Model-Based Traces ⋆

(preliminary version)

Shahar Maoz

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
shahar.maoz@weizmann.ac.il

Abstract. We introduce *model-based traces*, which trace behavioral models of a system's design during its execution, allowing to combine model-driven engineering with dynamic analysis. Specifically, we take visual inter-object scenario-based and intra-object state-based models (sequence charts and statecharts) used for a system's design, and follow their activation and progress as they come to life at runtime, during the system's execution. Thus, a system's runtime is recorded and viewed through abstractions provided by behavioral models used for its design. We present two example applications related to the automatic generation and visual exploration of model-based traces and suggest a list of related challenges.

1 Introduction

Transferring model-driven engineering artifacts and methods from the early stages of requirements and specification, during a system's design, to the later stages of the lifecycle, where they would aid in the testing, analysis, maintenance, evolution, comprehension, and manipulation of running programs, is an important challenge in current model-driven engineering research.

In this paper, as a means towards this end, we introduce *model-based traces*, which trace behavioral models from a system's design during its execution, allowing to combine model-driven engineering with dynamic analysis. Specifically, we take visual inter-object scenario-based and intra-object state-based models (sequence diagrams and statecharts) used for a system's design, and follow their activation and progress as they come to life at runtime, during the execution of the system under investigation. Thus, a system's runtime is recorded and viewed through abstractions provided by models used for its design.

An important feature of model-based traces is that they provide enough information to reason about the executions of the system and to reconstruct and replay an execution (symbolically or concretely), exactly at the abstraction level defined by its models. This level of model-based reflection seems to be a necessary requisite for the kind of visibility into a system's runtime required for model-based dynamic analysis and adaptation.

Additional features worth noting. First, model-based traces can be generated and defined based on partial models; the level of abstraction is defined by the

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

modeler. Second, the models used for tracing are not necessarily reflected explicitly in the running program's code; rather, they define a separate viewpoint, which in the process of model-based trace generation is put against the concrete runtime of the program under investigation. Third, the same concrete runtime trace may result in different model-based traces, based on the models used for tracing; and vice versa, different concrete runtime traces may result in equal model-based traces, if the concrete runs are equivalent from the more abstract point of view of the model used for tracing.

In the next section we briefly introduce, informally define, and discuss the format and features of model-based traces, using a simple example. We then present two example applications related to the automatic generation and visual exploration of model-based traces. Finally, we suggest a list of related challenges.

2 Model-Based Traces

The use of system's execution traces for different analysis purposes requires different levels of abstraction, e.g., recording CPU register assignments, recording virtual machine commands, or recording statements at the code level. We suggest a higher level of abstraction over execution traces, based on behavioral models typically used for a system's design, such as sequence diagrams and statecharts.

In this work we present two types of model-based traces, inter-object scenario-based traces and intra-object state-based traces. Additional types may be created by combining variants of the two or using other modeling techniques[1].

Given a program P and a behavioral model M, a model-based execution trace records a run r of P at the level of abstraction induced by M. A unification mechanism is defined, which statically and dynamically maps concrete elements of the run to elements in the model. The type of the model used, the artifacts and their semantics, define the types of entries that appear in the model-based trace. We demonstrate our ideas using two concrete examples of a scenario-based trace and a state-based trace, taken from a small example system.

Note that although there are code generation schemes for the *execution* of the models we use, we do not, in general and in the example given here, consider tracing programs whose code was automatically generated from models. On the contrary, we believe that one of the strengths of our approach is that it can be applied to systems in general, not only to ones where the implementation explicitly reflects certain high-level models.

Also note that the model-based traces we present are not mere projections of the concrete runtime information onto some limited domain. Rather, we use *stateful* abstractions, where trace entries depend on the history and context of the run and the model; the model-based trace not only filters out irrelevant information but also adds model specific information (e.g., information about entering and exiting 'states' that do not appear explicitly in the program).

