GAMMATELLA: Visualization of Program-Execution Data for Deployed Software

Alessandro Orso, James A. Jones, Mary Jean Harrold, and John Stasko
College of Computing – Georgia Institute of Technology
{orso,jjones,harrold,stasko}@cc.gatech.edu

1 Introduction

The development of reliable and safe software is difficult. Quality assurance tasks, such as testing, analysis, and performance optimization, are often constrained because of time-to-market pressure and because products must function in a number of variable configurations. Consequently, released software products may exhibit missing functionality, errors, incompatibility with the running environment, security holes, or inferior performance and usability.

Many of these problems arise only when the software runs in the users' environments and cannot be easily detected in-house. To analyze and investigate the behavior of deployed software, we developed the GAMMA technology [5]. GAMMA is based on the use of lightweight instrumentation to collect various kinds of *program-execution data*—information about deployed software gathered during its execution in the field. Some examples of program-execution data are coverage data, exception-related information, profiling information, and performance data, such as memory and CPU usage.

Unfortunately, the monitoring of a high number of deployed instances of a software product can produce a huge amount of program-execution data. For example, in our preliminary study that involves only a few users and only one system, we collected more than 1,000 program-execution data in less than a month. If we multiply that number by a realistic number of users for an average system, we can easily see that the quantities involved are on the order of millions of program-execution data per system. Furthermore, when collecting more and more kinds of program-execution data from the field, not only does the size of the data grow, but also their complexity: different kinds of data may have intricate relations that require such data to be analyzed together to be understood.

Obviously, such an avalanche of data cannot be analyzed manually. To be able to extract meaningful information about the program behavior from the raw data and exploit their potential, we need suitable data-mining and visualization techniques. In particular, visualization techniques can be very effective in transforming program-execution data into visual information that can be explored and easily understood [6, 8].

In this demo, we present a new approach that can efficiently represent different kinds of program-execution data and lets us investigate the data to study the behavior of programs in the field. The approach is defined for a situation in which a number of instances of a program are continuously monitored, and has the following characteristics: (1) it provides a hierarchical view of the code, so that the user can navigate the program at different levels of detail while studying the program-execution data; (2) it is flexible in the kind of program-execution data it can show for each execution; and (3) it accounts for a dynamic, constantly increasing, and possibly very large number of executions through the use of *filters* and *summarizers*. The technique is described in detail in Reference [4].

In the demo, we show our prototype toolset, GAMMATELLA, that implements the visualization approach and provides capabilities for instrumenting the code, collecting program-execution data from the field, and storing and retrieving the data locally. We show two possible applications of our visualization technique: exception analysis and profiling.

2 Visualization Technique

To investigate the program-execution data efficiently, we must be able to view the data at different levels of detail. In our visualization approach, we represent software systems at three different levels: statement level, file level, and system level. At the statement level, we represent the actual code. The representation at the file level provides a miniaturized view of the source code similar to the one used in the SeeSoft system [2]. The system level uses treemaps [7, 1] to represent the software and is the most abstracted level in our visualization. At each level, coloring is used to represent one- or two-dimensional information about the code, using the colors' hue and brightness components. The coloring technique that we apply is a generalization of the coloring technique defined for fault-localization by Jones and colleagues [3].

The coloring applies differently to the different representation levels. For the statement-level and the file-level representations, a color is assigned to each statement, and its visual representation is rendered in this color. For the system-level representation, we defined a mapping to maintain color-related information in the treemap view. Our mapping results in differently colored blocks, within each treemap node, that are proportional in size to the number of

statements represented with that color. Our mapping maintains the same layout of the colored blocks across all nodes.

Our visualization technique represents executions using an *execution bar*: a virtually infinite rectangular bar that consists of possibly colored bands; each band represents a different execution of the monitored program in the field.

To support the investigation of a possibly high number of program-execution data, our visualization technique encompasses filtering and summarization capabilities. Those capabilities leverage executions' properties (e.g., hardware platform on which the execution was performed) expressed as a set of alphanumeric pairs $(key, value)$. Filtering can help the user focus on only a subset of executions (selected based on their properties), whereas summarization can help the user identify correlations among executions by aggregating them based on common properties.

