Application of Helix Cone Tree Visualizations to Dynamic Call Graph Illustration

Jyoti Joshi, Brendan Cleary, Chris Exton
Department of Computer Science and Information Systems
University of Limerick
Ireland

Brendan.Cleary@ul.ie

Abstract

We describe a tool that enables users to record and visualise runtime behaviour of software applications developed in Java. The execution trace, stored in the form of an XML file is visualized using 3D call graphs that are an extension of the Cone Tree information visualisation technique. This tool gives the user the ability to create several call graph views from a program's execution trace, providing additional representations of the program execution to both novice and expert programmers for the purposes of program execution analysis.

1 Introduction

As Java programs get larger and more complex, they become more difficult to understand. The maintainer, who tries to understand a program, reads some code, asks questions about this code, conjectures answers, and searches the code and documentation for a confirmation of the conjectures. In developing this prototype tool our primary goal is to help maintainers to easily verify conjectures on the runtime behaviour of the program with respect to the program structure. Our project focuses on tracing the dynamic behaviour of Java programs and using innovative 3D graph visualizations to achieve these goals. Our tool is functionally divided into two components namely: the Java Trace Generator (JTG) and the Call Graph Viewer (CGV). The JTG records program execution events in an XML format. The CGV viewer provides a 3D visualization media to which a program's execution trace, derived using the JTG, can be rendered. The next section presents related work; section 3 presents work on the JTG and its implementation details. Section 4 then presents the CGV and its implementation.

2 Related Work

A number of visualization tools exist that focus on program execution and a few of the more notable recent examples are Jinsight (IBM Research), IsVis (Jerding and Rugaber, 1997), JavaVis (Oechsle, 2001).

Jinsight is a tool for visualising the execution of Java Programs developed by IBM. Jinsight offers functionality for collecting Java program trace and also provides an environment for visualising program execution. It includes various views for examining several different aspects of run-time behaviour.

The IsVis visualization tool supports the browsing and analysis of execution scenarios. A source code instrumentation technique is used to produce the execution scenarios. In IsVis, the dynamic event trace can be analysed using a variation of Message Sequence Charts called Scenario View.

JavaVis monitors a running Java program and visualizes its behaviour with two types of UML diagrams for describing dynamic aspects of the program, namely object and sequence diagrams. It is suitable for small sized programs.

VisiVue (VisiComp Inc) is a runtime visualisation tool that dynamically displays the operations of any Java program at runtime, creating animated diagrams of the data structures and providing a complete trace of the programs execution. This tool is particularly effective in the Java educational environment and for smaller applications.

Reiss (Reiss, 2003) has developed a dynamic Java visualizer that provides a Box display view of the program and attempts to provide high- level program-specific information in real time.

Our tool is different from the above in that it makes use of a variation of the Cone Tree visualisation technique called the Helix Cone Tree for visualising program execution, which we feel provides a better technique for viewing a large amount of hierarchical, sequential data simultaneously.

3 Java Trace Generator

Program execution monitoring can be divided broadly into two categories: time driven monitoring and event driven monitoring (Jain, 1991). Time driven monitoring also known as sampling observes the state of the monitoring system at certain time intervals. Sampling however provides only summary statistical information about program execution. Event driven monitoring reveals the dynamic behaviour of program activities by events (Oechsle, 2001) as we require behavioural information not just snapshots, the JTG implements an event driven monitoring approach. The Java Trace Generator architecture (Figure 1) comprises the following elements:

- Debugger - The Debugger represents the starting point of the Java Trace Generator, it launches a primary Virtual Machine (VM) then launches a secondary VM which executes the Debugee whilst maintaining a connection to the primary VM. The Debugger also reads the event filter configuration file and sets the filter options for event requests. Events can be of one of the following types: Class Prepare Event, Class Unload Event, Method Entry Event, Method Exit Event, Thread Start Event, Thread Death Event and Exception Event.

- Debugee - This component represents the target application to be monitored. This is written in standard Java and compiled with '-g' option to facilitate the generation of all debugging information.

