

Doctoral Dissertation Proposal
Towards Declarative and Visual Execution Environments

Jeffrey K. Cxyz
Department of Computer Science and Engineering
University at Buffalo
The State University of New York

Committee:
Dr. Bharat Jayaraman, Chair
Dr. Michalis Petropoulos
Dr. Bina Ramamurthy
Dr. Carl Alphonse

November 20, 2008

Abstract

Traditional debugging is a *procedural* process in that a programmer must proceed step-by-step and object-by-object in order to uncover the cause of an error. In contrast, we explore a *declarative* approach to debugging in which the execution states of a program are queried through a novel temporal query language. In this proposal, we describe such a language by means of examples and show its potential for declarative debugging of object-oriented programs. The queries may examine individual execution states as well as the history of execution. In this research, we also explore techniques for visualizing the runtime states and execution history of object-oriented programs. The current state of execution is depicted through an enhanced UML object diagram in which both object structure and method activation are represented. The history of execution is depicted by a sequence diagram which clarifies object interactions. A major contribution of our work lies in showing that declarative queries and runtime visualizations complement each other: the queries help reduce the amount of information to be visualized, while the visualizations provide a framework for reporting the answers to queries as well as for formulating the queries themselves. Another important contribution of our research is in devising efficient techniques for reverse stepping and jumping over previous execution states. We are developing an experimental framework for incorporating these ideas as an extension of the Eclipse Integrated Development Environment (IDE) for Java. This dissertation proposal describes our results to date and also outlines our proposed research in the areas of scalable visualizations, interactive execution, and declarative query languages.

1 Significance and Objectives

There has been considerable interest in recent years in object-oriented technologies and integrated development environments. While much attention has been given to the areas of software design and implementation, tools that enhance understanding program behavior at runtime have been lagging behind. According to a recent survey, the cost of software failures on the U.S. economy is estimated at nearly \$60 billion annually [28]. Hence, there is a critical need for improved techniques for debugging object-oriented software.

The state of the art in runtime environments for object-oriented programs is exemplified by IDEs such as Eclipse, NetBeans, IntelliJ IDEA, 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. Using these features 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 focus of our research is on developing a declarative and visual environment for program comprehension and debugging.

There have been a few recent studies on the nature of errors in object-oriented programs. In general, errors may be due to a flawed design, incorrect use of language constructs, algorithmic errors, and oversights in coding [14]. To identify the cause of a runtime error in modern debugging environments, a programmer only has access to the current state, i.e., the outstanding (incomplete) method invocations leading up to the current method call and all objects that are accessible from these calls. However, often the cause of an error lies in a completed method call rather than just the outstanding method calls. Thus, an execution environment should ideally provide access to the entire history of execution rather than just the current state, something typical debuggers do not support.

Central to understanding a program, whether object-oriented or not, is comprehending how variables take on certain values. Often a program error arises because a variable has taken on an unexpected value, resulting in an exception such as a null pointer exception. In order to identify such errors, it is desirable that the runtime environment provides comprehensive information about variables, e.g., all changes to a variable; when (at what execution points) a variable took on certain values; when an invariant involving a certain variable was first violated; etc. A manual traversal of the execution history to elicit this information is clearly infeasible due to its size, hence there is a need for a query-based debugging environment. Such a query-based debugger may be categorized as declarative in nature because it allows a programmer to search the entire execution history without engaging in a laborious step-by-step procedural process. We believe the declarative approach complements procedural debugging just as web searching complements web browsing.

Another common cause of program errors lies in the difference between a programmer's mental model of runtime states and the actual states. Visual depiction of runtime states can help highlight these difference more easily. In particular, the use of UML-like object diagrams and sequence diagrams to depict respectively the current state and execution history can greatly assist highlighting these differences. In designing an appropriate visualization, it should be noted that object-oriented programs differ from procedural programs in that objects are not just data structures but serve as environments within which method activations occur. Thus, the standard UML object diagrams need to be extended to capture both the object structure as well as the method activation structure. OO methodology also engenders the use of smaller methods and results in more complex object interactions. We propose the use of UML-like sequence diagrams in clarifying these interactions,

and in this manner we help close the loop between design time and runtime.

Practical experience with visualizations shows that they do not scale gracefully for large executions due to the large number of objects and method activations that arise [7]. In order to achieve scalable visualizations, we not only need the ability to construct concise and abstract diagrams, but also a query capability by means of which only relevant portions of the visualization are depicted. Thus, declarative queries and visualization work in concert to provide a more effective runtime comprehension and debugging environment.

An initial step towards the realization of such a runtime environment was taken in previous research on the Java Interactive Visualization Environment (JIVE) [12, 13]. JIVE depicts the current state through an enhanced object diagram (showing objects and method activations) and the history of execution through a UML-like sequence diagram. It is interactive in that the user may step forward or backward in the execution history to revisit previous runtime states. This feature is invaluable as programmers often step past errant statements while debugging and must re-execute the program to examine its state [2]. JIVE currently performs incremental state-saving during forward execution and performs incremental state-restoration during reverse execution. While this approach incurs minimal storage overhead, it is less efficient for performing jumps back to distant runtime states because each state must be restored one-by-one until the desired state is reached. Thus, one of the goals of our research is in developing efficient reverse (and forward) execution techniques. In contrast with previous research on JIVE, which has focused mainly on interactive visualization, our current research is aimed at providing a more comprehensive execution environment supporting both interactive visualization and declarative debugging.

The rest of this proposal is organized as follows. Section 2 surveys closely related work; Section 3 describes our current approach to interactive visualization including an overview of JIVE; Section 4 describes our current approach to declarative debugging; and Section 5 outlines the remaining research to be completed.

2 Related Work

We survey and organize the closely related work under three main areas: (i) query-based debugging; (ii) program visualization; and (iii) reverse execution. In the area of query-based debugging, our approach extends previous work by providing much higher-level declarative queries over the history of program states. In the area of program visualization, JIVE’s object and sequence diagrams provide a much richer visualization of the execution history and program states compared to the related approaches. In the area of reverse execution, our research advances existing work by examining both static and dynamic techniques for efficient reverse jumping.

Query-based Debugging Query-based debugging was proposed by Lencevicius [24] to understand object relationships. Here, the query language is the same as the target object-oriented language (Self), i.e., it is an imperative language. The advantage of this approach is that the programmer does not need to learn a new language. Queries consist of a search domain and a constraint. In order to realize the semantics of a query, both the compiler and the underlying virtual machine need to be modified. While the execution of the query can modify the state of the target program, it offers the benefit of facilitating incremental answers to queries.

