Visualization of Program-Execution Data for Deployed Software

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

James Jones
College of Computing
Georgia Institute of Technology
jjones@cc.gatech.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
harrold@cc.gatech.edu

Abstract

Software products are often released with missing functionality, errors, or incompatibilities that may result in failures in the field, inferior performances, or, more generally, user dissatisfaction. In previous work, we presented the GAMMA technology, which facilitates remote analysis and measurement of deployed software and allows for gathering program-execution data from the field. When monitoring a high number of deployed instances of a software product, however, a large amount of data is collected. Such raw data are useless in the absence of a suitable data-mining and visualization technique that supports exploration and understanding of the data. In this paper, we present a new technique for collecting, storing, and visualizing program-execution data gathered from deployed instances of a software product. We also present a prototype toolset, GAMMATELLA, that implements the technique. We show how the visualization capabilities of GAMMATELLA allows for effectively investigating several kinds of execution-related information in an interactive fashion.

CR Categories: D.2.5 [Software Engineering]: Testing and Debugging—Monitors, Testing tools, Tracing

Keywords: Gamma technology, software visualization, remote monitoring

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 pressures 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 or investigated in-house. To analyze and investigate the behavior of deployed software, we developed the GAMMA technology [Bowring et al. 2002; Orso et al. 2002]. 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. 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 [Baker and Eick 1995; Gray et al. 2002; Reiss and Renieris 2001; Stasko et al. 1998; Storey and Müller 1995].

In this paper, we present a new visualization approach that can efficiently represent different kinds of program-execution data and allows for investigating the data to study the behavior of programs in the field. The approach is defined for a context 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*.

We also present a 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.

Finally, we report two possible applications of our visualization technique, exceptions and profiling analyses, and a feasibility study. In the study, we use GAMMATELLA to collect, store, visualize, and investigate program-execution data gathered from instances of a real software program distributed to a set of users. The study shows how the visualization capabilities of GAMMATELLA let us effectively investigate several kinds of program-execution data in an interac-

© 2003 ACM 1-58113-642-0/03/0006 $5.00
ACM Symposium on Software Visualization, San Diego, CA

tive fashion and get meaningful insights into the monitored program's behavior.

The main contributions of the paper are:

- a new visualization approach that allows for visualizing various kinds of program-execution data and for interactively studying a program's behavior;
- a toolset, GAMMATELLA, that implements the visualization approach and provides instrumentation and data-collection capabilities; and
- a case study that shows the feasibility of the approach.

2 Visualization Technique

In this section, we describe the visualization approach that we defined to enable continuous monitoring and exploration of program-execution data collected from deployed software.

One goal of our work is to provide an interface that can scale to large programs and that can handle a number of executions by many users. To achieve this goal, we defined a visualization approach that provides:

- representation of software systems at different levels of detail;
- use of coloring to represent program-execution data;
- explicit representation and visualization of program-execution data about each execution together with its properties; and
- capabilities for filtering and summarizing the program-execution data in an interactive way.

2.1 Representation Levels

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.

Statement level The lowest level of representation in our visualization is the statement level. At this level, we represent actual source code, and each line of code is suitably colored (in cases where the information that we are representing does involve coloring). Figure 1 shows an example of a colored set of statements in this view. The statement level is the level in which users can get the most detail about the code. However, directly viewing the code is not efficient for any program of non-trivial size. To alleviate this problem, our visualization approach provides representations at higher levels of abstraction.

```
...
finallyMethod.setName(
   handlers.getFinallyNameForCFGStartOffset(finallyStartOffsets[i] ));
if ( numFinallyBlocks != 0 ) {
   finallyMethod.setType(Primitive.valueOf(Primitive.VOID));
   finallyMethod.setContainingType(parentMethod.getContainingType());
}
finallyMethod.getContainingType().getProgram().addSymbol( finallyMethod );
finallyMethod.setDescriptor( new String("()V") );
finallyMethod.setSignature( parentMethod.
...
```

Figure 1: Example of statement-level view.

File level The representation at the file level provides a miniaturized view of the source code. We use a technique similar to the one introduced by Eick and colleagues in the SeeSoft system [Ball and Eick 1996; Eick et al. 1992]. We map each line in the source code to a short, horizontal line of pixels. Figure 2 shows an example of a file-level view for the statements in Figure 1. This "zoomed away" perspective lets more of the software system be presented on one screen. Colors of the statements are still visible at this scale, and the relative colorings of many statements can be compared. This approach presents the source code in a fashion that is intuitive and familiar because it has the same visual structure as the source code viewed in a text editor. This miniaturized view can display many statements at once. However, even for medium size programs, a significant amount of scrolling is still necessary to view the entire system. For example, the subject program for our feasibility study, which consists of about 60KLOC, requires several full screens to be represented with this view. Monitoring a program of this size would require scrolling back and forth across the file-level view of the entire program, which may cause users to miss important details of the visualization.

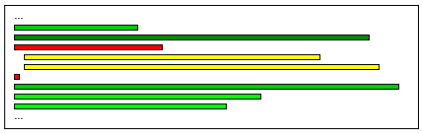


Figure 2: Example of file-level view.

System level The system level is the most abstracted level in our visualization. For the representation at this level we use the treemap view developed by Schneiderman [Shneiderman 1992] as well as extensions to this view developed by Bruls and colleagues [Bruls et al. 2000]. The treemap visualization is a two-dimensional, space-filling approach to visualizing a tree structure in which each node is a rectangle whose area is proportional to some attribute of that node. Treemaps are extremely effective in showing attributes of leaf nodes by size and color coding. We chose to use treemaps because they are especially effective in letting users spot unusual patterns in the represented data. Moreover, treemaps can efficiently encode information for large-sized programs. In a 1280x1024 display, we can visualize, on average, programs of more than 4,000 files [Shneiderman 1992].

Figure 3 shows an example hierarchy and the resulting treemap. The traditional tree view shows eight nodes: the five leaf nodes representing program classes of varying sizes (shown in parentheses) and the non-leaf nodes representing their packages. The treemap view shows these eight nodes as rectangles whose area is proportional to the relative size of the file (or package). For example, the rectangle for `HashMap` occupies 40% of the treemap area because the size of `HashMap` is 40% of the size of the `java` package.