- EventObserver - The functionality of this element is to act as a main monitoring entity. It is event driven in operation and continuously monitors the EventQueue from the secondary VM; listeners are activated once a new event is received.

- Text Listener - This component transforms event information received from EventObserver according to an XML format and outputs it to the associated file. This XML debug trace is the intermediate medium of information exchange between the JTG and CGV.

The JTG is implemented using the JDI layer of the JPDA platform (Sun Microsystems, b). JPDA was developed to provide an infrastructure to build end-user debugger applications for the Java Platform, the JDI provides introspective access to a running virtual machine's state, class, array, interface, and primitive types, and instances of those types. The JTG using the JDI provides us in an XML format the behaviour of a java program, it's the task of the CGV to visualise and present that behavioural information to the user.

4 Call Graph Viewer

In theory, every problem can be encoded as a graph problem (Collberg et al., 2003). Call graphs represent execution relationships between discrete sections of software and are commonly used in aiding programmers understanding of code design and execution behaviour. There are two types of call graphs namely, Static Call Graphs and Dynamic Call Graphs. Call graphs generated by our tool fall into the second category as they are computed from

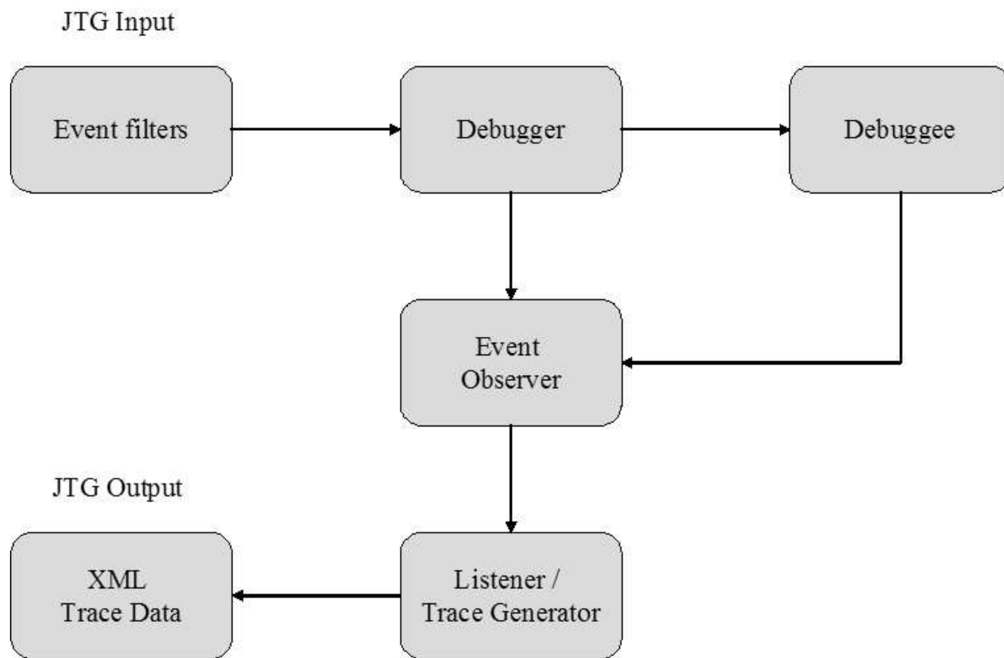

Figure 1: Java Trace Generator Architecture

the trace containing runtime execution details of the Java program. The Call Graph Viewer (CGV) provides the user with the ability to visualise these runtime execution details using a Helix Cone Tree.

4.1 The Helix Cone Tree

2-Dimensional call graph representations, while being an extremely useful and intuitive graph for the representation of small execution traces; are highly restricted in their ability to efficiently represent large data sets in finite display space. The Cone Tree (Robertson et al., 2003) graph visualization technique uses a 3-Dimensional rendering space to display hierarchies of information. The interactive nature of Cone Trees and this expansion of the traditional tree layout into the third dimension provide a significant improvement in the quantity of data displayable in a finite display space. Cone Trees however are not without their problems (Cockburn and McKenzie, 2000) specifically the occlusion of nodes and text labels within sub graphs.