Opium [9] is a trace analyzer for Prolog that uses a simple query model allowing the user to traverse the program execution both forward and backward. More complicated queries can be

formed by writing Prolog predicates in terms of the query model. Coca [8] is an automated debugger for C that allows setting event-based breakpoints prior to program execution. These breakpoints take the form of Prolog-like queries over the event history. When an event matching the breakpoint is emitted, the program suspends and the user is allowed to query the current state. Coca does not allow for examining past states.

The Trace-Oriented Debugger (TOD) [30] is a scalable ‘omniscient debugger’ for Java. Omniscient debuggers record state-altering events as the program runs and allow for revisiting past execution states using simple event queries [25, 26]. TOD uses bytecode instrumentation to generate events, which are then recorded to a specialized event database. Cursors are used by the queries to iterate over the event trace based on a certain condition. TOD’s queries are limited to support stepping, state reconstitution, control flow reconstitution, and root cause finding. TOD uses a diagrammatic technique called murals for high-level overviews of thread, object, and method activity. Compared with JIVE, TOD has no support for visualizing object structures or execution histories and has a less comprehensive query capability.

Whyline [19] is an interrogative debugger for the Alice programming environment. It allows the user to ask ‘why did’ or ‘why did not’ questions to determine the cause of an error. This aspect of Whyline is a potentially very powerful capability, and we hope to explore this through runtime flow analysis queries in JIVE which supports a more powerful set of language constructs compared to Whyline.

Program Query Language (PQL) [27] is a query language that allows a programmer to express application-specific code patterns for Java. Queries are expressed in a Java-like syntax that corresponds to the shortest piece of code that could match the pattern. A combination of static and dynamic techniques are used to identify query matches. Upon finding a match, an additional or alternative piece of code can be executed. The PQL compiler generates code that is weaved into the target application. This technique has been useful in finding a number of interesting security violations. Program Trace Query Language (PTQL) [15] is a relational query language designed to query program traces. PTQL queries are expressed using an SQL-like syntax over object allocation and method invocation tables. Other events such as variable assignments are not supported. Both PQL and PTQL use bytecode instrumentation to limit the trace size. Unlike JIVE, both PQL and PTQL are not interactive debuggers and thereby lack visualization and query capabilities on the execution history.

EndoScope [4] is a software monitoring tool that allows users to pose declarative queries about the state and performance of a program. EndoScope is used to implement specialized profilers by instrumenting only the relevant portion of a program. However, queries are limited to relations dealing with function start and end times, function duration, variable values, and CPU usage. EndoScope also does not support interactive visualization of execution states.

In comparison with some of the above approaches (PQL, PTQL, EndoScope), JIVE currently has a potential shortcoming in that it incurs a greater runtime penalty because it needs to collect more events generated by the JVM in order to support its runtime visualizations and declarative queries. In comparison with TOD, JIVE is currently less efficient in event collection because it depends on the Java Platform Debugger Architecture (JPDA) rather than on bytecode instrumentation.

Program Visualization Recent surveys have compared dynamic visualization tools for object-oriented programs [29, 16]. Sharp [31] describes interactive exploration of UML sequence diagrams using various zooming and filtering techniques. Program Explorer [22] 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. Ovation [6, 7] visualizes an execution trace using an execution pattern view, which is similar to a typical call tree representation. Pattern matching, filtering, collapsing, expanding, and other techniques are used to limit the size of the diagram. ISVis [18] uses both static and dynamic analysis 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. ISVis does not support interactive execution.

In comparison with the above approaches for visualizing object structures and interactions, JIVE additionally supports the visualization of method activations. We believe that this is an essential capability for program debugging. Like some of the above approaches, JIVE supports flexible granularity control, and the end-user may view the diagrams at a high level and also explore details as necessary.

Two important tools for pedagogic use are BlueJ [20] and jGRASP [17]. BlueJ was one of the first pedagogic tool for visualizing executions of object-oriented programs and teaching the object-first approach to programming. Compared with JIVE, BlueJ does not display structural relationships between objects, method activations, or the execution history. jGRASP focuses on intuitive visualizations of dynamic Java data structures such as lists and trees. It uses an ‘external viewer’ model for specifying how the data structure is to be visualized. Thus, such an approach is applicable for a predetermined set of data structures. In contrast, the visualizations of JIVE must work with arbitrary structures. Note that jGRASP does not provide visualization of the execution history.

Reverse Execution Techniques for reverse stepping have been studied in the context of C as well as Java. Most of the techniques focus on incremental state rewinding, while some approaches examine jumping over composite statements such as conditions, loops, and procedure calls.

Spyder [1] is an interactive debugger for C that uses a technique called structured backtracking. In their approach, each assignment statement is associated with a change set. The change set includes all the values the variable was assigned. Reverting to a program point entails restoring variables to their previous values. Composite statements such as conditionals, loops, and procedure calls are associated with a change set consisting of all variables that it can modify. This facilitates efficient backtracking. Once in a past state, Spyder moves forward by re-executing the program.

BDb [3] is a C debugger capable of reverse execution. The concept of an inverse of a statement is used to generate an inverse program. At any point during execution, the user may step backward to a previous state using the inverse program. This is clearly not a general technique since inverse functions are not always guaranteed to exist.

Flashback [33] is a Linux operating system extension for rollback and deterministic replay of software. It uses a state saving approach that duplicates the running process into a number of shadow processes as the program runs. The shadow processes are kept suspended until user needs to revisit the corresponding program state. The drawback of this approach is that each state save is a full checkpoint and thus is not efficient in memory usage.

Lawall and Muller [23] developed an efficient incremental checkpointing approach for the reverse execution of Java programs. They make use of automatic program specialization to optimize a generic checkpointing algorithm. Cook [5] presents a state restoration approach for reverse execution of Java programs. This method logs bytecode instructions using a stack implemented on a circular

buffer. The use of a circular buffer bounds the number of states that may be revisited.

While JIVE currently supports incremental state rewinding, or reverse stepping, as part of our proposed work we plan to explore efficient jumping to distant execution states using a combination of static and dynamic analysis.

3 Overview of JIVE

JIVE supports visualization of the runtime state and execution history, interactive forward and reverse stepping, and declarative queries. In this section, we provide an overview of these capabilities.

3.1 Interactive Visualization

We have incorporated our declarative and visual execution environment (JIVE) into Eclipse, which is an extensible development platform based upon a plug-in architecture [35]. An overview of the JIVE architecture can be found in Figure 1. An important component of Eclipse is the Java Development Tools (JDT), and hence using Eclipse allows JIVE to inherit the functionality of JDT. Since both JDT and JIVE are based upon the Java Platform Debugger Architecture (JPDA), Eclipse serves as an ideal experimental framework for our research.