A small example Consider an implementation of the classic PacMan game. PacMan consists of a maze, filled with dots, power-ups, fruit and four ghosts. A

[1] In principle, any representation of an execution trace may be considered a model-based trace, depending on the definition of what constitutes a model.

human player controls PacMan, who needs to collect as many points as possible by eating the objects in the maze. When a ghost collides with PacMan, it loses a life. When no lives are left, the game is over. However, if PacMan eats a power-up, it is temporarily able to eat the ghosts, thus reversing roles. When a ghost is eaten, it must go back to the jail at the center of the maze before leaving again to chase PacMan. When all dots are eaten, the game advances to the next – more difficult – level. We consider the PacMan game to be a well-known, intuitive, relatively small and yet complex enough reactive system, hence a good choice for the purpose of demonstrating model-based traces in this paper.

2.1 Scenario-based models

For inter-object scenario-based modeling, we use a UML2-compliant variant of Damm and Harel's *live sequence charts* (LSC) [4,9]. Roughly, LSC extends the partial order semantics of sequence diagrams in general with a universal interpretation and must/may (hot/cold) modalities, and thus allows to specify scenario-based liveness and safety properties. Must (hot) events and conditions are colored in red and use solid lines; may (cold) events and conditions are colored in blue and use dashed lines. A specification typically consists of many charts, possibly interdependent, divided between several use cases (our small PacMan example has 9 scenarios divided between 3 use cases).

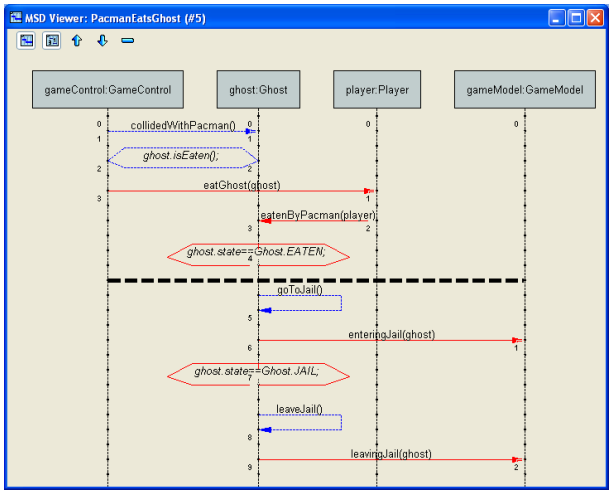

Fig. 1. The LSC for `PacManEatsGhost` with a cut displayed at (3,4,2,0).

Fig. 1 shows one LSC taken from our example model of PacMan. Vertical lines represent specific system objects and time goes from top to bottom. Roughly, this scenario specifies that *"whenever a* `gameControl` *calls a ghost's* `collidedWithPacman()` *method and the ghost's* `isEaten()` *method evaluates to TRUE, the* `gameControl` *must tell the* `player` *(PacMan) to eat the* `ghost`*, the* `player` *must tell the* `ghost` *it has been eaten, and the ghost's* `state` *must change to* `EATEN`*. Then, if and when the* `ghost` *goes to jail it must tell the* `gameModel` *it has gone there and its* `state` *should change to* `JAIL`*, etc..."* Note the use of hot

'must' elements and cold 'may' elements. Also, note the use of symbolic instances (see [15]): the lifeline representing `ghost` may bind at runtime to any of the four ghosts (all four are instances of the class `Ghost`).

An important concept in LSC semantics is the *cut*, which is a mapping from each lifeline to one of its locations (note the tiny location numbers along the lifelines in Fig. 1, representing the state of an active scenario during execution). The cut `(3,4,2,0)`, for example, comes immediately after the hot evaluation of the ghost's state. A cut induces a set of enabled events — those immediately after it in the partial order defined by the diagram. A cut is hot if any of its enabled events is hot (and is cold otherwise). When a chart's minimal event occurs, a new instance of it is activated. An occurrence of an enabled method or `true` evaluation of an enabled condition causes the cut to progress; an occurrence of a non-enabled method from the chart or a `false` evaluation of an enabled condition when the cut is *cold* is a *completion* and causes the chart's instance to close gracefully; an occurrence of a non-enabled method from the chart or a `false` evaluation of an enabled condition when the cut is *hot* is a *violation* and should never happen if the implementation is faithful to the specification model. A chart does not restrict events not explicitly mentioned in it to occur or not to occur during a run (including in between events mentioned in the chart).