The Helix Cone Tree (Figure 3) in an effort to alleviate these problems cuts and stretches the base of a cone along the y axis into a constrained helix thus reducing the amount of occlusion of nodes and text labels in a particular sub graph. This altering of the vertical positioning of nodes within a sub graph in a predictable order introduces an additional dimension of information to the traditional Cone Tree. This additional dimension can be mapped to various attributes of a dataset but in the CGV is used to represent the implicit chronological element present in program execution data.

4.2 Call Graph Viewer Technology

The CGV is an implementation of the CHIVE program source visualisation framework (Cleary and Exton, 2004). The CHIVE has at its core a customisable visualization framework that caters for the 3D visualization and manipulation of generic hierarchical data structures. In designing the CHIVE to satisfy the requirements of an adaptable, reusable visualisation framework it was partitioned into 3 subsystems each responsible for an aspect of the visualisation task (Figure 2).

- The Data Model provides a platform for presenting independent datasets in terms of nodes and relationships to the graph drawing and layout algorithm subsystem. The separation of the data model from problem domain specifics allows the user the latitude to quickly develop and modify algorithms for extracting the data and relationships independent of the data model implementation. The data model itself is an extension of the DefaultTreeModel class, taken from Java Swing, which is a common, highly documented generic tree data structure.

- The Graph Drawing and Layout Subsystem takes as input a data model and outputs a visualisation based on the application of a graph layout to that data model. This separation of the graph layout algorithm from the data model allows multiple layouts to be applied to a given data model resulting in many different visualisations, one such layout algorithm as used in the CGV is the Helix Cone Tree. Along with the relationships defined by the layout, essential to any graph visualisation is the information expressed by the nodes of that graph; this comprises not only the user data associated with the node but also the visual representation of that node. In the CHIVE this appearance information of a node is defined by a glyph (Telea et al., 2002). Glyphs encapsulate all the visual aspects (geometry, appearance and text components) of a node including any interactive behaviour.

- The Graph visualisation, Interaction and Manipulation Environment is responsible for providing an anchor for visualisations in a 3D rendering environment and for mediating user interaction with visualisations in that environment. To accomplish these goals it provides a Java 3D (Sun Microsystems, a) universe to which the graph layout subsystem renders its visualisations and also various interaction behaviour components. These interaction behaviour components can be used locally within visualisation and also globally within the environment. When used locally behaviours such as manual and animated rotation, selection and pruning effect only the select node and the sub graph rooted at that node. When used in a global context they effect the visualisation as a whole.

