Bee/Hive: A Software Visualization Back End

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

1. Introduction

Software understanding is the task of answering questions and gaining insights about software systems. In some cases it involves gathering broad-stroke information about what a system is doing; in other cases it involves understanding a particular aspect of the system; in still other cases it involves addressing very specific issues such as why was this routine called or what will happen if I change this input.

We are currently developing a comprehensive system, BLOOM, that uses visualization to facilitate software understanding [8-10]. The system starts with facilities for collecting information about the software in question. It then provides facilities for analyzing and summarizing this information. Next it provides a visual language to let the user specify what system information and analyses are relevant to the question at hand and how these should be related. Finally it provides a back end visualization system that is able to display the selected information in a variety of ways.

Using visualization for software understanding puts a number of requirements on the underlying visualization system. Software understanding implies being able to view a software system from many perspectives and at many levels of detail. It means getting overview information at times and answers to very specific questions at other times. It requires the ability to ask a wide range of questions and get the answers to those questions reasonably fast and efficiently. All this requires that a software visualization system provide:

- *Multiple visualizations*. No one visualization strategy can handle displaying information about the wide range of questions that software understanding implies. Instead, different types of visualizations and different visualization strategies are going to be most suitable for different types of data. The back end must offer a range of such strategies so that the one most appropriate to the question at hand can be used.
- *Browsing facilities*. The questions involved in software understanding are generally broad at first and then narrow down to specific issues. For example, one might want to know why some operation takes so long, and then narrow this down to why a particular routine takes so long and then why a particular line in that routine is executed so frequently. Software visualization should facilitate such searches. Moreover, specifying very specific questions for visualization is often difficult; it is generally much easier to specify a more general question and then use browsing facilities to isolate the particular data in question.
- *Coordinated views*. Many of the issues that arise in software understanding involve relationships between different analyses and program information. While such relationships can often be expressed in a single visualization, it is often easier to use multiple visualizations with one being used to navigate in the other. The visualization system must allow and facilitate such coordination.
- *Multiple data sources*. Finally, software visualization requires acquiring and correlating data from multiple data sources to answer the particular questions that arise. The back end system must be able to gather and combine such sources and then organize the resultant information for visualization.

In the remainder of this paper we describe the architecture of our visualization back end and how it meets these requirements.

2. Architecture Overview

The back end of our visualization system is composed of the five components as shown in Figure 1. The central component, BEE, controls the visualization process. It reads a visualization description file generated by the front end. This file describes both the data that is needed for the visualization and the visualization to be used to display the

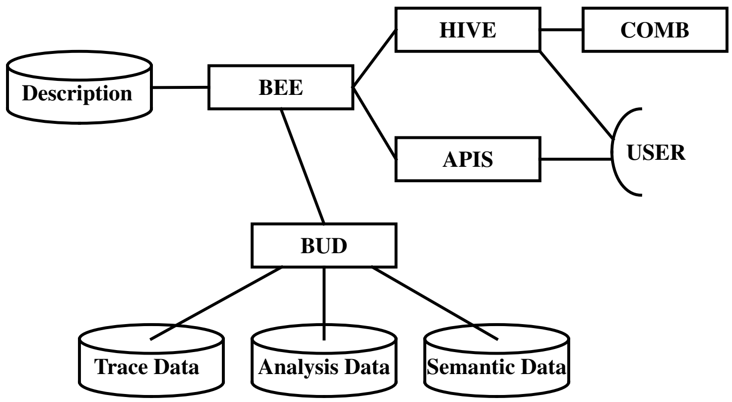

FIGURE 1. Overview of the back end visualization architecture.

data. BEE uses a data manager, BUD, to actually access the data from the various data sources that are available. It then provides a user interface, APIS, that lets the user specify how to correlate the data with the visualization and offers the appropriate browsing and view coordination facilities. Finally, it uses a common visualization back end, HIVE, to actually create the appropriate visualization. HIVE is a generic visualization engine that can support a wide variety of different visualizations. The actual visualization implementations are provided as plug-ins through the COMB package. We consider these packages in more detail in the next sections.