2.2 Scenario-based traces

Given a scenario-based specification consisting of a number of LSCs, a *scenario-based trace* includes the activation and progress information of the scenarios, relative to a given program run. A trace may be viewed as a projection of the full execution data onto the set of methods in the specification, plus, significantly, the activation, binding, and cut-state progress information of all the instances of the charts (including concurrently active multiple copies of the same chart). Thus, our scenario-based traces may include the following types of entries:

- **Event occurrence** representing the occurrence of an event. Events are timestamped and are numbered in order of occurrence. Only the events that explicitly appear in one of the scenarios in the model are recorded in the trace (one may add identifiers of participating objects, i.e., caller and callee, and parameter values). The format for an event occurrence entry is:
  `E: <timestamp> <event no.>: <event signature>`
- **Binding** representing the binding of a lifeline in one of the active scenario instances to an object. Its format is:
  `B: <scenario name>[instance no.] lifeline <no.> <- <object identifier>`
- **Cut change** representing a cut change in one of the active scenario instances. Its format is:
  `C: <scenario name>[instance no.] <cut tuple> [Hot|Cold]`
- **Finalization** representing a successful completion or a violation in an active scenario instance. Its format is:
  `F: <scenario name>[instance no.] [Completion|Violation]`

Fig. 2 shows an example short snippet from a scenario-based trace of PacMan. Note the different types of entries that appear in the trace.

```
...
E: 1172664920526 64: void pacman.classes.Ghost.slowDown()
B: PowerUpEaten[1] lifeline 6 <- pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 1 <- pacman.classes.Ghost@7e987e98
C: GhostStopsFleeing[7] (0,1) Hot
C: GhostFleeing[7] (1,3) Hot
E: 1172664920526 65: void pacman.classes.GameControl.ghostSlowedDown(Ghost) pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: GhostStopsFleeing[7] (1,2) Cold
C: GhostFleeing[7] (2,4) Cold
E: 1172664920526 66: void pacman.classes.GameModel.resetGhostPoints()
C: PowerUpEaten[1] (1,2,6,1,1,1,1) Cold
F: PowerUpEaten[1] Completion
E: 1172664921387 67: void pacman.classes.Fruit.enterScreen()
B: PacmanEatsFruit[0] lifeline 2 <- pacman.classes.Fruit@3360336
C: PacmanEatsFruit[0] (0,0,1,0) Hot
C: PacmanEatsFruit[0] (0,0,2,0) Cold
E: 1172664923360 68: void pacman.classes.Ghost.collidedWithPacman()
B: PacmanEatsGhost[2] lifeline 1 <- pacman.classes.Ghost@7d947d94
B: PacmanEatsGhost[2] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: PacmanEatsGhost[2] (1,1,0,0) Hot
C: PacmanEatsGhost[2] (1,2,0,0) Hot
C: GhostEatsPacman[2] (0,1,1,0) Cold
F: GhostEatsPacman[2] Violation
...
```

Fig. 2. Part of a textual representation of a scenario-based trace of PacMan.

2.3 State-based models

For intra-object state-based modeling, we use UML state machines (that is, the object based variant of Harel statecharts [7]). For lack of space, we assume the reader is partly familiar with the syntax and semantics of statecharts in general, at least to the level that allows to understand our example.

Fig. 3 shows an example statechart taken from a model of PacMan. It shows part of a statechart for the class `Ghost`.

2.4 State-based traces

Given a state-based specification consisting of a number statecharts, a *state-based trace* includes the creation and progress information of the statecharts, relative to a given program run. The trace includes information on events, guards evaluation, and the entering and exiting of states in all instances of the statecharts (including concurrently running instances of the same statechart). Thus, our state-based traces may include the following types of entries:

- **State entered** representing a statechart entering a state. The format is:

  `EN: <class_name>[instance no.] Entered state <state full name>`

- **State exited** representing a statechart existing a state. The format is:

