Temporal Model for Debugging and Visualizations

Demian Lessa
CSE Department
SUNY at Buffalo
dlessa@buffalo.edu

Bharat Jayaraman
CSE Department
SUNY at Buffalo
bharat@buffalo.edu

This paper discusses the benefits of a temporal data model for debugging and run-time visualization of object-oriented software. Current debugging models generally only provide access to the current program state and support manual exploration of the state to uncover the cause of program errors. However, often the cause of an error lies in distant previous states, and a more comprehensive view of the execution history is necessary to uncover such errors. This is achieved through our proposed temporal data model and query language. This paper also shows the benefits of UML-like object and sequence diagrams for representing respectively the current run-time state and execution history. The main contribution of this paper lies in showing that declarative temporal queries and (run-time) object/sequence diagrams work in a symbiotic manner to achieve a more effective debugging system: queries help the user to focus on specific regions of the diagrams, while the diagrams provide a framework for reporting the answers to queries. Since run-time visualizations become unwieldy for large executions, we propose two broad classes of techniques to achieve compact sequence diagrams: folding operations, for nested calls and also a sequence of calls; and filtering operations, to remove unnecessary or irrelevant calls relative to a debugging task. We introduce a refinement of the sequence diagram to account for missing calls, and regular-expression labels for compacted execution sequences. Together, these techniques have proven to be effective and they form part of JIVE, a state-of-the-art debugging system for Java.

Keywords

object-oriented programming, temporal data model, query-based debugging, object and sequence diagrams, folding and filtering

1. INTRODUCTION

Object-oriented methodology has become widespread in the software industry due to the practical impact of languages such as C++ and Java. Comprehending the structure of object-oriented systems via source code, design time diagrams, or other static analysis tools is significantly easier than understanding their dynamic behavior [3]. This gap is due in large measure to the nature of the object-oriented methodology, which promotes the definition of smaller methods and more complex interactions among them. It also encourages the use of dynamic dispatching and inversion of control patterns, making flow of control very hard to follow through an inspection of the source code. Coupled with more advanced features such as multi-threading– a feature that has become more pervasive with the advent of multi-core architectures– diagnosing and debugging errors in object-oriented software remain a challenging problem.

The state of the art in run-time environments for object-oriented programs is exemplified by IDEs such as Eclipse, NetBeans, and Visual Studio. Typical features found in such systems include setting of breakpoints, spying on variables, stepping forward in execution, and examining variables on the call stack. Still, the programmer must proceed step-by-step and object-by-object to uncover the cause of an error. While these IDEs also provide graphical interfaces, they serve mainly as front-ends for traditional text-based debugging. Such debuggers may be categorized as procedural and textual in nature. In contrast, the main contribution of our research is in providing a declarative and visual environment for program comprehension and debugging.

In order to identify the cause of an error, software developers today only have access to the *current state* of the program, i.e., stack and heap objects. However, the cause of an error often lies in an already completed method call. This problem could be solved in part if debuggers could record the *execution history* of a program, i.e., the entire sequence of method calls, variable assignments, etc. It would then be possible to answer questions such as:

1. When (at which instants in execution) did variable v change value?
2. Was there a *concurrent* update of any variable in the program?
3. Was the object o_2 *ever* reachable from object o_1?

These questions express *temporal* properties that are hard, if not impossible, to answer using traditional debugging tools. Stepping and jumping through the execution history to answer these questions are impractical: even a simple query such as query 1 above would require a full scan of the execution history! In order to answer questions such as the above, a dynamic analysis tool should support declarative temporal queries over execution histories. We show in this paper a temporal data model and query language that is capable of

formulating a rich collection of debugging queries. We also show how many of these queries can be easily expressed by the end-user (i.e., software developer) using a simple form.

In visualizing the run-time behavior of object-oriented programs, we have found two types of diagrams are especially useful: object diagrams for the current state, and sequence diagrams for the execution history [7]. Although object and sequence diagrams are traditionally used in UML to document use cases, our proposed visualization system (Jive) [2] displays these diagrams at run-time, thereby facilitating a comparison of design-time with run-time behavior in a uniform notation and helping "close the loop" between design-time and run-time. (In fact, Jive constructs extended object and sequence diagrams, we explain in section 3.) An important property of our proposed approach is that every point on the sequence diagram is associated with the object diagram that would have been in effect at that point in execution. Thus, the sequence diagram serves as an effective temporal navigation tool, allowing a user to access any point in the execution history.

A fundamental problem, however, with visualizations is that they tend to be become very large for even modestly sized programs [3]. In order to be useful and effective, a visualization tool must be able to determine how much information to display and how to display it without compromising the meaning of the program execution. We propose two broad classes of techniques to achieve compact sequence diagrams: folding operations, for nested calls (horizontal folding) as well as for a sequence of calls (vertical folding); and filtering operations, to remove unnecessary or irrelevant calls relative to a debugging task. (Our earlier paper [6] showed techniques for compact object diagrams.) We introduce a simple regular-expression-like notation, with sequencing and definite interaction, in order to compactly label execution sequences. We also introduce a refinement of the sequence diagram in order to account for calls from a method that is not filtered (i.e., 'in-model') to one that is filtered (i.e., 'out-of-model'), and vice-versa. Together, these techniques have proven to be effective on a number of examples. They have been implemented and tested as part of the dynamic analysis tool, Jive, Java Interactive Visualization Environment, which is available as a plug-in for Eclipse.

The main contribution of this paper lies in showing that *declarative temporal queries and object/sequence diagrams work in a symbiotic manner to produce an effective debugging system*: queries help the user to focus on specific regions of the diagrams, and the diagrams provide a framework for reporting the answers to debugging queries. While previous research has focused on one or the other of these two topics exclusively, our work shows the benefit of integrating these two techniques. In particular, sequence diagrams provide a visual time-line which is especially useful for reporting answers to 'when' queries, e.g., "when did variable x first become negative?", "when did two threads concurrently access x?", etc. Likewise, such queries help define the points of interest in the sequence diagram.

The rest of this paper is organized as follows: Section 2 surveys closely related work on run-time visualization and query-based debugging. Section 3 provides an overview of our previous work on Jive, an interactive visualization environment for Java. Section 4 describes our temporal data model and illustrates temporal queries for debugging. Section 5 describes techniques for compact visualizations. Section 6 presents conclusions and directions for further work.

2. RELATED WORK

One of the distinguishing aspects of Jive is that it integrates in a single tool techniques from query-based debugging and dynamic visualization of execution.

