USING MODEL-BASED TRACES AS RUNTIME MODELS

Shahar Maoz
Weizmann Institute of Science

Software engineers typically use code-level tracing to capture a running system's behavior. An alternative is to generate and analyze model-based traces, which contain rich semantic information about the system's runs at the abstraction level that its design models define. A set of metrics and operators can aid such trace analysis.

Model-based traces record a system's runtime through abstractions provided by the models used in its design. These traces include rich semantic information about the systems from which they were generated, the models that induced them, and the relationships between the two as they unfold at runtime. Using model-based traces, an engineer can follow and monitor system design models, such as sequence diagrams, class diagrams, and statecharts, as they come to life during the system's execution.

Metrics and operators for model-based traces can serve as tools for gaining insights into the running system's structure and behavior at the abstraction level its design models define. In essence, as the "How Can a Trace Be a Runtime Model?" sidebar describes, a model-based trace (and traces in general) can be viewed as a special type of runtime model. Thus, the analysis of these traces, as models of the systems from which they were generated, supports tasks related to model-based testing, comprehension, and evolution.

UNDERSTANDING MODEL-BASED TRACES

An important characteristic of model-based traces is that they are not mere projections of concrete runtime information onto some limited domain. Rather, they are stateful abstractions, in which trace entries depend on the history and context of the run and the model. The model-based trace not only filters irrelevant information but also adds model-specific information, such as data about entering and exiting states that does not appear explicitly in the program code. This level of model-based reflection provides a unique visibility into a system's runtime and enables model-based dynamic analysis.

Although simulation or code-generation schemes might exist for the execution of the models described in this article, model-based tracing is not limited to programs for which code was automatically generated from models. On the contrary, one of the strengths of model-based traces is that they apply to systems in general, not only to systems in which the implementation explicitly reflects certain high-level models.

Properties

Model-based traces have several important properties:[1]

 Published by the IEEE Computer Society

- *Engineers can define and generate model-based traces on the basis of partial models.* These can be some properties of a selected class or a state machine consisting of some abstract states. The model defines the traces' abstraction level. System run elements not represented in the model will not be part of the trace.
- *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 and implementation of the program under investigation.
- *The same concrete execution can result in different model-based traces, depending on the models used for tracing.* Conversely, different executions can result in identical model-based traces if the runs are equivalent from the more abstract point of view that the models used for tracing define.

Using a system's execution traces for different analyses requires the definition of an abstraction level. The abstraction level specifies the elements that the trace records, such as CPU register assignments, virtual machine commands, or statements at the code level. Unlike more concrete code-level traces, model-based traces use a higher abstraction level, typically defined by models used for high-level system design.

Definition and generation

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

Given a program P and a model M, a model-based execution trace records a run r of P at the abstraction level that M induces. To enable tracing, a unification mechanism maps concrete elements of the run to elements in the model. The trace entries can belong to several entry types. The model type used and the artifacts and their semantics define the types of entries that appear in the model-based trace.

In this article, the focus is on the *scenario-based* trace[1] as a runtime model and on the metrics and operators to analyze it. The "Property-Set Model-Based Traces" sidebar describes another type of model-based trace—the *property-set* trace. These two types are only examples; engineers can create additional types of model-based traces by combining variants of these types or by using other modeling techniques.[1]

Depending on the type of model-based trace and execution environment, it may be possible to automatically generate a model-based trace from a running system. This is important because, however theoretically interesting and semantically rich, model-based traces must also be easy to create or engineers will not find them effective. Previous work[1] outlines a prototype application that addresses automatic model-based trace generation by compiling models into aspect code; the automatically generated aspect code is weaved into the system code to create model-based monitors.[2,3] When the instrumented system executes, it can generate a model-based trace.

HOW CAN A TRACE BE A RUNTIME MODEL?

Following the classification of models into development and runtime models,[1] a system's execution trace can be viewed as a runtime model of the system from which it was generated. A trace records a system's execution at some level of detail; hence, it can be viewed as an abstract representation—an artifact that models some aspects of its system. Like any other system model, an execution trace represents information about the system from a certain viewpoint and omits (or abstracts away) other information.

