Visual Queries for Interactive Execution of Java Programs

Hani Z. Girgis, Akshay V. Hegde, Manu Pushpendran,
Paul V. Gestwicki, Bharat Jayaraman
[hzgirgis|ahegde2|mp67|pvg|bharat]@cse.buffalo.edu
Department of Computer Science and Engineering
University at Buffalo, State University of New York
201 Bell Hall, Box 60200
Buffalo, NY 14260-2000

December 13, 2004

Abstract

The behavior of object-oriented programs at runtime can be understood in terms of object states and the history of interaction between objects. Runtime behavior can be very complicated, and information visualization tools are invaluable for program development and debugging. While there has been considerable work in predicting execution patterns through slicing and program analysis, comparatively less has been done on analysis of program execution states and histories. We present a novel technique for visualizing queries over object-oriented program execution. Our technique is built upon the JIVE system of interactive program visualization, which depicts program state in object diagrams and execution history through sequence diagrams. The queries are posed through the graphical interface, and each view of a program (state, history, source code) provides different querying functionality. Furthermore, the results of the queries are shown in the graphical user-interface by annotating the diagrams or inducing new views. Our design allows for queries over variables, methods, objects, classes, and source code. This paper describes the domain of these queries as well as their interfaces, briefly outlines their implementation, and also provides a survey of related work and comparisons.

1 Introduction

Debugging is an unavoidable process during software development. Generally, when running a program or even when using standard debuggers, a bug can only be detected after it has occurred. This fact has spurred the development of various techniques: Program slicing can be used along with tools such as WhyLine in order to determine how specific parts of a program influence certain events [10]. Visualizations, such as those produced by Bloom, can help identify problems such as memory leaks or performance sinks [15]. Snapshot debuggers consider the runtime state and allow for queries to be posed over a specific state [14].

The motivation of our work is similar to those mentioned above: we wish to make debugging easier and more effective, and we wish to enhance a user's understanding of programs, through execution visualization and queries. To this end, we propose a novel technique for posing queries over the program execution history so that the queries and their results are both visual in nature. Our work is built upon the interactive visual execution architecture of JIVE [8], a tool for program visualization whose major features include:

+ multiple views of program state using a visual operational semantics for object-oriented languages [7];

+ visualization of program history through UML sequence diagrams [4];

+ automatic and dynamic layout of diagrams; and

+ interactive execution, which allows both forward and reverse execution of programs.

It is JIVE's capability for interactive execution that enables the instrumentation of the queries we describe.

A JIVE visualization has three basic components: a view of program state through an object diagram; a view of program history through a sequence diagram; and a view of program source code, with dynamic source highlighting during execution. In our development and testing of JIVE, we often posed questions about the program state that are fundamentally related to the multiple-view design of the visualization. For example, an object diagram shows the objects that are currently active, but we often wanted to know when an object was last active, meaning when did it last contain a method activation. It is possible, though perhaps inconvenient, to answer this question by scrolling through the sequence diagram. This process leads to the question of how many times a method is called. Again, the answer is implicit in the execution, and JIVE's reverse execution functionality allows us to answer the question by stepping through all the program states. However, this would be cumbersome, and it suggests that a better interface is needed.

In this paper, we describe a visual interface for queries in the JIVE visualization system. The execution log maintained by JIVE is, essentially, an object-oriented database of execution data. Given such a database, a system for querying is clearly desirable. The interface to such a system must be visual so that it integrates with and complements the existing JIVE architecture. Furthermore, a graphical interface to the query system is context-sensitive (that is, selected items are aware of the queries that can be posed on them). The user does not have to learn a new text-based query language; the experience of querying the program is integrated into the interactive execution and visualization environment.

The remainder of this paper is organized as follows: Section 2 describes our query system in terms of its graphical representation of queries and responses; Section 3 describes the architecture of JIVE's interactive execution, and specifically how queries are enabled by it; Section 4 presents a comparison with closely related work; and Section 5 describes our current status and future research plans for this work.

2 Queries

JIVE supports several different queries on the execution history and runtime state of a program. Some of the possible queries are outlined as an ontology of examples in figure 1. As seen in the ontology, a query can act on a variable, value, method, object or statement. The rich set of queries is made possible through a few primitive query operators that change context depending on the *query subject* in consideration. All queries are posed with respect to the time sequence diagram, object diagram or the source code viewer, and the results are displayed through a combination of them. In the following sections we discuss the features and capabilities of our graphical query system.

Query On	**To Know**
Variable	Its value
Value, Range	Variable(s) with this value
Method	Arguments' values, Number of calls
Statement	Number of executions
Object	Number of instances, Last activity

Table 1: Queries supported through the primary GQAS input interface

2.1 Graphical Query Answer System

The graphical query answer system (GQAS) is subsystem of JIVE that accepts queries graphically and displays the query results graphically as well. The system is composed of input and output interfaces. The input interface is very simple to use, as queries can be posed intuitively. A user can query on values, variables, methods, statements, and objects over a specific time interval. A time interval here refers to a section of the time region that scopes over the entire execution of a program. Table 1 shows some of the queries supported by GQAS. As JIVE depicts a program execution through object diagrams, time sequence diagrams, and source code stepping, the output interface also displays the results using the same methodologies.