Query-Based Debugging. Query-based debugging was first proposed by Lencevicius *et al* [14]. In their approach, a query is formulated in the programming language itself and run against the objects in the heap. There is no support for querying past program state and queries are not guaranteed to be side-effect free. Jive's data is in-memory as well, but it is constructed from program traces. Querying past state is supported and queries are side-effect free. TOD [15] is a scalable omniscient debugger that features query-based debugging and dynamic visualizations. It uses bytecode instrumentation to generate events which are recorded to a specialized database. This contrasts with Jive's current approach of relying on JPDA to trace events and storing events in-memory. TOD's query language is based on two low-level primitives, *cursor* and *count*, with higher-level queries being constructed algorithmically. On the other hand, Jive's query-based debugging features are based on a declarative temporal query language. TOD provides high-level visualizations in the form of murals, which are graphs showing the evolution of event density for a given class of events. These visualizations do not provide the detailed information about execution history that Jive's sequence diagram does.

Whyline [11] is an *interrogative* debugger supporting 'why did' and 'why did not' queries about program executions. It works on recorded rather than on live executions, therefore, online debugging is not supported. Whyline does not expose a query language. PTQL [9] is a relational query language with SQL-like syntax designed to query program traces online via instrumented code. PTQL supports conjunctive select-project-join queries against a schema consisting of two relations: method invocations and object allocations. Coca [4] is an automated debugger for C that allows setting event-based breakpoints prior to program execution in the form of Prolog-like queries. When an event matching the query is detected, the program suspends and the developer is allowed to query current and past program state.

To the best of our knowledge, none of the query-based debuggers described herein present query results on a sequence diagram. Jive displays query results in both tabular form and on the sequence diagram, providing rich context that facilitates the user's interpretation of the results.

Dynamic Visualizations Ovation [3] visualizes execution traces using an execution pattern view, a form of interaction diagram that depicts program behavior. Diagrams support a number of operations such as collapsing, expanding, filtering, and execution pattern detection (e.g., repetition). Ovation also supports searches for execution patterns on different criteria. Jive and Ovation both rely on trace data to construct their diagrams, provide search capabilities to explore the program execution via their respective diagrams, and support techniques to help users focus on regions of interest in the diagram. In contrast with Ovation, Jive is an online debugger, uses a declarative temporal query language for querying the underlying data model, and uses an enhanced sequence diagram to represent program execution.

Amida [21] extracts sequence diagrams from program traces and applies a dominance algorithm in order to detect and

remove local objects contributing to internal behavior of dominator objects. Experimental results show an average removal of 40% of all objects from execution traces. Amida processes traces offline while JIVE displays sequence diagrams during program tracing. JIVE scales sequence diagrams using folding and filtering and providing visual cues to indicate the existence of additional structure and/or information in the diagram. Amida excludes objects from the sequence diagrams but provides no indication that certain interactions are omitted.

Sharp [17] describes interactive exploration of UML sequence diagrams constructed by reverse engineering the source and rely on static analysis. Their filtering techniques include temporal, call depth, and interaction fragment filtering. Filtering may be realized by graying out the filtered out parts of the diagram, or by removing them altogether. JIVE's sequence diagrams provide a dynamic view of an actual program execution. With the exception of their interaction filters, our horizontal folding techniques can scale the sequence diagrams in a manner comparable to their filtering. Our vertical folding is semantically richer in that it reduces the diagram and provides an intuitive and concise description of the hidden substructure.

TPTP [5] is primarily concerned with collecting profiling data, but is able to represent the entire execution of a program as a sequence diagram, interactively. It supports filtering and hiding methods and objects, as well as collapsing entire call trees. However, the latter case is not automatic. Program Explorer [13] uses merging and filtering to reduce the size of its object and interaction graphs. Programs are visualized interactively and their execution traces can be viewed as interaction charts which are similar to sequence diagrams. ISVis [10] uses both static and dynamic analyses to construct message flow diagrams similar to sequence diagrams. These diagrams represent interaction patterns in the trace. A global view of the execution is displayed in its execution mural.

Homonyms. Two other research projects have adopted the name "Jive" for their tools. In [16] tracing is done over intervals and visualizations consist of box displays containing various statistics relevant to the particular box. This tool is somewhat related to our JIVE tool, but our approaches to limiting traces and visualizations are quite different. Further, [16] does not support queries. The tool described in [8] focuses on visualization of algorithms and data structures, so there is no relation with our JIVE tool.

3. OVERVIEW OF JIVE

The Java Interactive Visualization Environment [2], JIVE, is a versatile dynamic analysis tool suitable for a number of applications including visual and query-based debugging, program comprehension, and teaching programming languages and software engineering. JIVE is currently implemented in Java as an Eclipse plug-in.

In many aspects, JIVE works much like a traditional debugger, allowing one to define breakpoints, inspect variables, step into and over instructions, etc. JIVE also provides features that transcend the abilities of traditional debuggers: dynamic visualizations of the run-time state and execution history, query-based debugging, and interactive forward and reverse stepping. Figure 1 shows JIVE in action during a debug session. The usual debug windows are displayed on the left: call stacks (top left), source (middle left), breakpoints (bottom left). On the top right, an object diagram depicts the current state of the program's objects. On the bottom right, a sequence diagram displays the entire call history of the program.

3.1 Architecture

JIVE's implementation is based on a model-view-controller architecture, the main components of which are illustrated in Figure 2.

JPDA Debugger. The debugger part of JIVE is implemented on top of the Java Platform Debugger Architecture (JPDA), an event-based debugging architecture where debugger and debuggee tiers run in separate Java Virtual Machines (JVMs). The debugger front-end and back-end communicate using the Java Debug Wire Protocol (JDWP) and the debugger front-end communicates with JIVE using the Java Debug Interface (JDI). The types of event requests supported by JDI are: virtual machine start, death, and disconnect; class prepare and unload; thread start and death; method entry and exit; field access and modification; exception; step.

Controller. JIVE's controller has three modules: an event handler, a model manager, and a UI engine. The event handler requests events from JPDA and processes event notifications received from JPDA. The event handler is capable of inferring additional event types not directly supported by JPDA, such as local variable changes. The model manager receives events from the event handler and triggers appropriate model changes. Finally, the UI engine uses data contained in the models to update the object and sequence diagrams.

Temporal Data Model. JIVE's temporal data model consists of two submodels: design-time and run-time. The design-time submodel is the non-temporal part and stores information about the program's types and their relationships. The run-time submodel stores all the dynamic information about the behavior of the program, including events (e.g., variable changes, method calls), field and variable bindings, method execution intervals, etc. A more complete discussion of the temporal data model is provided in section 4.