As a model, a trace should be formally described using a modeling language with well-defined syntax and semantics. Given a syntax, an engineer can check if the trace is well formed, or syntactically correct. The semantics of a trace can be defined at two levels: in terms of its representation of the concrete run from which it was generated and, more generally, as consisting of the set of systems (and their sets of runs) from which the trace could have been generated. Thus, given a semantics, an engineer can check if a syntactically correct trace is consistent with regard to a concrete run (is the representation correct?), and, more generally, if it is realizable—that is, if a system (and a run) exists from which it could have been generated.

The trace is subject to investigation and exploration using various metrics and operators. It may undergo various transformations, have textual and visual representations, and be compared to other traces. Overall, its features as a model provide evidence of and insights into the structure and behavior of the system it models—insights that can support engineering tasks related to systems testing, comprehension, and evolution.

Reference

1. **R.B. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering* (FOSE 07), *Proc. Int'l Conf. Software Eng.* (ICSE 07), L.C. Briand and A.L. Wolf, eds., IEEE CS Press, 2007, pp. 37-54.**

MODELING AND TRACING A SAMPLE SYSTEM

A simple e-learning application, Well, serves here as a sample system for modeling and tracing, and as context for demonstrating the idea of using model-based traces as a runtime model. Well lets students follow structured tutorials, practice, or take exams. A teacher can monitor each student's activity in real time, take over control of selected student stations to help in problem solving, or broadcast class material to all students.

Scenario-based models

The first step in defining a model-based trace is to choose a modeling language. For the Well sample application, the choice was a UML2-compliant variant of Werner Damm and David Harel's live sequence charts (LSCs),[4,5] an

PROPERTY-SET MODEL-BASED TRACES

Property-set traces record value changes in selected class properties and in expressions that depend on their values (derived properties). Thus, the model inducing a property-set trace consists of a class diagram showing only selected classes of the system under investigation, selected properties of these classes, and some derived properties that the engineer defines. Derived properties might depend on several properties coming from different classes, so they could relate to more than one class. Engineers can use expressions in the Object Constraint Language to define the properties for tracing. An entry is added to the trace whenever a property (or a derived property) specified in the model changes its value during execution.

To illustrate, consider a property-set model-based trace induced by a model consisting of the class `Student` in the Well application example, with its `numOfCompletedExams` and `numOfCompletedTutorials` properties, and with a derived Boolean expression `ALL-IN-T-OR-E`, which states, "for all students, `currentTutorialID` is not null or `currentExamID` is not null." This model is intentionally partial; it does not describe the complete Well application or even all properties of the class `Student`. Moreover, the derived property `ALL-IN-T-OR-E` does not appear explicitly in the Well application code; it is only part of the viewpoint the model defines.

Figure A shows a snippet from a property-set trace of Well induced by this model.

To analyze property-set traces, an engineer can define vertical and horizontal filters, similar to those described for scenario-based traces in the main text. These would abstract away selected time intervals or hide entries related to selected model properties, according to the needs of the engineering task at hand. Metrics and comparators would be useful too. For example, a horizontal metric can count how many times a model property has changed its value or how many unique values it had during some execution interval. Comparators can compare these metrics' values in different traces, find matching property values between traces, and so on. Finally, it is possible to check if a property-set model-based trace satisfies certain invariants. For example, for the trace in Figure A, an engineer could check that `numOfCompletedExams` for each student never decreased during execution.

Property-set model-based traces are much simpler than the scenario-based traces described in the main text because they are nearly stateless (they are not completely stateless, since assignments that do not change a property's value are not reported). They induce a rather strong abstraction: The trace progresses only when one of the properties specified in the model changes its value.

Engineers might be able to create property-set model-based traces using automatically generated aspect-oriented code. Work in the context of scenario-based modeling, and more generally, in runtime verification using program instrumentation, shows the feasibility of automatically deriving the code responsible for this kind of model-based trace generation from the models used for trace definition. Similar technology can be developed for the property-set traces presented here.

```
...
P: 6534711 39: ALL-IN-T-OR-E Value=true
P: 8099789 40: numOfCompletedExams well.Student@8088348 Value=7
P: 8249842 41: numOfCompletedExams well.Student@8078122 Value=4
P: 11101610 42: numOfCompletedExams well.Student@8088348 Value=8
P: 11801625 43: numOfCompletedTutorials well.Student@9176521 Value=2
P: 11801625 44: ALL-IN-T-OR-E Value=false
P: 18597008 45: numOfCompletedExams well.Student@6114366 Value=3
P: 23417266 46: numOfCompletedExams well.Student@6114562 Value=3
...
```