There are several queries whose answers require multiple states to be displayed, e.g., *show all changes to a variable during a given time interval.* In such cases, there are two ways of displaying the answer: either in continuous mode or in jump mode. These two modes allow the user to study the results from different perspectives. The continuous mode displays the query results as well as all events that occurred in between the results in a chronological order. The jump mode displays only the query results and ignores all events that occurred in between them. These two modes are similar to the fast-forward and the rewind features in a VCR. The continuous mode is similar to rewinding/fast-forwarding except that all events are shown at normal speed. The jump mode is similar to rewinding/fast-forwarding the VCR without displaying anything on the screen except the events of the query result. The two modes are controlled by four stepping buttons as shown in the JIVE toolbar in the top part of figure 3. The play button (right arrow) steps forward over the specified time interval in a continuous manner. The fast forward button (double right arrow) jumps forward from one point

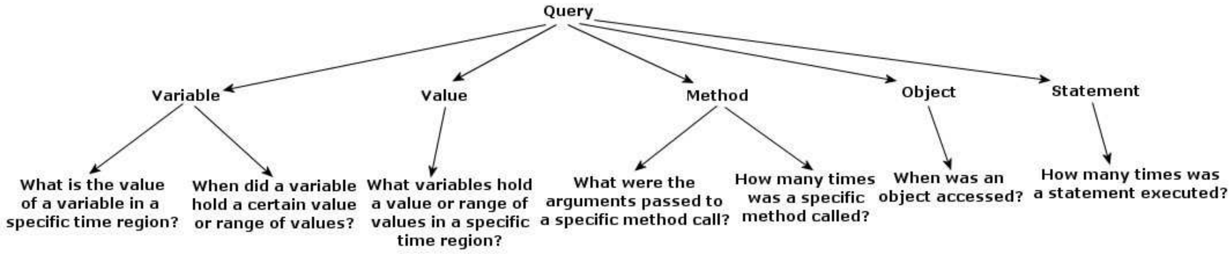

Figure 1: Ontology of query examples supported by JIVE

to the next point within the interval. A point here means one element of the query result. The step back button (left arrow) steps backwards in a continuous manner while the rewind button (double left arrow) jumps backwards to the previous element of the query result.

The queries supported by GQAS can be categorized into four types as follows; queries with respect to time sequence, queries with respect to object diagram, queries with respect to source code, and independent queries. These query types are further explained in the sections below.

2.2 Queries on the Sequence Diagram

A query on a variable, value, and method can be placed in the time sequence diagram. To query the run time state with respect to the time sequence a time interval needs to be specified. The user clicks in the time sequence diagram at the beginning of the time interval of interest, and as the user drags the mouse a rectangle appears and it expands till the user stops dragging the mouse. The rectangle represents the time interval that the user is interested in. When the user right clicks within the rectangle, an options list appears to allow the user to place any of the above mentioned queries. When the user chooses to query by value the list provides a text field to accept the user input. When the user chooses to query by variable or by method, a menu of all objects that are active within the specified time interval appears. The user then selects the object of interest and follows the nested list options to place the desired query.

GQAS outputs the answer for each query on a variable in a separate window. The results are shown as a time sequence diagram that specifies the different values assigned to the variable over the specified time interval. The object diagram section of the window initially shows a snapshot of the program execution corresponding to the value of the variable at the beginning of the time interval. The variable and its value are highlighted in the object diagram. Also, the code segment that assigned the variable the value of interest is highlighted in the source code viewer.

We consider an example query by variable. Figure 2 shows a query on the variable 'value'. 'value' is a data member of class instance 'BST:2'. The answer to this query is shown in figure 3. The object diagram shows the 'BST:2' object maximized and the variable 'value' is highlighted. The time sequence diagram complements this by displaying the method where the value was assigned. In this case the method is the constructor of object 'BST:2'. The source code viewer displays and highlights the line in the constructor where the 'value' was assigned the value of 'v' as shown in figure 4.

When JIVE processes a query on a value over a specific time interval, JIVE generates snapshot of the system for each variable that satisfies the specified value. Now let's consider the case when the query answer is more than one variable, say two variables. The first variable satisfied the desired value at the beginning of the interval and the other one satisfied the value at the end of the time interval. Also assume that the interval is long enough such that no single snapshot of the system can show both variables at once. JIVE displays the query results in multiple windows, one for each variable. Each sequence diagram, highlights the points at which the variable held the desired value by a different color. Also the code segment that resulted in the assignment of the value of interest is highlighted in the source code viewer. This query is immensely useful in answering questions such as *what variables are NULL in my program?*.