Debugging an application with JIVE proceeds as follows. Early during the debugging bootstrap process, JIVE's event handler module requests a number of debug event types from the debugger front-end. As debugging progresses, the event handler receives event notifications asynchronously, via the debugger front-end. Handled events are forwarded to the model manager which then updates the trace data model. Updates to the object and sequence models, if necessary, are also triggered at this time by the model manager. Finally, the UI engine updates the object and sequence diagrams displayed to the user, if necessary, using the updated models.

Reference Example. We designed a Binary Search Tree (BST) example in order to illustrate JIVE's integrated query-based debugging and scalable visualizations capabilities. The BST application uses a model-view-controller architecture: the model (instance of `Model`) maintains a binary search tree (instance of `BSTNode`) of `int` values; the view displays a BST widget and a "Load..." button that pops up a dialog from which the user can select a data file; the controller coordinates the interaction between the view and the model. A run of the BST application proceeds as follows. First, the controller is created, then it instantiates one view and one model, displays the view, and waits for a file to be loaded.

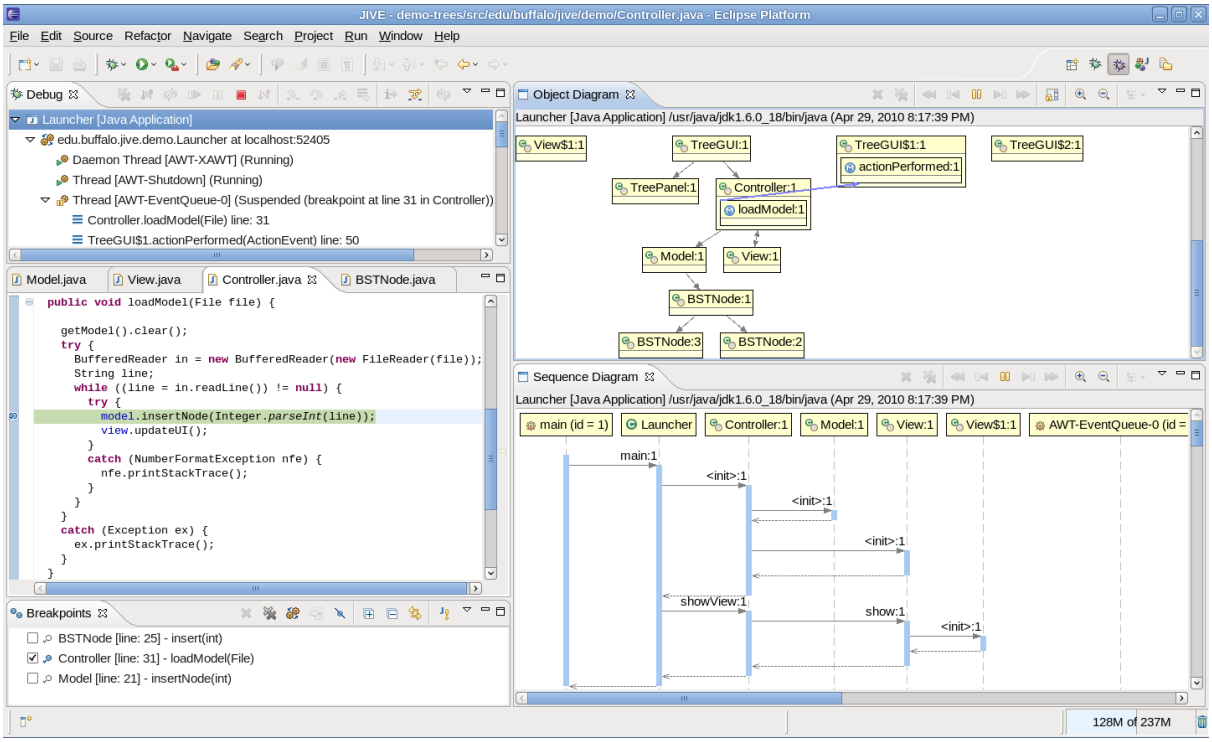

Figure 1: JIVE user interface.

After the user selects a file, the controller loads BST data from the file: for each line, the controller passes the new tree element to the model by calling `Model.insertNode(int)` and then notifies the view `View.udpateUI()`. The BST application was originally designed not to support insertion of duplicate data, however, we injected a bug in the `BSTNode.insert(int element)` method in order to allow duplicate insertion.

3.2 Visualization

Jive supports two kinds of visualizations: object and sequence diagrams. JIVE's object diagram captures runtime state by showing object states, structural links, and outstanding method activations within their respective object contexts. They provide considerably richer information when compared to UML's object diagrams. Such additional information augments program comprehension and facilitates debugging. A detailed discussion of JIVE's object diagrams, including graph drawing algorithms, scalability issues, and implementation challenges has been the subject of previous work [6].

JIVE constructs a sequence diagram dynamically at runtime in order to visualize the execution history of a program. JIVE's motivation differs fundamentally from that of UML in that the UML sequence diagram documents design-time considerations while JIVE's diagram captures runtime interactions between objects. Activation boxes in JIVE's sequence diagrams are colored based on the threads from which the activation is made. A fragment of a sequence diagram is shown at the bottom right of figure 1.

An important aspect of the sequence diagram is its dynamic nature. As program execution progresses, the diagram grows in both directions. Horizontal growth occurs at the rightmost end of the diagram as new life lines are appended to the diagram. Vertically, the diagram grows with the size of the trace: the more trace events the sequence diagram must represent, the larger it grows downward. An important consequence of this continuous growth of the sequence diagrams is that they quickly become very large and users must scroll the viewport in order to focus on a particular region of the diagram.

The sequence diagram also serves as an effective temporal navigation tool. Figure 4 shows the result of jumping back to a previous point in the execution: the horizontal dashed line indicates the current point in execution and the object diagram has been updated to reflect the program state at that past time, including all outstanding method activations (as indicated by the blue arrows). The navigation bars on top of both diagrams enable stepping and resuming both forward and backward in time.

Reverse Stepping. In JIVE, developers may inspect any past state of the debuggee without having to restart the program by using the reverse stepping feature. As reverse stepping is underway, JIVE updates the object diagram so it displays the state of the debuggee at the execution point being visited. The sequence diagram is also updated so it displays the current execution point being visited as a horizontal dashed line cutting across all life lines. For a complete discussion of how JIVE implements reverse stepping, we direct the reader to [2].

