Towards Trace Visualization and Exploration for Reactive Systems *
(preliminary version)

Shahar Maoz, Asaf Kleinbort, David Harel†
The Weizmann Institute of Science, Rehovot, Israel

1. Introduction

The design and development of reactive systems [10], discrete-event systems that maintain ongoing interaction with their environment, is a complex and challenging task. One way to address it is to use visual formalisms to model and describe the system's behavior. Two complementary approaches to model the behavior of reactive systems have been proposed – state-based intra-object modeling and scenario-based inter-object modeling – with corresponding visual languages [5, 7].

In this paper — as a natural extension of the idea of using visual formalisms for the modeling itself — we present a technique for the visualization and exploration of execution traces of such models. Our approach is different from previous approaches, most of which consider execution traces at the code level, look for interaction patterns in the traces, or generate concrete sequence diagrams from recorded execution traces. In contrast, we take an inter-object scenario-based behavioral model given by the designer as input, and visualize the activation and progress of the scenarios therein as they "come to life" during execution. We illustrate the ideas using modal scenarios, given in a UML-compliant dialect of *live sequence charts* (LSC) [5, 9].

The technique links the static and dynamic aspects of the system, and supports synchronic and diachronic trace exploration, concurrency, event-based and real-time-based tracing. It uses overviews, filters, details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods. We have implemented the ideas in a prototype tool we call the *Tracer*. The examples we show are based on a Java implementation of the PacMan game [2]. For lack of space this preliminary version reports only on the essentials. We refer the reader to [3] for a demonstration movie of the Tracer and for the Appendices to this paper, which include background material, preliminary ideas on state-based traces, and a comparison with related work. [1]

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

†Part of this author's work done on a visit to the School of Informatics at the University of Edinburgh, supported by an EPSRC grant.

[1]Thanks are due to Peter Kliem for making his jaret timebars [1] available under GPL and for the implementation of the variable scale feature.

2 Preliminaries

To specify modal scenarios, we use a UML-compliant, uniform, and slightly generalized version of Damm and Harel's *live sequence charts* (LSC), a visual formalism for scenario-based inter-object specifications [5], defined using the *Modal* profile [9]. The language extends the partial order semantics of sequence diagrams in general, by allowing to specify (sub) scenarios that "may happen", "must happen", or "should never happen". The notation is adopted from LSC: hot ("must") elements are colored in red, cold ("may") elements in blue (alternatively, hot use solid lines and cold use dashed ones). Vertical lifelines represent specific processes and time goes from top to bottom. Lifelines may be symbolic [12], and thus represent any object of the class associated with them. A specification typically consists of many scenarios, divided between several use cases.

An important concept in the semantics of modal scenarios is the *cut*, which is a mapping from each lifeline to one of its locations, representing the state of an active scenario during execution. A cut induces a set of enabled events — those immediately after it in the partial order defined by the scenario. A cut is hot if any of its enabled events 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 chart 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). We use the *S2A* compiler [8] to translate the diagrams into *scenario aspects* [11], and create the scenario-based execution traces that serve as the input for the visualization.

3 Scenario-Based Trace Visualization

Basically, we visualize a scenario-based program execution trace using a hierarchical Gantt chart, which we have enriched in several ways, discussed below. In it, time goes from left to right and the hierarchy is defined by the containment relation of the use cases and diagrams in the specification model. Thus, each leaf in the hierarchy represents a different sequence diagram, the horizontal rows represent specific active instances of a diagram (we call *scenario instances*), and the bars therein show the durations of being in the relevant cut states. The horizontal axis of the view allows to follow the progress of specific scenario instances over time, identify the events that caused progress, and locate completions and violations. The vertical axis allows a clear view of the synchronic characteristic of the trace, showing exactly what goes on at any given point in time.

By default, the basic view uses color coding to visually distinguish between existential (cold/blue) and universal (hot/red) cuts. A textual encoding of the cut into a tuple of integers representing locations on specific lifelines is displayed on each bar. Further details about a specific cut (e.g., the signature of its preceding event) are displayed in a tooltip over the bar. Other rendering functions, e.g., displaying the instance serial number or avoiding text labels entirely, are also available.