An algorithm to build treemaps (1) starts with a rectangle that represents the root node and occupies the whole visualization area, (2) divides the root-node rectangle so that each child is allotted an area proportional to its size (or the sum of the sizes of its leaves), and (3) recurses for each of its children until the leaf nodes are reached.

For our system-level view, we build a tree structure that represents the system. The root node represents the entire system. The intermediate non-leaf nodes represent modularizations of the system (e.g. Java packages). The leaf

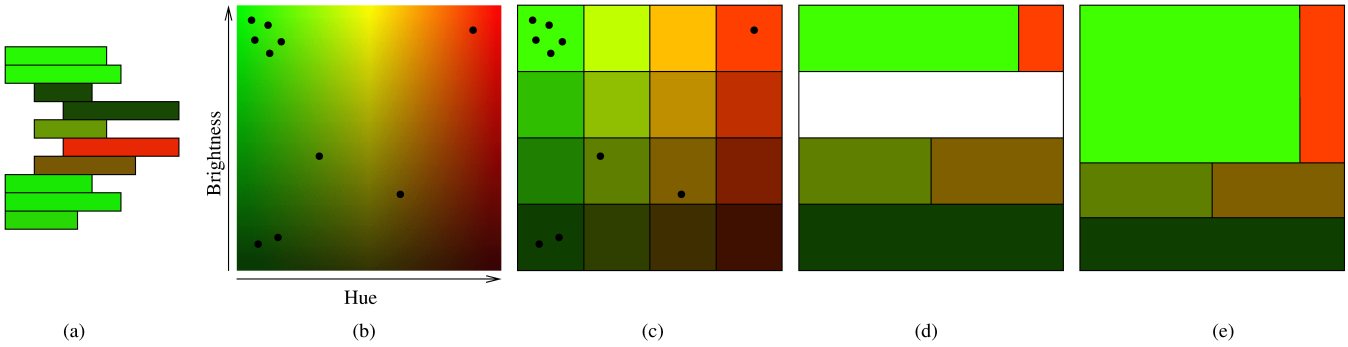


Figure 4: Example that illustrates the steps of the treemap node drawing.

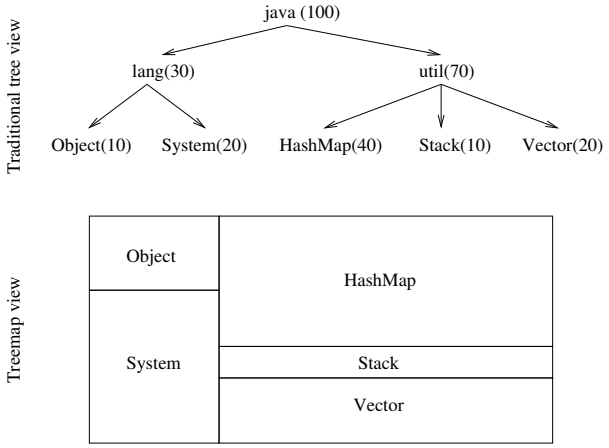


Figure 3: Example of treemap view applied to a sample hierarchy.

nodes represent source files in the system. We then apply the treemap visualization to this tree. The size of the leaf nodes is proportional to the number of executable statements in the source file that it represents.

2.2 Coloring

We use coloring to summarize information about the program-execution data. The coloring technique that we apply is a generalization of the coloring technique that we defined for fault-localization [Jones et al. 2001]. In the following, we first describe the general coloring mechanism, without considering the different levels of the representation. Then, we describe how the coloring approach maps to the three levels. The key idea of our coloring is to represent one- or two-dimensional information for each statement using the hue and the brightness components.

Hue component For the hue component, we use the continuous spectrum of colors that goes from red to yellow to green. The range of hues considered is therefore one third of the color wheel. The reason why we consider only this subset of colors is because of the analogy with a traffic light: colors red, yellow, and green intuitively convey the concepts of *danger*, *caution*, and *safety*, respectively, and can therefore be used to identify statements that require high, medium, or no attention. Although to account for different types of color-blindness, we could use other ranges of the color spectrum.

Without loss of generality, we express the hue in terms of degrees on the color wheel, with red represented by value 0 and green represented by value 120. Each statement is assigned a hue in such range. The way colors are assigned to statements depends on the kinds of program-execution data represented. Section 4 provides examples of uses of the coloring information for two applications.

Brightness component For the brightness component, we use the entire range of possible values. We express the brightness using a real number and assign value 0 to the minimum brightness and 1 to the maximum brightness. Again, the way the value for brightness is assigned to each statement depends on the kind of program-execution data represented, as shown in Section 4.

The coloring applies differently to the different representation levels. For the statement-level and the file-level representations, no mapping is necessary: for each statement, the color (i.e., hue and brightness) of the statement is used to color the corresponding line of code in the statement-level representation and the corresponding line of pixels in the file-level representation. (For the sake of simplicity, we assume that each line of code contains at most one statement. If this is not the case, the code can always be suitably formatted to satisfy this requirement, or the color can be averaged.)

For the system-level representation, there is no one-to-one mapping between statements and visual entities. Therefore, we defined a mapping to maintain color-related information in the treemap view. Each leaf node (i.e., rectangle) in the treemap view represents a source file.

To map the color distribution of the statements in a source file to the coloring of the node that represents that source file, we use, in turn, a treemap-like representation to further partition each node. (In this sense, we are embedding a treemap within each treemap node.) For example, if half of the statements in a source file were colored bright red, and the other half were colored dark green, the node would be colored as such—half of it would be colored bright red and half of it would be colored dark green.

