JIVE: Java Interactive Visualization Environment

Paul V. Gestwicki
pvg@cse.buffalo.edu

Bharat Jayaraman
bharat@cse.buffalo.edu

University at Buffalo
Department of Computer Science and Engineering
201 Bell Hall, Box 60200
Buffalo, NY 14260-2000

ABSTRACT

JIVE represents a novel approach to runtime visualization and analysis of Java programs. It facilitates program understanding and interactive debugging, featuring: multiple, customizable views of object structure; representation of execution history via sequence diagrams; interactive queries on runtime behavior; forward and reverse interactive execution. JIVE uses standard JVM and compilers.

Categories and Subject Descriptors: I.3.6 [Computer Graphics]: Methodology and Techniques; D.2.m [Software Engineering]: Miscellaneous

General Terms: Documentation, Languages, Human Factors

Keywords: Program Visualization, Interactive Reverse Execution, Object and Sequence Diagrams, Object-Oriented Programming, Java

1. INTRODUCTION

We present a novel approach to runtime visualization and analysis of object-oriented programs through a prototype system called *JIVE*: Java Interactive Visualization Environment. Our approach combines a visual operational semantics for Java with an interactive execution model, providing an environment that facilitates teaching and interactive debugging. Our methodology incorporates the following features, many of which are shown in Figure 1:

Depicts Objects as Environments. JIVE uses a novel notation for displaying runtime object structures, clarifying the important fact that objects are environments of program execution. This notation visually depicts the relationship between methods and their object and static contexts.

Provides Multiple Views of Object States. JIVE supports viewing object states in different granularities, giving the user the freedom to focus on aspects of interest. Object structures can become very complicated [2], so the capability to view them at varying levels of detail is important.

Captures History of Execution and Method Interaction. JIVE shows the history of program execution with dynamic, interactive sequence diagrams, like those of the Unified Modeling Language . Generating these diagrams at runtime helps close the loop between program design and program.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

Supports Forward and Backward Execution. JIVE makes it possible to step forward or backward through program execution. This interactive stepping is especially important when using JIVE for debugging, since errors are necessarily detected after they have occurred [1].

Supports Queries on the Runtime State. Understanding the behavior of variables over time is essential to understanding a program's execution. A user must be able to inquire of the system: when variables have changed; when variables have held certain values; when objects are created; and when methods are called. The results of these queries should be integrated into the visualization environment.

Produces Clear and Legible Drawings. JIVE automatically arranges diagram components so as to clarify the object structure and method-calling sequence. This is done using a combination of traditional, graph-theoretic criteria as well as properties specific to the program being visualized.

Uses Existing Java Virtual Machine. We use existing JVM technology in the JIVE architecture. No custom compilers or virtual machines are necessary.

2. APPROACH

JIVE provides visualizations of object structures and execution histories. The notation is generally applicable to object-oriented programs but has been customized for Java. Our methodology highlights the fact that objects are environments of program execution, with method activations nested within their proper object contexts. We provide intuitive visualizations of such Java features as objects, static contexts, inner classes, threads, and exceptions.

Execution history is visualized through time sequence diagrams. The view of execution history is coupled with views of object states, making it possible for a user to understand the current state in relation to the program's history. JIVE sequence diagrams are interactive: the user can select a state in the sequence diagram, and the corresponding object diagram is shown. Programs with Swing or AWT interfaces are fully supported in JIVE; the GUIs coexist with the visualization of the program's execution.

Support for forward and reverse execution is provided by incremental state saving and restoration mechanics. JIVE supports line-by-line stepping through execution, as well as running to breakpoints placed in the source code. Interaction execution is normally online (that is, the visualization runs in tandem with the program being visualized), but execution records can also be saved for future, offline visual-