3. HIVE and COMB

HIVE provides a generic framework to support a wide variety of 2D and 3D visualizations. It provides facilities for managing data objects representing information to visualize, for displaying and updating the visualization window, for letting the user pan, zoom, and otherwise manipulate the resultant visualization, for correlating positions with objects displayed in the window, for managing visualization parameters, and for supporting the actually drawing that the different visualization methods require. Its development was based on our previous experiences in developing a generic visualizatio back end [5-7].

HIVE offers two different interfaces. The first, used by BEE, is designed to let a client program easily create and use visualizations. This interface provides the abstractions of a visualization window and a visualization object. The client creates an appropriate set of objects to visualize which implement the visualization object interface. This interface gives HIVE a handle on the object and provides it with access to data associated with the object. It also allows individual objects to be quickly removed from or selected by the visualization. The visualization window interface provides the client with the ability to control the visualization. It offers facilities for editing the set of objects, for setting and getting properties of the actual visualization, for handling events using appropriate callbacks, and for handling selections.

The second interface is designed to support a variety of different plug-in visualizations. This interface provides a basic abstract class that the visualization plug-in actually needs to support. In addition, it provides support for maintaining the set of visualization objects, for drawing through a set of high-level primitives, for accessing parameters, f and for managing colors and textures. This interface was designed so that creating a new visualization would be relatively simple.

The actual visualization strategies are encoded in the COMB package. Actually, HIVE allows arbitrary plug-ins that can be specified and loaded at run time. However, we have developed an initial set of visualizations that cover a broad spectrum of software understanding problems. These include:

- *Box trees*. This is a tree representation where the nodes are drawn adjacent to their parents without having to display the arcs. Color, height, depth, and size can be used to represent various dimensions of data.
- *Tree maps*. This is an alternative tree representation where nodes are nested inside one another, alternating between the horizontal and vertical. Ours is a 3D variant of the display origianlly developed by Schneiderman [11].

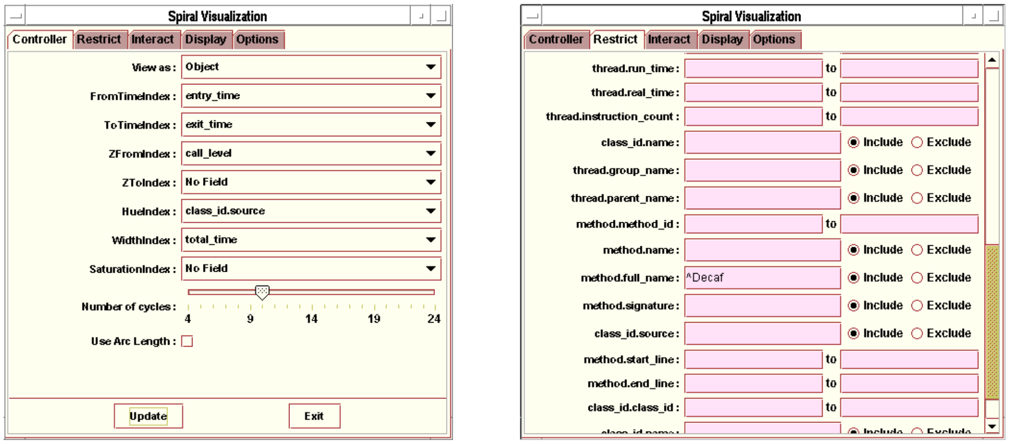

FIGURE 3. Two of the APIS control panels. The left one is the visualization control panel. The right one is the restriction control panel.

- *File maps*. This is a representation that is roughly equivalent to SeeSoft [2]. It can be used to display information relevant to a single file or a set of files. Color, height, and texture can be used to represent information that is associated with lines in the files.
- *Layouts*. We include the ability to display 2D or 3D graphs consisting of nodes and arcs where this is appropriate. These typically are only effective when the number of nodes is less than a hundred.
- *Point maps*. These are generalized 2D or 3D scatter or dot plots [1]. Hue, saturation, texture, size (X,Y, and Z separately), and rotation can be used to represent arbitrary information along with two or three positional dimensions. If appropriate, consecutive points can be connected by parameterized lines to show time series and other related data.
- *Spirals*. We use spirals to show time series data in a compact and space-efficient manner. Hue, saturation, height, and width can be used to show information relating to time. The spiral can be formed so that equal amounts of time are represented either by equal arcs or by equal lengths.

A set of visualizations illustrating some of the capabilities of the system is seen in Figure 2.