However, using a traditional treemap algorithm for coloring the nodes would likely cause the colors to be laid out in a different fashion for different files. For example, say the colors assigned to the statements in source file A were evenly distributed among four colors: bright red, dark red, bright green, and dark green. To color the node in the treemap view, we may use a traditional treemap algorithm to further divide node A (that represents source file A) into four equally-sized blocks, each colored by one of the specified colors. However in a traditional treemap algorithm, relative placement of nodes is not guarenteed. So, in node A, the bright red block may be placed in the upper-right corner, but in node B that represents similar proportions of colored

statements, the bright red block may be placed in the lower-left corner. In a treemap view that contains many nodes, a non-uniform look for all nodes will likely cause confusion as to where the boundaries of the nodes lie. Therefore, we chose to keep the same layout of colors within each node while still showing the color distribution in a treemap-like fashion. The layout we use is characterized by varying the hue across the horizontal axis and by varying the brightness across the vertical axis. Figure 4(b) shows an example of this layout.

This layout determines the relative placement of the colors within each treemap node, but does not define how the colors are mapped to colors assigned in the statement-level or file-level representations. We thus defined a technique for skewing the colors of Figure 4(b) to present the appropriate proportions of colors assigned while preserving the layout of the colors.

We explain this technique while illustrating it on the example in Figure 4. Assume that the sample miniaturized source-code view shown in Figure 4(a) is a source file composed of a set of statements, with related colorings, to be mapped into a treemap node.

The skewing of the color layout is performed in four steps. The first step plots the color of each statement onto a coordinate system with hue varying across the horizontal axis and brightness varying across the vertical axis. For the considered example, this step would result in the points plotted on the hue/brightness space in Figure 4(b), in which each point represents a statement in Figure 4(a) positioned at the appropriate hue and brightness.

The second step segments the space horizontally and vertically into equal-sized blocks to create a discrete bucket for each block, so as to categorize the statements' colors. This segmentation is shown in Figure 4(c). For the sake of simplicity, in this example, we use only four segments vertically and four segments horizontally, resulting in sixteen blocks; however, in a real application, we would normally perform a finer-grained categorization. After the segmentation is complete, each block is drawn with a representative color—the median color of the colors in the block.

The third step determines, for each row, the width of each block. To this end, we compute the ratio of the number of statements in the block to the number of statements in the entire row. The width of each block is proportional to this ratio. The widths of the blocks for the considered example is shown in Figure 4(d). The technique assigns the leftmost block in the first row 5/6th of the total width of the node because five of the six points in the row fall into this block. Likewise, the coloring technique assigns the rightmost block the remaining 1/6th of the width of the node. The middle two blocks in the first row are eliminated (i.e., they are assigned width 0) because they contain no points. Note that the technique assigns no widths for the second row because no points fall into this row.

The final step determines the height of each row by computing the ratio of the number of statements in the row to the number of statements in the entire node. The height of the blocks for the considered example is shown in Figure 4(e), which is the final representation of the node. The technique assigns the first row 6/10th of the total height of the node because six of the ten points in the node fall into this row. The last two rows are each assigned 2/10ths of the total height of the node.

This coloring technique results in blocks that are proportional in size to the number of statements plotted in them and, in addition, maintains the layout of the color blocks for each node. For example, the brightest green block, which contained five of the ten statements, results in half of the total area of the node ($5/6 * 6/10 = 1/2$).

2.3 Representation of Executions

To represent executions, we use an *execution bar*: a virtually infinite rectangular bar, of which only a subset is visible at any time. The bar consists of bands of the same height of the bar but of minimal width. *Minimal width* refers to a width that is as little as possible but can still be seen. The actual width depends on the characteristics of the graphical environment, such as the size and resolution of the display. Figure 5 shows a simple example of an execution bar.

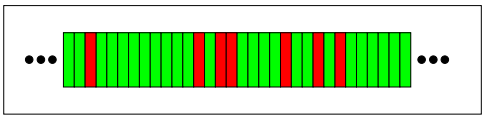


Figure 5: Example of execution bar.

Each band in the execution bar represents a different execution of the monitored program in the field (i.e., a run of the program and the data collected during such execution). Depending on the kind of program-execution data that we are representing, the bands in the execution bar may or may not be colored. For the coloring of the bands, we can use one or both of the two dimensions that we use for the code coloring: hue (from red to green) and brightness.

We defined the execution bar to be of *virtually infinite* size to account for a high and continuously increasing number of program-execution data collected from the field. Because we can show only a part of the execution bar on the screen, we assume the actual implementation of an execution bar to provide navigation capabilities, such as scroll bars.

2.4 Filtering and Summarization

To support the investigation of a possibly high number of program-execution data, our visualization technique encompasses filtering and summarization capabilities. Before describing filtering and summarization, we briefly discuss the concept of execution properties. *Execution properties* are properties that we associate with each execution. Examples of execution properties are the version of the Java Virtual Machine used to perform the execution, the ID of the user that performed the execution, and the name and version of the operating system used.

The set of execution properties collected may depend on the specific execution context and on the goal of the monitoring. For the discussion of filtering and summarization it is enough to know that we consider execution properties that can be expressed as a set of alphanumeric pairs $(key, value)$. Table 1 presents an example for the four properties mentioned above.

java.version	=	1.4.1_01
user.id	=	nXrPEQ7zq8w5JeY99FAxfThrFn
os.name	=	Linux
os.version	=	2.4.18-18.8.0

Table 1: Four example properties.

Section 3 discusses in greater detail the specific set of properties that we currently collect from deployed software.

Filters A filter lets us select only a subset of executions to be visualized. A user can include or exclude a set of executions based on the properties of such executions. For example, the user may choose to show only the executions that were run at a particular site, on a particular day, and to exclude those executions that raised a particular type of exception. More precisely, a filter is expressed as a disjunction or conjunction of predicates over the set of execution properties, with the syntax described in Table 2. For example, the following filter

$$(java.version \neq \text{'1.3.0'}) \; and \; (os.name = \text{'}Linux\text{'})$$

would select only those executions of the monitored program for which the version of the Java Virtual Machine used is *not* 1.3.0 and the operating system is *Linux*.

⟨*filter*⟩	::=	⟨*predicate_list*⟩
⟨*predicate_list*⟩	::=	⟨*predicate*⟩ | '(' ⟨*predicate_list*⟩ ⟨*bool_op*⟩ ⟨*predicate_list*⟩ ')'
⟨*predicate*⟩	::=	⟨*property*⟩ ⟨*op*⟩ ⟨*value*⟩
⟨*op*⟩	::=	'=' | '≠'
⟨*bool_op*⟩	::=	'and' | 'or'
⟨*value*⟩	::=	alphanumeric string
⟨*property*⟩	::=	property name

Table 2: Syntax for the filters.