JPDA is the key component of JIVE, allowing it to observe a running Java program without modifying the compiler or virtual machine. JIVE requests notification of certain runtime events using JPDA's Java Debug Interface (JDI). Such events include class preparation, step taken, variable modification, method entry and exit, thread start and death, etc. The user may specify whether to filter out events related to certain classes, such as those from the Java Platform. Upon event notification, JIVE may suspend the target virtual machine in order to glean any pertinent information from the call stack that may not be available from the JDI event. This together with the JDI event forms a JIVE event, and as the program executes a sequence of JIVE events is formed.

Figure 3 shows a snapshot from our current prototype which supports object and sequence diagrams with granularity control and zooming capability, query processing, and reverse stepping. We will explain each of these capabilities in the remainder of the section.

JIVE constructs two main models from the above event sequence: an object model and a sequence model. The object model represents the program's execution state, while the sequence model details its history of execution. These models serve as the basis from which JIVE's visualizations are derived. An object diagram depicts the program's execution state by showing objects and their structural links as well as outstanding method activations. In this sense, JIVE's object diagram differs from the UML object diagram. By depicting method activations in their object context, we not only clarify the semantics of method activations but also facilitate program comprehension and debugging. The object diagram is also helpful in clarifying Java's semantics for overriding and shadowing.

The JIVE sequence diagram is constructed interactively at execution time. Our approach differs fundamentally from the UML sequence diagram in that the latter documents design-time considerations, whereas the JIVE sequence diagram reflects the actual sequence of object interactions at execution time. In this respect, our approach also differs from other approaches that construct sequence diagrams by reverse engineering the source code [21]. JIVE constructs the diagram interactively rather than as a form of postmortem analysis after the program has completed execution [34]. In JIVE, every point on the sequence diagram is correlated with the object diagram that would have

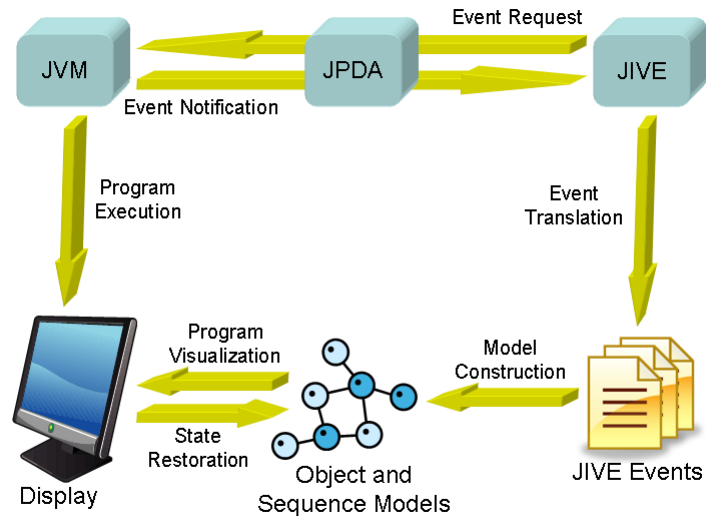


Figure 1: Overview of the JIVE architecture.

been in effect at that execution point. JIVE also supports interactive forward as well as reverse stepping of the program. Through the sequence diagram, a user may direct the JIVE engine to any previous point in the execution history in order to inspect the object diagram at that execution point.

3.2 Runtime Contexts

In this section, we explain the key components of JIVE’s object diagram, in particular the three types of contexts: object, method, and static contexts. These contexts play an important role in the declarative query language to be defined in Section 4.

An object serves as an environment within which method activations occur. When a method is invoked on an object, the method has access to some of the object’s fields. The fields that it can access depend on where it is defined. All fields of the defining class are accessible to the method. In the case of inheritance, the method can also access any field declared in a class from which its defining class inherits.¹

In JIVE, an object is conceptualized as a set of nested *contexts*, each of which contains a subset of the object’s fields. The innermost context corresponds to the class from which the object was instantiated. Each successively enclosing context corresponds to the class from which the enclosed class inherits. For example, if an object o is an instance of class C and if C inherits from class B which inherits from class A , then o will consist of three contexts: a , b , and c , with context c being the innermost context. Furthermore, context a , b , and c will contain only the variables declared in class A , B , and C respectively. We refer to these contexts as *object contexts*.

¹The presence of certain access modifiers may restrict which fields are accessible.

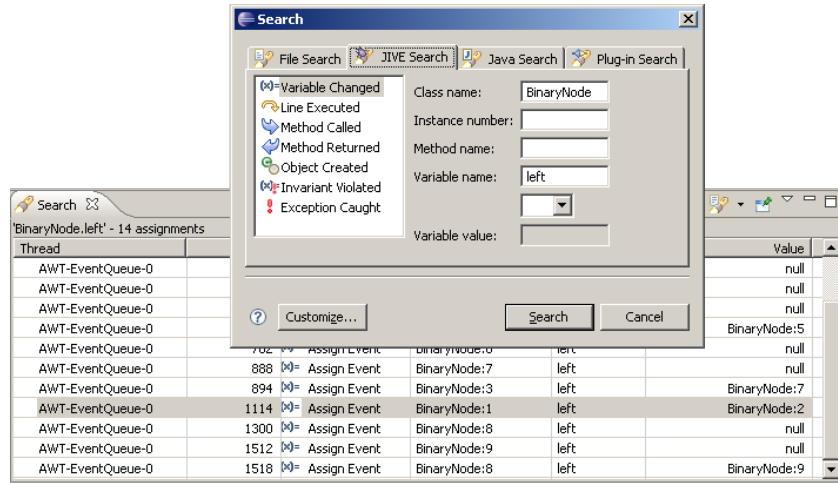


Figure 2: JIVE’s search dialog and result table.

Suppose that method m is defined in class B and is invoked on the above-mentioned object o . The resulting method activation m should be placed inside object context b (but outside context c) to clarify that it can only access fields in contexts b and a . We refer to this activation as a *method context*, and it contains the formal parameters and the local variables. In Java, method contexts cannot be nested because the language does not support nested methods.

We can extend the concept of contexts to static fields and methods. A static method is invoked on a class rather than on an object. A static method can only access the static fields of its defining class and superclasses. Suppose that we invoke a static method s defined in class C . The resulting method context s can access the static fields of classes C , B , and A . These three classes would have *static contexts* A , B , and C which contain the static fields of their respective classes. These contexts are nested to clarify scoping analogous to the nesting of object contexts. The method context s is nested within the static context C .

3.3 Declarative Debugging

In our approach to declarative debugging, the program’s execution history can be searched using an extensible set of queries. Queries are formulated using the source code or the diagrams, and the results are reported visually as diagram annotations and also summarized in a tabular or tree format. Each match in the search result corresponds to an event that occurred during the program’s execution. An event can be visualized as a point on the sequence diagram from which a corresponding program state (i.e., object diagram) can be easily revisited by JIVE. The programmer may then step through the execution history in either direction in order to uncover the cause of the error.