4. TEMPORAL DATA MODEL

We now present our temporal model for debugging. It consists of an *abstract temporal database*, a schema representing design-time and run-time entities, a concrete realization of the abstract database, and applications in debug-

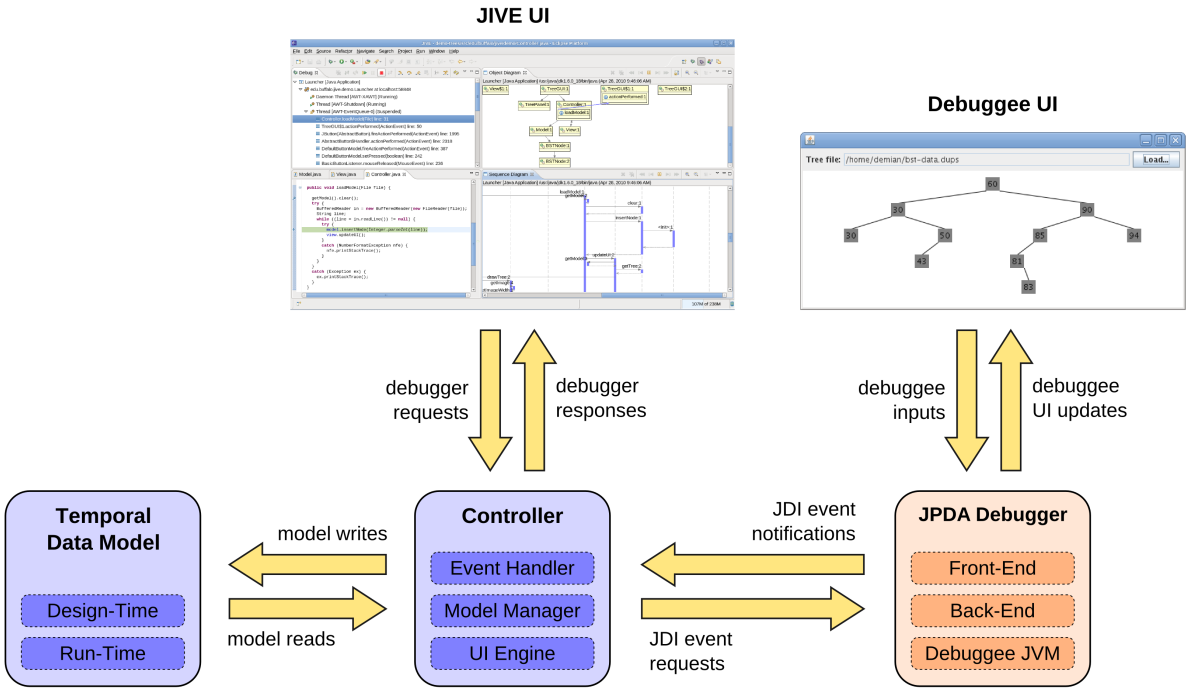

Figure 2: Jive's architecture.

ging. Our temporal database framework relies on earlier results from research in temporal databases, a summary of which we present next.

Background. A temporal database is a structure that supports some aspect of time (user-defined time excluded). Its *data model* defines the *temporal* and *data* values that may be stored and manipulated by the database, as well as their *supported operations*. A *query language* provides a standard mechanism to retrieve and modify data.

Temporal data models are usually based on time instants or intervals. An interval-based model provides compact representation of temporal data. However, it is subject to a number of problems: query languages with syntactic dependence on interval encodings; necessity of operations to consolidate representation; and, expressing representation-dependent queries [20]. On the other hand, a point-based data model allows for a clean, declarative syntax and simpler semantics, at the cost of a verbose representation of the data.

In order to benefit from the well-defined semantics and clean syntax of point-based models and the efficient encoding of interval-based models, we split the temporal database into a representation-independent *abstract temporal database* and a *concrete temporal database* for storage-efficient encoding of the temporal data [1]. A compilation technique [19] provides an elegant path for the evaluation of queries in the abstract query language and return answers *using the concrete encoding*.

4.1 Abstract Temporal Database

Our abstract temporal database supports time values that are discrete and linearly ordered (e.g., integers), and the expected uninterpreted types such as strings. Our temporal database schema contains entities pertaining to the runtime (temporal) and design-time (static).

Runtime Schema. At a high-level, the run-time consists of program states (e.g., temporal values of fields and local variables) and method activations (e.g., intervals between a method call and return). Program states are represented by *environments*, which are containers for bindings of names (e.g., fields or local variables) to values. Method activations play a dual role: they represent both an interval during which a particular method executes and also an environment for local variables. The execution flow a program (over time) may be conceived as a tree of method activations (one per thread), with nested calls represented as child nodes in the tree. Every method activation executes in a number of *steps*, each of which consists of one or more *events* (e.g., variable reads and writes, object construction, method call, etc). Note that the use of a step entity is a convenience for mapping events to specific points in the source code. A subset of the runtime part of our schema is presented below (in simplified form):

1. *Activation(aid, time, threadId, envId, mid)*
2. *Step(aid, time, lineId)*
3. *Event(time, eventId)*
4. *EventParam(time, paramId, value)*
5. *Environment(envId, time, tid, parentEnvId)*
6. *FieldBinding(envId, fid, value, time)*

We emphasize the point-based nature of the schema above. In particular, the binding of a field to a particular value may span several time instants. However, in the abstract schema we only represent information about bindings *at individual points in time* (as opposed to providing fields such as *fromTime* and *toTime*). When given queries referring to field bindings, the compiler guarantees that a correct query ranging over intervals is executed against the concrete database.

Design-Time Schema. This part of the schema represents program entities that are realized at runtime. They provide information that allows exploring relationships that would be otherwise difficult or even impossible (e.g., does a type implement a particular interface?). Additionally, because design-time information does not vary over time, it can be factored out from the temporal part of the schema. A subset of the design-time part of our schema is presented below (in simplified form):

1. *Type(tid, descriptor)*
2. *Field(fid, containerId, name, tid)*
3. *Method(mid, containerId, name, tid)*
4. *Local(varId, mid, name, tid)*
5. *Line(lineId, mid, lineNo)*
6. *TypeRelationship(baseId, relType, referenceId)*

In the above, a type is either a class, interface, or *method signature*. Fields and methods are named and typed members of some container type (by proper definition of key constraints, unique field names within types may be easily defined, and so can overloaded method names). In a similar way, Locals are variables defined in method bodies. A line represents an actual source code line of some method. Finally, type relationships represent the notions of class and interface inheritance.