Summarizers A summarizer lets us aggregate the program-execution data for a set of executions. A summarizer is simply expressed as a list of properties over which to aggregate:

⟨*summarizer*⟩ ::= (⟨*property*⟩)∗

For example, summarizer "*java.version*, *user.id*" would group all the executions for which the properties *java.version* and *user.id* have the same value. This operation corresponds to identifying equivalence sets in the executions with respect to the specified properties. Currently, the only equivalence relations that we consider is equality, but we plan to extend our technique to include a richer and more powerful set of relations. From the visualization standpoint, all the executions in an equivalence class are represented by only one band in the execution bar. If the summarization is performed for a representation that involves coloring of the execution bar, the color of each band is computed by averaging the color (i.e., hue and brightness) of all the bands whose executions are in the corresponding equivalence class.

Filtering and summarization are powerful instruments for managing, investigating, and understanding the large amount of program-execution data. Filtering can help the user focus on only a subset of executions at a time. Summarization can help the user identify correlations among executions. Section 4 provides examples of the usefulness of these two features.

3 The Toolset

In this section, we describe the GAMMATELLA toolset. Besides implementing the visualization approach described in Section 2, GAMMATELLA also 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.

3.1 InsECT

Before describing INSECT, we introduce the concepts of code coverage, profiling, and instrumentation. *Code coverage* is a measure of the extent to which some entities in a program have been exercised as a consequence of one or more executions of the program. In general, code coverage for a given type t of entities with respect to a set of executions E is expressed in terms of the number of t entities exercised by E over the total number of t entities. For example, statement coverage is expressed as the number of statements in the program exercised by the considered executions with respect to the total number of statements in the program.

Profiling is a measure of how much some entities in a program have been exercised during one or more executions of the program. In general, profiling for an entity e with respect to a set of executions E is expressed in terms of how many times e has been exercised by E.

The measuring of code coverage and profiling generally requires *instrumentation* of the code: probes are inserted in specific parts of the code prior to execution so as to report when they are executed. For example, for edge profiling, a probe can be inserted for each edge so that, as the program executes, it reports the number of times each edge is executed.

The *Instrumentation, Execution, and Coverage Tool* (INSECT) that we developed is a modular, extensible, and customizable instrumenter and coverage analyzer written in Java. Within GAMMATELLA, we use INSECT to instrument for statement coverage, branch coverage, call coverage, exception coverage, and statement and branch profiling. In addition, INSECT reports, for each execution, various information about the user environment, including a unique identifier for the machine and the user, the operating system brand and version, and the Java Virtual Machine brand and version.

It is worth noting that, with InsECT, we can instrument the whole program or only parts of it. For example, in the case of a program that consists of multiple components, we can instrument only a subset of the components (e.g., the ones developed in house or the most critical ones) and collect execution data only for those components.

All the above information is collected during the execution by monitor classes that are called by the probes inserted in the code. At the end of the execution, the information is dumped, compressed, and sent back to a central server over the network. For the sake of the description, and without loss of generality, we assume a network connection to be available. If this is not the case, we can store the information locally and send it when a connection is available.

To be general, we use the SMTP protocol [Postel 1982] to transfer the program-execution data from the users' machines to the central server collecting them (*collection server* hereafter). The compressed data are attached to a regular electronic-mail message whose recipient is a special user (*collection user* hereafter) on the collection server and whose subject contains a given label (*coverage label* hereafter) and an alphanumeric ID that uniquely identifies both the program that sent the data and its version. The only requirement for the collection server is thus to run an SMTP server.

3.2 Data Collection Daemon

The *Data Collection Daemon* (DCD) is a simple tool written in Java that runs as a daemon process on servers on which we store the execution data. Each instance of the tool monitors for information from all instances of a specific version of a specific program, provided to the tool in the form of the