Fig. 4 shows a representative screenshot of the default main view. Note the hierarchy of use cases and sequence diagrams on the left and the red and blue bars representing hot and cold cut-states. Note also the *Overview*, which displays the main execution trace in a smaller pixel per event scale, and the moving frame showing the borders of the interval currently visible in the main view.

When double-clicking a bar, a window opens, displaying the corresponding scenario instance with its dynamic cut (Fig. 4). Identifiers of bound objects and values of parameters and conditions are displayed in tooltips over the relevant elements in the diagram. 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.

The building blocks of reactive-system traces are discrete events. Indeed, the basic view presented above is event-based: the trace progresses if and when an event occurs (and only then), and all events are allocated the same horizontal distance on the view. In other words, time is abstracted away from the basic visualization of the trace; only the order remains. This kind of abstraction is not new and is typically reflected in the language chosen to specify a system's behavior. The basic variants of the *temporal logics*, for example, LTL and CTL, indeed do not consider the actual durations of happenings but only their order [6].

In many systems however, the real-time aspect of the trace is important. Thus, in addition to the default event-based view, we offer a time-based view, where the horizontal axis accurately reflects the progress of time, regardless of occurrence of events. The user can select the basic unit of time (millisecond, second, etc.) and a corresponding fixed pixel distance.

The time-based view of the trace correctly reflects its progress over time. However, in many cases, due to high variability in event duration and density, which can often span several orders of magnitude, the resulting view may be formally accurate but very difficult to comprehend and browse visually. Such high variability, e.g., short periods with many events and long periods with very few events, is typical to many real-world reactive systems that interact with their environment. To alleviate this problem, we provide a novel hybrid view, which we term *event-normalized* (Fig. 1). It combines the event-based and time-based presentations by allocating a fixed horizontal interval to each event, while programming the grid appearing in the background to draw vertical lines every fixed time unit. As a result, these time-based vertical lines are unevenly spaced.

Multiple instances of the same diagram, where lifelines bind to different objects, may be simultaneously active during program execution (see [12]). Consider the `PacManEatsGhost` scenario from Fig. 4: PacMan may eat a second ghost before the first one eaten has entered the jail. In this case, two instances of the `PacManEatsGhost` diagram, where the lifeline ghost binds to different objects, will be active simultaneously.

As a means to handle multiple instances we introduce a supporting view called *Multiplicities* (Fig. 2). On the main view, we hide the multiplicity from the user by covering the period where more than one instance has been active with a single grey bar. When the user double-clicks the grey bar, the specific instances are displayed in the supporting view. Thus, details about multiple copies are given on-demand.

The *Metrics View* (Fig. 3) shows various specification-wide synchronous (so called 'vertical') statistics, such as the total number of concurrently active scenarios (the *scenario bandwidth*), the total number of scenarios affected by the most recent event, etc., some of which are relevant to performance and resource allocation analysis. One of the more interesting ones is `duration`. As explained earlier, in the event-based tracing mode grid lines are evenly spaced between events and the data about the real durations is abstracted away. The `duration` metric displays the real duration of the periods between events, visually overlaying the real-time dimension on the fixed event-based view.

The metrics are displayed using user-adjustable color gradients; these are appropriate since for most purposes the metric's relative qualitative values, e.g., high/medium/low, are of interest, rather than the precise values.