To answer the query on argument values of a specific method, JIVE produces a snapshot of the program execution when this method was called. The object diagram shows the highlighted values of the arguments within the method. The user can use the stepping buttons to navigate through the values of the arguments for each call in a chronological order. The source code viewer highlights the method defini-

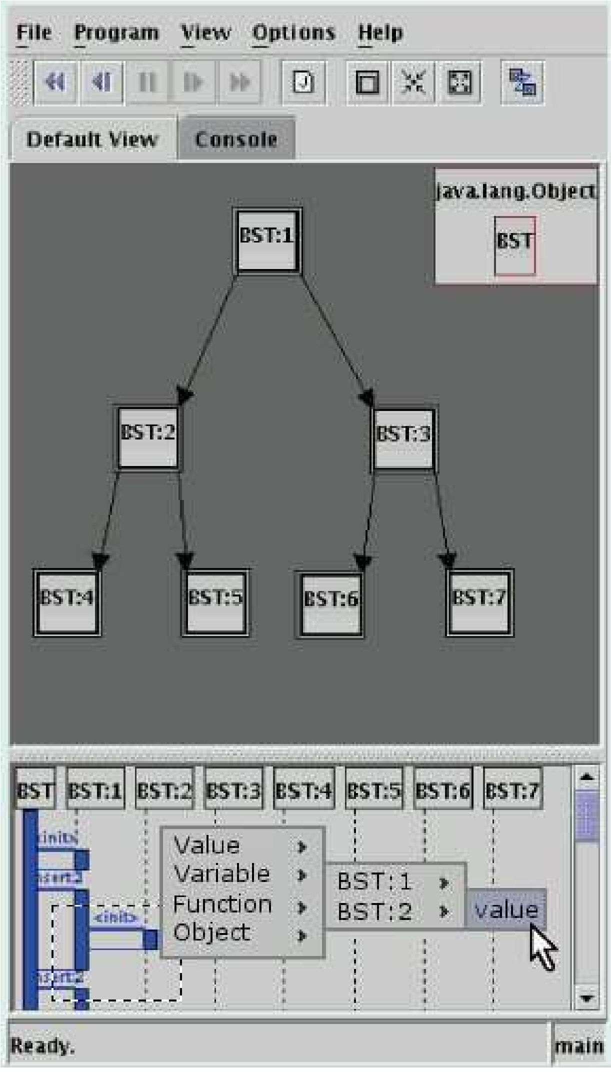

Figure 2: GQAS input interface for query on a variable. The object diagram shows a binary search tree in compact view; the object details are hidden; this view shows program state. The pop-up menu over the sequence diagram shows how variables can be selected over specific time intervals.

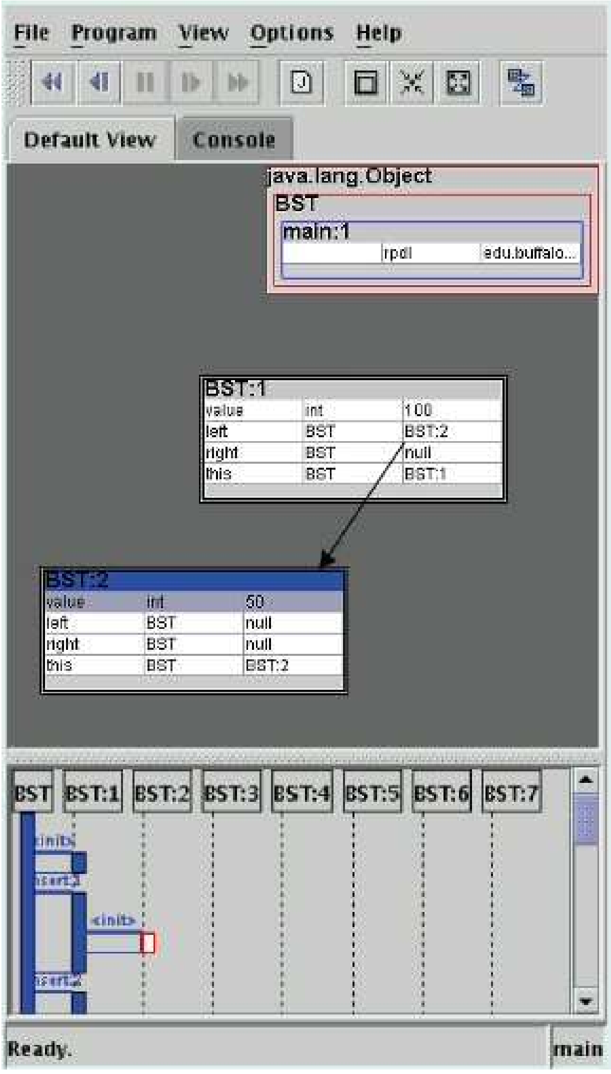

Figure 3: GQAS Output Interface for query on a variable. The red rectangle in the sequence diagram show the result of a query as a method activation, and the blue highlighted object in the object diagram shows the object in which the result holds.