4.2 Temporal Query Language

The selection of a query language should be motivated by the problems that it aims to solve. In that respect, our query language is applicable to a wide range of debug use cases which we present next, by example.

EXAMPLE 1. *The use of queries may help identify and resolve important concurrency errors and improve the design of concurrent programs. The query below shows two concurrent activations modifying the same instance:*

```
1  SELECT a1.aid, e1.time, a2.aid, e2.time
2  FROM
3   Activation a1 JOIN Event e1 JOIN
4   EventParam epi1,
5   Activation a2 JOIN Event e2 JOIN
6   EventParam epi2,
7   Activation a3
8  WHERE
9   a1.threadId <> a2.threadId  AND
10  a3.aid = a1.aid AND a2.time = a3.time AND
11  e1.eventId = "assignment" AND
12  e2.eventId = e1.eventId AND
13  epi1.paramId = "instance" AND
14  epi2.paramId = epi1.paramId AND
15  epi1.value = epi2.value;
```

The query above is interpreted as follows. Lines 9-10 guarantee that the activations represented by `a1` *and* `a2` *are concurrent (they are in different threads and have at least one time point in common). Lines 11-14 verify that, during each activation, an assignment to some instance field was performed. Line 15 guarantees that the updated instances were the same.* □

EXAMPLE 2. *Say a developer wishes to verify that method "int M(int n)" in a type "C" is monotonically increasing (i.e., M(x) < M(y) whenever x < y). The following query solves the problem by computing the complement of this property, i.e., finding all call pairs that violate monotonicity:*

```
1  SELECT a1.aid, a2.aid
2  FROM
3   Type T, Method m JOIN Local v,
4   Activation a1 JOIN ActivationParam ap1 JOIN
5   ActivationResult ar1,
6   Activation a2 JOIN ActivationParam ap2 JOIN
7   ActivationResult ar2
8  WHERE
9   t.descriptor = "C" AND
10  m.containerId = t.tid AND
11  m.name = "M" AND v.name = "n" AND
12  ap1.paramId = v.varId AND
13  ap2.paramId = v.varId AND
14  a1.aid <> a2.aid AND
15  ap1.value < ap2.value AND
16  ar1.value >= ar2.value;
```

The query above reads as follows. First, we identify the type, method, and local variable (lines 9-11). Lines 12-13 associates the input parameters with the corresponding local variable. Lines 14-15 identify distinct activations with inputs satisfying the monotonicity precondition. Line 16 checks for a violation in monotonicity. □

A large class of debug queries are expressible as *select-project-join* (SPJ) queries such as the ones above. More advanced debug queries may require features such as *aggregation* (e.g., when did the number of instances of a class exceed `k`?). An even more interesting class of debug queries would require recursion. For instance, design time data structures may be inherently recursive (e.g., trees, graphs) and many queries about their realization would be naturally recursive (e.g., what was the height of the binary tree instance `bt` at time `t`?). The run-time state of a program is also typically represented as an object graph. Analyzing the global program sate would require answering recursive queries such as: was object o_2 ever reachable from object o_1?

Our temporal query language supports all of the aforementioned features. Further, the syntax and semantics of the language is based on explicit quantification over time instants (e.g., example 1 above) rather than on intervals. Our system provides the necessary transformation to obtain answers in encoded in the form of intervals.

Query Formulation. Users must be provided with adequate interfaces so they may formulate and execute debug queries. Our current approach is to provide a template-based and a textual interface. The template-base interface provides a number of SPJ *query templates* for which users only need to provide the required parameters, if any. The textual query interface, on the other hand, enable users to formulate and evaluate any query.

Figure 3 illustrates the template-based interface. The "Variable Changed" template query has been parameterized to request all assignments of a negative value to the data field of a `BSTNode` instance.

Presenting Query Answers. Once a user executes a query, results must be presented in an adequate form. In typical database applications, results are presented in tabular form. In a debug application, however, we argue that a tabular output alone is not as effective. In order to make debugging more agile, query results should be presented with adequate context, such as the one provided by the sequence and object diagrams.

JIVE presents query answers as follows: each answer is

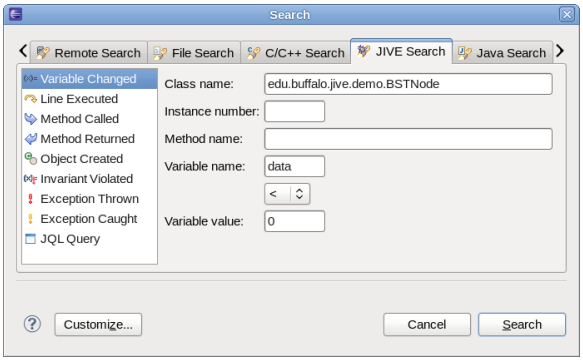

Figure 3: Template-based interface.

presented as a row in the search window and also marked on the sequence diagram as a small red box, in the appropriate activation box. When the user double-clicks a row in the search window, JIVE performs a number of tasks: 1) the sequence diagram *focuses* (see next section) on the activation box containing the result; 2) the activation box is brought into the current viewport, if not visible; 3) the *temporal context* of the debuggee is synchronized with the time corresponding to the query answer; 4) the state of the object diagram is reconstructed in order to match the new temporal context. Figure 4 shows the results of the "Variable Changed" query of figure 3, with the first result also selected on the sequence diagram.

5. SCALABLE VISUALIZATIONS

JIVE supports scalable visualizations of both object and sequence diagrams. Users may easily suppress the display of the internal details of objects and their interactions in the object diagram. This includes suppressing superclass details, hiding field tables, showing only objects involved in the call path, hiding aggregated objects, etc. Further details on scalable object diagrams were considered in our previous work [6]. In the remainder of the paper, we focus exclusively on scalability issues of sequence diagrams.

Although queries are effective in reducing the amount of data users must analyze, interpreting such data is not trivial. In particular, the tabular presentation provides very little context while the sequence diagram provides context that is hard to interpret due to the usually large dimensions and complexity of the diagram. In order to help users effectively interpret query results, JIVE supports focusing sequence diagrams on regions of interest via *diagram folding*. Folding aims at reducing sequence diagram size and clutter by removing uninteresting regions without compromising the meaning of the diagram as a whole. The portions of the diagram containing regions of interest and their relevant contexts are left intact.

