S2A: A Compiler for Multi-Modal UML Sequence Diagrams [⋆]

David Harel, Asaf Kleinbort, and Shahar Maoz

The Weizmann Institute of Science
{dharel,asaf.kleinbort,shahar.maoz}@weizmann.ac.il

Abstract. We report on S2A, a compiler that translates Modal UML Sequence Diagrams (MSDs), a UML-compliant version of Live Sequence Charts (LSCs), into AspectJ code. It thus provides full code generation of reactive behavior from visual inter-object scenario-based specifications. The S2A compiler is based on a compilation scheme presented by Maoz and Harel in [13].

1 Introduction

An important challenge of the inter-object, scenario-based approach to software specification is to find ways to construct executable systems based on it [3]. Many researchers have dealt with this challenge as a synthesis problem; see, e.g., [4, 10, 15], where inter-object specifications, given in variants of Message Sequence Charts (MSC) [9], are translated into intra-object state-based executable specifications for each of the participating objects. "Play-out" [8] is a recent example of a different approach. Instead of synthesizing intra-object state-based specifications for each of the objects, the play-out algorithm executes scenarios directly, keeping track of all user and system events for all objects 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 was defined for Live Sequence Charts (LSCs) [2, 7], a rich extension of MSC that supports multi-modal scenario specifications.

In this paper we present S2A, a compiler that implements the play-out execution mechanism by translating inter-object scenario-based specifications, given in a variant of LSC that is UML-compliant — Modal Sequence Diagrams (MSD) — into AspectJ code. S2A is based on a compilation scheme presented by Maoz and Harel in [13]. It exploits the inherent similarity between the scenario-based approach and the aspect-oriented approach to software specification — in both, part of the system's behavior is specified in a way that explicitly crosses the boundaries between objects — and takes advantage of the similar unification semantics of play-out and AspectJ pointcuts. We consider S2A a significant step towards realizing the promise of visual scenario-based programming within real world software engineering.

[⋆] This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

Section 2 very briefly reviews the MSD language, the play-out execution mechanism, and the compilation scheme; Section 3 concludes with a short discussion and future work directions.

2 Overview of S2A

MSD and LSC The language of Modal Sequence Diagrams (MSD) is a visual formalism for scenario-based inter-object specifications, defined as proper UML profile that extends UML 2 Interactions [14] with a `<<modal>>` stereotype consisting of two attributes: *mode* and *execution mode.* Each element in an MSD interaction, e.g., a message, a constraint, has a *mode* attribute which can be either *hot* (universal) or *cold* (existential), and an *execution mode*, which can be either *monitor* or *execute.* MSD is a more standardized and slightly generalized version of Live Sequence Charts (LSC) [2] (i.e., in MSD, monitoring and execution are not divided into pre-charts and main-charts, *mode* and *execution mode* are orthogonal). The semantics of MSD is based on that of LSC, whose expressive power is comparable to that of various temporal logics [11] (a trace-based semantics for MSD was given in [6] using alternating Büchi automata). Thus, MSD allows not only to specify traces that "may happen", "must happen", or "should never happen", but also to divide the responsibility for execution between the environment, the participating objects, and the coordination mechanism. MSD notation is adopted from LSC: hot (resp. cold) elements are colored in red (resp. blue), execution (resp. monitoring) elements use solid (resp. dashed) line.

MSD Play-Out MSD play-out is based on LSC play-out, presented by Harel and Marelly in [8][1]. Roughly, the execution mechanism reacts to events that are referenced in one or more of the MSDs; for each active MSD, instantiated following the occurrence of a minimal event in the partial-order induced by the diagram, 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 (a cut is cold if all its elements are cold and is hot otherwise), it discards this active MSD copy; if it is violating and the current cut is hot, program execution aborts; if the event does not appear in the MSD, 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; if it evaluates to false and the current cut is cold, the MSD copy is discarded; if it evaluates to false and the current cut it hot, program execution aborts. If the cut of an active MSD copy reaches maximal locations on all lifelines, the active MSD is discarded. Once all MSD's cuts have been updated, the execution mechanism chooses an event to execute from among the execution-enabled methods that are not violating any chart, if any.