Currently, the following set of queries are supported by JIVE: when did a variable change; when was a particular source code line executed; when was a given method called or returned; when was an object created; when was an invariant violated; and when was an exception caught. Figure 2 shows how queries are formulated using a simple search dialog box. The user selects a query and

provides the necessary input values. For some queries, conditions can be placed on the values of certain attributes, e.g., the variable’s value is greater than some specified value. The outcome of a query is often a set of results rather than a single result. Each result corresponds to an event that occurred during the program’s execution. The result set is presented in a tabular format and each event in the table is linked with a corresponding point on the sequence diagram. This is illustrated in Figure 3 as red markers on the sequence diagram. The table can be used to easily navigate through the sequence diagram from one result to the next. Additionally, navigating to an event will result in revisiting the program state at the time of the event, i.e., the object diagram existing at that state will be recreated. The user may then step forward or backward through the execution history to examine other states.

JIVE’s visualization and declarative debugging capabilities have been tested on several moderately large Java programs. The query capabilities are very useful in efficiently navigating through the execution history and pinpointing the location of errors. They also greatly aid in limiting the size of object and sequence diagrams by focusing the user’s attention at the points of errors.

4 Query Language for Declarative Debugging

The search interface currently provided by JIVE supports a small set of commonly needed queries. In this section we present a more comprehensive query language for declarative debugging. This language, called JQL (for JIVE Query Language), has an SQL-like syntax and is founded on a set of base event and interval relations, which will be explained further below. Our goal is to expand the search interface by incorporating these capabilities in a more succinct manner. JQL’s implementation and performance will be studied as part of our proposed research.

JQL queries are used to retrieve tuples of the base relations (and optionally their components) that satisfy certain conditions. The basic form of a JQL query is:

```
RANGE    rangeList
RETRIEVE retrieveList
WHERE    condition
```

where *rangeList* declares a list of variables and their associated event or interval relations, *retrieveList* declares a subset of the variables from the RANGE clause or their attributes, and *condition* is a boolean expression over the attributes of the variables declared by the RANGE clause. Note that the result of a query is the empty set if the condition in the WHERE clause is not satisfied. Also, the RETRIEVE clause permits the specification of a tuple of events or their attributes.

4.1 Event Relations

The execution of a Java program under the JIVE framework results in the generation of certain key events pertaining to variables, methods, and objects that are crucial to constructing the object and sequence models. The sequence of events generated effectively constitute the execution history of the program. JIVE events are based upon primitive events from the Java Debug Interface (JDI), which were described in section 3.1. The JIVE events include when a class was loaded, an object was created, a variable was assigned, a method was called/returned, and an exception was thrown/caught. These events contain all the necessary information that could serve as a basis for a declarative query language. We therefore formulate a set of event relations from the above JIVE events, as follows:

```

load(number, thread, activation, class, superclass)
assign(number, thread, activation, context, variable, rvalue)
new(number, thread, activation, objcontexts, enclosing)
call(number, thread, activation, caller, callee, formals, rvalues)
return(number, thread, activation, returner, rvalue)
throw(number, thread, activation, thrower, exception)
catch(number, thread, activation, catcher, variable, exception)

```

An *event relation* specifies a relationship between various entities: variables, objects, methods, threads, etc. Appendix A shows the event relations generated during the execution of a simple Java program. Each event is identified by an event number which specifies the sequence in which the event occurred relative to other events. Each event also specifies the method activation and thread on which it occurs.

As described in section 3.2, JIVE has three types of contexts: static, object, and method contexts. A static context corresponding to some class C is identified by its fully-qualified name.² An object context is identified as $C : n$, where C is the fully-qualified name of the corresponding class and n is a unique instance number for the object. A method context is identified as $C\#m : n$, where C is the enclosing object or static context, m is a method identifier, and n is an invocation number.

In the above relations, all fields are contexts with the exception of *number*, *variable*, *formals*, and (in some cases), *rvalue*. For example, *class* and *superclass* are static contexts; *objcontexts*, *enclosing*, and *exception* are object contexts; and *activation*, *caller*, *callee*, *returner*, *thrower*, and *catcher* are method contexts.

Variables—whether fields, parameters, or local variables—are realized at runtime as attributes of a context. In the case of a static context the variable is a static field; for an object context it is an instance field; and for a method context it is either a parameter or local variable. Since contexts are uniquely identifiable, a variable is uniquely identified by its name and its containing context. Variables may be assigned to either primitive values or object references. We use the term *rvalue* to refer to either of these two cases; ‘*rvalue*’ refers to the value that is permissible on the right-hand side of an assignment statement. JIVE events represent a primitive value by its string encoding and an object reference by the context identifier of its innermost context.

While the entities discussed thus far have been run-time entities, their identifiers are decomposable into both static (compile-time) and dynamic (run-time) components. The static components originate from their source code-level identifiers, e.g., class, method, and variables names. The dynamic components take the form of instance and invocation numbers. In our proposed query language, portions of these identifiers may be replaced by a wildcard character ($_$) or regular expression in order to match not only a single entity but instead an entire class of entities.

Example 1. Find all events where local variable x in method m of class C was assigned `null`.

```

RANGE      AssignEvent e
RETRIEVE  e
WHERE      e.context = 'C':_ # 'm':_ AND
           e.variable = 'x' AND
           e.rvalue = null

```

In the above query, the first conjunct of the **WHERE** clause specifies that the context for variable x is a method context for method m of class C . Note that **context**, **variable**, and **value** are fields of

²Here we assume only a single class loader is used, and therefore a class will only be loaded once.

the *assign* event relation introduced earlier.

Example 2. Find all events where static method $f(m,n)$ of class C was called with a negative value for parameter n .

```
RANGE      CallEvent e
RETRIEVE  e
WHERE      e.callee = 'C' # 'f':_ AND
           'n' ∈ e.formals AND
           e.rvalues('n') < 0
```

The conjunct $'n' ∈ e.formals$ checks whether n is a formal parameter of f and the term $e.rvalues('n')$ extracts the value passed to parameter n of method f .

Example 3. A function f is monotonic if $f(x) < f(y)$ whenever $x < y$. Find all call events of f violating this property.

```
RANGE      CallEvent c1, c2; ReturnEvent r1, r2
RETRIEVE  <c1, c2>
WHERE      c1.callee = r1.returner AND
           c2.callee = r2.returner AND
           c1.callee = 'MyMath' # 'f':_ AND
           c2.callee = 'MyMath' # 'f':_ AND
           c1.rvalues('n') < c2.rvalues('n') AND
           r1.rvalue >= r2.rvalue
```

The first two conjuncts of the WHERE clause identify a pair of matching *call* and *return* events; the next two conjuncts specify that these calls must pertain to function f ; and the last two conjuncts specify the violation criteria.