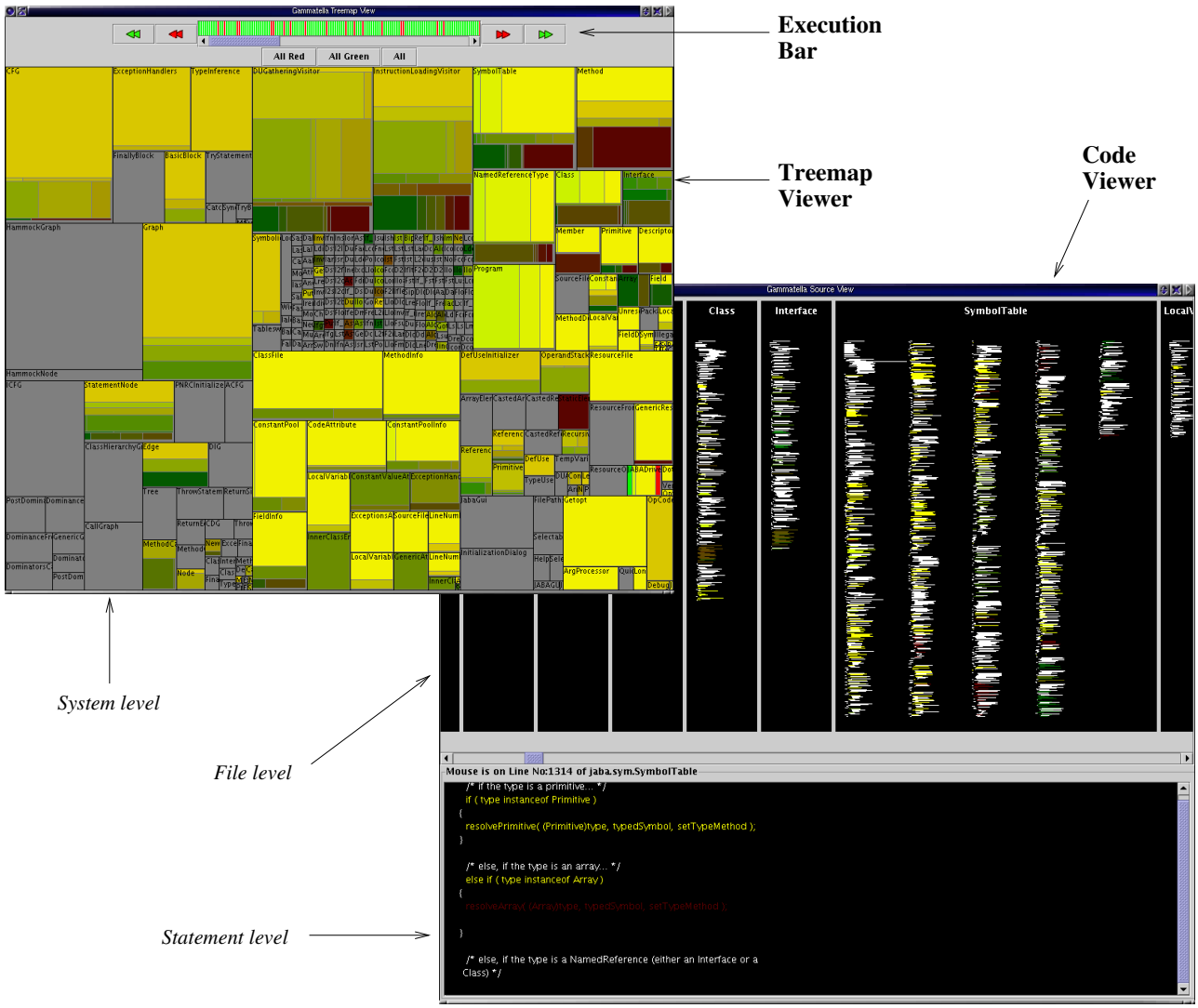


Figure 6: A screenshot of the GAMMATELLA Program Visualizer.

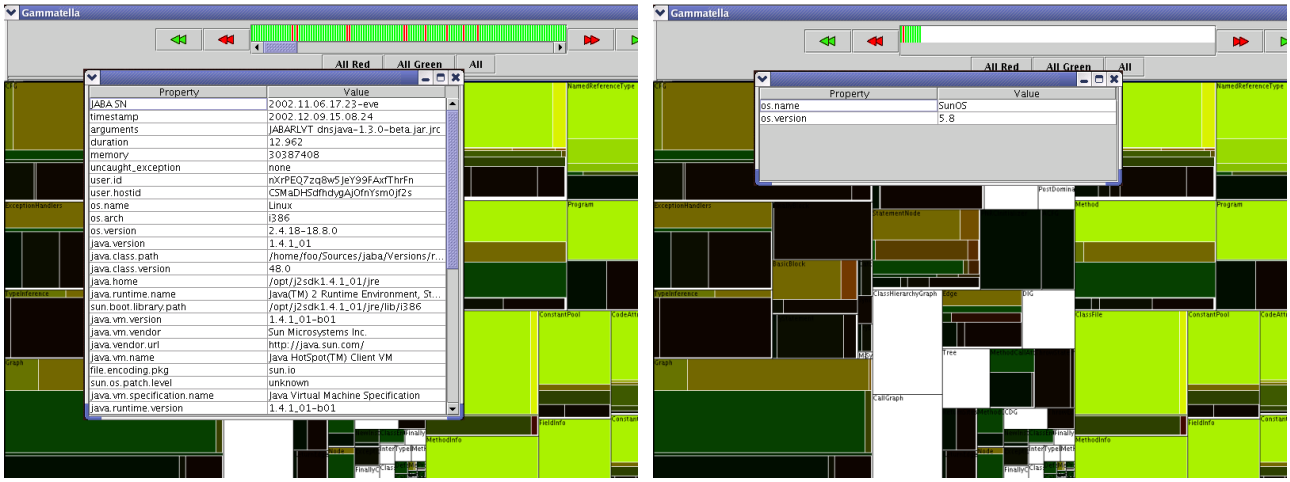
corresponding alphanumeric ID. The tool, upon execution, retrieves the incoming mail for the collection user from the collection server. To facilitate access of the data from different machines, we use the Internet Message Access Protocol (IMAP [University of Washington 2002]).

For each message retrieved that contains coverage data, DCD extracts the attachment from the message, uncompresses it, and suitably stores the program-execution data in a database. The additional information about each execution, such as the Java Virtual Machine version and the user ID, are stored as properties of the execution. This approach lets us efficiently perform filtering and summarization over the executions, as described in Section 2.

3.3 Program Visualizer

The *Program Visualizer* (*PV*) is the module of GAMMATELLA that implements the visualization technique described in Section 2. PV is divided into JavaBeans components written in Java using the graphical capabilities of the Swing toolkit. PV uses the coverage-analysis module of InsECT to retrieve and query the coverage data stored by the DCD. These data are used to update all appropriate views. PV is composed of the following components: Execution Bar, Code Viewer, and Treemap Viewer, shown together in Figure 6.

Execution Bar In the Execution Bar, executions are displayed as (possibly colored) vertical bands, as described in Section 2. Each band represents one or more executions (this latter case occurs when using summarizers). The user of our tool can interact with the execution bar in a variety of ways. The scroll bar below the Execution Bar lets the user quickly navigate the set of executions. The user can also use the two pairs of red and green arrows on each side of the bar to navigate to the previous (or next) red- and green-colored execution, respectively. Selecting an execution or a set of executions causes the other displays to update their views to show only the information pertaining to the selected executions. Executions can be selected by left-clicking with the mouse on the corresponding band(s). In addition, the three buttons under the execution bar let the user select all red-colored, all green-colored, or all executions. Right-clicking on a band causes a modal window to appear, which shows one of two possible types of information: (1) if the band represents only one execution, it shows all the properties of the execution in plain textual format (Figure 7(a)); (2) if the band represents the summary of more executions, it



(a) (b)

Figure 7: Windows that show executions properties.

shows only the common properties of those executions (Figure 7(b)). Filters and summarizers are currently provided to the tool using a configuration file that can be loaded dynamically. In the final implementation of the tool, filters and summarizers will be defined using the GUI as well.

