Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Shahar Maoz · David Harel

On Tracing Reactive Systems

Received: date / Accepted: date

Abstract We present a rich and highly dynamic technique for analyzing, visualizing, and exploring the execution traces of reactive systems. The two inputs are a designer's inter-object scenario-based behavioral model, visually described using a UML2-compliant dialect of live sequence charts (LSC), and an execution trace of the system. Our method allows one to visualize, navigate through, and explore, the activation and progress of the scenarios as they "come to life" during execution. Thus, a concrete system's runtime is recorded and viewed through abstractions provided by behavioral models used for its design, tying the visualization and exploration of system execution traces to model-driven engineering. We support both event-based and real-time-based tracing, and use details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods. Novel model exploration techniques include semantics-based navigation, filtering, and trace comparison. The ideas are implemented and tested in a prototype tool called the *Tracer*.

Keywords Software Visualization · UML Interactions · Sequence Diagrams · Live Sequence Charts · Model-Based Traces · Dynamic Analysis

Preliminary version appeared in VL/HCC '07: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (September 2007) [46]. This research was supported in part by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant from the European Research Council (ERC) under the European Community's Seventh Framework Programme (FP7/2007-2013).

Shahar Maoz
The Weizmann Institute of Science, Rehovot, Israel
E-mail: shahar.maoz@weizmann.ac.il

David Harel
The Weizmann Institute of Science, Rehovot, Israel
E-mail: dharel@weizmann.ac.il

1 Introduction

The design and development of reactive systems [28], discrete-event systems that maintain ongoing interaction with their environment, involves complex and challenging tasks. To list a few, these include the elicitation and formalization of the system's requirements, the translation of the requirements into a specification, the creation of an executable artifact, and the development of methods for checking that the resulting system indeed meets its requirements, specifically those related to the behavior of the system over time. One way to address these challenges is to use visual formalisms to model and describe the system's behavior. Among other things, such visual models provide a means to describe the system at various levels of abstraction and from different viewpoints, to communicate the system's description between stakeholders, to formally analyze the system and reason about its properties, and, in some cases, to directly generate an executable artifact. Two complementary approaches to model the behavior of reactive systems have been proposed – state-based intra-object modeling [20] and scenario-based inter-object modeling [15] – with the corresponding visual languages of *statecharts* and *live sequence charts*, respectively.

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 charts therein as they "come to life" during the execution of a reference program of a concrete system. Thus, a concrete system's runtime is recorded and viewed through abstractions provided by behavioral models used for its design. We illustrate the

ideas using a UML2-compliant dialect of *live sequence charts* (LSC) [15,24].

Our technique belongs to the domain of model-based dynamic analysis. In contrast to static analysis, which investigates a system by analyzing its code or model — without executing it, dynamic analysis considers, in addition to the system's code and model, runtime input coming from concrete executions of the system under development and investigation. Model-based dynamic analysis includes tasks related to the investigation of the relationships between a system's execution traces and its models, such as testing whether a system run satisfies a property that a certain model specifies, measuring how various model features materialize in a system run, and finding the differences between two or more system runs, in the context of model-level debugging or evolution.

Major challenges in model-based dynamic analysis are the complexity and length of the models and execution traces. Visualization in general, and our scenario-based trace version in particular, attempt to address these challenges by creating a scalable and visually appealing solution. That is, they assist the engineer in comprehending and analyzing the trace, the relationships between the model and the concrete application at hand, and the relationships between the different parts of the model itself. Our way of doing this is by adapting classical visualization paradigms and techniques to the specific needs of model-based dynamic analysis tasks.

Specifically, our work links the static and dynamic facets of the system, and supports synchronic and diachronic trace exploration, multiplicities (concurrently active scenario instances), and event-based and real-time-based tracing. It uses overviews, filters, details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods. Novel exploration techniques include semantics-based trace navigation, horizontal and vertical filtering, and various trace comparison mechanisms. Together, these provide a novel, rich, and highly dynamic interface for model-based dynamic analysis.

In order to evaluate our ideas, we have implemented them in a prototype tool we call the *Tracer*. We have tested the Tracer on a number of case study programs, spanning various application domains. Screenshots and screencasts of the Tracer are available at the Tracer website [9]. Our experience in using the Tracer is reviewed in Section 7, which also includes a list of lessons learned and a critical evaluation of our work and its limitations.

A necessary prerequisite for the analysis and visualization of scenario-based execution traces is the effective generation of the traces. We build here on earlier work of ours, namely the transformation of modal scenario-based specifications into aspects [45] and its implementation in the S2A compiler [22], which (among other things), enables the automatic generation of scenario-based traces. We briefly recall this work in subsection 2.2.3.

The visualization of execution traces, as a topic within software visualization in general, has been suggested and implemented before. Most previous execution trace visualization work, like ours, is based on a two dimensional representation. Time goes along one axis, and a certain hierarchy that is based on the structure of the system's implementation is depicted on the other axis (e.g., packages, classes, objects). In contrast, our hierarchy comes from the scenario-based specification model, which consists of use cases and sequence diagrams. These reflect the requirements perspective or the specification perspective of the system. They do not necessarily correspond to elements of the structure and the implementation of the system under investigation. We discuss and compare our ideas with earlier related work in Section 8.

While we concentrate on inter-object modal scenario-based specifications given in LSC, and take advantage of their expressive power with regard to temporal liveness/safety and polymorphic interpretation, our ideas are applicable also to *intra*-object *state*-based specifications, as well as to *model-based traces* [43] in general. We discuss the concept of model-based traces in subsection 2.2.1. The applicability of our ideas to general model-based traces is discussed in Section 9.

1.1 Example application

The examples throughout the paper are based on a model of the classic PacMan game, the Java implementation of which can be found in [5]. The PacMan game has been used in the past as an example in computer science research (see, e.g., [12,18,37]).

PacMan's game board consists of a maze, filled with dots, power-ups, fruit, and four ghosts. A human player controls PacMan, whose goal it is to collect as many points as possible by eating the objects in the maze. When a ghost collides with PacMan, the latter 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 its cage 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. Figure 1 shows a screenshot from the PacMan game.

We consider the PacMan game to be a well-known, intuitive, relatively small and yet complex enough reactive system. Hence it is a good choice for the purpose of demonstrating the model-based trace visualization techniques we present in this paper.

1.2 Paper organization

The paper is organized as follows. In the next section we provide background material on LSC and on scenario-based traces. The next three sections consist of the main ideas of our work: Section 3 presents the basics of scenario-based trace visualization; Section 4 continues with more

Fig. 1 A screenshot from the PacMan game.

advanced visualization features, including multiplicities, time-based vs. event-based views, metrics, and completions; and Section 5 describes trace exploration features, covering navigation, filtering, and comparisons. Section 6 presents possible usage examples in the context of model-based dynamic analysis tasks. Section 7 describes the Tracer prototype implementation and provides an evaluation of our work by reviewing experience in applying it to traces from a number of applications. Following Section 8, which discusses related work, Section 9 discusses future work and Section 10 concludes.

2 Preliminaries

This section provides background material on live sequence charts and scenario-based traces.

2.1 Live sequence charts

Live sequence charts (LSC) [15] is a visual formalism for inter-object scenario-based specifications. The language extends the partial order semantics of classical message sequence charts (MSC) [32] mainly by adding universal/existential interpretations and must/may (hot/cold) modalities. It thus allows the specification of inter-object behaviors that may happen, must happen, or should never happen.

A UML2-compliant variant of LSC is defined in [24], and a translation of LSC into various temporal logics appears in [38]. An operational semantics for LCS, termed *play-out*, was defined and implemented in [26,27]. The language has been the subject of research in the areas of scenario-based programming, synthesis, verification, specification mining, and testing (see, e.g., [23,36,40,42,45]).

We give here only a brief background on LSC and its semantics, specifically covering the parts most relevant to the present paper. More thorough definitions of the language appear in [15,24,26].

An LSC consists of a set of lifelines, representing system objects, and events, specifically method calls and conditions, involving these objects. Lifelines are drawn using vertical lines, and each is labeled with a name and a type; method calls are drawn using horizontal arrows from locations on caller to callee lifelines and each is labeled with the method signature; conditions are drawn using hexagons and each is labeled with a Boolean expression. Like classical MSC, an LSC induces a partial order on its events; events covering the same lifelines are fully ordered from top to bottom, but events covering disjoint lifelines may be unordered.

Each event in an LSC, a method call or a condition, has a *mode*, which may be either *hot* or *cold*. Hot events are drawn using red lines; cold events are drawn using blue lines. The mode of an event carries a semantic meaning, as described below.

An important concept in the semantics of LSC 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. All events that appear in the chart but are not currently enabled are *violating events* (the intuition being that their occurrence violates the required behavior). However, events that do not appear explicitly in the chart are not restricted to occur or not to occur during a run, including in between the events that do appear explicitly in the chart. A cut is hot if at least one of its enabled events is hot and is cold otherwise. A hot cut represents an unstable state – one which, according to the specification, the system must eventually leave. A cold cut represents a stable state, in which the system may stay forever.

Whenever a scenario's minimal event occurs in a run of the system, a new instance of it is activated. An occurrence of an enabled event, or a TRUE evaluation of an enabled condition, causes the cut to progress. An occurrence of a violating event from the chart, or a FALSE evaluation of an enabled condition, does not cause the chart to progress; instead, if the cut is cold, the scenario instance closes gracefully (we call this a *cold violation*); if the cut is hot, this is considered a violation of the specification (a *hot violation*); in a run adhering to the specification, no hot violations should occur. When the cut reaches maximal locations on all lifelines, the chart instance closes with a *completion*.

Another important feature of the variant of LSC used in our work is its semantics of *symbolic instances* [48], specifically with its universal polymorphic interpretation [44]. Thus, an LSC lifeline labeled with the name of a class (or an interface) may represent any object whose class directly or indirectly inheriting from this class (or

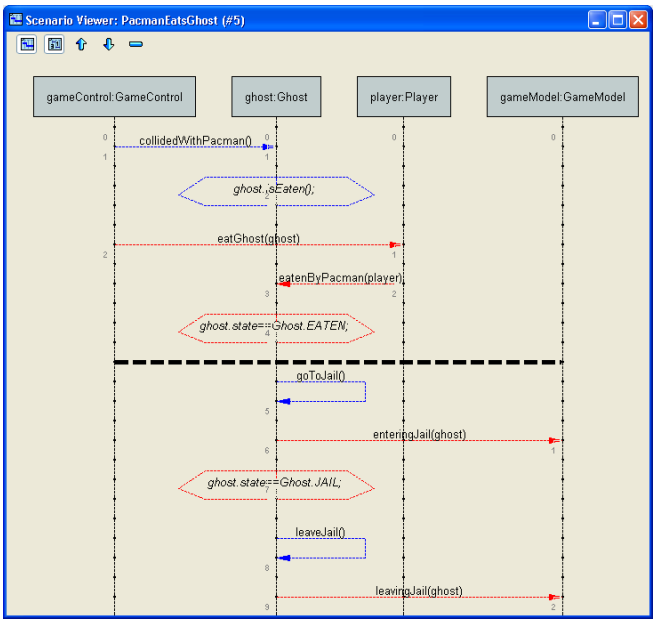

Fig. 2 An example LSC from the PacMan model: `PacmanEatsGhost`. Note the black dashed line indicating a cut drawn at location $\langle 2, 4, 2, 0 \rangle$.

implementing the interface). This allows the definition of succinct and expressive specifications in the context of object-oriented system models.