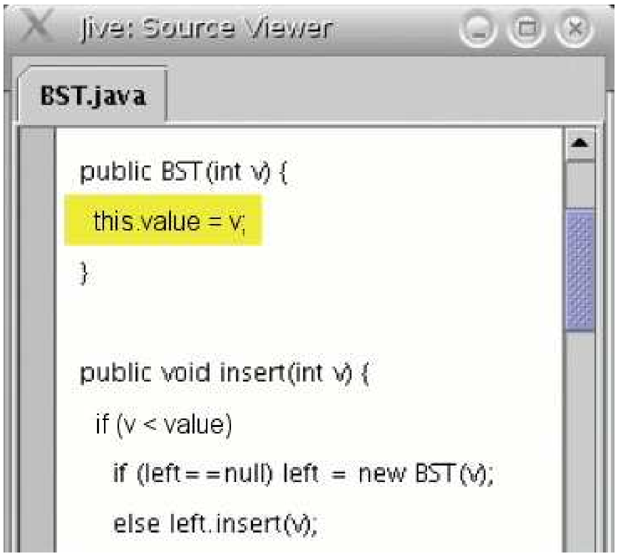

Figure 4: GQAS output interface for query on a variable. The segment of code that corresponds to the assignment of the variable is highlighted

tion. On the time sequence diagram the corresponding method call is highlighted in a different color. This query can be used very effectively for debugging and better comprehending programs involving recursive methods.

The query *when was this object last active?* can be placed by right clicking on the title of an object lifeline in the time sequence diagram. Object titles are located at the very top of object lifelines in the time sequence diagram. The answer to the query is displayed in separate window. The object diagram section shows a snapshot of the system execution when the object was last active. The object diagram of the object under consideration is highlighted. The time sequence diagram shows the last activity of this object by coloring the last method activation in different color. The code segment that corresponds to this activity is highlighted in the source code viewer.

2.3 Queries on the Object Diagram

A query such as *how many instances of this class exist?* is made in the object diagram. To place this query the user right clicks on the object diagram within the static context. JIVE responds to this query by displaying the number of instances in a pop up window. The visual results are shown in a different window. In the time sequence diagram, the object titles of all instances of this class are highlighted. The object diagram section displays a snapshot of the program execution when the first instance was created and its object diagram is highlighted. The code segment that caused the instantiation of this object is highlighted in the source code viewer. The user can use the stepping buttons to navigate through all instances. In the time sequence diagram, the title of the object highlighted in the object diagram section, is surrounded by a red border.

The following example considers a query on the number of instances of 'IntElem' class. The user places the query by right clicking within the static context of the object diagram section of the window as shown in the left part of figure 5. The visual query results are displayed in the right part of the figure. Notice the white titles for 'IntElem:1' and 'IntElem:2'. These results can be interpreted as there are two instances of 'IntElem'. These instances are 'IntElem:1' and 'IntElem:2'. The red border around 'IntElem:1' indicates that the object diagram is displaying the corresponding execution snapshot of class instance 'IntElem:1'.

2.4 Queries on the Source Code

When the user is interested in querying on a statement, GQAS directs the user to select a statement or block of code in the source code. When the user right clicks on the selected statement, an options menu appears to allow the user to place a query on that statement. The answer to the query *how many times was this statement executed?* is displayed as a count in a pop up window, while the visual results are shown in a separate window. JIVE produces a snapshot of the program execution at each point in time where this statement was executed. The user can step through these points via the stepping buttons. Each step highlights the object and corresponding method that contains statement of interest. On the time sequence diagram the point when this statement was executed is highlighted in a different color. Queries on statements are a feature unique to JIVE. They can be extremely useful in analyzing the execution flow of programs involving intricate control structures and loops.

To explain how query on statement works, we consider the following example. Consider the code shown in figure 7. The user is interested in knowing how many times the `else` statement was executed within the recursive method `f`. The method `f` computes the factorial for a given integer. The user can place the query by selecting the `else` statement within the body of the method `f` followed by a right click as shown in figure 7. Figure 6 shows the visual results of the query. These results can be interpreted as fol-