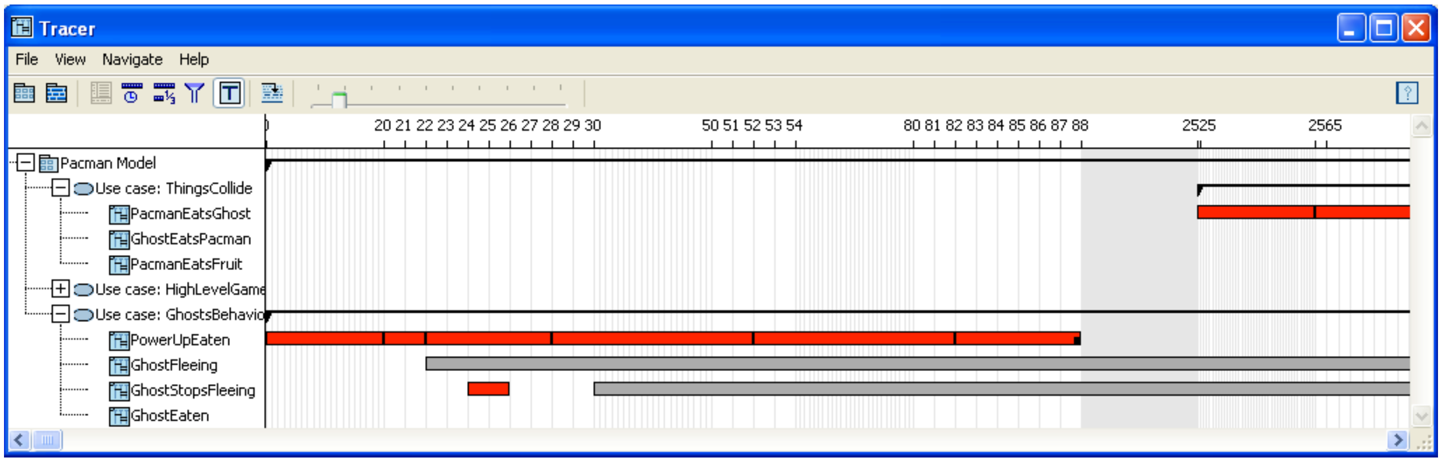


Figure 1. The event-normalized time-based view. Note the different scales between the 54th and 80th ms, and between the 88th and the 2525th ms. In the former interval, only 26 milliseconds passed between two consecutive events, but in the latter more than 2 seconds. In between