Because program traces tend to grow extremely large, folding alone may not fully realize scalable visualizations. Hence, JIVE allows users to filter out trace events that are unnecessary or irrelevant to the particular debug task at hand, thereby keeping the volume of trace information at a reasonable level. While folding is a dynamic operation in that it may be applied to a sequence diagram at any time, filtering affects the underlying execution trace so it must be defined prior to starting a debug session. With filtering enabled, sequence diagrams must handle missing information gracefully without compromising the meaning of the interactions displayed by the diagram. For example, JIVE is capable of inferring out-of-model (i.e., filtered out) calls and returns in order to correctly place "lost" and "found" messages in the diagram.

5.1 Sequence Diagram Folding

We now describe the two main types of folding: horizontal and vertical. Horizontal folding hides all nested activation boxes of a given activation box. This allows users to focus on the high-level meaning of the folded activation rather than on how its internal behavior is implemented. Vertical folding replaces a group of adjacent sibling activation boxes with a new activation box, which is labeled by a regular expression corresponding to the sequence of folded activation boxes. This type of folding is most effective when collapsing call sequences resulting from loops.

Fold operations are defined with respect to one or more activation boxes and their scope is limited to a single thread. They are reversible: for each fold operation, an unfold operation that reverts its effect on the sequence diagram is also defined. JIVE supports manual folding through context menu actions on the sequence diagram and automatic folding via preference settings.

5.1.1 Horizontal Folding

Given an activation box A in the sequence diagram, horizontal folding hides all child activation boxes of A. The folded activation box A is adorned with two '+' (plus) symbols, one on the top and one on the bottom. Structurally, this amounts to flattening the entire subtree rooted at A. JIVE implements this horizontal folding as the *Fold* operation. Five additional fold operations are defined: *FoldAfter* folds all activation boxes that started and terminated after A terminated; *FoldBefore* folds all activation boxes that terminated before A started; *FoldChildren* folds every child activation box of A; *Focus* composes *FoldBefore* and *FoldAfter*; *FocusLifeline* applies *FoldChildren* on all activation boxes of the specified life line and *Focus* on the activation boxes of all other unrelated life lines. Finally, given activation boxes A_1 and A_2, *FoldBetween* folds all activation boxes that started and terminated after A_1 terminated, and before A_2 started. *FoldBetween* may be naturally extended to any number of activation boxes. To reverse folding, we proceed in an analogous manner and define one inverse unfold operation for each fold operation defined above, using the obvious naming scheme, i.e., *Unfold*, *UnfoldAfter*, etc.

Double clicking a query result on the search window causes JIVE to perform a *Focus* operation on the respective activation box. Figure 4 shows a fragment of a sequence diagram after double clicking on the first result row in the search window. The query result is displayed as a red box within the activation box labeled `<init>:3`. All activation boxes before and after the subtree containing `<init>:3` have been horizontally folded, namely, all `insertNode` and `updateUI` activations.

Finally, when JIVE is configured to fold the sequence diagram automatically, it performs a *FoldBefore* operation after every method return event notification. This results in a sequence diagram that is completely folded, except for the activation boxes corresponding to the outstanding method calls in every active thread call stack.

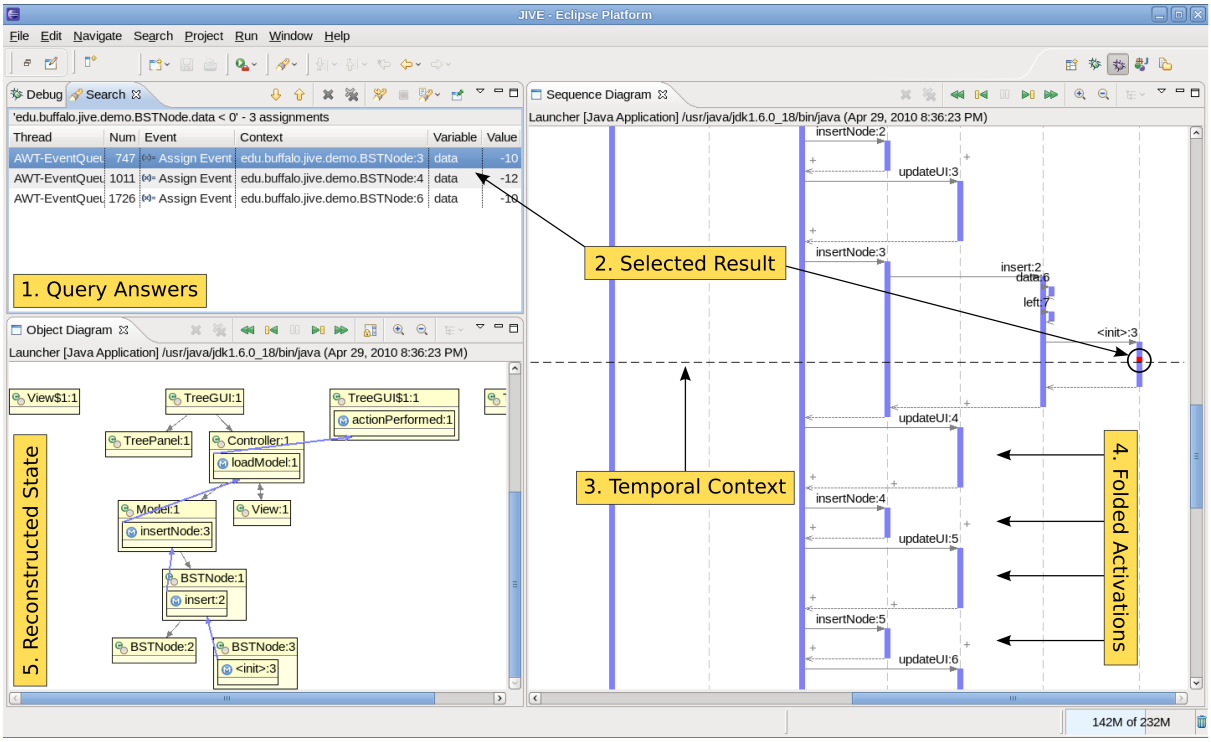

Figure 4: Query results in tabular form (top left) and the first result marked as a red box on the sequence diagram (right).

