Appendices for:
S. Maoz, A. Kleinbort, and D. Harel. Towards Trace Visualization and Exploration for Reactive Systems. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC'07)*, 2007.

Appendix A: Scenario-Based Traces

An important concept in modal scenarios 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, used to represent the state of an active scenario during execution). For example, in the modal scenario `PacManEatsGhost` shown in Fig. 1, the cut `(3,4,2,0)` 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. In Fig. 1, a single enabled event is induced by the cut at `(3,4,2,0)`, namely the ghost's self method call `goToJail()`. A cut is hot if any of its enabled events

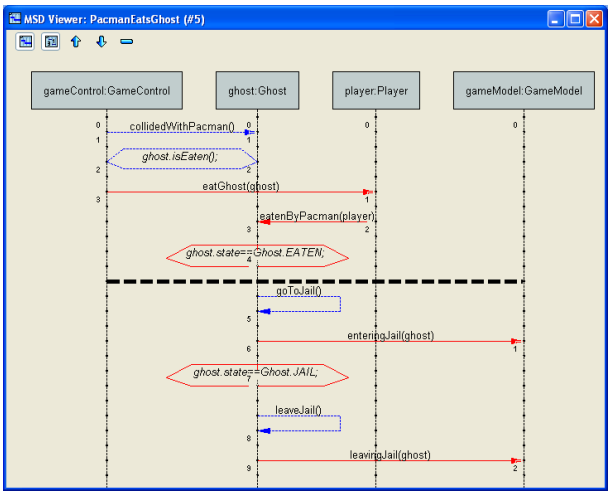

Figure 1. The modal scenario for `PacManEatsGhost` with a cut displayed at (3,4,2,0).

is hot (and is cold otherwise). When a scenario'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 scenario or a FALSE evaluation of an enabled condition when the cut is cold (hot) is a *completion* (*violation*) and causes the scenario's instance to close.

Given a scenario-based specification consisting of a number of modal scenarios, a *scenario-based execution 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 scenarios (including concurrently active multiple copies of the same scenario).

A snippet from a textual representation of a scenario-based execution trace is given in Fig. 2. Events (method calls) are time stamped and include data about the source and target objects (caller and callee), the method signature, and the values of parameters. Cut progress data include the scenario name, instance number, a tuple of integers representing locations on corresponding lifelines, and the current cut's mode (hot or cold). Binding information links scenario lifelines to objects. Finally, the text includes completion and violation information, as applicable.

In [6, 13] we presented a compilation scheme and a prototype compiler implementation called *S2A* for the translation of modal scenarios into *scenario aspects*, implemented in AspectJ . Each scenario aspect simulates an abstract automaton whose states represent cuts along the partial order defined by the diagram, and whose transitions represent enabled events (we refer the reader to [13] for details). We thus use these generated scenario aspects to instrument programs and create traces annotated with scenario-based information (such as the one shown in Fig. 2, which was created using *S2A*) that serve as the input for the visualization.[1]

Appendix B: State-Based Trace Visualization

While the main concern of the paper is the visualization of execution traces induced by inter-object scenario-based specifications, the ideas can be applied to intra-object state-based specifications too. We now sketch the adaptation.

We propose to generate *state-based execution traces*, i.e., execution traces that include information about the states of (selected) objects during a run of the program. These are then visualized using an appropriate variant of the Tracer, where the hierarchy reflects the object composition relation, and the horizontal bars represent the durations of being in states of specific object instances, as they change over time. Moreover, if Statecharts [4, 5] are used to describe the intra-object behavior of the system (as in, e.g., Rhapsody [2]), the trace visualization would further reflect the orthogonal components of an object's statechart as sibling nodes in the Gantt hierarchy, while the depth of the states would be indicated on the horizontal bars themselves.

This state-based trace visualization may take advantage of the techniques described in the paper: handling concurrency, details-on-demand (from a horizontal bar on the Gantt to a Statechart diagram where the current state is highlighted), event/time-based tracing, variable time-scale mode, etc.

[1] Other means could have been used to generate similar scenario-based traces (e.g., the Play-Engine [8]) and other textual formats (e.g., XML) could have been used to represent the data. Our techniques are independent of the method used for trace generation and of the textual format used.

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

Figure 2. Part of a text file for a trace of PacMan

Interestingly, our trace visualization methods for the two approaches – intra-object and inter-object – can be combined. This allows the user to switch between state-based and scenario-based traces of the same program run, explore them simultaneously in different synchronized views, etc. This can have significant benefits both in system development stages and in system testing and maintenance.