Figure 2 shows an example LSC taken from our PacMan application. Roughly, it specifies that *'whenever the* `GameControll` *tells a* `ghost` *it has collided with PacMan, and the ghost's* `isEaten()` *method returns TRUE, the game control must tell the* `player` *to eat the ghost, the* `player` *must tell the* `ghost` *it has been eaten, and the* `ghost`*'s state should be equal to* `EATEN`*. If and when the* `ghost` *goes to jail, it must inform the* `GameModel` *when it enters the jail, etc.'*. Note how the difference between hot and cold events and conditions is reflected in the semantics of this chart. Also note the use of symbolic lifelines and the application of the polymorphic interpretation: the chart refers to any of the four ghosts participating in the game. In the figure, a cut is drawn at location $\langle 2, 4, 2, 0 \rangle$ (note the tiny location numbers on the lifelines), which comes immediately after the hot evaluation of the ghost's state. A single enabled event is induced by this cut, namely the ghost's self method call `goToJail()`. This cut is cold, since it has no enabled hot events.

Finally, a *scenario-based specification model* consists of a number of LSCs, divided between one or more use cases. In our settings, use cases do not carry semantic meaning. They are used as means for the designer to organize the scenarios into groups of related functionality.

2.2 Scenario-based traces

Scenario-based traces constitute a specialization of model-based traces. We start by describing model-based traces and their features in general and then continue with scenario-based traces.

2.2.1 Model-Based Traces

Model-based traces, introduced in [43], are aimed at tracing behavioral models of a system's design during its execution, allowing one to combine model-driven engineering with dynamic analysis. Specifically, model-based traces follow the activation and progress of models as they come to life at runtime, during an execution of a reference program. Thus, a system's runtime is recorded and viewed through abstractions provided by behavioral 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.

The following features of model-based traces are noteworthy. First, they can be generated and defined based on partial models; the level of abstraction is defined by the 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 identical model-based traces, if the concrete runs are equivalent from the more abstract point of view of the model used for tracing.

2.2.2 Scenario-based traces

Scenario-based traces constitute a specialization of model-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 the 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

```
Event occurrence:       E: <timestamp> <event no.>: <event signature>
Binding:                B: <scenario name>[instance no.] lifeline <no.> <- <object identifier>
Cut change:             C: <scenario name>[instance no.] <cut tuple> [Hot|Cold]
Finalization:           F: <scenario name>[instance no.] [Completion|Violation]
```

Fig. 3 The four different entry types in a scenario-based trace.

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).
- **Binding**, representing the binding of a lifeline in one of the active scenario instances to an object.
- **Cut change**, representing a cut change in one of the active scenario instances.
- **Finalization**, representing a successful completion or a violation in an active scenario instance.

Figure 3 shows the syntax we use for the different entries. Figure 4 shows a short snippet from a scenario-based trace of PacMan. Note the different types of entries that appear in the trace.

2.2.3 Generating scenario-based traces

A necessary prerequisite for the analysis and visualization of scenario-based execution traces is their effective generation. We build here on previous work of ours, described briefly below.

S2A [22] (for *Scenarios to Aspects*) is a compiler that translates live sequence charts, given in their UML2-compliant variant using the *modal* profile [24], into AspectJ code [7,35]. It thus provides full code generation of reactive behavior from visual declarative scenario-based specifications. The S2A compiler implements the transformation/compilation scheme presented in [45], in which each sequence diagram is translated into a *scenario aspect*, implemented in AspectJ. The scenario aspect 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. Moreover, following the play-out algorithm of [27], we construct another generated aspect, a *coordinator*, which collects cut-state information from all active scenarios, uses a *strategy* to decide on the next event to execute, and executes the selected event using inter-type declarations.

Most important in the context of this paper, though, is that in addition to scenario-based execution (which follows the play-out algorithm of [27]), S2A provides a mechanism for scenario-based monitoring and runtime verification. Indeed, the scenario-based trace shown in Figure 4 is taken from an actual execution log of a real Java program of the PacMan game adapted from [5], (reverse) modeled using a set of live sequence charts (drawn inside IBM Rational SA [3] as modal sequence diagrams), and automatically instrumented by the AspectJ code generated by S2A.

More on S2A and its use for scenario-based execution can be found in [6,45].

3 Trace Visualization: The Basics

We now set out to present our approach to trace visualization and exploration. Many elements of the description refer to the actual displays in the Tracer. Thus, the principles and concepts we present are intermixed with, and demonstrated using, certain high-level aspects of the prototype tool.

3.1 The main view

Basically, we visualize a scenario-based program execution trace using a hierarchical Gantt chart, where time goes from left to right and the hierarchy is defined by the containment relation of the use cases and the sequence diagrams in the specification model. Thus, each leaf in the hierarchy represents a different sequence diagram; the horizontal rows representing specific active instances of a diagram (which we call *scenario instances*), and the bars therein showing the durations of being in the relevant cut states.

The horizontal axis of the main view allows one to easily follow the progress of specific scenario instances over time, to identify the events that caused progress, and to locate completions and violations. The vertical axis provides a clear view of the synchronic characteristic of the trace, by showing exactly what goes on at any given point in time.

The main view uses color coding to visually distinguish between stable (cold) and unstable (hot) cuts: cold cuts are colored blue and hot cuts are colored red. A textual encoding of the cut into a tuple of integers representing locations on specific lifelines may be 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. Alternative rendering functions are also available, e.g., displaying the scenario instance's serial number in the trace, displaying its real-time duration, or avoiding text labels entirely to reduce visual clutter.