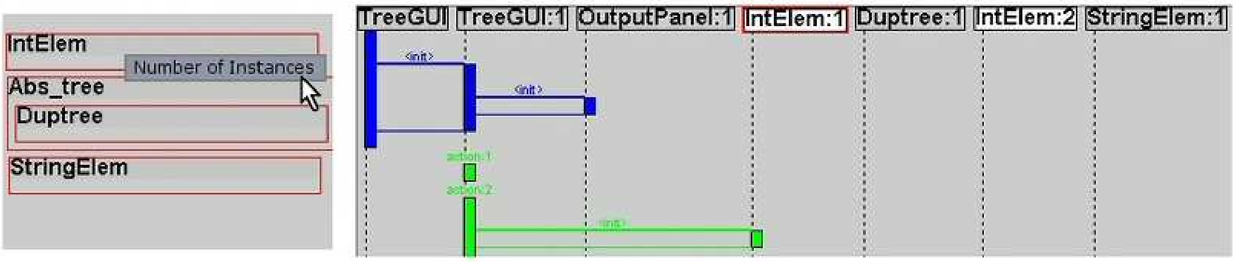

Figure 5: The left diagram is the GQAS Input Interface for the query “how many instances of IntElem”. The query is placed over the static context.The right diagram shows the result for this query. The white object titles are the instances of IntElem

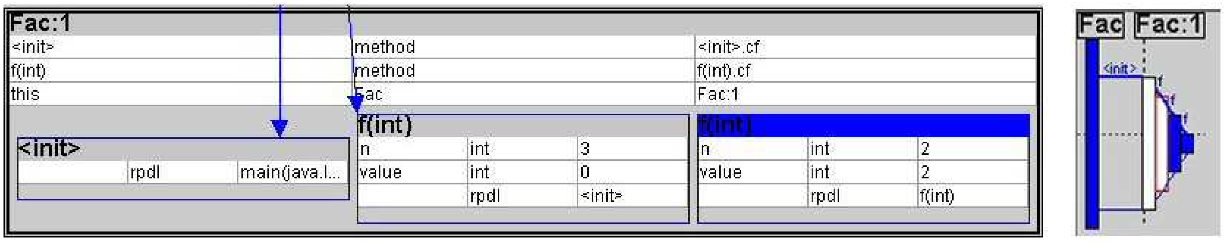

Figure 6: GQAS Output Interface for the results of the query “how many times were this statement executed?”. The left diagram shows that this statement was executed twice within the `f` method via the two white rectangles. The rectangle with the red boundary corresponds to the second time when the `else` statement was executed. The object diagrams on the right side also correspond to the second time of execution

lows; The right part of the figure indicates that the statement was executed twice within the `f` method as indicated by the two white rectangles. The red border around the white rectangle indicates the state under consideration in the object diagram. The left part of figure shows this state in the object diagram with second call of method `f` with blue header. In other words a white rectangle with red borders in the time sequence diagram corresponds to a method with blue header in the object diagram.

2.5 Independent Queries

Independent queries can be defined as queries that do not depend on the sequence diagram, object diagram, or the source code for input or output purposes. JIVE provides a standard interface that works independently from the graphical query answer system. The user can query the runtime state of a program through a standard database interface. The interface allows the user to query by variable or by value and returns the results in a tabular fashion. The user can interact with the interface by clicking on the ”Query” tab or the ”Q” button. In case of query by variable, the result contains all values that were assigned to the variable of interest over the entire execution time of a program. The table also shows the objects that contain these variables. The answer to a query on a value or range of values is also displayed in a tabular format. The table shows each variable and its value and the corresponding object. The interface will be extended to support query by method, statement, and object. Figure 8 shows the interface for the independent queries. The user placed a query on value ’50’. The result of the query reveals that there is one variable ’value’ that has value 50 and it is a data member of the ’BST:2’ object.

By adding the capability to support query on the runtime state, JIVE proves to be a useful educational application. Using the feature of query by value over a specific time interval is helpful to students in early programming classes. When beginner programmers face a null pointer exception, the compiler does not inform which variable is null, but JIVE does. This feature of JIVE helps the student detect the null variable in order to get their program to work. JIVE also informs the user where to look at in the source code by highlighting the code segment that caused a specific value to be assigned to a variable. Further the exact instant when this assignment occurred is also