Figure A. Part of a property-set trace induced by a model for the Well e-learning application.

expressive interobject scenario-based specification language. LSCs extend the classical partial-order semantics of sequence diagrams in general with a universal interpretation and must/may (hot/cold) modalities, which means that engineers can use it to specify scenario-based liveness and safety properties: what could happen, what must happen, and what should never happen. An LSC specification model typically consists of many charts, possibly interdependent, divided among several use cases. The sample model of Well, which is relatively small, has 21 scenarios divided among five use cases.

Figure 1 shows an instance of an LSC from the sample model of Well. Vertical lines, or *lifelines*, represent specific system objects, and time goes from top to bottom. Roughly, the scenario in the figure specifies that, whenever a teacher calls a student's `reqTakeover()` method and the student's `isInExam()` method evaluates to `FALSE` (the condition `!st.isInExam()` evaluates to `TRUE`), the student must tell the database to save its current state, the student must set its own controlled property using the `setControlled()` method, and the teacher must eventually take over the student's station.

An important concept in LSC semantics is the *cut*, a

mapping from each lifeline to one of its locations, which represents the scenario's state during execution. The cut (2,2,0) in Figure 1, for example, comes immediately after the evaluation of the student's `!st.isInExam()` cold condition. A cut induces a set of *enabled events*—those immediately after it in the partial order that the chart defines. A cut is hot if any of its enabled events is hot and is cold otherwise.

The occurrence of a chart's minimal event activates a new instance of it. An occurrence of an enabled method or true evaluation of an enabled condition causes the cut to progress. An occurrence of a nonenabled method from the chart or a false evaluation of an enabled condition when the cut is cold is a *completion*, which causes the chart's instance to close gracefully. An occurrence of a nonenabled method from the chart or a false evaluation of an enabled condition when the cut is hot is a *violation* and should never happen if the implementation is faithful to the specification model.

Thus, a chart restricts the order of occurrence of events mentioned in it. However, it does not restrict the occurrence or nonoccurrence of events not explicitly mentioned in it, including any events between the occurrences of mentioned events.

Finally, in the chosen semantics of LSC, scenarios use a polymorphic interpretation for symbolic lifelines.[6] In the Takeover scenario in Figure 1, for example, the lifeline `st:Student`, which represents a student, can bind at runtime to any instance of the class `Student` or its subclasses.

Scenario-based traces

Given a scenario-based specification comprising a set of LSCs, a *scenario-based trace* includes the activation and progress information of the scenarios, relative to a given program run. A trace can be viewed as a projection of the full execution data onto the set of methods in the specification, plus the activation, binding, and cut-state progress information of all the instances of all the charts in the set, including concurrently active multiple copies of the same chart. Thus, a scenario-based trace is made of several entry types: event occurrence, binding, cut-state change, and finalization.

As its name implies, an *event occurrence* represents an occurrence of an event. Event occurrences are time-stamped and numbered. The trace records only the events that explicitly appear in the model. An engineer can add details of interest, such as identifiers of participating objects (caller and callee) and parameter values. The entry format is

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

Binding represents the binding of a lifeline in one of the active scenario instances to an object. The entry format is