Figure 6 shows a representative screenshot of the Tracer's main view. Note the hierarchy of use cases and sequence diagrams on the left and the red and blue horizontal bars representing hot and cold cut states. Also

```
...
E: 1172664920526 64: void pacman.classes.Ghost.slowDown()
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. 4 Part of a textual representation of a scenario-based trace of PacMan.

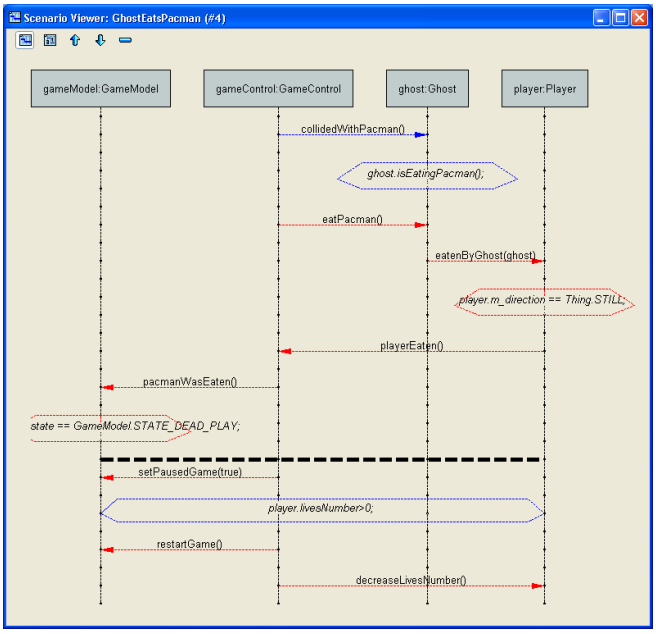

Fig. 5 The 4th instance of the `GhostEatsPacman` LSC, opened as a result of double-clicking the corresponding bar in the trace shown in Figure 6.

note the name of the currently loaded model and trace on the application's window caption (top left corner) and the status line (bottom), among other things showing the total number of events in the trace and the range of events currently visible in the main view.

3.2 Zooming in: Additional details on demand

Additional details are available to the user on demand.

First, when double-clicking a bar, a window opens, displaying the corresponding scenario instance as a sequence diagram, together with its dynamic cut state. Identifiers of bound objects and values of parameters and conditions are displayed when applicable in tooltips over the relevant elements in the diagram. In addition, one can travel back and forth along the cuts of the specific opened instance (using the keyboard or the arrows in the upper left part of the window).

Figure 5 shows an example scenario instance view, opened as a result of double-clicking the corresponding bar in the trace of Figure 6. Note the cut, drawn as a dashed black line, and the toolbar icons allowing to browse back and forth along the different cuts.

Multiple windows displaying a dynamic view of several different scenario instances can be opened simultaneously to allow, e.g., 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.

Second, the user may select an entire column – representing a global cut – and open a dialog window that presents the selected global cut's properties. The dialog shows the list of local cut states of all the scenarios that were active at the global cut represented by the selected column. Figure 7 shows an example of a global cut properties dialog, which was opened for a user-selected column.

3.3 Zooming out: The overview

The *Overview* supporting view, which appears in the bottom part of the application window, displays the main execution trace in a smaller pixel-per-event scale, based

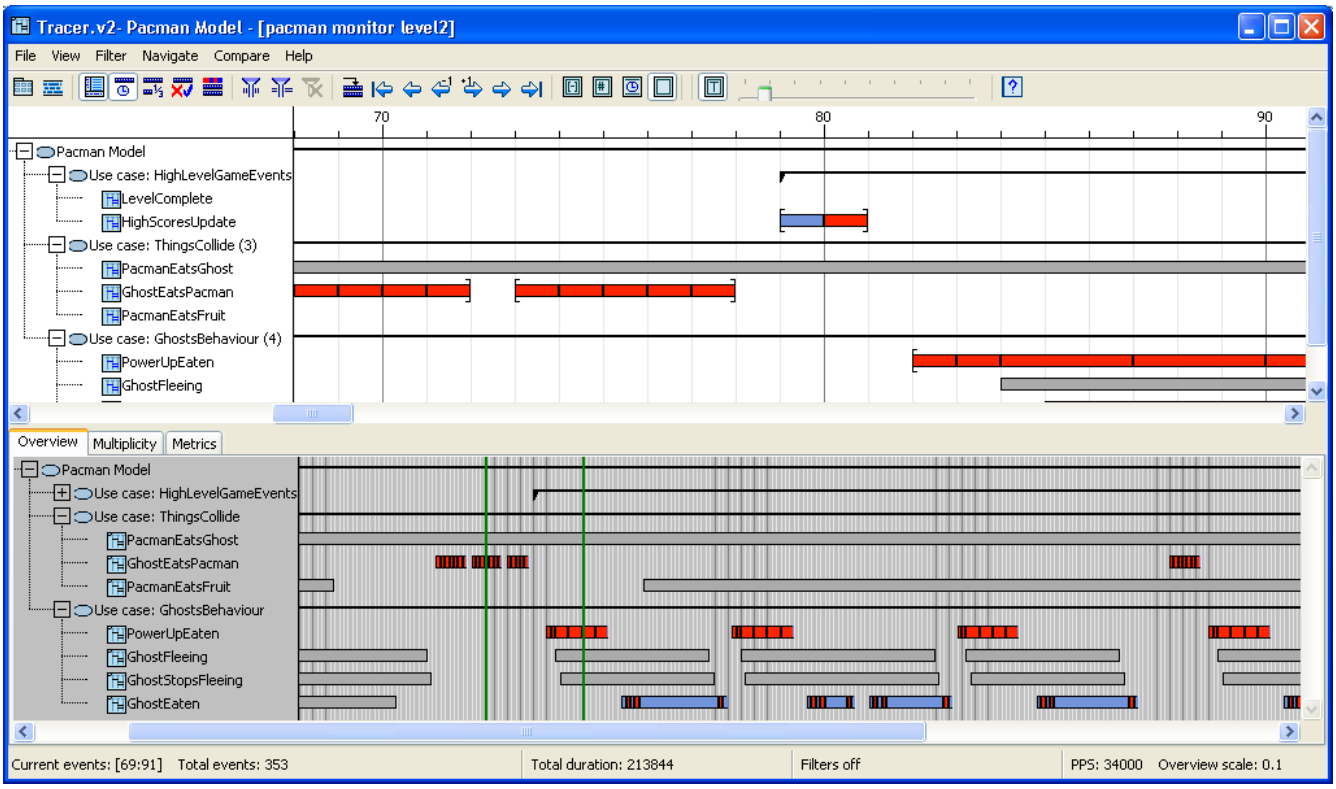

Fig. 6 The Tracer main view and Overview. Note the hierarchy of use cases and sequence diagrams on the left, and the red and blue horizontal bars representing hot and cold cut states. When double-clicking the red horizontal bar shown in the 69th location, a window opens, displaying the corresponding scenario instance of the `GhostEatsPacman` LSC, together with its dynamic cut state (see Figure 5).

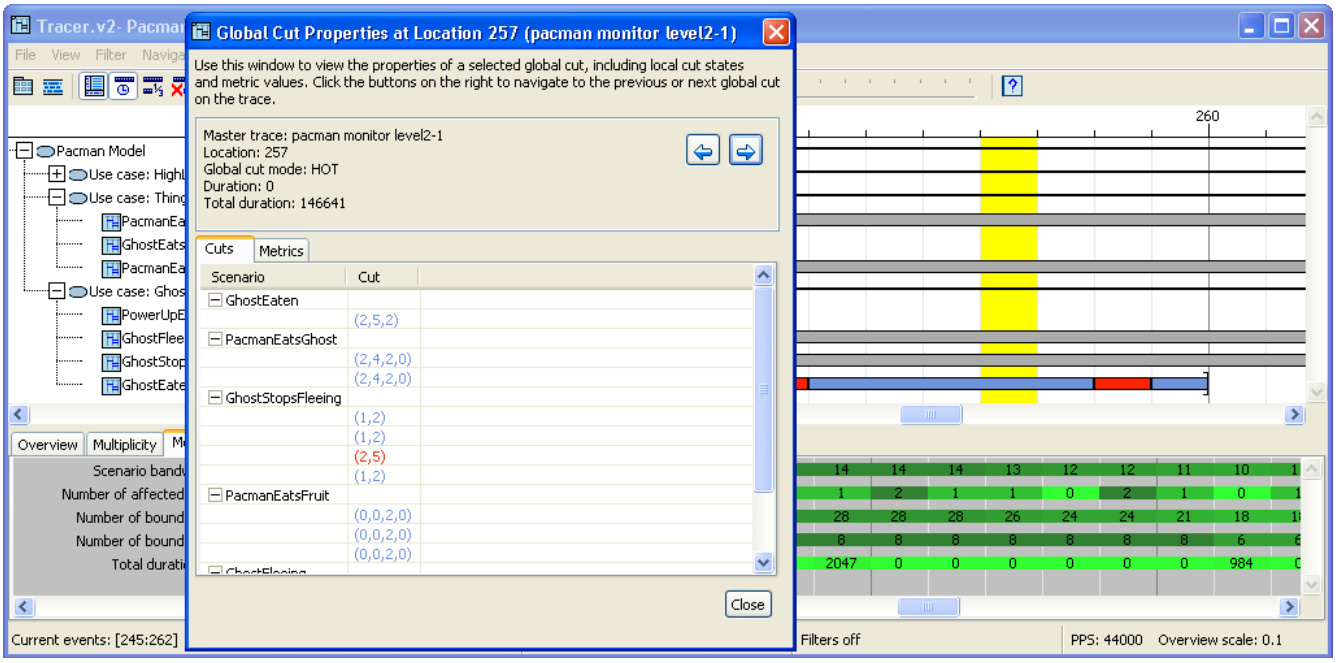

Fig. 7 An example of a global cut properties dialog, which was opened for a user-selected column. The selected column is visible in the main view, behind the opened dialog, highlighted with a yellow background.

on a user defined ratio. Thus, it shows a zoomed-out overview of the execution trace that can assist in, e.g., identifying long scale behaviors, repeated patterns, etc. As expected, the Overview is synchronized with the main view over horizontal scrolls. In addition, it has a moving window frame that shows the borders of the time interval – the range of events – currently visible in the main view. This becomes critical for user orientation when visualizing real-world scale long traces that feature thousands of events.

For example, see the bottom part of Figure 6. Note how the zoomed-out overview allows the user to identify the repeated behavioral pattern between the scenarios in the `GhostBehaviour` use case: after the power up is eaten, the scenarios for ghosts fleeing, starting and stopping, become active. During the activation period of these scenarios, some ghosts may be eaten by PacMan (see the bottom-most LSC `GhostEaten`). This high-level repeated pattern, exposing relationships between several scenarios, with variations, is difficult to identify from the main view but is clearly visible in the Overview.

4 Trace Visualization: Advanced Features

This section presents some of the more advanced visualization features of our work, including multiplicities, time-based and event-normalized tracing, and metrics.

4.1 Handling multiplicities

Multiple instances of the same diagram, where lifelines bind to different objects, may be simultaneously active during program execution (see [48]). Consider the scenario `PacManEatsGhost` from Figure 6: PacMan may eat a second ghost before the first one eaten has entered 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 concurrent instances in the trace's visualization, we introduce a supporting view called *Multiplicity*. On the main view, we hide the details of the multiplicity from the user: we use a single grey bar to cover the row representing the static scenario over the period where more than one of its instances has been active. When the user double-clicks the grey bar, the corresponding instances are displayed in the supporting view. Thus, details about multiple copies of active scenario instances are given on-demand. This feature can be considered a special kind of semantic zoom-to-details from classes to instances, where classes here are 'classes of scenarios' and instances are 'active scenarios'.

Figure 8 shows a sample screenshot of the multiplicities view. There are four concurrent active copies of the `GhostFleeing` diagram, which are displayed in the lower view as a result of double-clicking the corresponding grey bar in the main view.

As an alternative solution, we could have extended the hierarchy of the specification model (use cases, sequence diagrams) with leafs representing scenario instances. When only one scenario is active, the row representing the static diagram would suffice, while when multiple scenarios of the same diagram are simultaneously active, they would appear as sibling rows under the diagram's row.

This may be considered to be simpler and perhaps more user-friendly than our solution, since the user would not need to look for details in other views: all the information about the concurrently active scenarios would be available in the main view. Our solution, however, is more scalable. When the number of concurrently active scenarios grows, the view described in the alternative solution above may become difficult to browse. By displaying the details about the active copies in a separate supporting view, our solution keeps the main view more abstract, and thus easier to browse and understand.

4.2 Time-based and event-normalized tracing

The building blocks of reactive-system traces are discrete events. Indeed, the Tracer's basic view 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, although the input trace includes time-related data (note the timestamps on events in Figures 3 and 4), the 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. For example, the basic variants of *temporal logics*, LTL (linear temporal logic) and CTL (computation tree logic), indeed do not consider the actual durations of happenings but only their order [17]. Similarly, the variant of LSC used in our work does not consider real-time. For example, the LSC shown in Figure 5 specifies that whenever a collision happens, eventually the ghost should eat PacMan, but it does not specify how much time, at most or at least, may elapse between the two events.

In many systems however, the real-time aspect of the trace is important and is then reflected in the specification language chosen, e.g., TPTL [11]; also, the full version of LSC supported by the Play-Engine includes a powerful notion of time (see [25,26]). For example, one may require that not only every request be eventually granted but also that the duration between the requesting and granting events will not exceed a certain duration, specified in actual time units. To support the real-time aspect of the trace, we offer, in addition to the default event-based view, a time-based view, where the horizontal axis accurately reflects the progress of time, regardless of event occurrences.

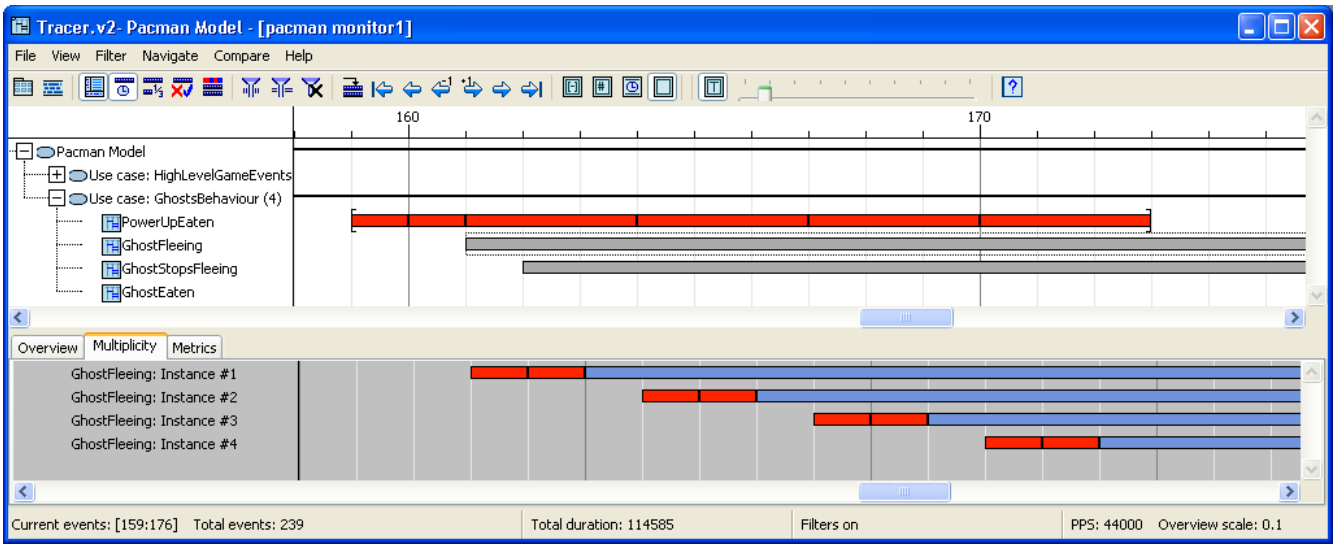

Fig. 8 A screenshot of the multiplicity view. Note the four concurrent active copies of the `GhostFleeing` scenario, displayed in the lower view as a result of double-clicking the corresponding grey bar in the main view.

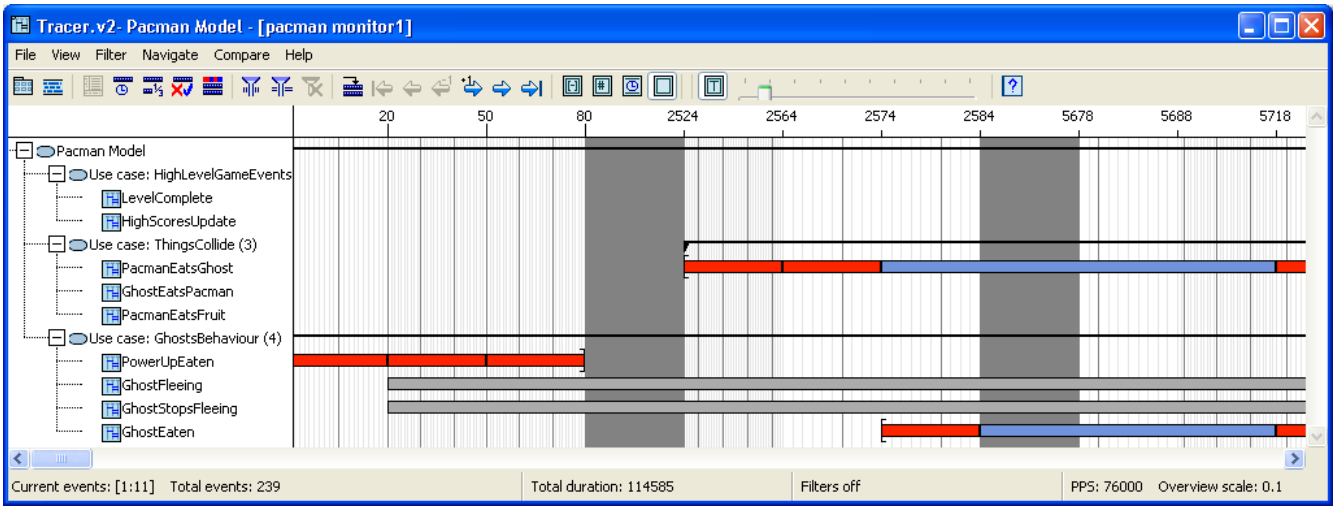

Fig. 9 A screenshot of the event-normalized view. Note the different scales represented by the variable density of the vertical grid lines. For example, consider the different scales between the 2524th and 2564th milliseconds, and between the 2584th and the 5678th milliseconds. In the former, 40 milliseconds have passed between two consecutive events, but in the latter, more than 3 seconds. In the real time-based view (not shown here), 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.

This time-based view correctly reflects the trace's 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 view may be formally accurate but very difficult to comprehend and browse visually. Such high variability, i.e., short periods with many events and lengthy 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*. This view combines the event-based and time-based presentations by allocating a fixed horizontal interval to each event (more precisely, to each set of events having the same timestamp), while programming the grid appearing in the background to draw vertical lines at every fixed time unit. As a result, the displayed time-based vertical lines are unevenly spaced: in periods where very few events occurred, the lines are dense, while in periods where many events occurred, they are sparse. See Figure 9.

4.3 Metrics

The *Metrics* view of the Tracer, shown in Figure 10, displays 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 these metrics may be relevant to performance and resource allocation analysis, and others may be relevant to better understanding of the relationships between the scenarios and the system's objects.

The metrics are displayed using color gradients. We believe they are appropriate, since for most purposes it is the metric's relative qualitative values, e.g., high/ medium/ low, that are of interest, not the precise values. The user may switch between two display modes, with or without the actual number values.

One of the more interesting metrics we have is *duration*. As explained earlier, in the basic event-based tracing mode grid lines are evenly spaced between events and the data about the real durations is abstracted away. The duration metric compensates for this abstraction. It displays the real duration of the periods between events, visually overlaying the real-time dimension onto the fixed event-based view.

Figure 10 shows two possibilities of the metrics view: without values on the left and with values and in a different scale on the right. Note the dark colored bar for the 149th event in the duration metric (bottom right). This clearly indicates a long period with no events, more precisely, a period in the execution with no event occurrences that are relevant to the progress of any of the scenarios in the specification model. This time-related information about the trace is otherwise abstracted away in the event-based view.