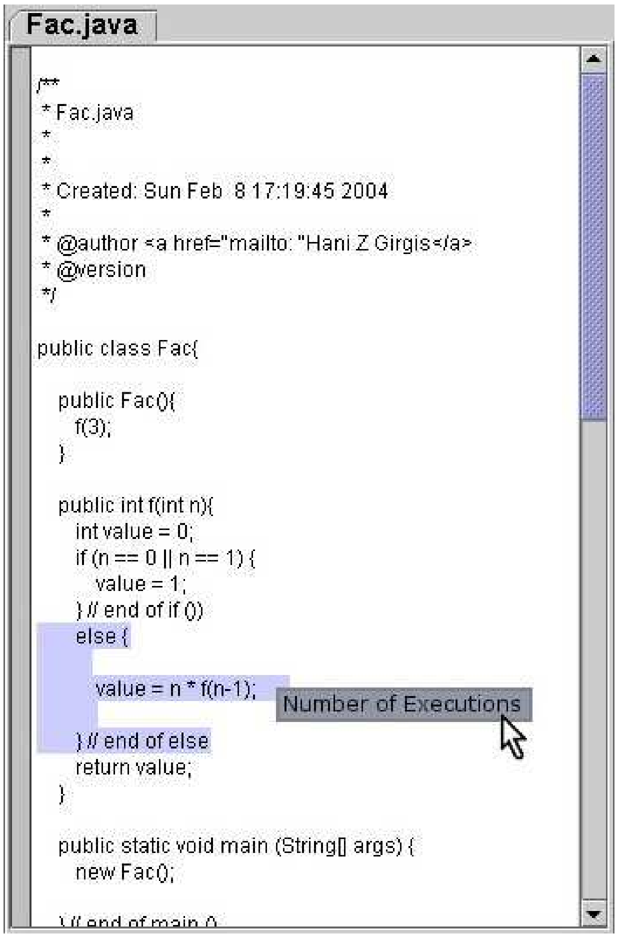

Figure 7: GQAS input interface for query on number of executions of a statement

revealed through a time sequence diagram designed specifically to monitor the changes in this variable alone. The query system of JIVE proves to be a strong tool to understand recursion using query by method and query by statement.

3 Architecture

The query system we have described is a module of JIVE, which is an application that enhances program understanding through interactive program visualization [8]. JIVE allows the user to step forward or backward through execution states. This is a useful debugging functionality since, at runtime, bugs are necessarily found after the erroneous code has executed, and allowing the user to step backwards helps significantly with error identification [1]. In this section, we describe the architecture of JIVE's interactive execution module and how it facilitates the visual queries

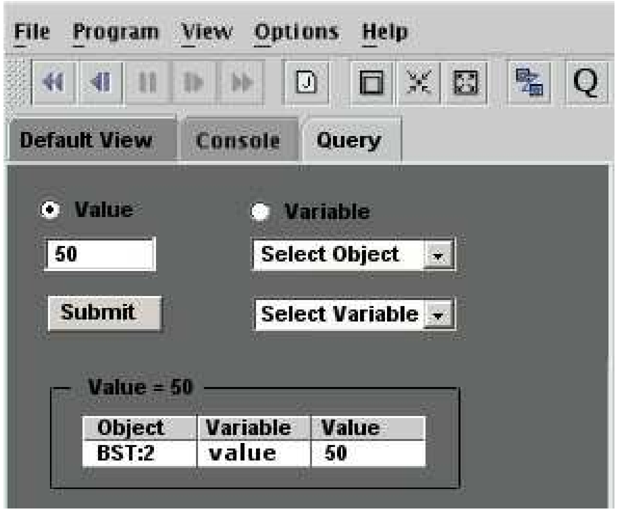

Figure 8: Independent Queries Interface

described above.

It is important to distinguish between JIVE's reverse execution functionality and actual un-execution. Stepping backward in JIVE does not affect the program being visualized (the *client program*), only the current visualization. The client program is suspended while the user inspects past states, and it is resumed when necessary. This model precludes the possibility of altering program inputs during visualization, but it also avoids other problems such as maintaining synchronization with external resources. For example, a program may read from a network stream that does not support push-back and rereading. If JIVE attempted to rewind the actual state of this program to before the stream was read, then the data originally read from the stream would be irrevocably lost. This motivates our current design of state recording, in which “reverse execution” is a rewinding through recorded states, and “forward execution” is either playing back through recorded states, or it involves notifying the client program to resume generating new state data.

3.1 Execution Database

JIVE maintains a single *execution database*, which stores all recorded program changes. The execution database is the data model from which JIVE constructs its sequence diagram view of program history. Any number of *execution models* can be spawned from the execution database. An execution model is a stateful representation of program execution, forming the data model for any specific state's visualization. The execution database does not maintain a complete execution history of a program; it only maintains the information necessary to construct or reconstruct JIVE visualizations, and so many libraries are automatically elided, including the class loaders and garbage collectors.

JIVE reads program state information from an active program via the Java Platform Debugger Architecture [17]. The changes are recorded into the execution database by recording these descriptors of these events, thereby maintaining a declarative history of program execution [16]. The events are stored with a sequence number and the thread on which the event occurred. The recording of each event type is optional, subject to its dependencies, based on the information the user wishes to visualize. By default, JIVE will record the following events:

Static context creation : A *static context* is created for each class that is loaded. The static context is the abstract space where static variables and the activation of static methods occur [7]. This event records the structure of the static context, which comprises of the static variables and methods of a class along with a reference to its superclass.

Object creation : This event records the structure of an object, defined by its instance variables and method declarations along with a reference to its superclass. Inner objects also contain a reference to their enclosing instance or, for static inner objects, their surrounding static context.

Method call : A method call event records the structure of a method, including its parameters and local variables, as well as its corresponding source code, if available. They also record the return point and dynamic link for the method.