them, there was an event (at least) every millisecond. In the real time-based view, where every millisecond gets a fixed width, these differences in event density would result in a display that is very difficult to comprehend and browse visually.

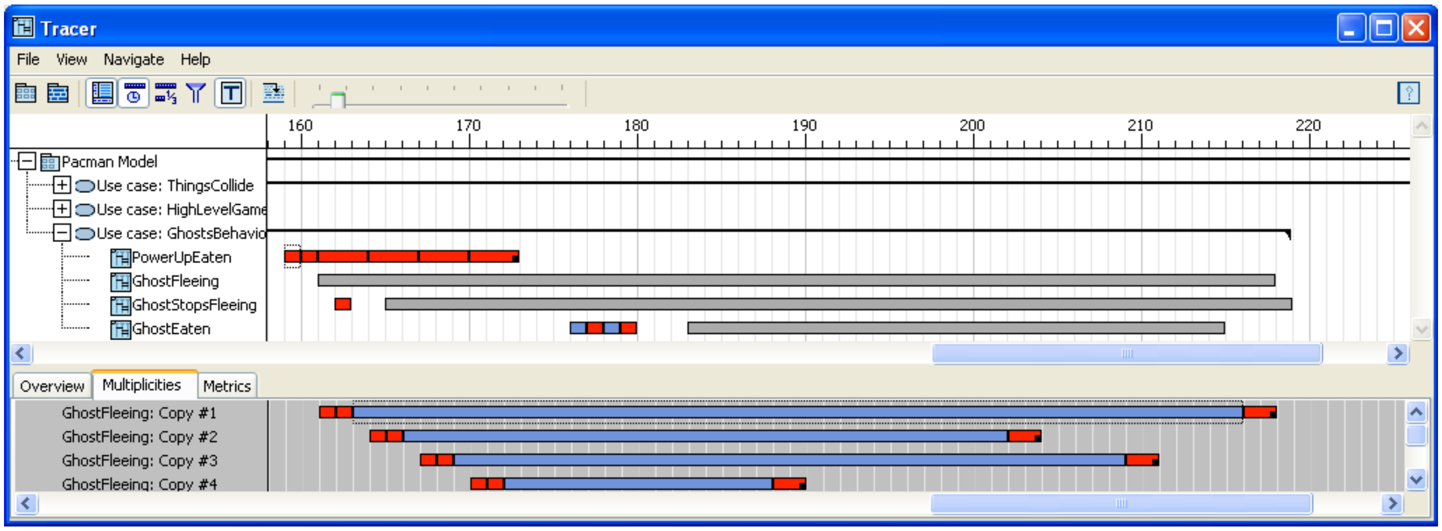
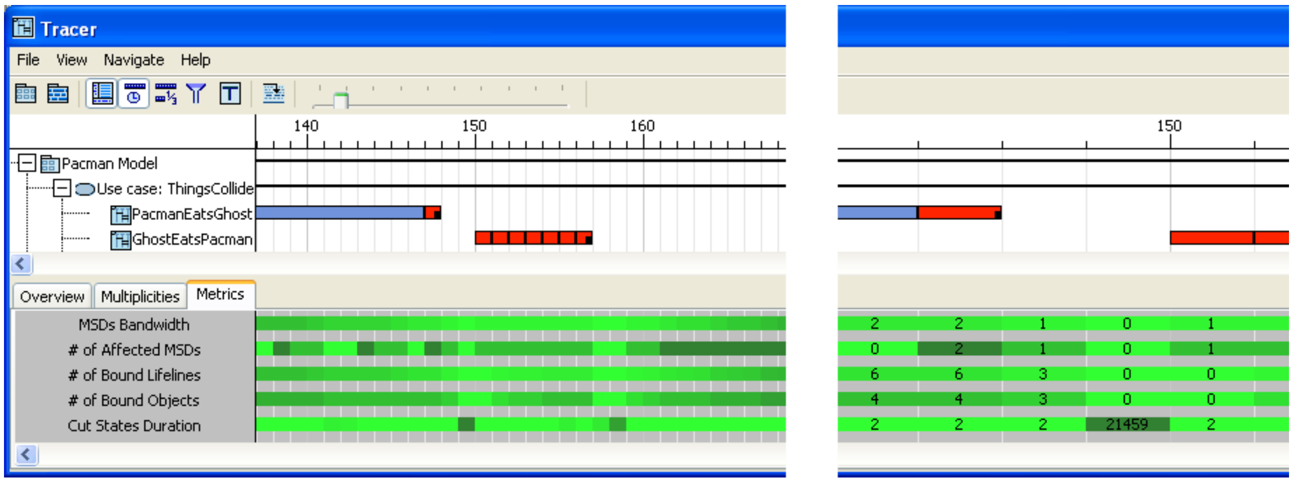