  `EX: <class_name>[instance no.] Exited state <state full name>`

- **Event occurrence** representing the occurrence of an event. Events are timestamped and are numbered in order of occurrence. Only the events that explicitly appear in one of the statecharts in the model are recorded in the trace. One may optionally add guards evaluation. The format is:

  `EV: <timestamp> <event no.>: <event signature>`

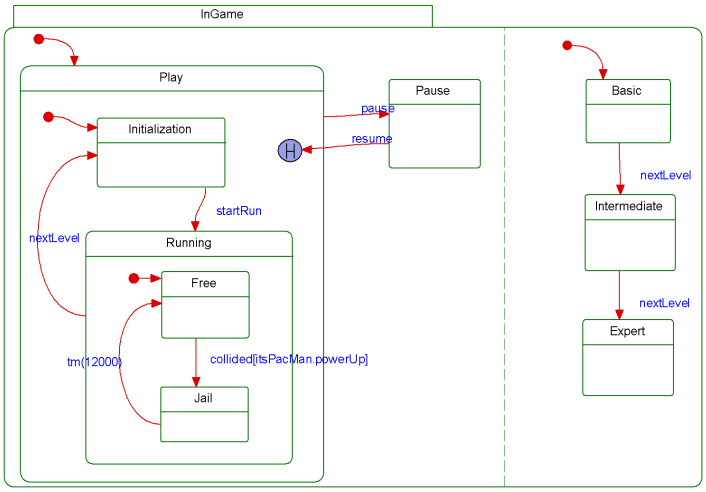

Fig. 3. Part of the `Ghost` statechart in the PacMan model.

Fig. 4 shows a snippet from a state-based trace of PacMan involving a number of statecharts. Note the different types of entries that appear in the trace.

We remark that the above scenario-based and state-based trace formats are presented as examples. Depending on the application, the trace generation mechanism available, and the kind of analysis and reasoning intended for the model-based traces, one may consider different formats, different entry types, different levels of succinctness etc. For example, whether to document the values of guards or the concrete values of parameters depends on the specific application and expected usage of the model-based trace.

3 Example Applications

We give a short overview of two example applications related to the generation of model-based traces and to their visualization and exploration.

3.1 Generating model-based traces

S2A [8] (for Scenarios to Aspects) is a compiler that translates live sequence charts, given in their UML2-compliant variant using the *modal* profile [9], into AspectJ code [1], and thus provides full code generation of reactive behavior from visual declarative scenario-based specifications. S2A implements a compilation scheme presented in [13]. Roughly, each sequence diagram is translated into a *scenario aspect*, implemented in AspectJ, which simulates an automaton whose states correspond to the scenario cuts; transitions are triggered by AspectJ pointcuts, and corresponding advice is responsible for advancing the automaton to the next cut state.

```
...
EV: 45632290 874: Ghost[3].collided
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644272 875: Ghost[2].collided
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644290 876: Ghost[3].timer
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Free
EV: PacMan[1] 877: Pacman[1].complete
EX: PacMan[1] Exited state PacMan.InPlay.Play
EN: PacMan[1] Entered state PacMan.InPlay.LevelInitalization
EV: 45664403 878: Ghost[1].nextLevel
EX: Ghost[1] Exited state Ghost.InGame.InPlay.Play.Running.Free
EX: Ghost[1] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[1] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[1] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664405 879: Ghost[2].nextLevel
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EX: Ghost[2] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[2] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664408 880: Ghost[3].nextLevel
...
```

Fig. 4. Part of a textual representation of a state-based trace of PacMan.

Most important in the context of this paper, though, is that in addition to scenario-based execution (following the play-out algorithm of [10]), S2A provides a mechanism for scenario-based monitoring and runtime verification. Indeed, the example scenario-based trace shown in Fig. 2 is taken from an actual execution log of a real Java program of the PacMan game adapted from [3], (reverse) modeled using a set of live sequence charts (drawn inside IBM Rational SA [2] as modal sequence diagrams), and automatically instrumented by the AspectJ code generated by S2A. More on S2A and its use for model-based trace generation can be found in `http://www.wisdom.weizmann.ac.il/~maozs/s2a/`.

