating aspect code from UML models. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th Aspect-Orianted Software Development Modeling With UML Workshop*, 2003.

[18] D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.

[19] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science (IJFCS)*, 13(1):5–51, Febuary 2002. (Also, *Proc. 5th Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS, Springer-Verlag, 2000).

[20] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *LNCS*, pages 378–398, 2002.

[21] D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In *Proc. 5th Int. Workshop on Scenarios and State-machines (SCESM'06) at the 28th Int. Conf. on Soft. Eng. (ICSE'06)*, Shanghai, 2006.

[22] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, Texas, 2002.

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

[24] D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling (SoSyM)*, 2(2):82–107, 2003.

[25] I-Logix,Inc. http://www.ilogix.com/.

[26] ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.

[27] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[28] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. 2072:327–355, 2001.

[30] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proc. European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, Berlin, 1997.

[31] K. Kiviluoma, J. Koskinen, and T. Mikkonen. Run-Time Monitoring of Behavioral Profiles with Aspects. In *Proc. 3rd Nordic Workshop on UML and Software Modeling*, pages 62–76. University of Tampere, August 2005.

[32] J. Klose and H. Wittke. An automata based interpretation of live sequence chart. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of LNCS, Springer-Verlag*, 2001.

[33] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *DIPES*, volume 155 of *IFIP Conf. Proc.*, pages 61–72. Kluwer, 1998.

[34] I. Krüger, R. Mathew, and M. Meisinger. From Scenarios to Aspects: Exploring Product Lines. In *Proc. 4th Int. Workshop on Scenarios and State-machines (SCESM'05) at the 27th Int. Conf. on Soft. Eng. (ICSE'05)*. ACM Press, 2005.

[35] I. Krüger, R. Mathew, and M. Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proc. 28th Int. Conf. on Soft. Eng. (ICSE'06)*, pages 62–71, 2006.

[36] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 445–460. Springer, 2005.

[37] M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time UML models. In *4th Int. Conf. on the Unified Modeling Language, Toronto*, October 2001.

[38] M. Mahoney. Modeling Reactive Systems and Aspect-Orientation. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 345–346. Springer, 2005.

[39] M. Mahoney and T. Elrad. Weaving crosscutting concerns into Live Sequence Charts using the Play-Engine. In *7th Int. Workshop on Aspect-Oriented Modeling*, October 2005.

[40] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proc. 17th ACM Conf. on Object-Oriented Prog., Systems, Lang. and App. (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.

[41] OCL. UML 2.0 Object Constraint Language. OMG specification, OMG, 2005.

[42] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In A. P. Black, editor, *ECOOP*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[43] IBM Rational Rose Technical Developer (includes Rational Rose RealTime). http://www-306.ibm.com/software/awdtools/developer/technical/.

[44] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *5th Workshop on Runtime Verification (RV'05)*, July 2005.

[45] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD*, pages 21–29, 2003.

[46] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. System architecture: the context for scenario-based model synthesis. In *Proc. 12th ACM SIGSOFT Int. Symp. on Foundations of Soft. Eng. (FSE-12)*, pages 33–42, New York, NY, 2004.

[47] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Soft. Eng. Method.*, 13(1):37–85, 2004.

[48] UML. Unified Modeling Language Superstructure Spec., v2.0. OMG specification, OMG, August 2005.

[49] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful Aspects in JAsCo. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, volume 3628 of *LNCS*, pages 167–181. Springer, 2005.

[50] J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: An air traffic control case study. *Software and System Modeling*, 4(1):71–93, 2005.