Appendix C: Related Work

We briefly discuss recent related work in the areas of execution traces, sequence diagrams, and time-series data visualization.

The visualization of execution traces, as a topic within software visualization in general, has often been suggested and implemented. However, most trace extraction and visualization efforts to date (e.g., [11, 12, 14, 15, 18]) consider the trace at the code level, while our traces are abstracted to the scenario (or state) level. Thus, our approach combines tracing with model-driven design. Moreover, instead of looking for interaction patterns in the extracted traces (as in, e.g., [12]), or visualizing the recorded trace using a sequence diagram (as in, e.g., VET [14], Eclipse TPTP [1], or I-Logix Rhapsody [2]), we take the scenario-based specification given by the user as input and visualize the activation, progress, and interaction of the specified inter-object scenarios as they "come to life" during execution. Finally, we consider not only the partial-order semantics of sequence diagrams, but significantly also the modal, universal/existential, hot/cold semantics of live sequence charts (LSC) [3, 8] and their UML-compliant version [7], which allow for stronger expressive power with regard to temporal, functional, and structural properties.

The *Play-Engine* [8] is an interpreter-style simulation engine built in our group for LSCs, based on the *play-in/play-out* approach [9]. As the simulation progresses, the Play-Engine follows and displays all the active LSCs and their cuts. User experience shows, however, that in terms of comprehending the execution there can be much information overload: when many LSC windows open and close rapidly during execution the effectiveness of the visualization decreases. The approach presented in the paper seems to constitute the much-needed aid for such comprehension. Thus, for example, we allow the user to choose a preferred level of detail and to zoom from the black-box Gantt view to the detailed view where the active scenarios complete sequence diagrams and their cut information is shown (and these cuts can be visualized diachronically in the context of the execution's past and future). Also, the Play-Engine is an LSC-specific closed environment, whereas the Tracer can be used for the scenario-based analysis and visualization of any third-party Java program.

Recent work by Reiss [16, 17] proposes visualizing program execution by following the states of a user-defined automaton on the traces. This work has some similarities with ours. Some key differences are our use of sequence diagrams as visual specifications, the link from the bars to the diagram displaying the cut, the event/time-based combinations, and the additional multiplicities and metrics views.

Finally, analysis and visualization of program execution traces is related to the exploration and visualization of time-series data in general. In [10], Hochheiser and Shneiderman present Timeboxes, which are rectangular widgets used in direct-manipulation graphical user interfaces to specify query constraints on time-series data sets. It is possible to combine TimeBoxes with the Tracer to apply this and other time-related techniques to our model-based execution traces

(both event-based and time-based), thus gaining dynamic querying capabilities and greater visual insight into the execution of complex reactive systems.

References

[1] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.

[2] I-Logix Rhapsody. http://www.i-logix.com.

[3] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer, 1999, pp. 293-312.

[4] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[5] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, pages 31–42, July 1997.

[6] D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *Proc. 10th Int. Conf. Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 121–124, 2007.

[7] 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 Software Engineering (ICSE'06)*, pages 13–20, New York, NY, USA, May 2006. ACM Press.

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

[9] 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.

[10] H. Hochheiser and B. Shneiderman. Dynamic query tools for time series data sets: timebox widgets for interactive exploration. *Information Visualization*, 3(1):1–18, 2004.

[11] J. G. Hosking. Visualisation of object oriented program execution. In *Proc. 1996 IEEE Symp. on Visual Languages*, pages 190–191. IEEE Computer Society, 1996.

[12] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 360–370, New York, NY, USA, 1997. ACM Press.

[13] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT'06/FSE-14)*, pages 219–230, New York, NY, USA, 2006. ACM Press.

[14] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (VET): an interactive plugin-based visualisation tool. In *Proc. 7th Australasian User Interface Conf. (AUIC'06)*, pages 153–160, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[15] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lect. on Software Visualization, Int. Seminar*, pages 151–162, London, UK, 2002. Springer-Verlag.

[16] S. P. Reiss. Checking event-based specifications in java systems. *ENTCS*, 144(3):107–132, 2006.

[17] S. P. Reiss. Visualizing program execution using user abstractions. In *Proc. 2006 ACM Symp. on Software Visualization (SoftVis'06)*, pages 125–134. ACM Press, 2006.

[18] S. P. Reiss and M. Renieris. Jove: java as it happens. In *Proc. 2005 ACM Symp. on Software Visualization (SoftVis'05)*, pages 115–124. ACM Press, 2005.