3 The Toolset

GAMMATELLA is a toolset that implements our visualization approach and provides capabilities for instrumenting the code, collecting program-execution data from the field, and storing and retrieving the data locally. GAMMATELLA is written in Java, supports the monitoring of Java programs, and consists of three main components: an Instrumentation, Execution, and Coverage Tool, a Data Collection Daemon, and a Program Visualizer.

The *Instrumentation, Execution, and Coverage Tool* (INS-ECT)[1] that we developed can instrument for code coverage and profiling. During the execution, coverage and profiling information is collected by probes inserted in the code. At the end of the execution, the information is dumped, compressed, and sent back to a central server using the Simple Mail Transfer Protocol.

The *Data Collection Daemon* (DCD), written in Java, runs as a daemon process on servers on which we store the execution data. The tool retrieves the incoming mail with coverage information from the central server and stores the program-execution data in a database.

The *Program Visualizer* (*PV*), written in Java using the Swing toolkit, implements our visualization technique. PV retrieves coverage data stored by the DCD in real time, and uses them to update statement-level, file-level, and system-level view. A snapshot of the Program Visualizer is shown in Figure 1. As the figure shows, the Program Visualizer consists of three main components (Execution Bar, Code Viewer, and Treemap Viewer) and a set of additional widgets (interactive color legend, statistics pane, color slider, and color-space control menus).

4 Demo

We demo our data collection and visualization technique for two tasks: investigation of exceptions generated during users' executions and profiling analysis. To visualize exception-related information, we assign a color to each statement in the program to represent how likely it is for the statement to be responsible for the behavior that led to the throwing of an exception. To visualize profiling information, we assign a color to each statement in the program to represent how often the statement is executed. The coloring lets the user identify *hot spots* in the programs according to the way the program is used in the field.

In the demo, we show such applications using a real subject, a 70-KLOC program-analysis tool written in Java, and real execution data that we collected from 14 users over a period of four months.

[1] `http://sourceforge.net/projects/insectj`

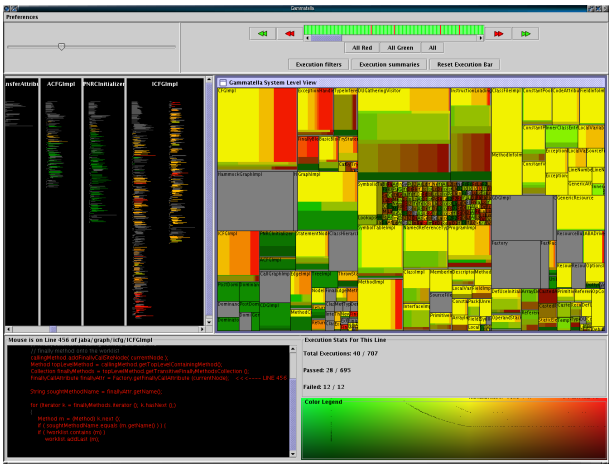

Figure 1. A screenshot of the GAMMATELLA Program Visualizer.

References

[1] M. Bruls, K. Huizing, and J. J. van Wijk. Squarifi ed treemaps. In *Proc. of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, 2000.

[2] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft – a tool for visualizing line oriented software. *IEEE Transactions On Software Engineering*, 18(11):957–968, Nov 1992.

[3] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of the 24th International Conference on Software Engineering (ICSE'01)*, pages 467–477, May 2001.

[4] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proc. of the ACM symposium on Software Visualization*, San Diego, CA, USA, June 2003.

[5] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 65–69, Jul 2002.

[6] S. P. Reiss and M. Renieris. Encoding program executions. *Proc. of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 221–230, may 2001.

[7] B. Shneiderman. Tree visualization with tree-maps: A 2-D space-fi lling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

[8] J. Stasko, J. Domingue, M. Brown, and B. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.