4.4 Completion-related metrics

We use three notions to formalize and summarize the relationship between the scenario-based specification model and the trace.

A scenario is considered *completed* in a trace, if it has at least one instance that has reached its maximal cut state and no instance with a hot violation (an instance of a scenario is defined as a *completion* of this scenario if it reaches its maximal cut state). A scenario that has at least one hot violation in the trace is considered *violated*, even if it also has completions. *Vacuous* scenarios are those that are neither completed nor violated. Completion, violation, and vacuity information is aggregated from the scenario level through the use case level up to the entire specification model level.

Note that not all scenario instances are completed or violated. Some instances may be closed gracefully with a cold violation before reaching their maximal cut state. Moreover, since the trace is finite, at the end of trace some instances may be truncated without closing.

The completion properties of all the scenarios and use cases in the specification model are summarized and visualized using representative symbols, optionally displayed on the specification model tree itself and at the end of every scenario instance in the main view of the trace. An example of these is shown in Figure 11. In addition, completion related metrics for the entire trace (e.g., the number of vacuous scenarios) are shown in the scenario, use case, and model properties windows (see, e.g., Figure 15). Additional completion related features are briefly described in Section 5 below, as part of the general discussion on trace exploration techniques.

Finally, we define the *completion coverage* of a trace vis-à-vis a model (or a use case) to be its ratio of completed scenarios; that is, the number of completed scenarios, divided by the total number of scenarios in the model (or the use case). One may view this ratio as providing a high-level yardstick by which to measure the "quality" of an execution trace versus a scenario-based specification model. The completion coverage ratio is displayed in the use case and model properties windows (see Figure 15).

Note that the completion related metrics are defined for the trace as a whole, not for a single point in time. Thus, in contrast to the metrics of subsection 4.3, the completion related metrics are 'horizontal' rather than 'vertical'.

Taken together, the completion related metrics provide an overview of the trace's characteristics in relation to the specification model. They may be useful in the context of testing, where LSCs are used to specify required testing scenarios. They may also be useful in the context of scenario-based programming, as they can help in identifying redundancies in the specification or other gaps between expected and actual executions.

5 Trace Exploration

Our work goes beyond static visualization techniques and presents various interactive trace exploration features, including navigation, filtering, and comparisons.

5.1 Navigation and additional information

We consider three modes of navigation features: general, semantics-based, and metric-based.

General navigation features include basic 'go to's to next/previous views and beginning/end of trace. In addition, the user may navigate to a specific location on the trace by specifying its number.

Additional navigation features are semantics-based; allowing the user to navigate the trace by specifying a semantic criteria related to the trace, including, e.g., 'go to next hot violation', 'go to last completion', 'go to first instance' etc. These navigation options can be applied at each of the three levels: a selected scenario level, a selected use case level, or the entire model level. The selected level defines the scope by which the semantic criteria is evaluated.

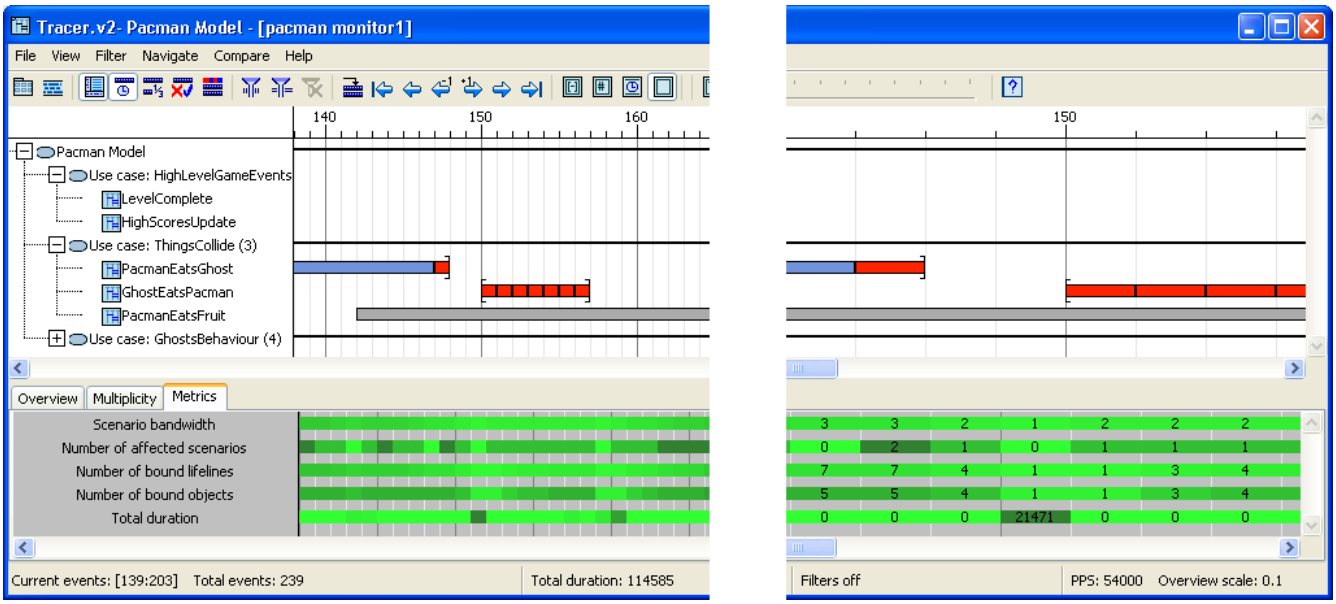

Fig. 10 The metrics view, without values on the left and with values and in a different scale on the right. Note the dark colored bar for the 149th event in the duration metric (bottom right), indicating a relatively long period of more than 21 seconds, where no event relevant to the progress of the trace has occurred.

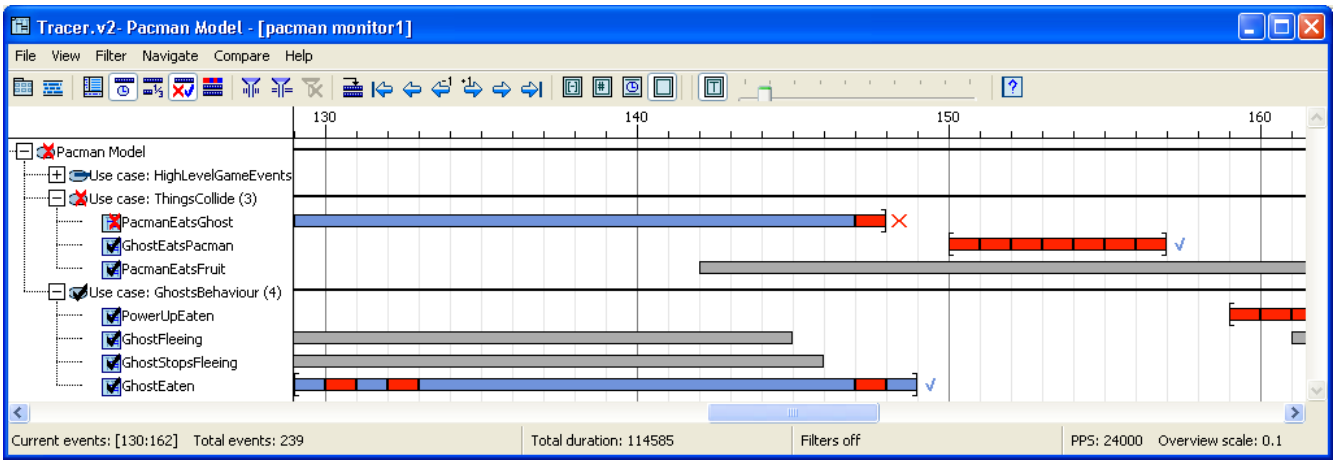

Fig. 11 The main view with completion information. Note the $\times$ and $\surd$ symbols representing violations and completions, respectively, at the end of scenario instances bars. Also note the aggregated completion status information represented by similar symbols on the icons in the specification hierarchy on the left.

Finally, the user may ask for aggregated information about the vertical metrics, including maximal, minimal, average, median, and most common value for each metric, as computed over the complete trace (see Figure 12). This additional information has corresponding metric-based navigation features, such as 'go to next maximum', 'go to last minimum' etc. (available from a context menu), and a 'search for' first/ previous/ next/ last occurrence feature for user-defined values, per metric. For example, one can navigate to the location with the maximal cut-state duration, or to the next location on the trace where no scenario was active.

Together, these navigation features support a convenient and focused trace browsing experience, assisting the user in quickly finding locations of interest on the trace. The use of the different navigation features is critical for effective exploration of long and complex traces.

5.2 Filtering

Filtering is used to exclude elements from the view, allowing the user to hide certain elements or parts of the trace in order to better focus on ones that are relevant to a specific task. We consider two general types of filters – horizontal and vertical – each of which may be custom-defined by the user or pre-defined and calculated based on semantic criteria.

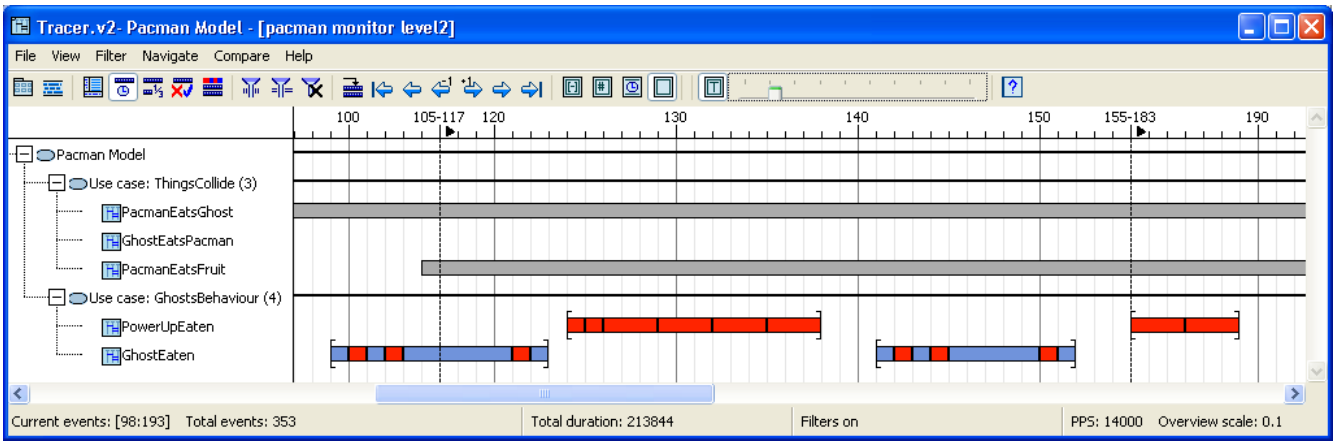

Fig. 13 Horizontal and vertical filters. Note that the first use case, `HighLevelGameEvents`, and two of the scenarios in the last use case, `GhostBehaviour`, have been filtered out from the view. Compare with the complete specification model shown in Figure 6. Also note the application of vertical filters, between locations 105 and 117 and between locations 155 and 183.

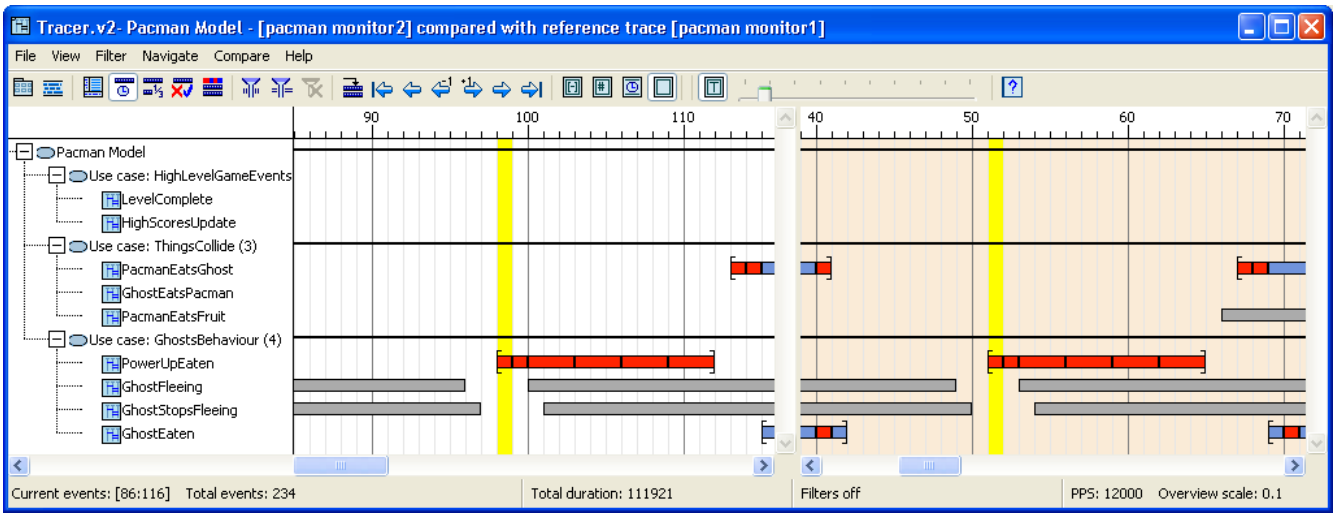

Fig. 14 Comparing traces of the same model. Note the highlighted columns at locations 99 and 51 of the master (left) and reference (right) traces, respectively, indicating that the Tracer has found them equivalent. Note also that the two traces seem to remain equivalent until 15 locations later, at location 66 of the reference trace (an instance of `PacManEatsFruit` starts only in the reference trace). This can be verified by using the highlighted cuts as anchors and choosing the Tracer's 'search forward difference' feature. Also, it may be the case that if the `PacManEatsFruit` scenario is filtered out, the next global cut states would become equivalent. Again, this can be checked by the Tracer.

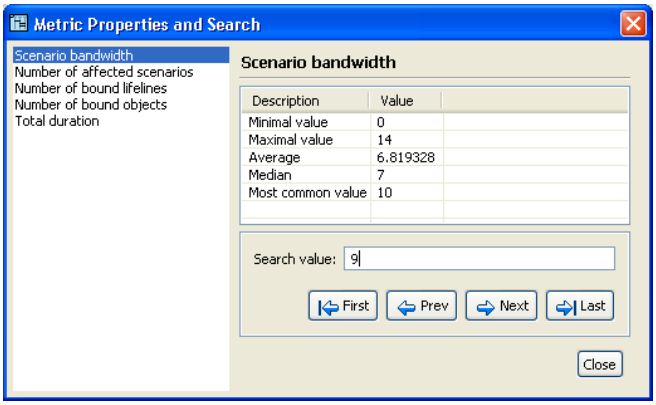

Fig. 12 Aggregated metric properties and search.

5.2.1 Horizontal filtering

Horizontal filters deal with the exclusion of scenario instances, scenarios, or entire use cases from the view.

Custom user-defined exclusion of scenarios or use cases is done in a rather standard way, by selecting specific items directly from the model.

Pre-defined semantics-based criteria for the filtering may also be employed. They include, for example, the exclusion of all scenarios (entire rows) with no hot violations, which renders visible only the scenarios that have been violated at least once during the recorded run, or hiding all completed instances, which leaves the trace with violating and incomplete (vacuous) instances only. Filtering is applied to the entire trace, automatically re-

moving from the view all use cases, scenarios or instances meeting the filter's criteria.

Figure 13 shows a sample screenshot of a PacMan trace after the application of several filters.

5.2.2 Vertical filtering

Vertical filters deal with the exclusion of ranges of trace locations from the view.

Custom user-defined vertical filters are set by selecting ranges of columns directly on the trace or by choosing location range numbers in a dedicated dialog window.

Pre-defined semantics-based criteria for vertical filtering may also be employed. Such filters include, for example, the exclusion of all ranges where no instance of a selected scenario was active, or the exclusion of all ranges where no multiple instances of a selected scenario were simultaneously active.

Excluded ranges are removed from the view and are replaced by dashed vertical lines. A dashed vertical line is headed by a label showing the location numbers of the hidden range. An example trace view with two hidden ranges is shown in Figure 13.

5.2.3 Visual vs. logical filter application

The filters described above are primarily visual; that is, they are used to exclude elements or parts of the trace from the view. Yet, one may consider not only the visual impact of these filters but also their possible logical impact. In other words, the filters may be employed also as logical abstraction mechanisms over the model and trace. We consider this to be in important facet of our work.

To support the application of filters as logical abstraction mechanisms, all properties of the model, trace, global cut states, and metrics, may be calculated with and without the applied filters. For example, Figure 15 shows the properties of a selected use case. The second column on the right shows the value without the application of filters and the value with the application of current filters (in parenthesis).

The application of filters as a logical abstraction mechanism allows the user to find answers to 'what-if' questions over the model and trace. Examples include: If I exclude this scenario or that use case, will the trace still include violations? If I ignore this range of the trace, would the trace still include concurrently active scenarios?

5.3 Comparisons

It seems clear that a lot can be learned about a system under investigation by comparing different traces, especially model-based ones. Obviously, we are not interested in a Boolean comparison, which tells only whether two

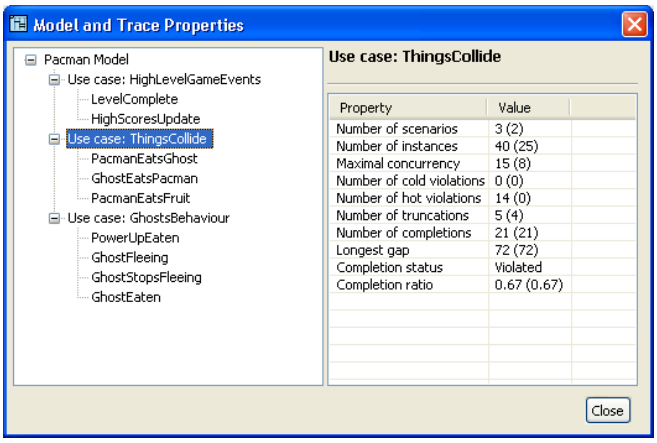

Fig. 15 Model and Trace Properties window, showing the properties of a selected use case. The second column on the right shows the value without the application of filters and the value with the application of current filters (in parenthesis).

traces are equal or not, but in much richer kinds of comparisons that cover various similarities and differences between traces and their elements.

The need for trace comparison capabilities may arise in practice in various contexts. For example, for evolution and version control, one may be interested in comparing various traces related to different yet very similar models – different versions of the same model. On the other hand, the same program may yield very different execution traces based on its initial configuration and environment inputs or behavior over time. Thus the need also arises to compare different execution traces of the same model.