4.2 Interval Relations

An event relation holds between specific entities at the point of occurrence of the event. Using these relations we may define higher-level relations that hold over a sequence of consecutive events. For example, an *assign* event results in a *value* relation holding between a variable and its assigned value over an interval of consecutive events, i.e., until the variable is assigned another value. We refer to these higher-level relations as *interval relations*. An interval relation holds so long as the occurrence of another event does not nullify the relation. Or, more appropriately, it exists between the given entities during a specific event interval. Each interval relation captures a portion of the runtime state over a sequence of consecutive events and is useful for limiting the scope of a query to a particular state or range of states. We define the following three interval relations:

```
enclose(start, end, outer, inner)
invocation(start, end, caller, callee, thread)
value(start, end, context, variable, rvalue)
```

In the above relations, *start* and *end* are event numbers that delimit the interval. It should be noted that *start* is inclusive while *end* is exclusive, and this is denoted by $[start, end)$.

The *enclose* relation specifies that context *inner* is nested within context *outer* during a specified event interval. An *enclose* relation holds between two static contexts over an interval starting with a *load* event and until program completion. This relation holds between two object contexts over

an interval starting with a *new* event and until program completion. This relation holds between an object (or static) context and a method context over an interval starting with a *call* event and ending with a *return* or *throw* event.

An *invocation* relation holds between two method contexts, *caller* and *callee*, over an interval starting with a *call* event and ending with a *return* or *throw* event. The relation also identifies the thread (object context) on which the invocation occurred.

The *value* relation specifies that a *variable* in a given *context* holds an *rvalue* over an event interval starting with an *assign* event for the variable and ending with the next *assign* event for the same variable. For the case of local variables and parameters, the ending event can also be either a *return* or *throw* event.

Example 4. *Find all event ranges during which variable v of class C has a value of null.*

```
RANGE      ValueInterval v
RETRIEVE  v
WHERE      v.context = 'C':_ AND
           v.variable = 'v' AND
           v.rvalue = null
```

The result of this query is a set of intervals which in turn could be used to limit the scope of other queries.

Example 5. *Find all event ranges during which a circular list of size 2 exists, assuming that lists are built from a class ListNode with element and next fields.*

```
RANGE      ValueInterval v1, v2
RETRIEVE  <v1, v2>
WHERE      v1.context = 'ListNode':_ AND
           v2.context = 'ListNode':_ AND
           v1.context != v2.context AND
           v1.context = v2.rvalue AND
           v2.context = v1.rvalue AND
           v1.variable = 'next' AND v2.variable = 'next' AND
           v1.start <= v2.start AND v2.start <= v1.end
```

The first three conjuncts of the above WHERE clause identify two distinct ListNodes; the next three conjuncts checks for the circularity condition through the field *next*; and the last two conjuncts ensure that the two list nodes are referencing each other during a common event interval. In this example, the result of the query is a set of pairs of *value* intervals satisfying the desired criteria.

Example 6. *Find all instances of a concurrent update of a field in class C by method m on two threads.*

```

RANGE      InvocationInterval i1, i2; AssignEvent e1, e2
RETRIEVE <e1, e2>
WHERE      i1.thread != i2.thread AND
           i1.start < i2.start AND i2.start < i1.end AND
           i1.callee = 'C':N#'m':_ AND i2.callee = 'C':N#'m':_ AND
           e1.thread = i1.thread AND
           e2.thread = i2.thread AND
           e1.variable = e2.variable AND
           e1.context = e2.context AND
           i2.start < e1.number AND e1.number < min(i1.end, i2.end) AND
           i2.start < e2.number AND e2.number < min(i1.end, i2.end)

```

The first three conjuncts identify method invocations on two threads that overlap each other. The next two conjuncts specify that the method invocations for `m` occur on the same object—note that the variable `N` stands for the object instance number. The next four conjuncts identify assignments to a common variable (i.e., having the same name and same object or static context) on these two threads; and the remaining conjuncts specify that the assignments occur over a common event interval. We will show an approach to a more succinct formulation in the next section.

The results of the queries in the foregoing examples are either events or event intervals. In both cases, the results could be displayed by JIVE by suitably annotating the sequence diagram.

5 Proposed Research

Scalable Visualizations We propose to develop algorithms for reduced object and sequence diagrams with the overall goal of obtaining efficient, scalable, and aesthetic visualizations. In general, there could be a large number of objects and method activations arising during program execution. Broadly speaking, there are three approaches to obtaining compact diagrams: filtering, abstracting, and zooming. Filtering involves eliminating entire sub-diagrams from being displayed. The results of queries serve as a good basis for filtering, in that they help depict only those portions of the object and sequence diagrams that are relevant to the debugging task of the programmer. Abstracting is the suppression of internal details of objects and their interactions. These techniques include suppressing superclass contexts, hiding the fields of objects and method activations, showing only objects involved in the call path, etc. Finally, a zooming out operation is also very useful in obtaining compact presentation of object and sequence diagrams, and it complements filtering and abstraction. These techniques are applicable to both object and sequence diagrams.

With respect to object diagrams, we are interested in upward, centered, layered drawings with minimal area and line crossings. By upward we mean that most of the object references are directed from top to bottom; by centered we mean that the child (aggregated) objects are centered below the parent (aggregator) object; and, by layered we mean that all children of the same parent should be placed on the same level. However, it has been shown recently that drawing object diagrams while preserving these layout properties is NP-hard [11]. Typically, when the graph structure has cycles, the back edges that create the cycles are first removed in order to obtain a directed acyclic graph, then a layout is obtained, and finally the back edges are inserted back into the diagram.

It has also been shown recently that a purely graph-theoretic treatment of the layout of object diagrams tends to produce layouts that are not always aesthetic [10]. It is important to combine both program-specific criteria together with the above graph-theoretic criteria. For example, knowing

the recursive class structure facilitates the design of better layouts for object diagrams. In the JIVE context, often the object sub-diagram to be drawn is determined as the result of a declarative query, and hence the number of objects to be drawn is much less than the full object diagram. This permits us to use more sophisticated techniques, which may have higher asymptotic complexity, without adversely impacting interactive performance. Hence we propose to explore algorithms for obtaining aesthetic drawings taking into account the recursive class structure.

With respect to sequence diagrams, optimal placement of object lifelines (i.e., minimizing awkward line crossings due to method calls) has been shown to be NP-hard [10]. We propose to explore how to obtain reduced sequence diagrams in which sub-computations corresponding to large call trees are replaced by a single, abstract node. This in turn helps compact the timeline and obtain a concise drawing. In general, we are interested in both ‘vertical’ and ‘horizontal’ compaction of the sequence diagram. Vertical compaction entails reducing the timeline while horizontal compaction involves eliminating unnecessary object lifelines from being displayed. As with object diagrams, the result of a query helps focus the user’s attention on a sub-sequence diagram which limits the number of objects as well as method activations to be displayed. We are exploring a hierarchic technique for sequence diagram compaction taking into account program-specific and graph-theoretic properties.