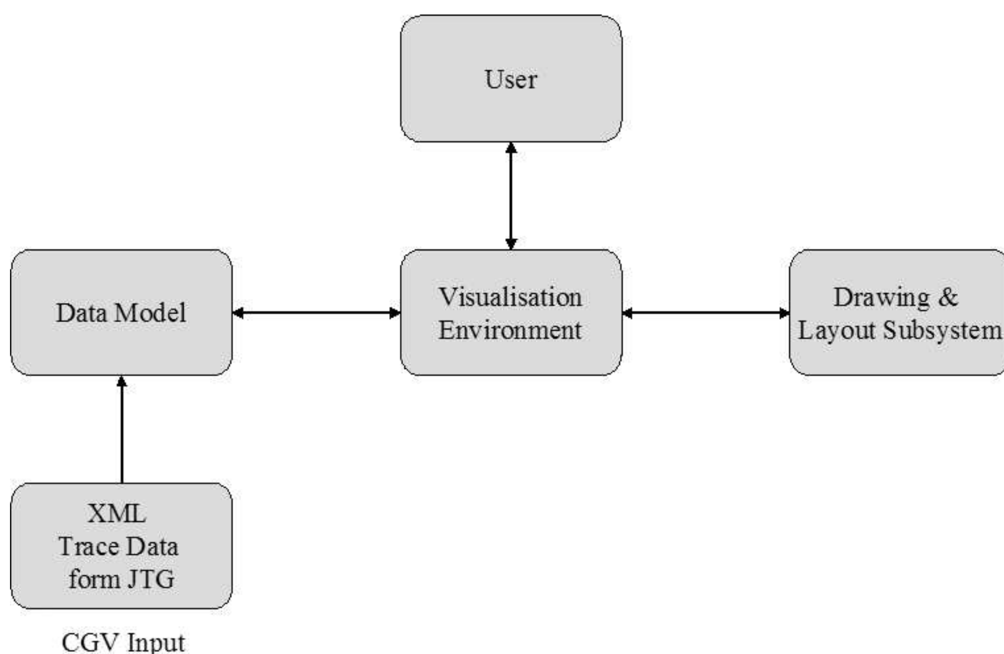

Figure 2: CHIVE Architecture

4.3 3D Call Graph Viewer Implementation

For an execution trace to be visualised that trace data needs to be transformed into the data model acceptable by graph drawing and layout subsystem. This data model serves as the interface between the CGV and the JTG. The different sets of relations that can be defined in the data model specify the different types of views (described next) in the form of call graph trees that can be displayed to the user. Call graphs however generally contain cycles. Traditional 2-Dimensional call graphs represent cycles using additional links between nodes; this however would violate the simplicity of the Cone Tree layout and render its advantages in terms of graph clarity obsolete. Cone Trees are at their most effective when displaying simple hierarchies without cycles, so in our implementation we chose to repeat nodes where a cycle occurs. This solution however, due to the size of tree produced, is not viable when very deep cycles are encountered.

4.4 Call Graph Views

The CGV lets you work with the trace information through various views, each depicting different aspects of your program's objects and execution sequence.

- The CGV's Main View displays a complete call graph tree with all execution details. Here the root node represents start of process, the first level of child nodes represent all the thread start events from the program and subsequent levels indicate start of method and end of method events. Different types of event nodes are represented by different glyphs. On selection of a node, the information pane of the user interface gives detailed information about that node. For example if a node for a method start event is selected, the selected method details displayed in the information pane will include; method name, class name, thread name, method arguments and time at which method was called. Figure 3 shows the main view of the CGV.

- The Thread View allows the user to select one or more threads from the list of threads executed through out the program execution, this view then provides a Thread Execution Tree for the selected thread or threads.

- The Object Instantiation View allows the user to select a root object, the system then provides the user with a view of the object instantiation sequence derived from that root object. Figure 4 shows the thread and object views of the CGV.

Finally the user can interact with these views in a variety of different ways; for example the user can select any of the above views for visualisation of trace data, when selecting a node the node details are displayed in the information pane, the user can select a node and get a highlighted view of sub tree starting from the selected node and the system allows the user to browse through previously visited views.

5 Conclusions and Future Work

In this paper we present a program visualisation tool that is based on the combination of technologies we have developed for capturing the runtime behaviour of java programs and the visualisation of hierarchical, sequenced data.

The Java Trace Generator (JTG) captures the runtime behaviour of executing java programs by implementing an event driven monitoring approach and exports that behavioural information as an XML execution trace. The Call Graph Viewer (CGV) represents the execution traces generated by the JTG as Helix Cone Trees and provides an environment for the user to interact with and inspect several views of the subject programs execution behaviour.

Significant future work includes the synchronization and tighter integration of the JTG and CGV. This involves the concurrent generation of call graph views as the monitored program is

being executed and changing monitoring program parameters from the CGV to alter the future execution sequence of the monitoring program for different purposes like better performance, memory management and exception handling. Other important areas of future work include the development of new prototypes that expand on the number of views and representations used, evaluation of these prototypes and the integration of the JTG and CGV into a popular Integrated Development Environment (IDE).