Figure 2. The *Multiplicities* view. Note the four active copies of the `GhostFleeing` scenario.



MSDs Bandwidth	2	2	1	0	1
# of Affected MSDs	0	2	1	0	1
# of Bound Lifelines	6	6	3	0	0
# of Bound Objects	4	4	3	0	0
Cut States Duration	2	2	2	21459	2

Figure 3. *Metrics* view without values (left) and with values and different scale (right). Note the dark colored bar for the 149th event in the duration metric (bottom), indicating a long period with no events, otherwise abstracted away in the event-based view.

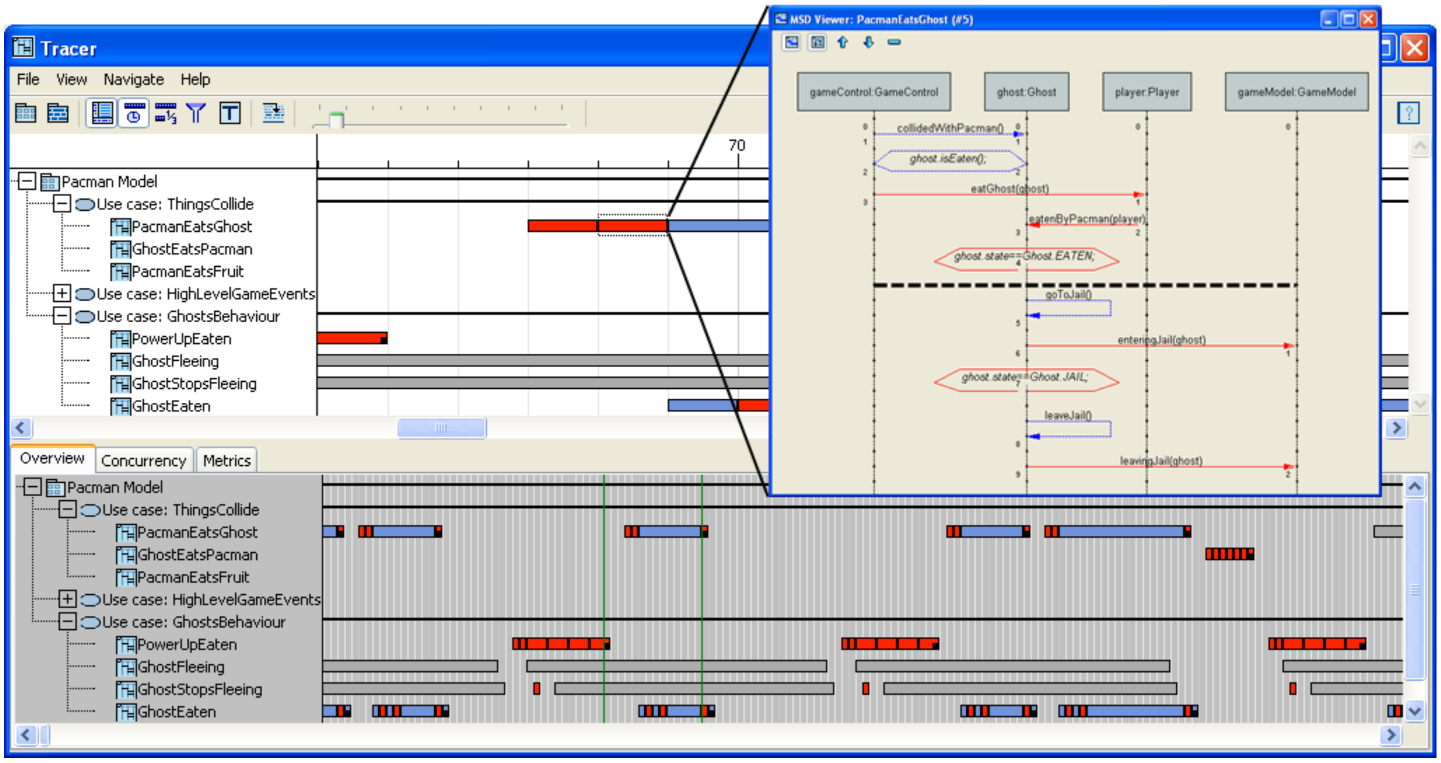


Figure 4. The Tracer's main view, an opened scenario instance with cut displayed at (3,4,2,0), and the *Overview*.

4 Conclusion and Future Work

The main contribution of our work is in providing new techniques for scenario-based visualization and exploration of reactive system execution traces. By considering traces not at the code level but at a higher behavioral abstract level, we are able to connect dynamic analysis with model-driven development. The separate event-based and time-based tracing modes, as well as the combined event-normalized multi-scale mode, appear to be another novel contribution.

Our work follows the *overview first, zoom and filter, details-on-demand* paradigm [4], and the concept of *semantic zooming* [13] in a number of ways. First, by the use of the *Overview* supporting view and its main view frame, and second, by the zoom from classes to instances (of concurrent scenarios!) to scenario instance details on demand. Also, the multi-scale presentation may be viewed as a special kind of automated semantic zooming, where although the real duration between events is explicitly displayed, fragments of the execution trace receive horizontal space according to the level of activity they contain, rather than according to their real-time duration.

Planned future work includes completing the Tracer's packaging and conducting usability studies. In addition, we have begun to consider the application of our ideas to state-based behavioral models. We believe that the Tracer, or a similar tool based on our ideas, can be used effectively to improve the activities involved in the development of complex reactive systems, specifically in simulation, testing, and dynamic analysis.

References

[1] Jaret timebars package. http://jaret.de/timebars/.
[2] PacMan game code. http://www.bennychow.com/.
[3] Tracer website. http://www.wisdom.weizmann.ac.il/˜maozs/tracer/.
[4] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization*. 1999.
[5] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 2001.
[6] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990.
[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.
[8] D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *FASE'07*, 2007.
[9] D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In *SCESM'06*, 2006.
[10] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, 1985.
[11] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
[12] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *OOPSLA'02*.
[13] K. Perlin and D. Fox. Pad: an alternative approach to the computer interface. In *SIGGRAPH'93*, 1993.