Interactive Execution Another important component of our debugging environment is the support for both forward and reverse interactive execution. As noted earlier, reverse stepping (or state rewinding) is very valuable for debugging since a user often realizes that an error has occurred only after the errant statement has been executed. JIVE currently performs incremental state-saving during forward execution and performs incremental state-restoration during reverse stepping. A state change could be a modification in the value of a variable or the addition or removal of an object or method activation. While incremental state-saving incurs minimal storage overhead and is efficient for forward execution, it is inefficient for performing jumps back to distant runtime states because each state must be restored one-by-one until the desired state is reached. Query results may also refer to points in the execution history, necessitating jumps to disparate execution states.

We are exploring an approach to efficient reverse jumping through a system of checkpoints. An ‘incremental checkpoint’ records changes in program state after each step in execution. This is the current mechanism used in JIVE. A ‘full checkpoint’ contains all the information necessary to recreate a program state. Such checkpoints should be used infrequently in order not to slow down execution excessively. As an intermediate approach, we are exploring the idea of a ‘differential checkpoint’ which will record changes in program state since the last differential or full checkpoint. Given a sequence of incremental checkpoints c_1, \dots, c_k , the differential checkpoint after c_k is the set $\{c_1, \dots, c_k\}$ but excluding any c_i for which there is a $j > i$ such that c_j assigns the same variable as c_i . Note that incremental checkpoints will also record the addition and removal of contexts. Given a sequence of differential checkpoints d_1, \dots, d_k , the full checkpoint after d_k is $d_1 \sqcup \dots \sqcup d_k$, where $x \sqcup y$ is the union of x and y but excluding any incremental checkpoint c_i for which there is a later checkpoint that assigns the same variable.

A key issue with differential checkpoints lies in deciding at what execution events and at what intervals these checkpoints should be taken. A natural choice would be at the end of each method call. However, object-oriented programs engender the use of many small methods, and hence it may be more appropriate to choose the interval size based upon the number of variables that changed during the interval. Through experimentation we propose to arrive at suitable intervals for taking differential checkpoints, and also to quantify the space-time overhead incurred by our checkpointing

methodology.

Finally, we would also like to explore a combined static and dynamic analysis of Java programs to facilitate reverse jumping. For example, a method activation that does not modify any object state can be jumped over during reverse stepping without traversal through each of the incremental checkpoints that occur within the activation. The detection of such methods can benefit from both compile-time and run-time analysis since it requires the consideration of nested method calls. The presence of multiple threads further complicates this analysis in that we must ensure that the overlapping method activations also do not modify object state. We believe that efficient reverse jumping will benefit from such static and dynamic analysis of programs.

Query Language We are evaluating the introduction of two useful relations, $p \rightarrow q$ and $p \rightarrow^* q$, applicable to objects, method calls, and variable assignments.

- For two objects, o_1 and o_2 , the relation $o_1 \rightarrow o_2$ means that o_1 directly references o_2 through one of its fields, and $o_1 \rightarrow^* o_2$ means that o_1 transitively references o_2 through zero or more intermediary objects. For example, $o \rightarrow^* o$ means that object o is part of a cycle. And, $\{x \mid o_1 \rightarrow^* x\}$ defines the set of all objects reachable from o_1 .
- For two method activations, m_1 and m_2 , the relation $m_1 \rightarrow m_2$ means that there is a direct call that links m_1 to m_2 , and $m_1 \rightarrow^* m_2$ means that m_1 transitively calls m_2 through zero or more intermediary calls. For example, $\text{Driver\#main:1} \rightarrow \dots \rightarrow m_k$ defines a calling path from method `main` to m_k . And, $\{x \mid m_1 \rightarrow^* x\}$ defines the set of all calls in the call tree rooted at m_1 .
- For two assignments events, a_1 and a_2 to variables v_1 and v_2 respectively, the relation $a_1 \rightarrow a_2$ means that the rvalue of v_1 is used to define v_2 without any intervening assignment to v_1 between a_1 and a_2 , and the relation $a_1 \rightarrow^* a_2$ means that the rvalue of v_1 is transitively used to define v_2 in terms of zero or more intermediary assignments. This relation is useful for dynamic flow analysis of variables.

Another useful capability that we are exploring is the `WHEN` clause, which helps to narrow down the temporal scope of the `WHERE` clause and thereby facilitates a more efficient search. The `WHEN` clause specifies an event interval over which the `WHERE` clause is tested. An interval may be specified through a condition on the program state, and intervals may be composed through operations such as union, intersection, and difference. Consider the following example:

Example 7. *Find all events where method `insert` of class `Queue` is called when the head of the queue is null.*

```

RANGE   ValueInterval i, CallEvent e
RETRIEVE e
WHERE   e.callee = 'Queue':N#'insert':_
WHEN    i↓[i.context = 'Queue':N AND
         i.variable = 'head' AND
         i.value = null]

```

We are considering the use of a `↓` operator which will determine a subset of intervals that satisfy a specified condition. In the above example, the condition refers to states of a `Queue` when its head

is `null`. The `WHEN` clause thereby determines a smaller range of events over which the `WHERE` clause is tested. The use of variable `N` in both the `WHERE` and `WHEN` clauses ensures that the call event for `insert` occurs within the context of a `Queue` object whose head is `null`.

The next example illustrates interval composition through an intersection operation and is a more succinct formulation of the query in Example 6.

Example 8. *Find all instances of a concurrent update of a field in class `C` by method `m` on two threads.*

```

RANGE      InvocationInterval i1, i2; AssignEvent e1, e2
RETRIEVE  <e1, e2>
WHERE      i1.thread != i2.thread AND
           e1.thread = i1.thread AND
           e2.thread = i2.thread AND
           e1.variable = e2.variable AND
           e1.context = e2.context AND
WHEN      i1↓[i1.callee = 'C':N#'m':_] ∩ i2↓[i2.callee = 'C':N#'m':_]

```

The above is a sketch of a possible approach to this problem. The expression `i1↓[i1.callee = 'C':N#'m':_]` specifies a subset of the intervals of `i1` satisfying the restriction that the invocation is for method `m` of class `C`. The expression `i2↓[i2.callee = 'C':N#'m':_]` specifies a similar restriction on `i2`. The intersection operator `∩` joins the resulting subsets of intervals on the variable `N` and limits each matching pair of intervals to a common start and end range.