Method return : This event references a method activation that has terminated.

Change in source line : This event records a change from one source line to another, and it is stored as a pair of source code references: the old location and the new location. Recording both the old and new location facilitates efficient commit and rollback of this event.

Change in variable value : Variable values are categorized as *primitive*, *internal reference*, or *external reference*. Primitive values are stored as Strings for simplicity of rendering in the visualization. Internal reference values are stored as links to objects that were defined previously with object creation events. External references are links to objects that are not stored in the database, based on the default filters and user-specified preferences. Like primitive values, these are stored as strings that describe the object being referenced. This event records the old and new value of the variable, which enables efficient rollback and commit of the change in the execution model.

Exception caught : Exception handling is recorded with a reference to the exception object, the method in which the exception was thrown, and the method in which the method was caught, if applicable. Exceptions in Java are simply objects with prescribed semantics, and so the exception object itself is treated as any other object.

Each of these events is encapsulated as an object, and each object contains enough information to commit or un-commit itself from the execution log. The event descriptors are essentially the mementos of a Memento pattern [6]. Since the domain is object-oriented program execution, each of these events can also be conceptualized as prototypes of the changes that occur in any single execution model. In this way, the execution database is like a sequential repository of objects for prototype-driven programming [18].

3.2 Query Implementation

The query system is designed so that the queries themselves are separate from the visual interface. Queries are created as objects, which in turn are passed to the execution log. A query object collects the information it requires from the database, and is returned to its source. The graphical front-end then interprets the query in the appropriate section of the user-interface. The graphical interface is a convenient means for posing a query, but it is not a requirement of the query system; queries can be expressed in text or programmatically through other JIVE modules. This has important implications on the extensibility and robustness of our approach. For example, execution data can be serialized, allowing queries to be run offline and without the overhead of visualization. Furthermore, queries can be run on several execution logs in order to compare executions or to build a profile of normal execution patterns.

Maintaining redundant or cached data can improve the performance of querying the runtime database. The interface between queries and the execution database allows for mediators that intercept families of queries. For example, a mediator can observe the execution database and log the number of instances of each class that are created. A query regarding how many instances are created would then be answered in constant time by the mediator rather than requiring linear processing of the entire execution log. Additionally, queries that are bounded in time need only consider the pertinent portions of the execution database. This occurs, for example, when queries are posed over selected portions of the sequence diagram. These optimizations affect the responsiveness of the query system, but they may require redundant storage. Determining precisely which types of queries are worth specific optimization requires a formal analysis of which queries are most often used for categories of programs or groups of users.

The current JIVE prototype uses an in-memory model for fastest queries and processing, but the entire model can be externalized when dealing with excessive data or limited memory. Additionally, the execution models can be serialized for offline analysis.

4 Related Work

Related research can be broadly organized into the following categories: Dynamic Querying, Snapshot Debugging, Debugging by Program Slicing, and Software Understanding by Visualization. While a few systems have been built with similar querying functionality, none support the idea of visual queries and visual answers that capture the dynamic state of a program over the entire execution history.

Dynamic Querying. WhyLine is an Interrogative Debugger that allows programmers to ask *why did* and *why didn't* questions about their programs' runtime failures, and provides direct access to the runtime data that they need to debug [10]. It visualizes answers in terms of runtime events directly relevant to a programmer's question. The results achieved through Whyline would be more specific to the actual cause of program failure and hence more detailed. However, it has limited support for visualizing and understanding the run-time relationship between various program objects. Also it is bit restrictive when we consider the kinds of queries that it allows the user to pose on the system. There are a few other systems built with similar approaches. Lencevicius *et al.* discuss Query-Based Debugging, where programmers form textual queries on objects' runtime relationships [12]. However, it forces programmers to guess what relationships might exist, and requires learning an unfamiliar query language. Briggs *et al.* discuss a task timeline for debugging distributed Ada programs [5]. Their visualization highlights a dynamic slice, but it does not relate runtime events to the code. Zeller's work on cause-effect chains and the AskIgor debugger is also a related diagnosis tool [21]. Object Visualizer [13] and Program Explorer [11] provide numerous graphical and statistical run-time views with class-dependent filtering but do not allow general queries. Visualizing debuggers gather information by either instrumenting the source code or by using program traces. Debuggers that gather information by either instrumenting the source code or by using program traces usually require program recompilation for each different view and do not allow on-the-fly modifications.

Snapshot Debugging. Fox is a general purpose snapshot query-based debugger that supports queries over standard heap snapshots [14]. Basically, this can be thought of as pausing a program and studying its state in great detail to look for causes of program behavior using all the information available at that instant from the heap. This makes the timing of when a heap snapshot is taken critically affect whether we are able to catch a particular state that we want to examine in detail. Besides it requires programmers to know about what filters to apply to extract information from the heap snapshot and also requires learning of new query language (Fox Query Language) to obtain meaningful information from the program snapshot, which could turn out to be a potential deterrent. DINO is a tool, developed by Trent Hill, that visualizes an ownership tree - a structure derived from the object graph used in the aliasing research [9]. DINO allows the user to concentrate only on the user-created objects that are relevant to the code being executed, and avoid looking at the internal processes inside system libraries which may or may not have an adverse effect on the code in question.