Code Viewer The Code Viewer displays both the file-level view and the statement-level view described in Section 2. Right-clicking on a statement in the file-level view causes a context menu to appear that permits the viewing of different types of information about the statement, such as the number executions that covered it or the types of exceptions that were thrown by the executions that covered it. The statement-level view shows a small number of statements in its full-sized text, at the bottom of the Code Viewer window. Moving the mouse cursor over the file-level view causes the statement-level view to display those statements under the cursor, so allowing the user of the PV to investigate sections of code in detail.

Treemap Viewer The Treemap Viewer displays the system-level view described in Section 2. On the upper-left corner, each node shows the name of the file it represents (without the `.java` extension), which also corresponds to the name of the public class in the file. Hovering the mouse over a node causes a tool-tip to appear describing the name of the package to which the represented file (i.e., the classes in the file) belongs. We utilized the TreeMap Java Library by Bouthier [Bouthier 2002] to implement the treemap algorithm that performs the layout of the source-file and package nodes. We also utilized the squarified treemap algorithm built into the library [Bruls et al. 2000] to present more visible nodes. The Treemap Viewer displays a treemap at the finest level of detail. In future work, we will extend the Treemap Viewer to provide capabilities for viewing the package structure at different levels (e.g., by collapsing all the files in one package into only one node and incrementally expanding a package into its next-level sub-packages and/or source files).

The three visual components in the PV communicate and interact with one another. For example, the selection of executions in the execution bar causes the source-level, file-level, and system-level views to update their displays to display only the information about those executions. Due to the component-based architecture of our implementation, additional views can be integrated and the current components can be updated and substituted with low effort.

The PV dynamically updates the information displayed to reflect the latest data. As the DCD receives additional executions from the field, the visualizations are updated based on the new information. This approach permits an almost real-time monitoring of the behavior of the monitored program by developers and maintainers.

4 Applications

To investigate the usefulness of our data collection and visualization technique, we applied it to two tasks: investigation of exceptions generated during users' executions and profiling analysis. We also performed a feasibility study for the former. In the rest of this section, we first describe the two applications and then present the feasibility study.

4.1 Exceptions Analysis

We applied our technique to the visualization of exception-related information. To this end, we used an approach similar to the approach that we previously used for fault localization [Jones et al. 2001]. The idea is to 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. Red, yellow, and green are used in this case to represent "very likely," "possibly," and "unlikely," respectively.

Consider a statement s, a set of executions that result in an uncaught exception F, and a set of executions that do not result in an uncaught exception P. Let f represent the percentage of executions in F that execute s, and let p represent the percentage of executions in P that execute s. We assign to s a hue value based on the percentages f and p. As a result, if p is larger than f (that is, if a larger percentage of the executions terminating without an uncaught exception executed s than the percentage of executions terminating with an uncaught exception) s is assigned a more green hue to represent some confidence in its correctness. Conversely, if f is larger than p for s, a more red hue is assigned to represent suspiciousness of the correctness of s.

We use the brightness component to encode the relevance of the information represented by statement s. More pre-

cisely, we use the larger of the two percentages f and p. Reference [Jones et al. 2001] provides additional details on the described coloring technique.

4.2 Profiling Analysis

The second application of our technique is the visualization of profiling information. The goal is to let the user identify *hot spots* in the programs (i.e., places in the code that are executed most often). This information is valuable for several software-related tasks such as targetting parts of code for optimization, determining feature-usage, aiding in the reduction of software-bloat, and aiding the guidance of future enhancements.

In this case, we assign a color to each statement in the program to represent how often the statement is executed: a red statement is executed very often, a yellow statement is executed often, and a green statement is executed rarely. For each statement s and set of executions E that traverse s, we first assign to s a score by adding the number of times s is traversed in all executions in E. Then, we normalize the computed score for all statements over the range 0–120, and we assign to each statement a hue corresponding to the normalized score.

For this application, we do not currently need to represent two-dimensional information. Therefore, we assign a constant value to the brightness component of the coloring. In future work, we will investigate the usefulness of the brightness component to represent additional information about the profiling. First, we will investigate the use of brightness to distinguish between statements that are executed by only a small number of executions and statements that are executed by most executions. Second, we will use the brightness to dim the information pertaining to older executions. (Because the profiling information is likely to change over time, it is important to characterize the time frame of the visualized information.)

4.3 Feasibility Study

We implemented in Gammatella the visualization for exceptions analysis described in Section 4.1 and performed a feasibility study using a real system: Jaba. Jaba (Java Architecture for Bytecode Analysis [Aristotle Research Group 2003]) is a framework for analyzing Java programs developed in Java within our research group that consists of 550 classes and approximately 60KLOC. Jaba consists of components that read bytecode from Java class files and perform analyses such as control flow and data flow, thus enabling the development of program-analysis techniques and program-analysis-based software-engineering tools for Java.

We instrumented Jaba using the InsECT component of Gammatella and released it to a set of users who agreed to have information collected during execution and sent back to our server. Although instrumentation overhead is not the main concern for this feasibility study, it is worth reporting that the instrumentation caused a 28% degrade in Jaba's performances.

We distributed the instrumented version of Jaba to nine people who already used Jaba for their work. Five of the nine people involved in the study are working in our department: two are part of our group and use Jaba for their research; another two are students working in our department who use Jaba for a graduate-level project; the last one is an M.S. student who developed a regression testing tool on top of Jaba. The remaining four people involved in the study are two researchers and two students working on a research project in two different universities abroad.

After releasing the instrumented version of Jaba, we started the DCD and the PV on a dedicated machine in our lab. We used a different machine as the collection server. While users used Jaba for their work, the program-execution data was sent to the collection server, the DCD retrieved and stored the data, and PV visualized the corresponding information on a monitor located in the common area of our lab.

Within the first month, we collected more that 1,000 executions. Using Gammatella, we have been able to save the information about the executions automatically and visualize them. Figure 8 shows a screenshot of Gammatella visualizing all the execution data received at the time of the submission (1,214). We have also been able to use Gammatella to perform an initial investigation of the data.