Our work is deliberately limited to the abstraction level defined by the specification model used for tracing, and the kinds of comparisons we are discussing here are no exception; they are carried out with respect to the same abstraction. Thus, as mentioned in subsection 2.2.1, different concrete runtime traces may result in identical model-based traces – in our case, scenario-based traces – if the concrete runs are equivalent from the more abstract point of view of the model used for trace generation. Our comparison features indeed reveal this important property.

When a model and a trace are opened, the Tracer allows the opening of a second (reference) trace, and compares it with the main (master) trace. Below we describe a number of features related to various aspects of trace comparison.

5.3.1 Comparing global cut states

A comparison of selected global cut states, one from the master trace and one from the reference trace, can reveal various similarities and differences. For example, showing which scenarios were active in one but not the other, showing which scenarios were active in both, and show-

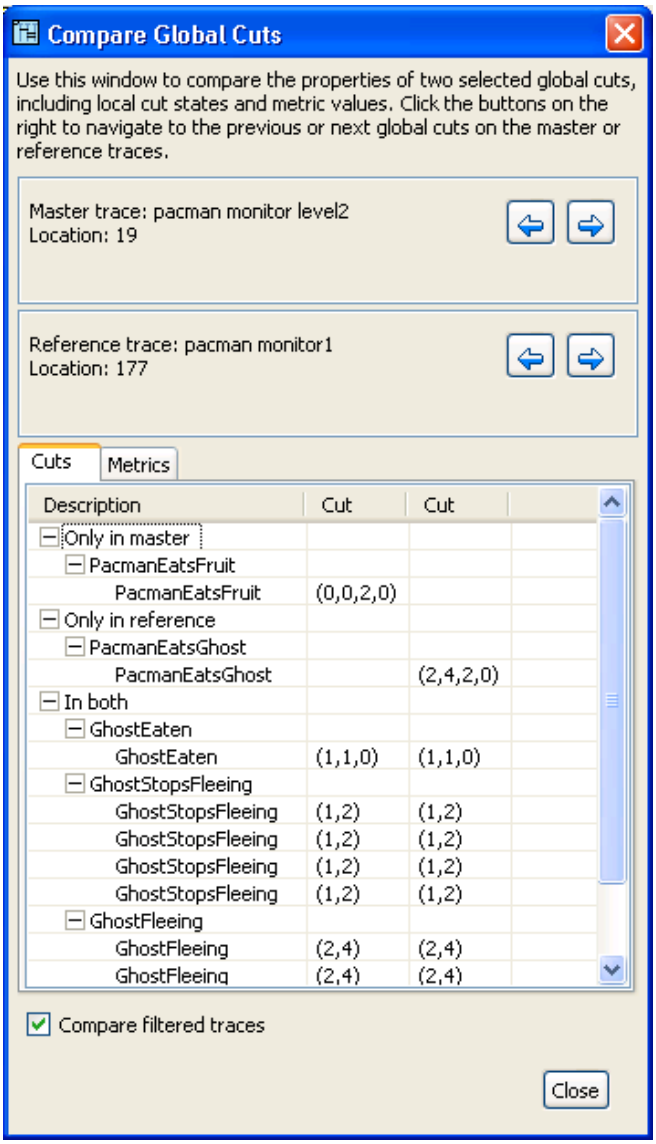

Fig. 16 Compare Global Cuts window, showing a comparison between two selected global cuts. The arrows on the right allow the user to browse backward and forward for global cuts on the master and reference traces.

ing the synchronous metrics of the two selected cuts side by side to allow for visual comparison. Figure 16 shows an example screenshot from the dialog window responsible for the global cut-state comparison.

5.3.2 Searching for equivalent global cuts

One way to relate two traces is to look for locations where they are equivalent. Equivalence is defined modulo the abstraction induced by the scenario-based model; that is, by comparing active scenario local cut states.

To this end, the user may select a column – representing a global cut – in the master trace, and ask the Tracer to find all equivalent global cut states in the reference trace. These may then be used as starting points or anchors for further investigation of the similarities and differences between the traces (see Figure 14).

Note that global cuts equivalence can be calculated with or without the application of filters (see subsection 5.2.3). This may be of interest, since two global cuts may be concretely different but equivalent under a stronger abstraction defined by certain filters (horizontal filters in this case).

5.3.3 Searching forward / backward

As a kind of a generalization of searching for equivalent cuts, we can start with two equivalent selected global cut columns used as anchors. The Tracer then can look for maximal forward or backward ranges of trace locations where the equivalence still holds. That is, given equivalent locations on the two traces, the Tracer finds the closest differing location.

The search results are presented to the user by highlighting the equivalent ranges found in the two traces. Like the other comparison features, searching forward and backward differences can be performed with or without the application of filters.

5.3.4 Comparing trace metrics and completion coverage

The master and reference traces may also be compared with respect to global horizontal properties. These include the comparison between, e.g., the total number of instances throughout the trace, the minimal, maximal and average cut-state duration, and the traces' completion coverage. Again, the user can choose to employ the comparison with or without the application of filters.

6 Usage Examples

To demonstrate the utility of our approach in supporting model-based dynamic analysis tasks, we briefly discuss two examples of specific usages: scenario-based testing and program evolution review. Similar usages of the Tracer's features apply to program comprehension and requirements traceability tasks.

6.1 Scenario-based testing

In scenario-based testing, the engineer defines one or more scenarios and captures them as LSCs, each of which specifies an expected or a forbidden behavior of the system under test. Using the S2A compiler, the scenarios are transformed into aspect code, which, at runtime, drives and monitors the progress of the specified testing scenarios, producing a scenario-based trace, which is given to the Tracer as input together with the set of scenarios.

In the Tracer's main view, the engineer follows the instantiations of the different testing scenarios during the run, and identifies completions and violations (see subsection 4.4 and Figure 11). When a violation is identified — representing a testing scenario that has failed to complete as expected — the engineer may zoom-in for more details by double-clicking the corresponding bar and opening the violated scenario instance (see subsection 3.2 and Figure 5).

If the trace is long, as is usually the case, the Overview supporting view (see subsection 3.3 and Figure 6) assists the engineer's orientation. Moreover, the semantics-based navigation features, such as 'go to next violation' (see subsection 5.1), are used to quickly navigate between locations of interest.

The filters described in subsection 5.2, such as the horizontal filter that excludes all scenarios with no hot violations, are used to delete irrelevant information from the view. These can help the engineer focus on the parts relevant to the investigation of the test results at hand; that is, the violated testing scenarios.

Finally, aggregated summary data about the test results (e.g., the total number of violations), is available in the Model and Trace Properties Window (see Figure 15).

6.2 Program evolution review

The goal of a program evolution review is to identify the differences between two versions of a program under investigation. In the context of our work, we are not interested in the syntactic differences between the two versions of the program at the code level but in the different behaviors that the two versions admit during their execution. Thus, we would like to compare an execution of one version of a program with an execution of another version of the same program, and want the results of the comparison to be presented at the level of abstraction of the models used in the design of these programs. The syntactic difference at the code level may be known and perhaps simple, but we are interested in the effect this difference may have on the program's behavior.

The engineer defines a model consisting of a set of scenarios specifying different behavioral properties of the program under investigation. The model is used to instrument the code of the two program versions at hand, so that their execution creates two scenario-based traces (if the program interacts with the environment, it may be necessary to simulate the same environment behavior during the two runs). The two traces are loaded into the Tracer as master and reference traces (see subsection 5.3).

To start off, the engineer may ask the Tracer to find the first difference between the two traces, starting from their initial global cut state. If no difference is found, it means the two traces are equivalent from the point of view of the model used for tracing. Otherwise, if a difference is found, the engineer can identify the differencing scenario instance or instances, zoom-in for further investigation or continue looking for differences along the two traces.

In some cases, several differences between the two traces are known *a-priori*, and are thus of not much interest (except to confirm that indeed a required change was made). Using the various filters presented in subsection 5.2, the engineer can ask the Tracer to abstract away and ignore these differencing scenarios or fragments of the trace in the comparison. Then, additional differences that may be unexpected or accidental could be revealed, hinting at some program behaviors the engineer was not aware of.

7 Evaluation

7.1 The Tracer prototype implementation

In order to evaluate and test our ideas we have implemented a prototype tool called the *Tracer*. All screenshots shown in this paper are taken from the Tracer. Additional documentation, including screenshots and screencasts of the Tracer demonstrating its various features, are available on our Tracer website [9].

The Tracer reads UML2 models containing sequence diagrams extended with the *modal* profile of [24]. Its input scenario-based traces are given in a simple text format, similar to the one shown in Figure 4. As mentioned earlier, we have generated such traces from various applications using the S2A compiler and the Play-Engine.

The Tracer is written in Java, using SWT. It uses the Eclipse UML2 project API [2] to read the UML models. The Gantt charts are based on the jaret timebars component of [8].

We have made an effort to render the Tracer's prototype implementation scalable, allowing it to handle long traces of rather complex models. As the examples below show, the current implementation can handle traces of 10K events, spanning 50 different sequence diagrams with a total of approximately 500 scenario instances, on a regular personal computer, while maintaining very reasonable user experience. We acknowledge, however, that industrial usage will involve longer and more complex traces. These will require special technical solutions for optimized, scalable performance, which are beyond the scope of the present work.

7.2 Experience

We describe some of our experience in experimenting with the Tracer on a number of applications. We deliberately choose to present here three cases from different application domains and of different technologies: a desktop application written in Java and executed using code generated by the S2A compiler [22]; a Nokia smartphone

application written in C++, executing scenario-based test aspects generated by S2A; and a biological system simulated by the Play-Engine [26]. Selected screenshots from the different experiences described in short below are available in [9].

7.2.1 Case study 1: An RSS News Ticker

We have used the Tracer to visualize some execution traces of an RSS News Ticker application, previously developed as an example case study for scenario-based execution using the S2A compiler [22]. The News Ticker is a small desktop application; it downloads RSS news from user defined websites and presents them to the user as continuously scrolling text. Additional features include switching between horizontal and vertical presentation modes, switching between several predefined scrolling speeds, changing the URL for the RSS feeds, and, when a headline is clicked, opening the corresponding news item in the browser window. The model and source code for the News Ticker are available from the S2A website [6].