Given the system's modular design it is hoped that new visualisation techniques can be easily applied to the existing trace mechanism to quickly develop future prototypes. As for the evaluation of these prototypes several issues need to be overcome, including the ability of the technologies to scale to larger program sizes, the validity of the experiments and the environment in which they are performed and the selection of participants to the experiments.

Work is currently underway on the development of a CHIVE eclipse plugin, we hope to capitalise on this work in integrating the CGV and JTG with the eclipse IDE (The Eclipse Project). We see integration of our tool with a popular and modern IDE as being key both to the successful adoption of our tool and also in providing a realistic environment in which to perform experiments. Tools succeed in being adopted by conveying a perception of usefulness and increased efficiency (Zayour and Lethbridge, 2001). By integrating our tool with a successful IDE such as eclipse we hope to reduce the burden of switching between applications on the user and capitalise on user and participant familiarity with the IDE encouraging the use of our tool and also providing a more realistic context for conducting experiments.

References

B. Cleary and C. Exton. Chive - a program source visualisation framework. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC 04)*, pages 268–269, Bari, Italy, June 24-26 2004. IEEE Computer Society Press.

A. Cockburn and B. McKenzie. An evaluation of cone trees. In *Proceedings of the 2000 British Computer Society Conference on Human Computer Interaction*, University of Sunderland, 2000.

C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph based visualization of the evaluation of software. In *Proceedings of the ACM symposium on Software Visualization*, pages 77–86, San Diego, CA, USA, June 2003. ACM Press, New York, NY.

IBM Research. Jinsight visualizing the execution of java programs, 2001. `http://www.research.ibm.com/jinsight/`.

R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design*. John Wiley & Sons, 1991.

D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE)*, pages 56–65, Amsterdam, The Netherlands, October 6-8 1997. IEEE Computer Society Press.

R. Oechsle. Automatic visualization with object and sequence diagrams using the java debug interface (jdi). In *Proceedings of the Software Visualization: International Seminar*, pages 176–190, Dagstuhl Castle, Germany, May 20-25 2001. Springer-Verlag Heidelberg.

S. Reiss. Bee/hive: A software visualization back end. In *Proceedings of the ACM symposium on Software visualization*, pages 57–65, San Diego, CA, USA, June 2003. ACM Press, New York, NY.

G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on human factors in computing systems: Reaching through technology*, pages 189–194, New Orleans, Lousiana, USA, 2003. ACM Press, New York, NY.

Sun Microsystems. Java3d, 2003a. `http://java.sun.com/products/java-media/3D/`.

Sun Microsystems. Java platform debugger architecture documentation, 2004b. `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

A. Telea, A. Maccari, and A. Riva. An open visualization toolkit for reverse architecting. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 02)*, pages 3–10, Paris, France, June 27-29 2002. IEEE Computer Society Press.

The Eclipse Project. Eclipse, 2004. `http://www.eclipse.org/`.

VisiComp Inc. Visivue java software visualization tool, 2004. `http://www.visicomp.com/product/visivue.html`.

L. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 01)*, pages 245–255, Toronto, Canada, June 12-13 2001. IEEE Computer Society Press.