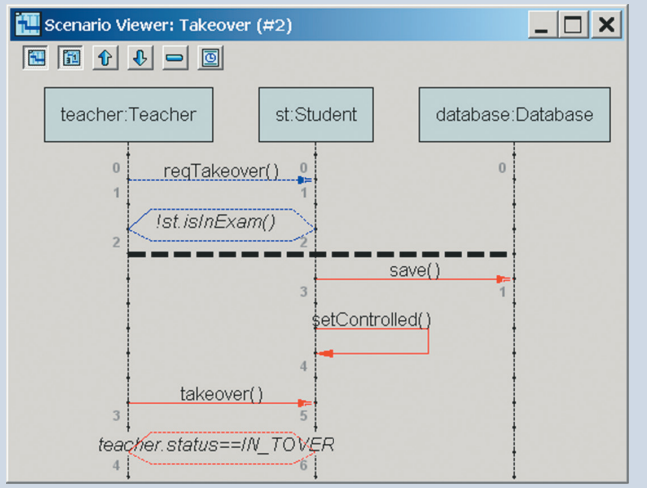

Figure 1. Live sequence chart depicting an active instance of the Takeover scenario from the sample model of Well. Vertical lines, or lifelines, represent participating objects: a teacher, a student, and a database. Horizontal lines represent method calls, and hexagons represent conditions. Red lines denote must (hot) elements; blue lines denote may (cold) elements. Numbers along lifelines denote locations. The LSC instance displays a cut, which is a mapping from each lifeline to one of its locations, representing the scenario's state during execution. In this case, the cut of the active instance shown is at (2,2,0), immediately after the evaluation of the student's `!st.isInExam()` cold condition.

```
B: <scenario name> [instance no.]
lifeline <no.> <- <object identifier>
```

Cut-state change represents a change of cut-state in one of the active scenario instances. The entry format is

```
C: <scenario name> [instance no.]
<cut tuple> [Hot|Cold]
```

Finalization represents a successful completion or a violation in an active scenario instance. The entry format is

```
F: <scenario name> [instance no.]
[Completion|Violation]
```

Figure 2 shows a snippet from a scenario-based trace of Well.

ANALYZING MODEL-BASED TRACES

The analysis of model-based traces is based on metrics and operators that provide insights into the structure and behavior of the systems from which these traces were generated. Although the metrics and operators described here are for analyzing a scenario-based trace, similar ones are useful in analyzing other types of model-based traces.

```
...
E: 2133609971871 305: void well.Student.startExam()
B: StartExam[4] lifeline 1 <- well.Student@890445
B: StartExam[4] lifeline 2 <- well.exams.ExamsController@945022
C: StartExam[4] (1,1,0,0) Cold
E: 2133609971871 306: void well.exams.ExamsDB.init()
B: StartExam[4] lifeline 3 <- well.exams.ExamsDB@704551
C: StartExam[4] (1,2,1,0) Hot
E: 2133609971871 307: void well.exams.ExamsController.init()
B: ExamInProgress[4] lifeline 1 <- well.Student@890445
B: ExamInProgress[4] lifeline 2 <- well.exams.ExamsController@945022
C: ExamInProgress[4] (1,1,0) Cold
C: StartExam[4] (2,2,2,0) Hot
E: 2133609971872 308: void well.Student.reqTakeover()
B: Takeover[1] lifeline 0 <- well.Teacher@984112
B: Takeover[1] lifeline 1 <- well.Student@709859
C: Takeover[1] (1,1,0) Cold
C: Takeover[1] (2,2,0) Hot
E: 2133609971873 309: void well.Student.endPractice()
F: FreePractice[6] Violation
E: 2133609971873 310: void well.datamodel.Database.save()
B: Takeover[1] lifeline 2 <- well.datamodel.Database@900426
C: Takeover[1] (2,3,1) Hot
...
```

Figure 2. Part of a textual representation of a scenario-based trace of Well. The trace includes event occurrences, bindings, cut-state changes, and finalizations.

Metrics

Model-based trace analysis takes advantage of vertical and horizontal metrics. *Vertical* metrics relate to a snapshot of an execution. *Horizontal* metrics relate to an interval of an execution. Together they provide a way to formally and quantitatively characterize and investigate the relationships between the model used for tracing, the concrete execution the trace was generated from, and—most important—the running system under investigation.

Vertical metrics. Vertical metrics quantitatively reflect the features of the trace at certain time points in the execution; their value depends on model-level information available at execution snapshots. One example of a vertical metric is *scenario bandwidth*, which measures the number of sequence diagrams open at a given point in the execution—that is, how many stories are simultaneously in progress. The scenario bandwidth metric is useful in program comprehension and in analyzing resource allocations in the running system.

Another example of a vertical metric is the *number of affected scenarios*, which is the number of scenario instances that the most recent event has affected. This metric can help identify significant events—those that have caused a state change in many scenarios.

Finally, *duration* between events measured in some real-time unit is an important metric because it can help identify performance-related issues.