4. BUD

BUD provides the ability to access and organize the various data sources needed for visualization. It provides a common interface to data that is stored in several different forms. Some of the data is kept in databases and accessed via OQL or SQL queries. Names and related information associated with trace data is accessed by ID through a programmatic interface to a shared memory segment. The analysis results are stored as XML files. Bud offers a common interface to this data. In addition it provides the facilities for constraining and relating different data sources and for creating virtual objects through such relationships. A detailed description of BUD and its entity-relationship based data model will be presented in a later paper.

5. APIS

The user interface to the visualization display is provided through APIS. APIS provides a set of control panels for each visualization. These include:

- *Visualization Control*. This panel lets the user control the visualization by both setting properties of the visualization and by specifying what data fields are associated with what visualization properties. An example of this panel is shown in Figure 3.
- *Restriction Control*. This panel provides browsing support by letting the user restrict what data objects should be displayed and which should be elided. The panel allows the user to set upper and lower bounds for numeric data fields and to specify regular expression patterns of items to either include or exclude for string fields. An example of this panel is shown in Figure 3.

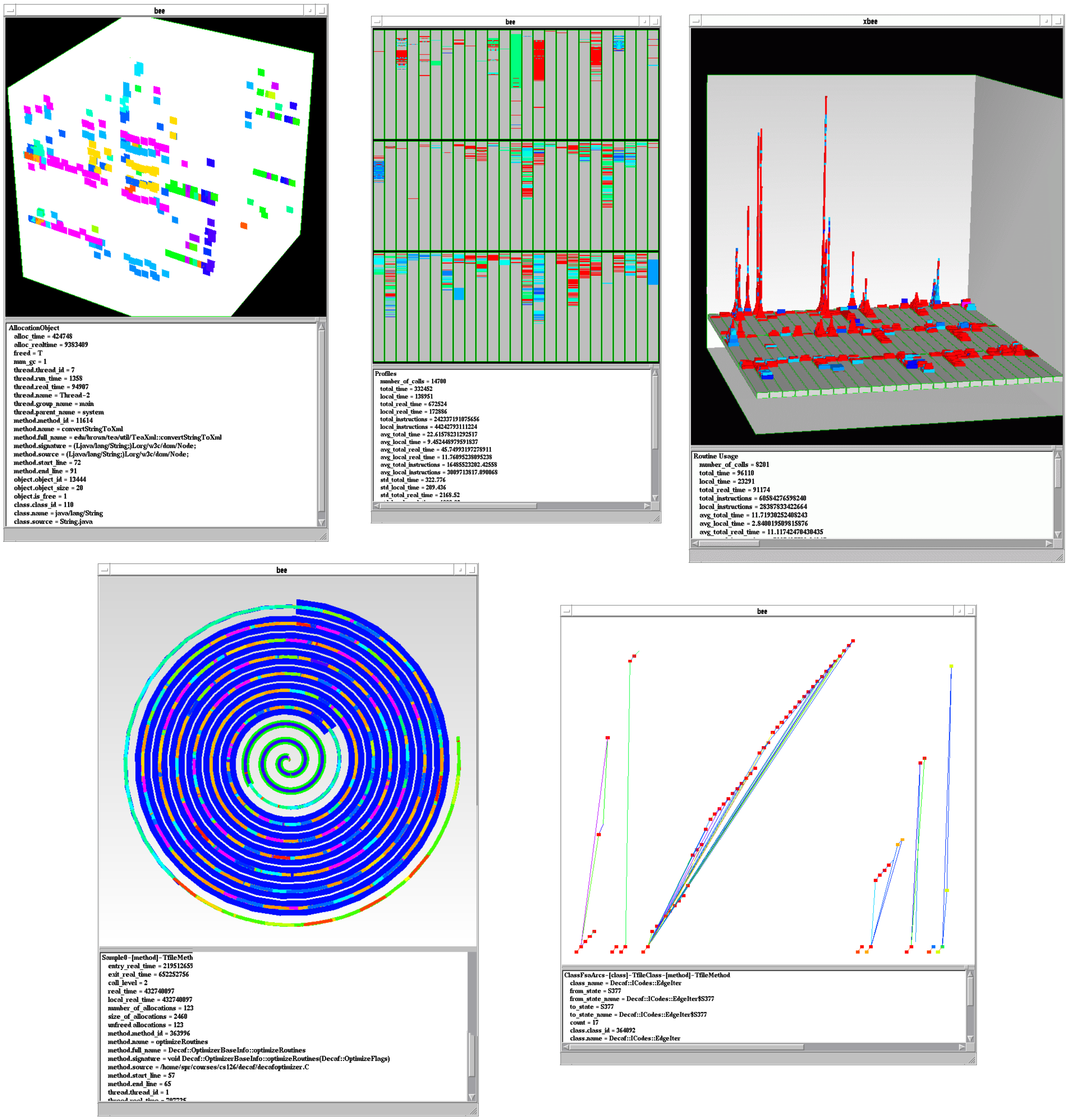

FIGURE 2. Sample visualizations. Clockwise, they represent a point map of a restricted set of object allocations located by time and colored by class, a 2D file view showing real time performance in an optimizing compiler, a 3D file view showing other performance information for the same compiler, a dot plot illustrating a subset of class usage automata, and a spiral showing a restricted and sampled call graph of the compiler.

- *Interaction Control*. This panel provides support for correlating multiple visualizations using semantic information. It lets the user specify what properties of the current visualization should be sent to other visualizations when the visualization is clicked on. It also lets the users specify whether and how the focus and selection of the current visualization should change when information is received from other visualizations.
- *Grouping Control*. This panel provides another form of browsing support. It lets the user dynamically specify groupings of the data objects, for example grouping all data for methods by their class. The user can specify how the various fields should be treated when grouped as well as the actual pattern-based criteria for grouping. This is based on the visualizations that were supported by the FIELD environment for call graphs and the class hierarchy [3,4,6].