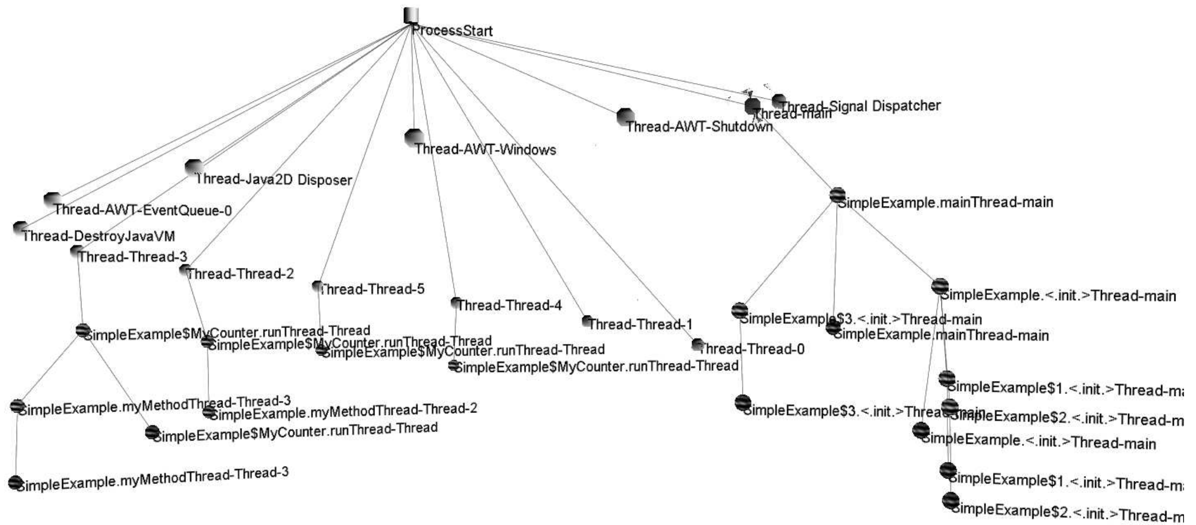

Figure 3: Call Graph Viewer - Main View

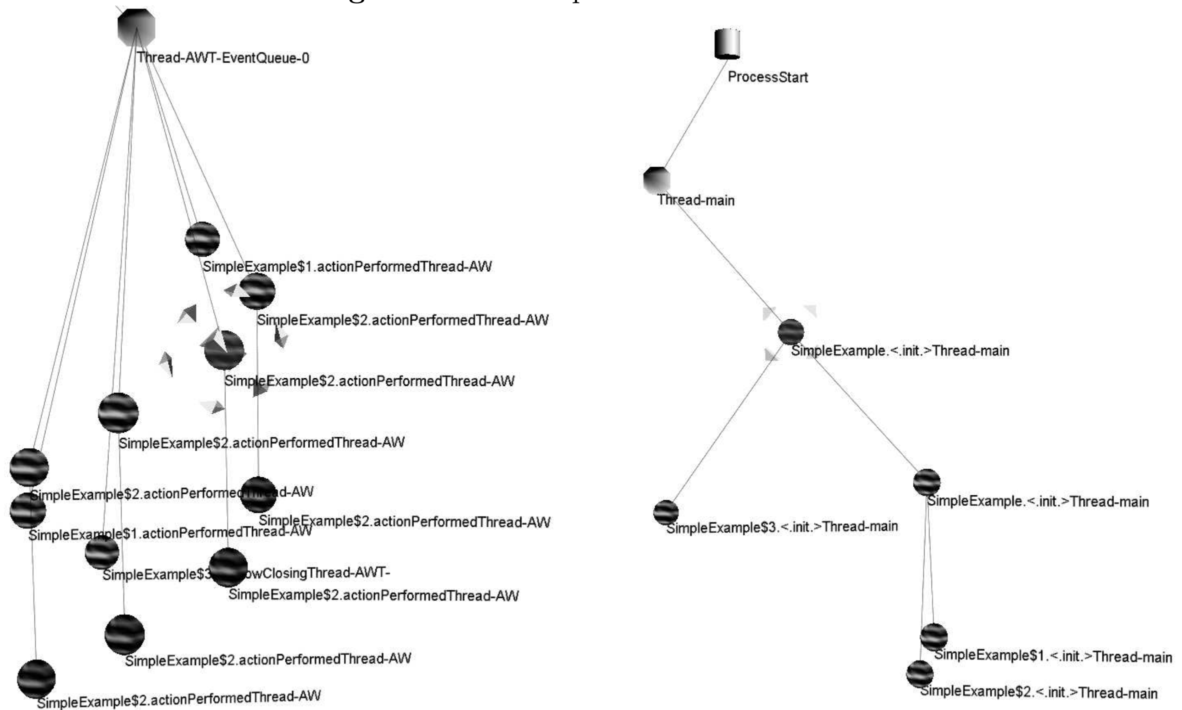

Figure 4: Call Graph Viewer - Thread View and Object View