Horizontal metrics. Horizontal metrics are evaluated over a time interval, typically a complete trace or a trace fragment. Their definition can be extended from a single model-based trace to a set of model-based traces. A horizontal metric can be any aggregation function over a

vertical metric—maximum, minimum, average, median, and so on—for values spanning a given time interval. For example, an engineer might be interested in computing the maximum value of the scenario bandwidth metric over a selected trace segment.

Another horizontal metric is *coverage*, which characterizes various fragments of the model that were met during a given system execution. An engineer can define coverage in terms of a ratio or by using some concrete or symbolic representation. For example, given a scenario-based trace, an engineer might be interested in measuring how many unique states from the model's states (global cut states) have been visited, or how many unique events from the events specified in the scenarios (the model's alphabet) have occurred. These may be used to quantitatively evaluate the quality of the information embedded in a system's run relative to a scenario-based specification.

The *number of completions and violations* in a trace segment could also serve as a horizontal metric and would be particularly relevant in model-based testing.

Operators

Engineers can also define various operators over model-based traces, which can be characterized as filters or comparators.

Filters. As with metrics, the two main filter types are vertical and horizontal. *Vertical* filters hide, or abstract away, selected trace segments, thus hiding certain time intervals. In contrast, *horizontal* filters abstract away selected model elements, deleting all the trace entries that reflect the state or progress of these elements over an entire trace.

Filters differ according to filtering criteria, which can be predefined (fixed) or calculated (derived). All filtering criteria are based on the specific features of the trace type. Following the view of traces as models, a filter is essentially a special kind of a model transformation.

An example of a vertical filter is a filter that abstracts away all trace segments in which a certain scenario was not active. Thus, the trace reduces to the concatenation of the segments during which the selected scenario had at least one active instance. Such a vertical filter might be useful for tasks that are local to a selected scenario or set of scenarios.

An example of a horizontal filter is a filter that hides all nonviolated scenario instances. Thus, the resulting trace includes only the violated instances. Such a filter is of interest in testing related tasks.

The application of filters to model-based traces provides an abstraction mechanism with two major advantages. First, model-based traces tend to be very long and complex, so some abstraction mechanism is needed to hide the segments or aspects of the trace that are irrelevant to the engineering task at hand. Second, the filters' abstraction mechanism lets an engineer conduct what-if investigations by calculating metrics for the original and filtered traces and then comparing the two.

Comparators. Comparators compare two or more model-based traces generated using the same models either from different runs of the same system or from runs of different system versions. Two versions of the same program, traced using identical models and equal input or environment stimuli, could result in identical model-based traces, if the models used for tracing cannot distinguish between the two versions (that is, if the runs were equivalent from the abstract view that the tracing models define). If they do distinguish between the two versions, a comparison of the two traces makes the version difference explicit and presents it at the abstraction level of the models used for tracing. This is useful in the context of tasks related to software evolution and regression testing.

The number of completions and violations in a trace segment would be particularly relevant in model-based testing.

A Boolean comparison tells only whether or not two model-based traces are identical. However, model-based trace analysis can use comparators that provide much richer comparisons, which cover various similarities and differences between the traces. Two examples illustrate the insights possible in using comparators:

- Given two model-based traces and a point in time in one of them, what are the points in time (if any) in the other trace where the two traces arrive at identical global states?
- Given two model-based traces with corresponding points in time where the two traces represent identical global states, what is the nearest forward point in time where their global states differ?

The second comparator is useful with or without considering real-time durations between events. In many cases, only the event order would be relevant; thus an engineer could perform this comparison while abstracting away the possible differences in real-time durations between the two traces. In any case, the comparison is conducted at the abstraction level of the models used for tracing.