The first, immediate finding of our investigation, not directly related to the exceptions analysis, was that a number of classes were never used in any of the executions, illustrated by gray nodes in the treemap view. In particular, the entire package of Jaba responsible for performing dominance analysis was never utilized. The treemap view provided by Gammatella let us spot immediately the large uncovered parts and identify the corresponding parts in the code. Such information, if confirmed by subsequent executions, could motivate the next release of a trimmed-down or optimized version of Jaba—one without the unused package and/or classes.

Another finding of our investigation is related to the occurrence of exceptions and their meaning in terms of anomalies in the program behavior. By inspecting the program-execution data using Gammatella, we realized that in most cases exceptions are raised because of trivial errors on the user side (e.g., errors in the parameters passed to Jaba and errors in setting the classpath). In all such cases, considering the corresponding execution as a failure is misleading and distracting from real sources of errors. Using the tool, we have been able to identify at least two exceptions that are always generated due to users' errors. Then, we used such information to filter out all the executions resulting in an uncaught exception of one of those two types, so reducing the amount of spurious information.

Finally, an important finding was that there is a specific combination of operating system and Java Virtual Machine for which executions of Jaba fail systematically. Using the summarization facilities of the tool and summarizing per user, we discovered that all executions for one of the students in our lab were terminating with an exception. By looking at the execution properties for the executions coming from that user, we discovered that all the failing executions were run using the Sun Java Virtual Machine version 1.4.0 on Solaris 2.8, a combination that no other user was using and that caused Jaba to fail. It is worth noting that this kind of problem is common for software that have to function in several different environments and configurations, and therefore cannot be adequately tested in-house.

5 Related Work

There are several visualization techniques that are related to our approach.

Eick and colleagues developed the SeeSoft system [Eick et al. 1992], which shows source code by mapping each line of code to a row of pixels. We utilize a similar technique for our file-level view of the code. We have extended this work by applying our coloring technique to the visualization, as well as applying it to a new domain.

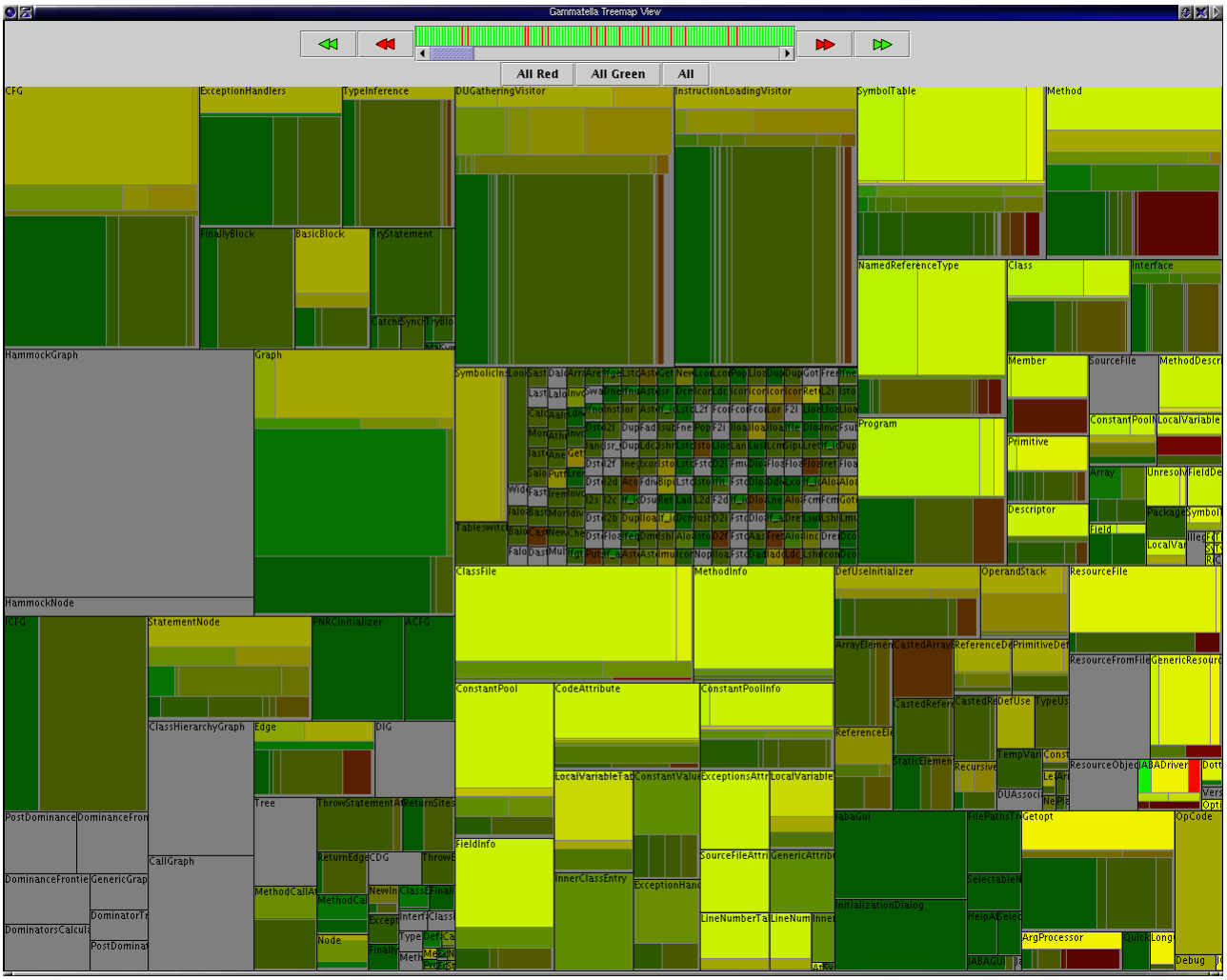


Figure 8: JABA execution data visualized in GAMMATELLA.

Shneiderman developed the treemap visualization [Shneiderman 1992] for visualizing hierarchical data in a space-filling manner. Bruls and colleagues developed an approach [Bruls et al. 2000] to display treemaps in a "squarified" fashion to reduce the aspect ratio of the nodes. We utilized both techniques for our system-level view of the code. We have extended this work by defining a technique for coloring the nodes of the treemap in a treemap-like fashion that has two properties: (1) preservation the color layout within the nodes, and (2) visualization of the appropriate proportions of colors to reflect the coloring of the entities represented by each node. Such a technique can be applied in general for the layout of treemaps that represent flat hierarchies (i.e., with depth of one) in situations in which preservation of node layout is important.