APIS also provides the user interfaces needed to select the system to be visualized and the particular trace to be examined as appropriate.

6. BEE

BEE is the glue that holds the above packages together. It reads a file generated by the visualization front end that describes what data should be visualized and what visualization strategy should be used. It sets up BUD to access and combine the required data sources after using APIS to let the user specify what program and trace files should be viewed. It then uses HIVE to set up the actual visualization window, plugging in the appropriate visualization method from COMB. Finally, it puts up and manages an appropriate APIS interface for the given data sources and the given visualization. It also takes care of passing events from the visualization through APIS to either the display panel of the visualization or to other visualizations.

7. Current Status and Future Work

The visualization back end described here is nearly complete. We are currently working on the underlying mechanisms to support interaction and grouping both in BUD and in APIS. We are also designing and working on a small set of additional visualization plug-ins. The current system runs on Solaris/Sparc and we are porting it to Linux.

Most of our future work on the back end will involve validating the utility of the different visualizations and the support that the back end provides for browsing, grouping, and correlated views. We are planning to make the system available both inside and outside of Brown this summer and, once we do so, are starting to plan some user studies.

8. References

1. Kenneth W. Church and Jonathan I. Helfman, “Dotplot: a program for exploring self-similarity in millions of lines for text and code,” *Journal of Computational and Graphical Statistics* Vol. **2** pp. 153-174 (1993).
2. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., “Seesoft - a tool for visualizing software,” AT&T Bell Laboratories (1991).
3. Steven P. Reiss, Scott Meyers, and Carolyn Duby, “Using GELO to visualize software systems,” *Proc. UIST ’89*, pp. 149-157 (November 1989).
4. Steven P. Reiss, *FIELD*: *A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).
5. Steven P. Reiss, “Cacti: a front end for program visualization,” *IEEE Symp. on Information Visualization*, pp. 46-50 (October 1997).
6. Steven P. Reiss, “Software tools and environments,” in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).
7. Steven P. Reiss, “Software visualization in the Desert environment,” *Proc. PASTE ’98*, pp. 59-66 (June 1998).
8. Steven P. Reiss and Manos Renieris, “Generating Java trace data,” *Proc Java Grande*, (June 2000).
9. Steven P. Reiss, “An overview of BLOOOM,” *Submitted to PASTE ’01*, (June 2001).
10. Steven P. Reiss and Manos Renieris, “Encoding program executions,” *Proc ICSE 2001*, (May 2001).
11. Ben Schneiderman, “Tree visualization with tree-maps: a 2-D space-filling approach,” *ACM Transactions on Graphics* Vol. **11**(1) pp. 92-99 (January 1992).