The model for the News Ticker application consists of seven scenarios divided into two use cases. The typical traces we used were 8K-10K events long. These rather long traces are due mainly to repeated time tick and text scrolling events. Other events, such as the ones involved in changing the scrolling speed, are relatively rare. By applying the pre-defined vertical filter option 'hide inactive ranges' to the use case in which these more rare scenarios are grouped, we were able to automatically exclude most of the trace from the view, leaving a filtered trace showing only the very few and relatively short ranges where the scenarios of interest were active. When this filter was not applied, given the length of the trace, the Overview supporting view and the semantics-based navigation options such as 'go to next instance' were helpful in browsing the lengthy trace and looking for locations of interest.

7.2.2 Case study 2: Testing a Nokia smartphone

In [47], a modified version of the S2A compiler, which generates AspectC++ code rather than AspectJ code, was used to execute and monitor scenario-based test cases of a C++ application running on Symbian OS inside a Nokia smartphone (specifically, Nokia model N96). In this work, the Tracer was used to visualize and explore the progress and results of the different test execution traces.

Most relevant features in this context included the completion information displayed on the main view, which allowed us to identify completed, violated, and vacuous test scenarios, and the Model and Trace Properties window, showing aggregated completion-related information for the entire trace. Thus, we could easily answer questions such as which test scenarios have been violated during a run (if any), how many times has each test scenario been completed, etc.

The semantics-based filters, specifically the ones related to completion metrics, allowed us to hide the completed scenarios from the view and focus on the violated ones – those representing tests that have failed and hence require further investigation.

7.2.3 Case study 3: Simulation of a biological system

Finally, we have used the Tracer to visualize and explore different execution traces of the biological system model described in [34], which deals with the process of vulval precursor cell fate determination in the development of the *C. elegans* nematode worm. The model was implemented in the Play-Engine tool. It consists of more than 400 scenarios, divided into 22 use cases. Typical traces are 300-400 events long, and involve 40-80 different scenarios from the model.

Due to the large number of scenarios in the C. elegans model, using the various horizontal filters proved crucial. Moreover, typical traces included several periods with large numbers of multiple active instances of the same scenario (up to 30 such). The Multiplicities supporting view helped us in exploring the details of these multiple copies.

We examined a number of execution traces of this model. Differences between the traces were due to the probabilistic choices inside the LSCs in the model, as played-out by the Play-Engine, and due to our use of different initial configurations, simulating experiments with the worm's wild-type and the various mutations and cell ablations that appear in the *C. elegans* literature. Using the trace comparison features, we were able to find where such different 'experiments' show different behavior and where their behaviors are equivalent.

7.3 Limitations and challenges

Some limitations of our approach in general and its implementation in the Tracer in particular are important to note.

First, following the scenario-based approach to modeling, we focus on *inter-object* reactive behavior, that is, on the temporal properties of the interactions between the objects in the system as they materialize at runtime in the method calls between them. Except for the use of conditions (and guards in interaction fragments, and methods' arguments) within scenarios, we do not cover data, in particular, data structures and their creation and manipulation at runtime. Thus, our approach is not suitable for visualizing the execution of data intensive computations, e.g., sorting, graph traversals, numerical algorithms, etc. If tracing and visualization mechanisms for data and data structures manipulation are developed,

perhaps they can be integrated with our work to create a combined solution.

Second, as with any dynamic analysis approach, the value of our technique depends first and foremost on the information embedded in the traces provided to the Tracer as input. The Tracer is limited to analyzing the concrete traces it gets as input; it cannot be used to check invariants or to extract properties that apply to all possible executions. For example, if a scenario is not violated in any of the traces analyzed by the Tracer, one cannot infer that it will never be violated in any trace of the system under investigation. Furthermore, important issues, such as the consistency and completeness of the scenario-based specification model, are not covered at all.

Third, the dependency of our work on input traces limits its applicability to systems for which adequate tracing mechanisms are available and where the tracing mechanism itself does not significantly affect the resulting execution. Our work uses S2A for automatic instrumentation of tracing code from input models, using generated aspects. The general availability, efficiency, and scalability of the trace generation mechanisms, although important, are outside the scope of our work.

Finally, here are some of the challenges we identified during our experiments with the Tracer. Some of them relate to the future work directions we suggest in Section 9.

One feature we found missing is a link back to the executing code. That is, given a location in the trace, we would like the tool to open up the original program at the exact point in the code corresponding to it. We note that this requires more information to be saved in the trace, but it is indeed possible. A related feature that is missing is on-line tracing, possibly when running the reference program in debug mode. In this case, trace visualization may be done in real-time and the link back to the code may take advantage of debugging information, such as the program's stack trace. In the case of traces generated from an interpreter-style simulation tool, such as the Play-Engine [26], the link back to the model may reproduce the configuration corresponding to the user-selected global cut, so that execution can actually continue from that point on. We leave these for future research work.

Another feature we found missing is the ability to analyze and aggregate statistics from multiple traces. We typically have many traces of the same model; summarizing their similarities and differences in a concise way could have been helpful.

In terms of exposing the information embedded in the traces, we found the objects and events perspective somewhat lacking. That is, the Tracer provides good tools for navigation and exploration, from the scenarios' point of view. However, some basic, more traditional information is not easily accessible. For example, we would like to have the event and object information more readily available. This may include additional navigation and filtering features, such as 'go to next occurrence of...' for a user-selected event, or 'hide all instances involving object...' for a user-selected object.

Some other usability features seem natural to add, such as a bookmarking feature (so that the user can mark locations of interest on the traces with notes), or adding views persistence (the ability to save and reload specific views). We leave these too for future implementation.

In subsection 7.2 we described our experience with the Tracer, in different application domains and for different purposes. We acknowledge, however, that further experiments, in particular, with software engineers, are required in order to better assess the value of our work to specific engineering tasks. Such experiments would certainly provide us with a more realistic evaluation of the advantages and limitations of our work as it may be applied in practice.

8 Related Work

We now discuss related work in the areas of execution trace visualization, scenario-based programming, and time-series data visualization.

8.1 Execution trace visualization

Software visualization [16,60], is a research field concerned with the visual representation of information about software systems. It is aimed at supporting software engineering tasks, such as development, comprehension, and analysis, mainly by providing visual representations of the structure, the behavior, and the history of a software system's code.

The visualization of execution traces, as a topic within software visualization in general, has already been suggested and implemented (for a survey, see [19]). However, most trace extraction and visualization efforts to date (e.g., [14,31,33,49,53,56,57]) consider the trace at the code level. In contrast, our traces are abstracted to the model level; we only trace events that are relevant to the model defined by the user, and our traces embed information about the progress of the model, its states etc., which do not always exist explicitly in the code of the reference program. In this way, our approach combines tracing with model-driven design.

Moreover, instead of looking for interaction patterns in the extracted traces (as in, e.g., [33]), or visualizing parts of the recorded trace using a sequence diagram (as in, e.g., VET [49], Eclipse TPTP [1], or IBM Rhapsody [4]), 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 during the execution of the reference program. Finally, we consider not only the partial-order semantics of sequence diagrams in general, but significantly

also the modal, hot/cold semantics of the live sequence charts language (in its UML2-compliant variant) and its symbolic polymorphic interpretation. These allow for stronger expressive power with regard to temporal, functional, and structural properties.

Some previous work in the area of execution trace visualization considers the data at the level of the code's actual syntax (see, e.g., [52]), referencing source code statements and line numbers. In contrast, our work abstracts away from the source code; we do not differentiate between two calls to the same method coming from different locations in the source code of the same class. Moreover, due to the polymorphic interpretation, we do not differentiate even between two calls to the same method coming from or implemented in *different* classes, as long as the method appears in the specification model on symbolic lifelines whose (ad-hoc) polymorphic interpretation covers the same concrete objects involved in the interaction.

Regardless of the level of abstraction used, most previous execution trace visualization work, like ours, is based on a two dimensional representation. Time goes along one axis, and a certain hierarchy that is based on the structure of the system's implementation is depicted on the other axis (e.g., packages, classes, objects). In contrast, our hierarchy comes from the scenario-based specification model, which consists of use cases and sequence diagrams. These reflect the requirements perspective or the specification perspective of the system. They not necessarily correspond to elements of the structure and implementation of the system under investigation. We consider this to be a fundamental differentiating aspect of our work.