Debugging by Program Slicing. A dynamic program slice presents a summary of the relevant computation by simply showing the relevant statements involved. The concept of program slicing was first proposed by Weiser [19, 20]; this original notion of a program slice envisioned a static slice obtained irrespective of the values of the input variables. Agrawal *et al.* have built a prototype debugger, named SPYDER that can, given a variable and a program location, determine which statements in the program affected the value of that variable at that location [2]. However, it requires that the programmer translates the externally visible symptoms of the fault into an internal program symptom in terms of a data or a control problem before these techniques may be used.

This translation may not always be an easy one to make.

Software Understanding by Visualization. BLOOM is a software tool aimed at using visualization for software understanding [15]. It uses MURAL as the visual query environment for gathering structural data from the source program. Data is accrued from static analysis of the source and through trace data collection. Once users have selected the data of interest by choosing an appropriate set of entities constrained by a combination of relationships and restrictions, MURAL provides an internal mechanism that finds appropriate visualizations and offers the user an ordered choice of what it deems appropriate for viewing the specified data. The problem with MURAL has been the heuristics used in selecting the appropriate visualization. The heuristics currently used by MURAL for selecting appropriate visualizations do not always select what we would consider to be the correct visualization. Another major problem in using BLOOM to everyday software understanding has been the lack of the data necessary to address the problem. Many of the problems that we come across need to be viewed at the level of source lines rather than functions. Moreover, variables and data flow are often important elements of the *understanding* problems. Also, the overhead for trace data collection is too large for some applications, especially long-running ones. VISUAL is another object-oriented graphical database query language designed for scientific databases where the data has spatial properties, includes complex objects, and queries are of exploratory in nature [3].

5 Conclusions and Future Work

The main contribution of this paper is a novel technique that enables programmers to pose queries on the actual run-time execution state of a program. Rather than posing queries that guess how a state was reached, the visual query system built on top of JIVE can tell a user exactly how a state or a statement was reached. Moreover, the ability to pose these queries visually eliminates the need to know or understand a complicated text-based querying language. The queries are associated with the diagram that engenders their usage.

Traditional debuggers allow programmers to find causes for program failure by using breakpoints and code stepping. These techniques, although helpful, require a lot of intuition and analysis on part of the programmer towards deciding where breakpoints should be set. Furthermore, if programmers have a weak hypothesis about the cause of a failure, any implicit assumptions about what did or did not happen at runtime will go unchecked. These unchecked assumptions in turn will prolong the debugging of programs. Dynamic program slicing and snapshot querying essentially generate are static views of the program run-time for a given set of conditions. These restrict the programmer from understanding the runtime relationships between program objects.

Our implementation of these techniques is currently prototypical. We have built a data model and query interface using JIVE's execution database. The data is available for running any of the queries we have described; we are currently integrating the graphical query interface with the programming query interfaces. Limited support for queries will be available in the next test release of JIVE. We are also investigating extensions of JIVE's visual operational semantics for Java that would more clearly show compound query results.

References

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990.

[3] N. H. Balkir, G. Ozsoyoglu, and Z. M. Ozsoyoglu. A graphical query language: Visual and its query processing. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):955–978, 2002.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison–Wesley, Reading, MA, USA, 1999.

[5] J. S. Briggs, S. D. Jamieson, G. W. Randall, and I. C. Wand. Task time lines as a debugging tool. *Ada Lett.*, XVI(2):50–69, 1996.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman, Inc., Reading, Massachusetts, 1995.

[7] P. Gestwicki and B. Jayaraman. Interactive visualization of Java programs. In *IEEE Symposium on Human-Centric Computing, Languages, and Environments*, pages 226–235, September 2002.

[8] P. Gestwicki and B. Jayaraman. Jive: Java interactive visualization environment. In *OOPSLA 2004 Conference Companion*, pages 615–616, 2004.

[9] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *TOOLS-Pacific 2000. Proceedings*. IEEE CS Press, 2000.

[10] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 conference on Human factors in computing systems*, pages 151–158. ACM Press, 2004.

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

[12] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 304–317. ACM Press, 1997.

[13] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337. ACM Press, 1993.

[14] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Australian Software Engineering Conference*, pages 251–261, 2004.

[15] S. P. Reiss. An overview of bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001.

[16] T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs with declarative event analysis. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*. Springer-Verlag, July 1998.

[17] Sun Microsystems, Inc. *Java Platform Debugger Architecture Overview*, 2001.

[18] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[19] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[20] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[21] A. Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.