A TRACE ANALYSIS TOOL

The Tracer,[7] developed at the Weizmann Institute of Science, is a prototype tool for analyzing scenario-based traces using visualization and interactive exploration techniques (www.wisdom.weizmann.ac.il/~maozs/tracer). Its input includes a scenario-based model of a system,

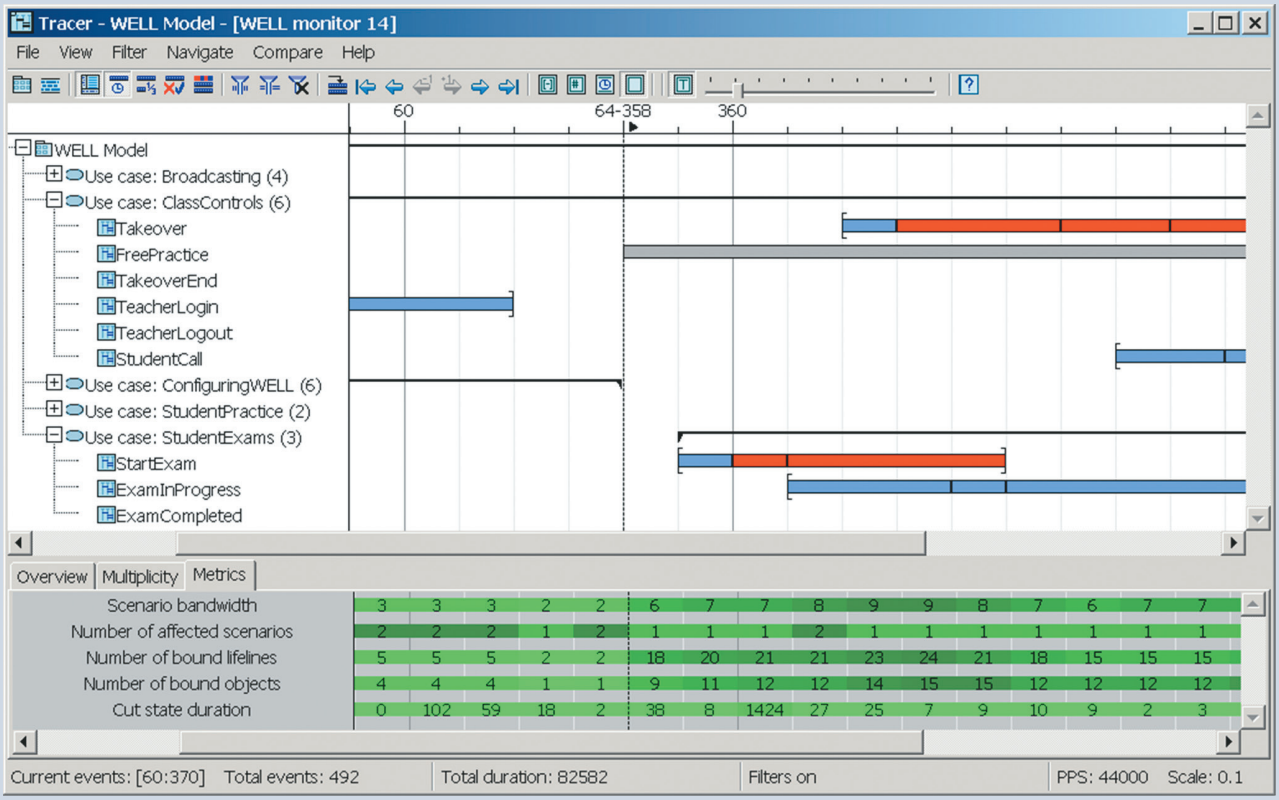

Figure 3. The main view in the Tracer, a tool for analyzing, visualizing, and exploring scenario-based traces. The scale at the top shows event numbers; the Tracer has abstracted away real time in this view, displaying only the order of event occurrences in the trace. The hierarchy at left comprises the list of sequence diagrams in the model, divided among use cases. Each horizontal row is a placeholder for specific active instances of the corresponding scenario, with blue and red bars showing the durations spent in specific cold and hot relevant cuts and gray bars depicting durations with simultaneously active scenario instances. The bottom pane shows the vertical metrics that the Tracer has calculated. A user-defined vertical filter hides a trace segment, from the 64th to the 358th event.

given as a set of UML2-compliant LSCs, and a scenario-based trace, representing a system execution. The Tracer implements some of the metrics and operators described here.

Figure 3 displays the scenario-based model of Well and a trace similar to the trace in Figure 2. This main view of the Tracer is based on an extended hierarchical Gantt chart, where time goes from left to right and a two-level hierarchy is defined by the containment relation of use cases and sequence diagrams in the model. Each leaf in the hierarchy represents a sequence diagram: The horizontal rows are placeholders for specific active instances of the corresponding scenario.