3.2 Exploring model-based traces

The Tracer [14] is a prototype tool for the visualization and interactive exploration of model-based traces. The input for the Tracer is a scenario-based model of a system given as a set of UML2-compliant live sequence charts, and a scenario-based trace, generated from an execution of the system under investigation.

Fig. 5 shows a screenshot of the main view of the Tracer, displaying a scenario-based model and trace similar to the one shown in Fig. 2. Roughly, the main view is based on an extended hierarchical Gantt chart, where time goes from left to right and a two-level hierarchy is defined by the containment relation of use cases and sequence diagrams in the model. Each leaf in the hierarchy represents a sequence diagram, the horizontal rows represent specific active instances of a diagram, and the blue and red bars show the duration of being in a specific cold and hot relevant cuts. The horizontal axis of the view allows to follow the progress of specific scenario instances over time, identify events that caused progress, and locate completions and violations. The vertical axis allows

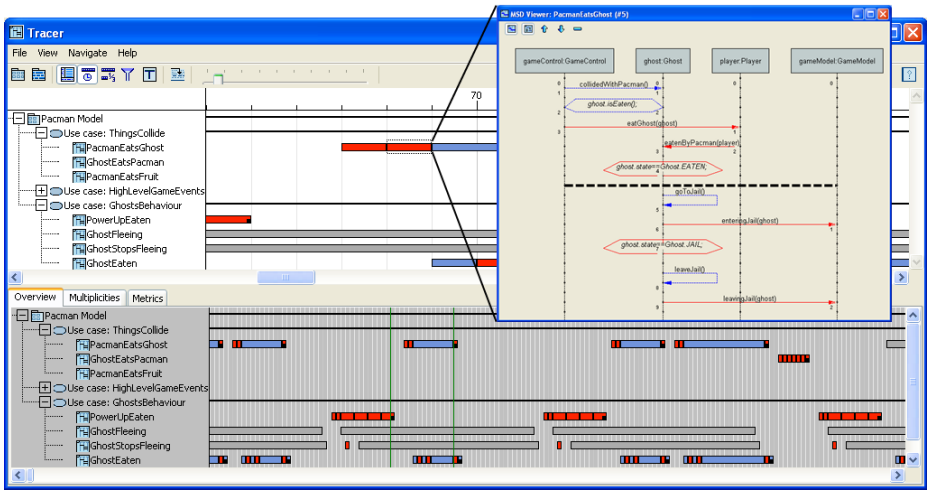

Fig. 5. The Tracer's main view, an opened scenario instance with its cut displayed at (3,4,2,0), and the Overview pane (at the bottom). The example trace and model are taken from an implementation of the PacMan game, see [14].

a clear view of the synchronic characteristic of the trace, showing exactly what goes on, at the models abstraction level, at any given point in time.

When double-clicking a bar, a window opens, displaying the corresponding scenario instance with its dynamic cut shown in a dashed black line. Identifiers of bound objects and values of parameters and conditions are displayed in tooltips over the relevant elements in the diagram. In addition, one can travel back and forth along the cuts of the specific instance (using the keyboard or the arrows in the upper-left part of the window). Multiple windows displaying the dynamic view of different scenario instances can be opened simultaneously to allow for a more global synchronic (vertical) view of a specific point in the execution, or for a diachronic (horizontal) comparison between the executions of different instances of the same scenario at different points in time during the execution.

Note the Overview pane (bottom of Fig. 5), which displays the main execution trace in a smaller pixel per event scale, and the moving window frame showing the borders of the interval currently visible in the main view. The Overview allows to identify high level recurring behavioral patterns, at the abstract level of the scenarios in the model. Additional views are available, supporting multiplicities, event-based and real-time based tracing, and the presentation of various synchronous metrics (e.g., how many scenarios have been affected by the most recent event?). Overall, the technique links the static and dynamic aspects of the system, and supports synchronic and diachronic trace exploration. It uses overviews, filters, details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods.

The Tracer was first presented in [14]. More on the Tracer, including additional screenshots and screencasts can be found in `http://www.wisdom.weizmann.ac.il/~maozs/tracer/`.