Other work uses a graph representation, where nodes represent participating objects and edges represent the calls between them (see, e.g., [39]). Some recent work combines extended variations of the two representations, the two dimensional representation and the graph representation (see, e.g., [14]). It may be interesting to examine the visualization of scenario-based traces using these techniques.

Recent work by Reiss [55] proposes visualizing program execution by following the states of a user-defined automaton on the traces. This is very interesting work that has some similarities with ours. Indeed, it seems that our work and [55] share the intuition that using a user-defined behavioral model as the basis for tracing has important advantages: trace generation focuses only on events that are relevant to the model and thus results in a reduced runtime overhead, and the generated traces are presented at a level of abstraction that is meaningful and useful for the engineer analyzing the program.

That said, some key differences between our work and the work described in [55] should be noted. First, we use a visual language to specify the user-defined behavioral models. Second, the models we use for tracing have rich semantics (e.g., stable/unstable cold/hot cut states, conditions, polymorphic interpretation). Third, our work contains many additional features, e.g., the details-on-demand link from the bars to the diagram displaying the cut, the event/time-based viewing combinations, the multiplicities and metrics views, the horizontal and vertical filters, and the various comparison features. Fourth, [55] emphasizes the performance of the trace generation technology, which is important when considering large and distributed systems. Since the focus of our work is on trace visualization and exploration, we consider neither trace generation performance issues nor tracing technologies for distributed systems. Our tracing technology was briefly presented in subsection 2.2.3, with the relevant references, and its details are outside the scope of the present paper.

8.2 Scenario-based program visualization

The Play-Engine [26] is an interpreter-style simulation engine built in our group for LSCs, based on the *play-in/play-out* approach [27]. The Play-Engine and the Tracer are very different: while the Play-Engine focuses on execution of LSCs, that is, by implementing the play-out mechanism, the Tracer does not execute a scenario-based specification. The input for the Tracer includes a recording of a "scenario-based" run (generated by a program created using S2A or by the Play-Engine or by some other mechanism) and the Tracer focuses on analyzing and visualizing that run.

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 execution comprehension, the result is often information overload: when many LSC windows open and close rapidly during execution, the effectiveness of the visualization decreases. The approach presented in this paper and implemented in the Tracer seems to constitute a 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-state information are visually shown (and these cuts can be viewed diachronically in the context of the execution's past and future). In addition, the Play-Engine is an LSC-specific closed environment. In contrast, the Tracer can be used for the scenario-based analysis and visualization of third-party programs, written, e.g., in Java, provided that an appropriate scenario-based tracing mechanism (such as the one implemented by S2A) is available.

SIV [29] (for *scenario inter-dependency visualization*) is a tool aimed at visualizing inter-dependencies between scenarios in a scenario-based specification. Based on input from the Play-Engine, SIV displays a graph, where each node represents a scenario and edges between nodes represent various types of dependencies, such as possible

causality and synchronization relations. The tool supports a number of filtering and aggregation operations on the graph. It thus allows the presentation and exploration of aspects of the specification that are otherwise hidden or difficult to identify. While it has some dynamic aspects, SIV focuses on a static (or a snapshot-based) view of a scenario-based specification model. In contrast, the Tracer focuses on specific 'runs' of the model and their progress over time. Thus, the two tools may be viewed as complementary.

8.3 Time-series data visualization

Finally, the analysis and visualization of program execution traces is related to the exploration and visualization of time-series data in general. Time-series visualization has been the subject of much previous work in the context of information visualization and data mining, e.g., of financial or health care related data; see, e.g., [10,41,61]. One recent approach is [30], where Hochheiser and Shneiderman present Timeboxes; rectangular widgets used in direct-manipulation graphical user interfaces to specify query constraints on time-series data sets. It is an interesting direction for future work to combine TimeBoxes with the Tracer, making it possible to apply time-related direct-manipulation graphical user interfaces techniques to our model-based execution traces (both event-based and time-based). This will yield dynamic querying capabilities and greater visual insight into the execution of complex reactive systems.

9 Future Work

Some of the ideas for future work we now discuss follow the challenges described earlier, in subsection 7.3.

9.1 Richer scenario-based traces

Enriching our scenario-based traces with additional information is a natural possible extension of the work. That is, we would like to handle a larger subset of the LSC language during trace generation and in the visualization and exploration phase, which includes, for example, values of variables and method parameters.

In [13], the syntax and semantics of the LSC language is extended to support the classical notion of object composition. Specifically, the extended language allows the specification and interpretation of scenario hierarchies – trees of scenarios – which are based on the object composition hierarchy in the underlying structural model. The work has been recently implemented as an optional extension to the S2A compiler. It would be interesting to consider the possible visualizations of scenario-based traces in the context of these hierarchies.

9.2 State-based traces

Going beyond scenario-based traces to other model-based traces, specifically *state-based* ones (see [43]), is another natural extension of our work. That is, while the main concern of the present 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 as well. We now sketch such an adaptation.

We propose to generate state-based execution traces, i.e., ones that include information about the states of (selected) objects during a run of the program. These can then be visualized using an appropriate variant of the Tracer, where the hierarchy reflects the object composition relation, and the horizontal bars represent the duration of being in the states of specific object instances, as they change over time. Moreover, if the object based variant of Statecharts [21] (the so-called UML state machines) is used to describe the intra-object behavior of the system (as in, e.g., Rhapsody [4]), 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 can take advantage of many of the techniques described here, such as handling multiplicities, details-on-demand (from a horizontal bar on the Gantt to a Statechart diagram where the current state is highlighted), event-based and time-based tracing etc.

9.3 Filtering and comparison techniques

Regarding filtering, our work on horizontal and vertical pre-defined calculated filters, reported on in subsection 5.2, calls for generalization and formalization. Specifically, we may consider defining *a query language for model-based traces*, which would allow us to formally express properties that can be used as constraints over model-based traces. The language should come with efficient means for computing its queries, so that it may be used as the formal basis for a generic navigation and filtering tool.

In the context of comparison techniques, we consider the extension of the trace comparison features reported on in subsection 5.3 with techniques based on best alignment algorithms. Specifically, it seems that one can adapt and apply algorithms inspired by the classical diff algorithm [50] or by variants of global and local sequence alignment algorithms (popular in bioinformatics [51,59]) to support various best alignment features between two or more model-based traces or parts thereof. Like our current comparison features, these can be exploited in the context of maintenance and version control tasks. Applying best alignment algorithms to model-based

traces in general seems interesting due to the multi-layer characteristics of the traces and the rich semantic information that is embedded in them. We leave this too for future work.

Additional comparison techniques may be developed to compare not traces but scenario instances. For example, one may be interested in a succinct summary of the different instances of the same scenario over a trace, specifically to find the differences between completed and violated ones. An example would be a question like: "what are the differences between the execution fragments that caused one instance to be completed while another instance to be violated?"

10 Conclusion

The main contribution of our work is in providing new techniques for model-based (specifically, scenario-based) visualization and exploration of reactive system execution traces. By considering traces not at the code level but at a higher abstract behavioral level, we are able to connect dynamic analysis with model-driven development. Additional contributions include the separate event-based and time-based tracing modes, as well as the combined event-normalized multi-scale visualization mode, the various semantics-based filters and model-based traces comparisons, and the vertical and horizontal metrics.

Our technique follows the classic *overview first, zoom and filter, details-on-demand* paradigm [58], and the concept of *semantic zooming* [54], 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. In addition, the event-normalized time-based view with the multi-scale presentation may be considered a special kind of automated semantic zooming: 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. Finally, the various horizontal and vertical semantics-based automated filters are additional examples of semantic zooming.

Moreover, our approach fits into an end-to-end visual framework for model-driven development, focusing on model-driven dynamic analysis tasks. First, the specification of the model is done using a diagrammatic language. Second, the code implementing the model-based trace generation through instrumentation is automatically generated from the visual specification model and is weaved into the program under investigation. Finally, the generated model-based execution traces are presented and investigated visually. Neither writing code nor browsing textual files are involved in the process. We consider this end-to-end visual characteristic of both model specification and model-based execution trace analysis to be an important feature of our work. It is designed to render the use of models in program analysis tasks more effective by making it more accessible, attractive, and usable to engineers.