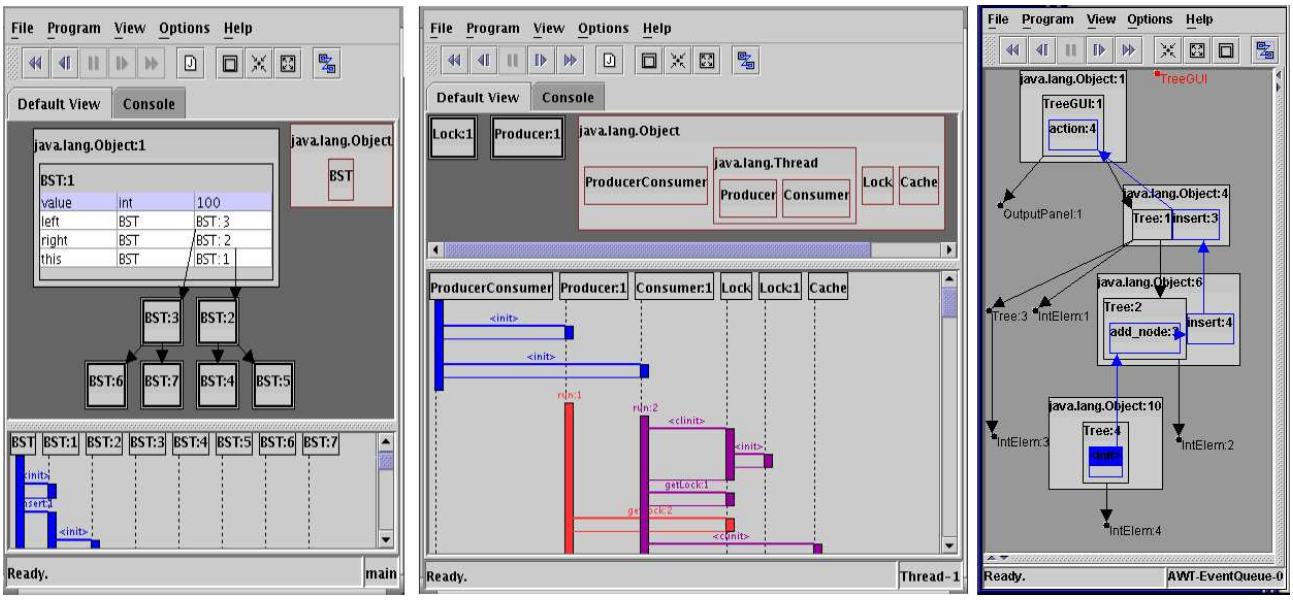

Figure 1: JIVE Screenshots. The leftmost screenshot shows a tree structure with the root node in full detail. The middle image shows multithreaded interaction in a producer/consumer application. The rightmost image shows the method's execution through different contexts, with all inactive objects minimized. The left and right images show how different levels of detail can be combined within views of the object diagram.

ization. Program logs are saved as XML data, which makes the data readily available for processing by other utilities.

JIVE incorporates a novel mechanism for performing interactive queries on the behavior of variables at runtime. The interaction model is built upon a database of execution history. Queries are processed on this database, and the results are displayed on the object and sequence diagrams. For example, if a user asks when a variable falls into a given range, the answer is shown by highlighting ranges of time in the sequence diagram. The user can then select any event in the sequence diagram, triggering JIVE to "jump" to the corresponding execution state.

3. STATUS AND FUTURE WORK

Our preliminary investigations of JIVE's effectiveness have been very positive. We have used the notation and the tool in a few undergraduate and graduate courses in programming languages. Object diagrams have proven useful for debugging, especially in cases where the user has a mental map of the desired structure, and JIVE clarifies the program's actual behavior. The object diagrams produced by JIVE serve as a visual operational semantics for Java. JIVE-generated sequence diagrams have proven invaluable in explaining the behavior of various design patterns and program constructs, and they are further clarified by the capability to view the details of the object and method calling structure at varying levels of detail. This has been especially true for programs with graphical interfaces: the colors clarify how the event-processing thread handles user input and drawing while other threads perform their own computations.

Sequence diagrams are traditionally used only in system design, when there is an inherent level of abstraction. At runtime, the sequence diagrams quickly become very complicated. We are investigating means by which the program history information can be modularized and multiple views can be provided, akin to the compact, minimized, and detailed views of contour diagrams. In addition to the aesthetic problem of providing a comprehensible diagram, there is a theoretical problem of how to most efficiently structure a dynamically-growing sequence diagram.

We use a combination of graph-theoretic and program-specific techniques to produce legible, meaningful diagrams. Analysis of the source code (class diagram) can give valuable information about the shape and dynamic behavior of the object diagrams. Converting diagrams into directed multigraphs and applying known graph drawing algorithms yields good drawings of hierarchical structures such as GUI frameworks. Our approach also the handles the nested object structures that occur when inner classes are used.

The use of a database model for execution history has potential applications outside of visualization. The behavior of variables can be recorded over multiple executions, making it possible to potentially detect abnormal behavior.

Our current implementation efforts on JIVE are focused on the following areas: integration of source-code analysis for improved graph drawing; efficient models for interactive execution and querying; and visual interfaces for queries.

4. 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] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the eighth annual OOPSLA conference*, pages 326–337. ACM Press, 1993.

[3] S. Mukherjea and J. T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1(3):215–244, 1994.