4 Related work

We briefly discuss related work. Generating model-based traces requires an observer with monitoring and decision-making capabilities; a so called 'runtime awareness' component (see, e.g., [5,11]). However, while model-based traces can be used for error detection and runtime verification, the rich information embedded in them supports more general program comprehension and analysis tasks and allows the reconstruction and symbolic replay of a program's run at the abstraction level defined by the model used for tracing.

The use of AOP in general and AspectJ in particular to monitor program behavior based on behavioral properties specified in (variants of) LTL has been suggested before (see, e.g., [5,16]). As LSCs can be translated into LTL (see [12]), these work have similarities with our use here of S2A. Like [16], S2A automatically generates the AspectJ code which simulates the scenario automaton (see [13]). Unlike both work however, S2A outputs a rich trace reflecting state changes and related data (binding etc.), to serve our goal of generating model-based traces that allow visibility and replaying, not only error detection.

Many work suggest various trace generation and visual exploration techniques (e.g., for a survey, see, [6]). Most consider code level concrete traces. Some attempt to extract models from these traces. In contrast, model-based traces use an abstraction given by user-defined models. They are generated by symbolically running these models simultaneously with a concrete program execution.

5 Discussion and Challenges for Future Work

We introduced model-based traces and presented two example applications. The focus of model-based traces is on providing visibility into an execution of a program at the abstraction level defined by a model, enabling a combination of dynamic analysis and model-driven engineering. Below we discuss our approach and list related challenges.

Trace generation S2A provides an example of a model-based trace generation technology, based on programmatically generated aspects. Two major advantages of this approach are that the monitoring code is automatically generated from the models, and that the code of the system under investigation itself is oblivious to the models 'watching' it. Related challenges include minimizing runtime overhead, scalability in terms of trace length and model size, and the application of similar technology to domains where aspect technology is not readily available (e.g., various embedded or distributed systems).

Analysis and reasoning We consider the development of analysis methods for model-based traces. For example, define and measure various vertical and horizontal metrics (e.g., 'bandwidth', state / transition coverage per trace per model, how many times was each state visited), abstraction and refinement operators (e.g., hide events and keep states, hide sub states of composite states), ways to represent and compare different model-based runtime configurations ('snapshots', perhaps most important for dynamic adaptation), or ways to align and compare between traces of different runs of the same system, or very similar

runs of different versions of the same system. In addition, we consider additional types of abstractions over the same models, e.g., real-time based vs. event-based trace representation (as is supported by the Tracer (see [14])). Also, an agreeable, common representation format for model-based traces, where applicable (e.g., for specific types of models), should perhaps be defined and agreed upon, so that not only models but also their traces may be exchanged between tools in a standard format like XMI.

Visualization and interaction The visualization and interaction supported by the Tracer allows a human user to explore and analyze long and complex model-based traces that are otherwise very hard to handle manually in their textual form. Still, a lot more can be done on this front, from finding "economic" visualizations for model-based snapshots to animation to visual filters etc.

Acknowledgements I would like to thank David Harel, David Lo, Itai Segall, Yaki Setty, and the anonymous reviewers for comments on a draft of this paper.

References

1. AspectJ. http://www.eclipse.org/aspectj/.
2. IBM Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.
3. PacMan. Java implementation of the classic PacMan game. http://www.bennychow.com.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
5. H. Goldsby, B. H. C. Cheng, and J. Zhang. AMOEBA-RT: Run-Time Verification of Adaptive Software. In *Models@run.time, MoDELS Workshops*, 2007.
6. A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON*, 2004.
7. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
8. D. Harel, A. Kleinbort, and S. Maoz. S2A: A Compiler for Multi-Modal UML Sequence Diagrams. In *FASE*, 2007.
9. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.
10. D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling (SoSyM)*, 2(2):82–107, 2003.
11. J. Hooman and T. Hendriks. Model-Based Run-Time Error Detection. In *Models@run.time, MoDELS Workshops*, 2007.
12. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.
13. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
14. S. Maoz, A. Kleinbort, and D. Harel. Towards Trace Visualization and Exploration for Reactive Systems. In *IEEE VL/HCC*, 2007.
15. R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *OOPSLA*, 2002.
16. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. *Electr. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.