By looking horizontally, an engineer can follow the progress of specific scenario instances over time, identify events that caused progress, and locate completions and violations. By looking vertically, an engineer can see exactly what goes on at the model's abstraction level at any given point in time.

Supporting metrics and filters

The Tracer calculates several vertical metrics, which it displays in a Metrics pane in color gradients, normalized per metric. The Metrics pane uses color gradients, since a metric's relative qualitative value (high, medium, low) often is of more interest than its precise value.

Although not shown in the figure, the Tracer also supports the calculation and application of some horizontal metrics and filters.

Supporting trace comparison

Figure 4 shows a comparison of two Well traces, representing two different executions. The yellow highlighting identifies fragments, one in each trace, that the Tracer found to be equivalent. The Tracer cannot extend equivalence further to the right, since the instance of the Takeover scenario (top of left gray bar) starting on the left trace at location 362 does not have a counterpart in the other trace.

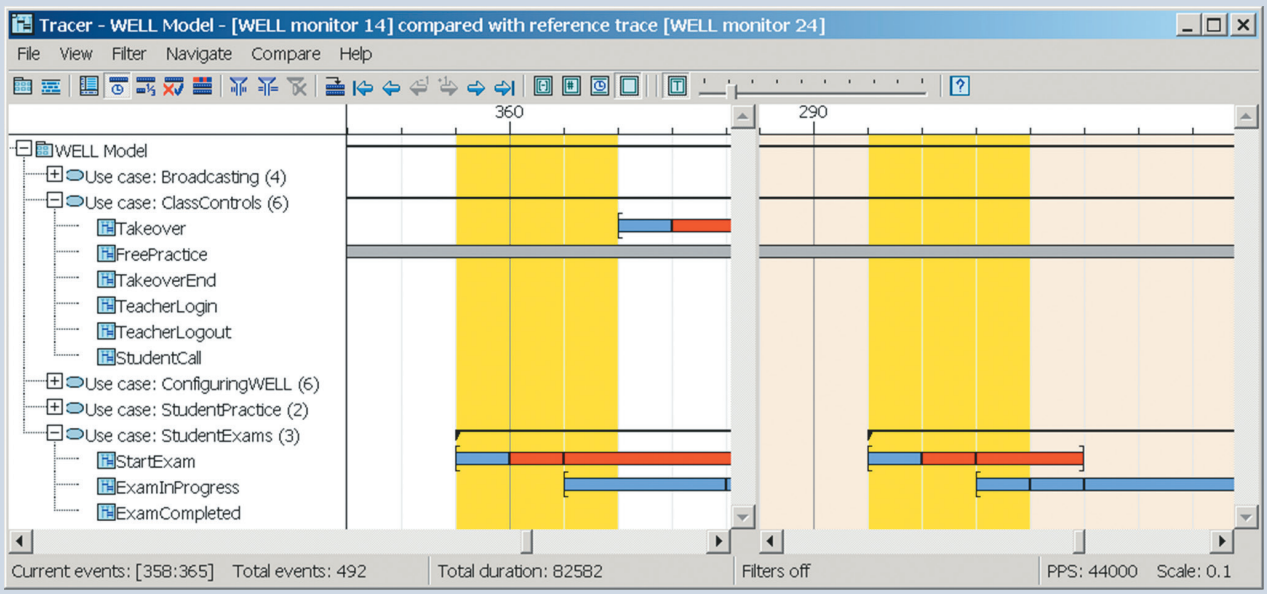

Figure 4. Using the Tracer to compare two scenario-based traces of the Well application. Yellow highlights show trace fragments that the Tracer found to be equivalent between the two executions.

MOVING AHEAD

The presentation of model-based traces as runtime models and the analysis methods described suggest challenging directions for future work.

The first is to formally define the syntax and semantics for various types of model-based traces. Once trace-modeling languages are formally defined, it might be possible to develop a generic mechanism to automatically derive a tracing-code-generation scheme for each type of model-based trace.

Another direction is to define additional operators for model-based traces: for example, comparators that take advantage of alignment algorithms to align similar trace segments, identify repeated patterns, and so on. A query language for model-based traces might allow the formal expression of properties that can be evaluated over such models.