We have provided a preliminary description of a declarative query language (JQL) for program debugging, and we propose to refine and finalize the language as part of our proposed research. This language has a strong temporal flavor and is related to other temporal query languages that are based on event and interval relations [32], etc. However, there are important differences arising due to our focus on program debugging. A basic property of our approach is that the program state at event number n can be defined as all instances of interval relations such that n falls within the event interval of the relation. We plan to formalize the semantics of JQL as part of our proposed research.

We also plan to explore implementation techniques and performance analysis of declarative debugging. The implementation of JQL queries necessitates an efficient iteration over the set of event and interval relations specified in the `RANGE` clause. The conjuncts in the `WHERE` clause need to be processed in a suitable order to achieve maximal reduction in the number of tuples that need to be processed. As noted earlier, the `WHEN` clause also facilitates efficient iteration over interval relations. The implementation of JQL also involves translating the events generated by the JVM into a set of base relations as described in Section 4. The efficiency of query processing depends on the data structures that are used to represent the execution history. While a sequence is a natural representation for the events that arise during debugging, the call tree structure together with an indexable table of events would facilitate more efficient query processing. Aggregate queries also require a more extensive traversal over the execution history.

Proposed Outline of Dissertation The following are the tentative chapters of the dissertation:

1. Introduction
2. Related Work
3. Overview of JIVE

4. Query Language for Debugging
5. Scalable Visualizations
6. Efficient Reverse Execution
7. Implementation and Performance Analysis
8. Conclusions and Contributions

References

- [1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26, 1991.
- [2] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Soft.—Prac. & Exper.*, 23(6):589–616, June 1993.
- [3] Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999.
- [4] Alvin Cheung and Samuel Madden. Performance profiling with EndoScope, an acquisitional software monitoring framework. *Proc. VLDB Endow.*, 1(1):42–53, 2008.
- [5] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45:2002, 2002.
- [6] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 326–337, September 1993.
- [7] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, April 1998.
- [8] Mireille Ducassé. Coca: An automated debugger for C. In *21st IEEE ICSE*, pages 504–513, May 1999.
- [9] Mireille Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic Programming*, 1999.
- [10] Ashim Garg, Paul V. Gestwicki, and Bharat Jayaraman. Interactive program visualization and graph drawing. In *Discrete Mathematics and Its Applications*, pages 36–52, December 2004.
- [11] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001.
- [12] Paul V. Gestwicki and Bharat Jayaraman. Interactive visualization of Java programs. In *Proceedings of the IEEE 2002 Symposium on Human-Centric Computing, Languages, and Environments (HCC '02)*, pages 226–235, September 2002.
- [13] Paul V. Gestwicki and Bharat Jayaraman. Methodology and architecture of JIVE. In *SoftVis*, pages 95–104, 2005.

- [14] Hani Z. Girgis, Bharat Jayaraman, and Paul V. Gestwicki. Visualizing errors in object-oriented programs. In *OOPSLA '05 Companion*, pages 156–157, 2005.
- [15] Simon Goldsmith, Robert O’Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402, October 2005.
- [16] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2004)*, pages 42–55, October 2004.
- [17] T. Dean Hendrix, II James H. Cross, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. *SIGCSE Bull.*, 36(1):387–391, 2004.
- [18] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE 97)*, pages 360–370, May 1997.
- [19] Andrew J. Ko and Brad A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *CHI*, pages 151–158, April 2004.
- [20] Michael Kölling and John Rosenberg. Guidelines for teaching object-orientation with Java. *ACM SIGCSE Bulletin*, 33(3):33–36, 2001.
- [21] Ralf Kollmann and Martin Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *Proc. CSMR '01*, page 58, 2001.
- [22] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, May 1997.
- [23] Julia L. Lawall and Gilles Muller. Efficient incremental checkpointing of Java programs. *Dependable Systems and Networks, International Conference on*, 0:61, 2000.
- [24] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. *SIGPLAN Not.*, 32(10):304–317, 1997.
- [25] Bil Lewis. Debugging backwards in time. In *Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 225–235, 2003.
- [26] Bil Lewis and Mireille Ducassé. Using events to debug Java programs backwards in time. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 96–97, New York, NY, USA, 2003. ACM.
- [27] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, pages 365–383, October 2005.
- [28] National Institute of Standards and Technology (NIST). The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, May 2002.

- [29] Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE '00)*, pages 80–89, November 2003.
- [30] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 535–552, New York, NY, USA, 2007. ACM.
- [31] Richard Sharp and Atanas Rountev. Interactive exploration of UML sequence diagrams. In *VisSoft*, pages 8–13, 2005.
- [32] Richard Snodgrass. The temporal query language TQuel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.
- [33] Sudarshan M. Srinivasan, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In Proceedings of the 2004 USENIX Technical Conference*, pages 29–44, 2004.
- [34] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba—an environment for reverse engineering Java software systems. *Soft.—Prac. & Exper.*, 31(4):371–394, 2001.
- [35] The Eclipse Foundation. Eclipse Platform. <http://www.eclipse.org/>.

A Event Trace

```

load(1, java.lang.Thread:1, <system>, java.lang.Object, null)
load(2, java.lang.Thread:1, <system>, BST, java.lang.Object)
call(3, java.lang.Thread:1, <system>, system, BST#main:1, [[]])
load(4, java.lang.Thread:1, BST#main:1, AbsTree, java.lang.Object)
load(5, java.lang.Thread:1, BST#main:1, Tree, AbsTree)
new(6, java.lang.Thread:1, BST#main:1, [Tree:1, AbsTree:1, java.lang.Object:1], null)
call(7, java.lang.Thread:1, BST#main:1, BST#main:1, Tree:1#<init> :1, [100])
call(8, java.lang.Thread:1, Tree:1#<init>:1, Tree:1#<init>:1, AbsTree:1#<init>:1, [100])
assign(9, java.lang.Thread:1, AbsTree:1#<init>:1, AbsTree:1, value, 100)
assign(10, java.lang.Thread:1, AbsTree:1#<init>:1, AbsTree:1, left, null)
assign(11, java.lang.Thread:1, AbsTree:1#<init>:1, AbsTree:1, right, null)
return(12, java.lang.Thread:1, AbsTree:1#<init>:1, AbsTree:1#<init>:1, <void>)
return(13, java.lang.Thread:1, Tree:1#<init>:1, Tree:1#<init>:1, <void>)
call(14, java.lang.Thread:1, BST#main:1, BST#main:1, AbsTree:1#insert:1, [50])
call(15, java.lang.Thread:1, AbsTree:1#insert:1, AbsTree:1#insert:1, Tree:1#createNode:1, [50])
new(16, java.lang.Thread:1, Tree:1#createNode:1, [Tree:2, AbsTree:2, java.lang.Object:2], null)
call(17, java.lang.Thread:1, Tree:1#createNode:1, Tree:1#createNode:2, Tree:2#<init>:2, [50])
call(18, java.lang.Thread:1, Tree:2#<init>:2, Tree:2#<init>:2, AbsTree:2#<init>:2, [50])
assign(19, java.lang.Thread:1, AbsTree:2#<init>:2, AbsTree:2, value, 50)
assign(20, java.lang.Thread:1, AbsTree:2#<init>:2, AbsTree:2, left, null)
assign(21, java.lang.Thread:1, AbsTree:2#<init>:2, AbsTree:2, right, null)
return(22, java.lang.Thread:1, AbsTree:2#<init>:2, AbsTree:2#<init>:2, <void>)
return(23, java.lang.Thread:1, Tree:2#<init>:2, Tree:2#<init>:2, <void>)
return(24, java.lang.Thread:1, Tree:1#createNode:1, Tree:1#createNode:1, Tree:2)