5.1.2 *Vertical Folding*

The goal of vertical folding is to collapse repeating patterns of calls within a single activation box in the sequence diagram. It takes an activation box A as input and replaces all child activation boxes with a new activation box F labeled with a regular expression $R(A)$ encoding the sequence of calls made in A. The regular expression $R(A)$ has the form $(s_1)^{i_1} \cdots (s_n)^{i_n}$, where each $(s_k)^{i_k}$ consists of a primitive call sequence s_k and a repeat count $i_k \geq 1$. Primitive call sequences are those that cannot be expressed as a smaller call sequence and a repeat count, e.g., $(update; update)$ is not primitive because it can be expressed as $(update)^2$. As with horizontal folding, the vertically folded activation box A is adorned with two '+' (plus) symbols, one on the top and one on the bottom. JIVE implements vertical folding as the *RegexFold* operation. For every horizontal fold (unfold) operation, JIVE implements an analogous vertical fold (unfold) operation, with the expected name, i.e., *RegexFocus*, *RegexUnfoldAfter*, etc.

Figure 5 shows the sequence diagram of the BST application after applying *RegexFocus* to the activation box `<init>:4` on the life line `BSTNode:4`. Vertically folded activation boxes are displayed in red: the top one, labeled $(\texttt{insertNode}; \texttt{updateUI})^3$, replaces six activation boxes; the bottom one is labeled $(\texttt{updateUI}; \texttt{insertNode})^6$ and replaces another twelve. All of these activation boxes are contained in the context of `loadModel:1` and represent the loop for inserting elements read from file into the tree while updating the tree widget for every insert. It is clear from the diagram that exactly 10 nodes were inserted into the tree.

5.2 Filtering

Filtering is motivated by the observation that users may know beforehand that certain parts of the code are uninteresting. For instance, in a debugging scenario, parts of the code may be trusted to be bug free. In a program comprehension scenario, the user may be interested in the public interactions among objects. Regardless of the actual user motivation, this opens the possibility for reducing the amount of trace information collected by the tracer. As previously noted, a consequence of filtering out trace events is that sequence diagrams must deal with missing information.

Regular Expression Filters. JIVE supports filtering of types based on their names or the package in which they are defined. A simple regular expression may be given, say, `java.*`, to filter out all trace events from types defined in any package matching the expression (e.g., `java.util.List`). JIVE provides sensible default package filters for applications, applets, and unit tests.

JIVE also supports regular expression filters on method names. This allows users to filter out parts of behavior considered uniniteresting for the task at hand. For instance, the user may filter out getter and setter methods by defining the filters: `MyClass.get*` and `MyClass.set*`. These filters effectively eliminate from JIVE's trace any getter and setter calls made by instances of `MyClass`.

Visibility Filters. In the object oriented paradigm, visibility scopes help users separate design from implementation concerns. Implementation details are hidden from view by declaring them either as private (or protected), while visible behavior is declared as public. JIVE enables users to focus on interactions happening at any visibility scope. For instance, users trying to gain a high-level understanding of a software may choose to view only public interactions involving public classes. On the other hand, users trying to debug the implementation of a particular class may choose to view

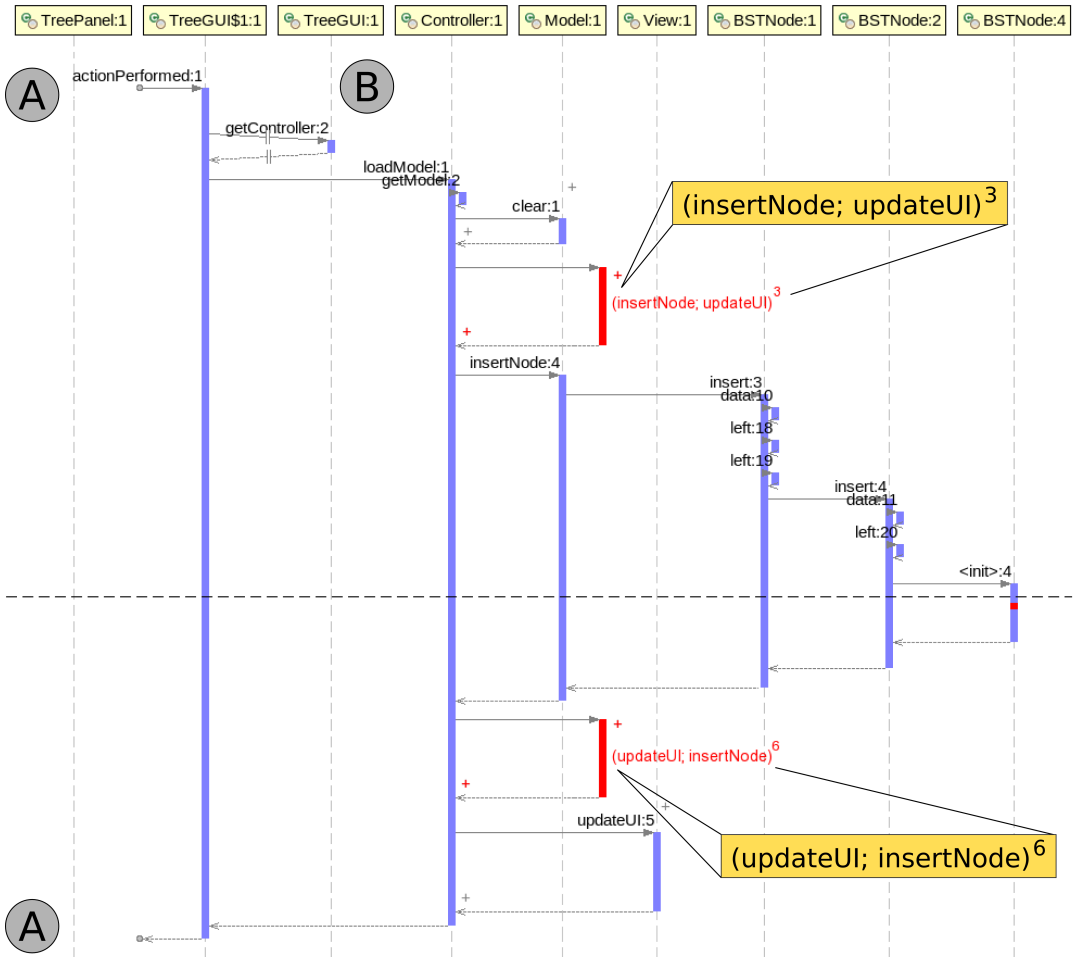

Figure 5: Sequence diagram illustrating vertically folded activation boxes in red, with regular expression labels (zoomed in). Region (A) shows an out-of-model call (top) and a matching out-of-model return (bottom). Region (B) shows an out-of-model call and return pair.

methods defined with any visibility scope.

JIVE supports additional filters based on specific attributes of types and methods. For instance, in the default configuration, JIVE suppresses trace events for all synthetic (compiler generated) method calls. It is also possible to filter out trace events from nested types. Typical use cases for nested classes in Java include the implementation of public interfaces and the subscription of observers for event notifications. Cases such as these ones may not be interesting for users who wish to gain a high-level understanding of the system.