Further work is also required to define methodologies that will instruct engineers in using model-based trace analysis methods and their implementation in tools like the Tracer within a model-based software development life cycle. Performing case studies with real-world projects would aid in evaluating alternative applications.

Finally, an important use of runtime models is to support (self)-adaptive systems.[8,9] Because model-based traces are generated during a system's runtime, in principle, part of their analysis could potentially be performed in real time. The results of such an analysis could provide feedback stimuli to the running system, and thus aid in dynamically adapting its behavior.

Many have suggested trace generation, analysis, and visual exploration techniques,[10] but most consider code-level concrete traces or attempt to extract models from such traces. In contrast, model-based traces use an abstraction provided by user-defined models. The analysis of model-based traces using various metrics and operators gives insights into the system's execution at the abstraction level defined by the models used for tracing. Hence, the traces enable a model-based analysis of running programs.

Viewing traces as runtime models appears to be novel. Although much work remains to make model-based trace generation and analysis practical, the benefits of deeply understanding a running program using models of its design are compelling reasons to push forward.

Acknowledgments

I thank David Harel, Assaf Marron, Michal Gordon, the volunteer editors of this special issue of *Computer*, and the anonymous referees for their helpful comments on early drafts of this article. Thanks also to Asaf Kleinbort and Evyatar Shoresh for contributing to the implementation of the Tracer and to Peter Kliem for the open source software used to implement the Gantt charts in the Tracer (www.jaret.de/timebars).

The research described in this article was supported 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 under the European Community's 7th Framework Programme (FP7/2007-2013).

References

1. S. Maoz, "Model-Based Traces," *Proc. Models in Software Eng. Workshops and Symposia at MoDELS 2008*, M.R.V. Chaudron, ed., LNCS 5421, Springer, 2008, pp. 109-119.
2. D. Harel, A. Kleinbort, and S. Maoz, "S2A: A Compiler for Multi-Modal UML Sequence Diagrams," *Proc. 10th Int'l Conf. Fundamental Approaches to Software Eng.* (FASE 07), M.B. Dwyer and A. Lopes, eds., LNCS 4422, Springer, 2007, pp. 121-124.
3. S. Maoz and D. Harel, "From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ," *Proc. 14th Int'l ACM/SIGSOFT Symp. Foundations of Software Eng.* (FSE 06), M. Young and P.T. Devanbu, eds., ACM Press, 2006, pp. 219-230.
4. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *J. Formal Methods in System Design*, vol. 19, no. 1, 2001, pp. 45-80.
5. D. Harel and S. Maoz, "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams," *Software and Systems Modeling*, vol. 7, no. 2, 2008, pp. 237-252.
6. S. Maoz, "Polymorphic Scenario-Based Specification Models: Semantics and Applications," *Proc. 12th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 09), A. Schürr and B. Selic, eds., LNCS 5795, Springer, 2009, pp. 499-513.
7. S. Maoz, A. Kleinbort, and D. Harel, "Towards Trace Visualization and Exploration for Reactive Systems," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing* (VL/HCC 07), IEEE CS Press, 2007, pp. 153-156.
8. N. Bencomo et al., "Third International Workshop on Models@runtime," *Proc. Models in Software Eng. Workshops and Symposia at MoDELS 2008*, M.R.V. Chaudron, ed., LNCS 5421, Springer, 2008, pp. 90-96.
9. B.H.C. Cheng et al., eds., *Software Engineering for Self-Adaptive Systems*, LNCS 5525, Springer, 2009.
10. A. Hamou-Lhadj and T.C. Lethbridge. "A Survey of Trace Exploration Tools and Techniques," *Proc. Conf. Centre for Advanced Studies on Collaborative Reseach* (CASCON 04), H. Lutfiyya, J. Singer, and D.A. Stewart eds., IBM, 2004, pp. 42-55.

Shahar Maoz *is a postdoctoral research fellow in the Department of Computer Science and Applied Mathematics at the Weizmann Institute of Science. His research interests include software and systems modeling, static and dynamic analysis, and aspect-oriented software development. Maoz received a PhD in computer science from the Weizmann Institute of Science. Contact him at shahar.maoz@weizmann.ac.il.*