```

```
assign(25, java.lang.Thread:1, AbsTree:1#insert:1, AbsTree:1, left, Tree:2)
return(26, java.lang.Thread:1, AbsTree:1#insert:1, AbsTree:1#insert:1, <void>)
call(27, java.lang.Thread:1, BST#main:1, BST#main:1, AbsTree:1#insert:2, [150])
call(28, java.lang.Thread:1, AbsTree:1#insert:2, AbsTree:1#insert:2, Tree:1#createNode:2, [150])
new(29, java.lang.Thread:1, Tree:1#createNode:2, [Tree:3, AbsTree:3, Object:3], null)
call(30, java.lang.Thread:1, Tree:1#createNode:2, Tree:1#createNode:2, Tree:3#<init>:3, [150])
call(31, java.lang.Thread:1, Tree:3#<init>:3, Tree:3#<init>:3, AbsTree:3#<init>:3, [150])
assign(32, java.lang.Thread:1, AbsTree:3#<init>:3, AbsTree:3, value, 150)
assign(33, java.lang.Thread:1, AbsTree:3#<init>:3, AbsTree:3, left, null)
assign(34, java.lang.Thread:1, AbsTree:3#<init>:3, AbsTree:3, right, null)
return(35, java.lang.Thread:1, AbsTree:3#<init>:3, AbsTree:3#<init>:3, <void>)
return(36, java.lang.Thread:1, Tree:3#<init>:3, Tree:3#<init>:3, <void>)
return(37, java.lang.Thread:1, Tree:1#createNode:2, Tree:1#createNode:2, Tree:3)
assign(38, java.lang.Thread:1, AbsTree:1#insert:2, AbsTree:1, right, Tree:3)
return(39, java.lang.Thread:1, AbsTree:1#insert:2, AbsTree:1#insert:2, <void>)
return(40, java.lang.Thread:1, BST#main:1, BST#main:1, <void>)
```

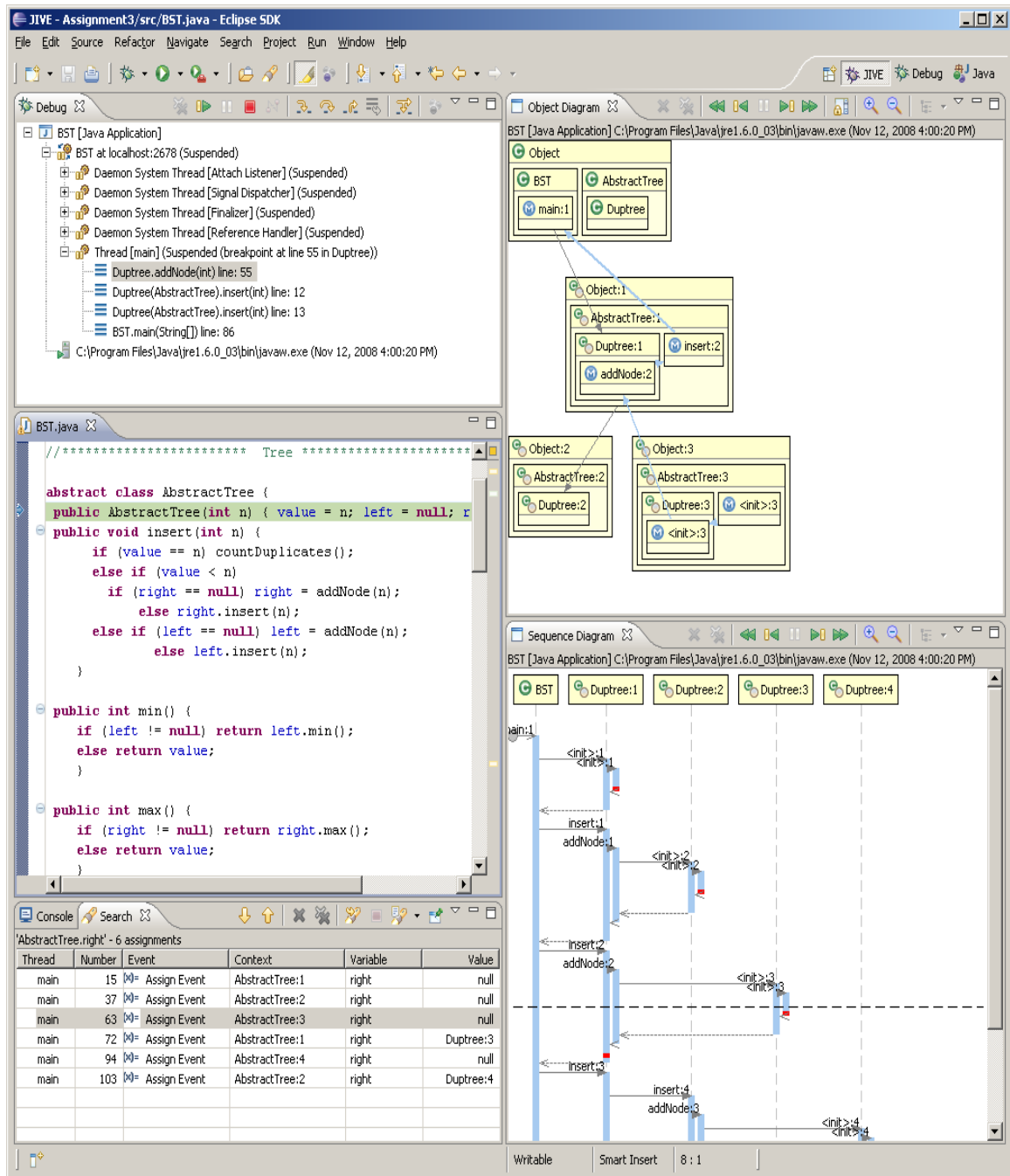


Figure 3: JIVE prototype showing object and sequence diagrams, call stack, source code, and query results.