Out-of-Model Calls and Returns. As mentioned earlier, the use of filters causes event traces to be incomplete. This means that every time JIVE receives a method call or return event notification, it must determine whether the event originated/terminated in-model (i.e., non-filtered class or method) or out-of-model (i.e., filtered class or method). This is accomplished by inspecting the debuggee's call stack and comparing it with corresponding stacks maintained by JIVE.

Once JIVE determines that a method call (return) originates in or out of model, it proceeds to draw the correct call (return) arrow in the sequence diagram. If a method is called from an out-of-model caller, JIVE uses a found message arrow. If a method returns to an out-of-model caller, it uses a lost message arrow. Lost and found messages are defined as part of UML's sequence diagrams. JIVE provides a variation on lost and found messages as follows: if a method m_2 is the first one called from an out-of-model caller within the execution of an in-model method m, then JIVE draws an 'in/out+/in' call arrow connecting the activation boxes corresponding to m and m'. Likewise, if m' returns to out-of-model and no other in-model method is called before an out-of-model caller returns to m, then JIVE draws an 'in-/out+/in' return arrow connecting the activation boxes corresponding to m' and m. Figure 6 shows the styles used for

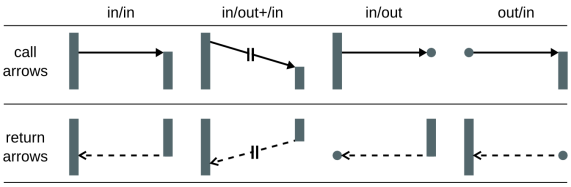

Figure 6: Sequence diagram arrow types.

all call and return arrows used by JIVE. Figure 5 contains two out-of-model call/return pairs: `actionPerformed`, called by the (filtered) `AWT` thread, and `getController`, called by a synthetic method of the anonymous `TreeGUI$1` class (an `ActionListener` object which listens to the "Load..." button click). The former corresponds to a 'out/in' call and a 'in/out' return; the latter to a 'in/out+/in' call/return pair.

6. CONCLUSIONS AND FURTHER WORK

We have described a temporal data model and query language for declaratively stating a rich collection of debugging queries over the execution history of a program. We also described UML-like object and sequence diagrams for representing the run-time state and execution history, and ways of depicting them in a more compact way to facilitate scalability for large executions. The main idea underlying our approach is that declarative queries not only facilitate efficient search for run-time information – akin to a web search engine – they also help the visualization system focus on what is to be displayed.

The sequence diagram is particularly helpful in providing a visual time-line for reporting answers to 'when' queries. In order to compactly depict the time-line, the paper also presented two broad classes of techniques (folding and filtering) for reducing the amount of information displayed. We introduced a regular-expression notation for concisely abstracting a sequence of calls in sequence subdiagram, and also a refinement of the sequence diagram itself for calls between methods that are filtered out and those that are not.

Most of the techniques described in this paper have been successfully incorporated in a state-of-the-art debugging system, JIVE, for the Java programming language. This is an evolving system which been in operation since 2007 and tested on a large number of programs and also been used for teaching Java. It may be obtained from `http://www.cse.buffalo.edu/jive`.

To further enhance the performance and usability of the JIVE system, we are exploring three approaches: (i) the use of byte-code instrumentation for minimizing the extent of process context-switching; (ii) the use of a visual query language for expressing more complex temporal queries; and (iii) the use of an external database for saving large execution histories. The database approach also allows us to explore debugging with multiple runs, a topic that we are also investigating.

7. REFERENCES

[1] Jan Chomicki. Temporal query languages: A survey. In *ICTL '94: Proc. of the 1st Intl. Conf. on Temporal Logic*, pages 506–534, London, UK, 1994. Springer-Verlag.

[2] J. K. Czyz and B. Jayaraman. Declarative and visual debugging in eclipse. In *Proc. Eclipse Technology eXchange*, pp. 31–35, Montreal, 2007.

[3] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. 4th COOTS*, pp. 219–234, April 1998.

[4] M. Ducassé. Coca: an automated debugger for c. In *Proc. 21st ICSE*, pp. 504–513, Los Angeles, CA, 1999.

[5] Eclipse. Eclipse Test and Performance Tools Platform. [Online; accessed 28-July-2006].

[6] P. V. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. In *Proc. 2nd SoftVis*, pp. 95–104, St. Louis, MO, 2005.

[7] P. V. Gestwicki and B. Jayaraman. Interactive Visualization of Java Programs. In *Proc 2nd HCC*, pp. 226–235, Arlington, VA, 2002.

[8] G. Cattaneo, P. Faruolo, U. F. Petrillo, and G. F. Italiano. JIVE: Java Interactive Software Visualization Environment. In *Proc 4th VL/HCC*, pp. 41–43, Rome, Italy, 2004.

[9] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proc. 20th OOPSLA*, pp. 385–402, San Diego, CA, 2005.

[10] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. 19th ICSE*, pp. 360–370, Boston, MA, 1997.

[11] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *Proc. 27th CHI*, pp. 1569–1578, Boston, MA, 2009.

[12] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[13] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, May 1997.

[14] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *12th OOPSLA*, pp. 304–317, Atlanta, GA, 1997.

[15] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. *SIGPLAN Notitces*, 42(10):535–552, 2007.

[16] S. P. Reiss. Visualizing Java in action. In *Proc. 1st SoftVis*, pp. 57–ff, San Diego, CA, 2003.

[17] R. Sharp and A. Rountev. Interactive exploration of uml sequence diagrams. In *Proc. 3rd VISSOFT*, page 8, Budapest, Hungary, 2005.

[18] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theor. Comput. Sci.*, 270(1-2):843–850, 2002.

[19] David Toman. Point vs. interval-based query languages for temporal databases (extended abstract). In *PODS '96: Proc. of the 15th ACM Symp. on Principles of Database Systems*, pages 58–67, New York, NY, USA, 1996. ACM.

[20] David Toman. Point-based temporal extension of temporal sql. In *DOOD '97: Proc. of the 5th Intl. Conf. on Deductive and Object-Oriented Databases*, pages 103–121, London, UK, 1997. Springer-Verlag.

[21] Y. Watanabe, T. Ishio, Y. Ito, and K. Inoue. Visualizing an execution trace as a compact sequence diagram using dominance algorithms. In *Proc. 4th PCODA*. Belgium, October 2008.