Finally, our work shows the potential of using visualization and interaction techniques in support of model-based dynamic analysis tasks. We believe that the Tracer, or a similar tool based on the ideas of this paper, can be used effectively to improve the activities involved in the development of complex reactive systems, specifically in model-based testing and simulation.

Acknowledgements We are grateful to Peter Kliem for making his excellent jaret timebar component [8] available under GPL, for adding the variable scale feature, and for his most friendly and effective technical support. We thank Asaf Kleinbort for contributing to the implementation of the first version of the Tracer presented in [46]. We thank Evyatar Shoresh for implementing the second major version of the Tracer. Thanks are due also to Itai Segall for implementing the scenario-based trace export from the Play-Engine and for his help with generating traces from the C. elegans model. We thank Hillel Kugler for early discussions on visualizations of LSC specifications.

References

1. Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/
2. Eclipse UML2 Project. http://www.eclipse.org/modeling/mdt/?project=uml2
3. IBM Rational Software Architect. http://www.ibm.com/software/awdtools/architect/swarchitect/
4. IBM (Telelogic) Rhapsody. http://modeling.telelogic.com/products/rhapsody/index.cfm
5. PacMan game code. http://www.bennychow.com/
6. S2A website. http://www.wisdom.weizmann.ac.il/~maozs/s2a/
7. The AspectJ project at Eclipse.org. http://www.eclipse.org/aspectj/
8. The jaret timebars. http://jaret.de/timebars/
9. Tracer website. http://www.wisdom.weizmann.ac.il/~maozs/tracer/
10. Aigner, W., Miksch, S., Müller, W., Schumann, H., Tominski, C.: Visualizing Time-Oriented Data - A Systematic View. Computers & Graphics **31**(3), 401–409 (2007)
11. Alur, R., Henzinger, T.A.: A Really Temporal Logic. J. ACM **41**(1), 181–203 (1994)
12. Asarin, E., Maler, O., Pnueli, A.: Reachability Analysis of Dynamical Systems Having Piecewise-Constant Derivatives. Theor. Comput. Sci. **138**(1), 35–65 (1995)
13. Atir, Y., Harel, D., Kleinbort, A., Maoz, S.: Object Composition in Scenario-Based Programming. In: J.L. Fiadeiro, P. Inverardi (eds.) Proc. 11th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'08), *Lecture Notes in Computer Science*, vol. 4961, pp. 301–316. Springer (2008)
14. Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., van Wijk, J.J.: Execution Trace Analysis Through Massive Sequence and Circular Bundle Views. Journal of Systems and Software **81**(12), 2252–2268 (2008)

15. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. J. on Formal Methods in System Design **19**(1), 45–80 (2001)
16. Diehl, S.: Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software. Springer (2007)
17. Emerson, E.A.: Temporal and Modal Logic. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. Elsevier and MIT Press (1990)
18. Gallagher, M., Ryan, A.: Learning to Play Pac-Man: An Evolutionary, Rule-Based Approach. In: Proc. Congress on Evolutionary Computation (CEC), pp. 2462–2469 (2003)
19. Hamou-Lhadj, A., Lethbridge, T.C.: A survey of trace exploration tools and techniques. In: H. Lutfiyya, J. Singer, D.A. Stewart (eds.) Proc. 2004 Conf. of the Centre for Advanced Studies on Collaborative research (CASCON'04), pp. 42–55. IBM (2004)
20. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8**, 231–274 (1987)
21. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. IEEE Computer **30**(7), 31–42 (1997)
22. Harel, D., Kleinbort, A., Maoz, S.: S2A: A Compiler for Multi-Modal UML Sequence Diagrams. In: M.B. Dwyer, A. Lopes (eds.) Proc. 10th Int. Conf. Fundamental Approaches to Software Engineering (FASE'07), *Lecture Notes in Computer Science*, vol. 4422, pp. 121–124. Springer (2007)
23. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. Int. J. Found. Comput. Sci. **13**(1), 5–51 (2002)
24. Harel, D., Maoz, S.: Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. Software and Systems Modeling (SoSyM) **7**(2), 237–252 (2008)
25. Harel, D., Marelly, R.: Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In: Proc. 10th Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'02), pp. 193–202. IEEE Computer Society (2002)
26. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
27. Harel, D., Marelly, R.: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Software and Systems Modeling (SoSyM) **2**(2), 82–107 (2003)
28. Harel, D., Pnueli, A.: On the Development of Reactive Systems. In: K.R. Apt (ed.) Logics and Models of Concurrent Systems, *NATO ASI Series*, vol. F-13, pp. 477–498. Springer (1985)
29. Harel, D., Segall, I.: Visualizing Inter-Dependencies Between Scenarios. In: R. Koschke, C.D. Hundhausen, A. Telea (eds.) Proc. of the ACM 2008 Symposium on Software Visualization (SoftVis'08), pp. 145–153. ACM (2008)
30. Hochheiser, H., Shneiderman, B.: Dynamic Query Tools for Time Series Data Sets: Timebox Widgets for Interactive Exploration. Information Visualization **3**(1), 1–18 (2004)
31. Hosking, J.G.: Visualisation of Object Oriented Program Execution. In: Proc. 1996 IEEE Symp. on Visual Languages, pp. 190–191. IEEE Computer Society (1996)
32. ITU: International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Tech. rep. (1996)
33. Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing Interactions in Program Executions. In: Proc. 19th Int. Conf. on Software Engineering (ICSE'97), pp. 360–370. ACM Press (1997)
34. Kam, N., Harel, D., Kugler, H., Marelly, R., Pnueli, A., Hubbard, E.J.A., Stern, M.J.: Formal Modeling of C. elegans Development: A Scenario-Based Approach. In: C. Priami (ed.) Proc. 1st Int. Workshop on Computational Methods in Systems Biology (CMSB'03), *Lecture Notes in Computer Science*, vol. 2602, pp. 4–20. Springer (2003)
35. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: J.L. Knudsen (ed.) Proc. 15th European Conf. on Object-Oriented Programming (ECOOP'01), *Lecture Notes in Computer Science*, vol. 2072, pp. 327–353. Springer (2001)
36. Klose, J., Toben, T., Westphal, B., Wittke, H.: Check It Out: On the Efficient Formal Verification of Live Sequence Charts. In: T. Ball, R.B. Jones (eds.) Proc. 18th Int. Conf. on Computer Aided Verification (CAV'06), *Lecture Notes in Computer Science*, vol. 4144, pp. 219–233. Springer (2006)
37. Koza, J.: Genetic Programming: On The Programming of Computers by Means of Natural Selection. MIT Press (1992)
38. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-Based Specifications. In: N. Halbwachs, L.D. Zuck (eds.) Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), *Lecture Notes in Computer Science*, vol. 3440, pp. 445–460. Springer (2005)
39. Lange, D.B., Nakamura, Y.: Object-Oriented Program Tracing and Visualization. IEEE Computer **30**(5), 63–70 (1997)
40. Lettrari, M., Klose, J.: Scenario-Based Monitoring and Testing of Real-Time UML Models. In: M. Gogolla, C. Kobryn (eds.) Proc. 4th Int. Conf. on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, *Lecture Notes in Computer Science*, vol. 2185, pp. 317–328. Springer (2001)
41. Lin, J., Keogh, E.J., Lonardi, S.: Visualizing and Discovering Non-Trivial Patterns in Large Time Series Databases. Information Visualization **4**(2), 61–82 (2005)
42. Lo, D., Maoz, S.: Mining Scenario-Based Triggers and Effects. In: Proc. 23rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2008), pp. 109–118. IEEE (2008)
43. Maoz, S.: Model-Based Traces. In: M. Chaudron (ed.) Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 5421, pp. 109–119. Springer (2009). (Presented at the 3rd Int. Workshop on Models at Runtime (Models@Run.time 2008), at MoDELS'08.)
44. Maoz, S.: Polymorphic Scenario-Based Specification Models: Semantics and Applications. In: A. Schürr, B. Selic (eds.) Proc. 12th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'09), *Lecture Notes in Computer Science*, vol. 5795, pp. 499–513. Springer (2009)
45. Maoz, S., Harel, D.: From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In: M. Young, P.T. Devanbu (eds.) Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE'06), pp. 219–230. ACM Press (2006)
46. Maoz, S., Kleinbort, A., Harel, D.: Towards Trace Visualization and Exploration for Reactive Systems. In: P. Cox, J. Hosking (eds.) Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC'07), pp. 153–156. IEEE Computer Society (2007)
47. Maoz, S., Metsä, J., Katara, M.: Model-Based Testing Using LSCs and S2A. In: A. Schürr, B. Selic (eds.) Proc. 12th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'09), *Lecture Notes in Computer Science*, vol. 5795, pp. 301–306. Springer (2009)

48. Marelly, R., Harel, D., Kugler, H.: Multiple Instances and Symbolic Variables in Executable Sequence Charts. In: Proc. Int. Conf. on Object-Oriented Programming, Languages, and Applications (OOPSLA'02), pp. 83–100. ACM (2002)
49. McGavin, M., Wright, T., Marshall, S.: Visualisations of Execution Traces (VET): An Interactive Plugin-Based Visualisation Tool. In: W. Piekarski (ed.) Proc. 7th Australasian User Interface Conf. (AUIC'06), *CRPIT*, vol. 50, pp. 153–160. Australian Computer Society (2006)
50. Myers, E.W.: An O(ND) Difference Algorithm and Its Variations. Algorithmica **1**(2), 251–266 (1986)
51. Needlemana, S.B., Wunscha, C.D.: A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology **48**(3), 443–453 (1970)
52. Orso, A., Jones, J.A., Harrold, M.J.: Visualization of Program-Execution Data for Deployed Software. In: S. Diehl, J.T. Stasko, S.N. Spencer (eds.) Proc. ACM Symp. on Software Visualization (SoftVis'03), pp. 67–76. ACM (2003)
53. Pauw, W.D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J.M., Yang, J.: Visualizing the Execution of Java Programs. In: S. Diehl (ed.) Revised Lect. on Software Visualization, Int. Seminar, *Lecture Notes in Computer Science*, vol. 2269, pp. 151–162. Springer (2002)
54. Perlin, K., Fox, D.: Pad: An Alternative Approach to the Computer Interface. In: Proc. 20th Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH'93), pp. 57–64. ACM Press (1993)
55. Reiss, S.P.: Visualizing program execution using user abstractions. In: E. Kraemer, M.M. Burnett, S. Diehl (eds.) Proc. 2006 ACM Symp. on Software Visualization (SoftVis'06), pp. 125–134. ACM Press (2006)
56. Reiss, S.P.: Visual Representations of Executing Programs. Journal of Visual Languages and Computing **18**(2), 126 – 148 (2007)
57. Reiss, S.P., Renieris, M.: Jove: Java as it Happens. In: T.L. Naps, W.D. Pauw (eds.) Proc. 2005 ACM Symp. on Software Visualization (SoftVis'05), pp. 115–124. ACM Press (2005)
58. Shneiderman, B.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: Proc. IEEE Symp. on Visual Languages (VL'96), pp. 336–343. IEEE Computer Society (1996)
59. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. Journal of Molecular Biology **147**(1), 195–197 (1981)
60. Stasko, J.T., Domingue, J.B., Brown, M.H., Price, B.A. (eds.): Software Visualization. MIT Press (1998)
61. van Wijk, J.J., van Selow, E.R.: Cluster and Calendar Based Visualization of Time Series Data. In: Proc. 1999 IEEE Symp. on Information Visualization (INFOVIS'99), pp. 4–9. IEEE Computer Society (1999)