Play-out requires careful event unification and dynamic binding mechanism. Roughly, two methods are unifiable if their senders (receivers) are concrete

[1] For a thorough definition of the LSC language and its operational semantics we refer the reader to [7].

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 that at least one of them is free[2]. To implement this, S2A exploits the similarity between the unification semantics of play-out and that of AOP.

The Compilation Scheme We briefly review the compilation scheme (see [13] for details). S2A translates each MSD into a *Scenario Aspect*, implemented in AspectJ. The Scenario Aspect simulates an alternating Büchi automaton whose states represent possible MSD cut states and whose transitions are triggered by aspect pointcuts. Each Scenario Aspect is locally responsible for listening to relevant events and advancing its cut accordingly. To construct the automaton, S2A statically analyzes the MSD by simulating a 'run' that captures all possible cuts; each cut is represented by a state; transitions correspond to enabled events.

In addition, and most importantly, S2A generates a single *coordinator* aspect, which 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, as necessary, uses a *strategy* (see below) to choose a method for execution. It then executes it using inter-type declarations inside generated wrapper methods.

The *strategy* is responsible for choosing the next method to execute, based on the information collected by the *coordinator*. S2A installation includes a default play-out strategy that implements basic (naïve) play-out (arbitrarily choosing a non-violating method from among the currently enabled methods). The user can implement a new strategy (by implementing the `IPlayOutStrategy` interface) and point the compiler to it in the compiler's configuration file.

3 Conclusions and Future Work

We presented S2A, a compiler for multi-modal UML Sequence Diagrams. S2A currently supports the following MSD language features: hot/cold (universal/existential) method calls and conditions, symbolic lifelines (class hierarchies and interfaces), symbolic, exact, and opaque method arguments, dynamic creation of objects, control structures (if-then-else, bounded and unbounded loops, switch-case), and (one step) anti-scenarios.

While we concentrate on scenario-based program *construction*, S2A can also be used for scenario-based *testing* of Java programs in general. This includes test execution and monitoring, similar to that implemented by tools such as the Rhapsody TestConductor [12]. No prior assumptions on the Java program under test are necessary.

We are continuing to develop S2A in a number of directions. First, the compilation scheme can be easily modified to generate code in aspect languages other than AspectJ (e.g., AspectC++) or extensions of AspectJ (e.g., TraceMatch [1]),

[2] The formal definitions of unification for LSCs can be found in [7].

possibly taking advantage of specific AOP features we have not yet exploited (e.g., aspect instantiation). This will create opportunities for code optimization as well as for widening the applicability of our approach to other application domains (e.g., embedded systems). Second, we are planning to implement better play-out strategies, such as *Smart Play-Out* [5], which uses model-checking to reduce nondeterminism in LSC execution. Third, we are enhancing the compiler with scenario-aware debugging capabilities (e.g., supporting breakpoints at the scenario's cut-state level).

UML models, source and generated code, executables of case studies, installation guide, as well as other resources, are available online at the S2A website: `http://www.wisdom.weizmann.ac.il/~maozs/s2a/`.

References

1. 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 *Proc. 20th Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 345–364, 2005.
2. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
3. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.
4. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Comp. Science (IJFCS)*, 13(1):5–51, Feb. 2002.
5. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Comp.-Aided Design (FMCAD'02), Portland, Or.*, volume 2517 of *LNCS*, pages 378–398, 2002.
6. 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.
7. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003.
8. D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling*, 2(2):82–107, 2003.
9. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.
10. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *DIPES*, volume 155 of *IFIP Proc.*, pages 61–72. Kluwer, 1998.
11. 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.
12. M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time UML models. In *Proc. 4th Int. Conf. on The UML*, Toronto, October 2001.
13. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proc. 14th Int. Symp. Foundations of Software Engineering (FSE-14)*, Portland, Oregon, November 2006.
14. UML. Unified Modeling Language Superstructure Spec., v2.0. OMG, August 2005.
15. 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.