Baker and Eick developed the SeeSys system [Baker and Eick 1995], which shows source code in a treemap fashion. They used this system to show various properties of the source code. We utilize this idea of applying treemaps to software to visualize properties of the software. In our approach, we use a different technique, based on visualization of two-dimensional data, to represent the information within the treemap nodes.

Leon and colleagues, in their work on observation-based testing, describe some uses of multivariate visualization [Leon et al. 2000] applied to execution profiles. They use multivariate visualization to project many-dimensional profiling information onto a specific visualization, a two-dimensional scatter plot, which is then used to perform tasks such as clustering. Our approach aims to provide a generic visualization framework that can be instantiated for different tasks.

Reiss and Renieris developed the Bloom system [Reiss and Renieris 2001], which provides a framework for software visualization and exploration. Similarly, we have several components that visualize software, its execution, and its properties. In fact, the visualization techniques described in this paper may probably be also implemented leveraging the Bloom framework.

Storey and colleagues developed the SHriMP Views system [Best et al. 2001; Storey and Müller 1995], which is a visualization based on zooming to display hierarchical views of software. Their work is mainly concerned with exploring the software itself and its hierarchical structure, whereas the technique described in this paper is directed at visualizing program-execution data and its relation to the program.

Jones and colleagues developed the Tarantula [Jones et al. 2001] system to visualize test-case information for fault localization. In this paper, we utilized and abstracted the color-mapping concepts from that work for a variety of purposes. In fact, Tarantula's fault-localization technique could be a specific instance of the approach described in this paper.

6 Conclusion

In this paper, we presented a new approach for visualizing program-execution data collected from deployed instances of a software system. Our technique is generic enough to allow for representing different kinds of data, and allows for investigating such data visually to study the software system's behavior. Furthermore, because of its hierarchical approach to visualization and its coloring, filtering, and summarization capabilities, the technique lets the user efficiently visualize and explore large amounts of data and large programs.

We also presented the GAMMATELLA toolset, which implements our approach, and a feasibility study in which we used the toolset on a real program deployed to a set of real users. Besides showing the feasibility of the approach, the study led to some initial discoveries about the subject program and the way it is used. Although we cannot consider such discoveries more than very preliminary results, they provide some evidence of the usefulness of the approach. The feasibility study also helped us identify a number of important directions for future work.

First, we will investigate scalability issues. To this end, we will expand the initial study to involve additional participants. We will also consider using other widely-used and freely-available subjects, such as open-source software systems. Finally, we will investigate monitoring at a higher level of abstraction than statements (e.g., procedures).

Second, we will further investigate the use of the approach for exception analysis. We will investigate the use of data-mining techniques to improve the visualization (e.g., by automatically grouping correlated executions) and consider monitoring and visualizing different kinds of information, such as features usage and memory layouts).

Third, we will investigate additional tasks to which our approach can be applied. During these investigations, we may discover the need for optimization of the visualization for the specific tasks, such as the need for different summary colorings in the treemap, or the need for new visualizations altogether. These investigations will also give us the opportunity to make our framework easier to customize, so as to let users develop their own visualization.

Finally, from a more practical standpoint, we are investigating the possibility of providing a web interface to some of the visualizations so that they can be publicly displayed and monitored.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. Preeti Bhat helped with the implementation of the visualization components in the Program Visualization module of GAMMATELLA. Anil Chawla helped with the development of INSECT and its integration in GAMMATELLA. John Stasko provided useful suggestions and comments.

References

ARISTOTLE RESEARCH GROUP, 2003. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

BAKER, M. J., AND EICK, S. G. 1995. Space-filling software visualization. *Journal of Visual Languages and Computing 6*, 2, 119–133.

BALL, T., AND EICK, S. G. 1996. Software visualization in the large. *Computer 29*, 4 (Apr.), 33–43.

BEST, C., STOREY, M.-A. D., AND MICHAUD, J. 2001. SHriMP views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*.

BOUTHIER, C., 2002. Treemap java library. `http://treemap.sourceforge.net/`.

BOWRING, J., ORSO, A., AND HARROLD, M. J. 2002. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, 2–8.

BRULS, M., HUIZING, K., AND VAN WIJK, J. J. 2000. Squarified treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, 33–42.

EICK, S. G., STEFFEN, J. L., AND SUMNER, E. E. 1992. Seesoft – a tool for visualizing line oriented software. *IEEE Transactions On Software Engineering 18*, 11 (Nov), 957–968.

GRAY, J., SLUTZ, D., SZALAY, A., THAKAR, A., VANDENBERG, J., KUNSZT, P., AND STOUGHTON, C. 2002. Data Mining the SDSS SkyServer Database. Tech. Rep. MSR-TR-2002-01, Microsoft Research, January.

JONES, J. A., HARROLD, M. J., AND STASKO, J. 2001. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'01)*, 467–477.

LEON, D., PODGURSKI, A., AND WHITE, L. J. 2000. Multivariate visualization in observation-based testing. In *Proceedings of the 22th International Conference on Software Engineering (ICSE'00)*, 116–125.

ORSO, A., LIANG, D., HARROLD, M. J., AND LIPTON, R. 2002. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, 65–69.

POSTEL, J. B., 1982. RFC821: Simple Mail Transfer Protocol. `http://www.ietf.org/rfc/rfc0821.txt`.

REISS, S. P., AND RENIERIS, M. 2001. Encoding program executions. *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)* (may), 221–230.

SHNEIDERMAN, B. 1992. Tree visualization with treemaps: A 2-D space-filling approach. *ACM Transactions on Graphics 11*, 1, 92–99.

STASKO, J., DOMINGUE, J., BROWN, M., AND PRICE, B., Eds. 1998. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA.

STOREY, M.-A. D., AND MÜLLER, H. A. 1995. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)* (Opio (Nice), France, October 16-20, 1995).

UNIVERSITY OF WASHINGTON, 2002. The IMAP Connection. `http://www.imap.org/`.

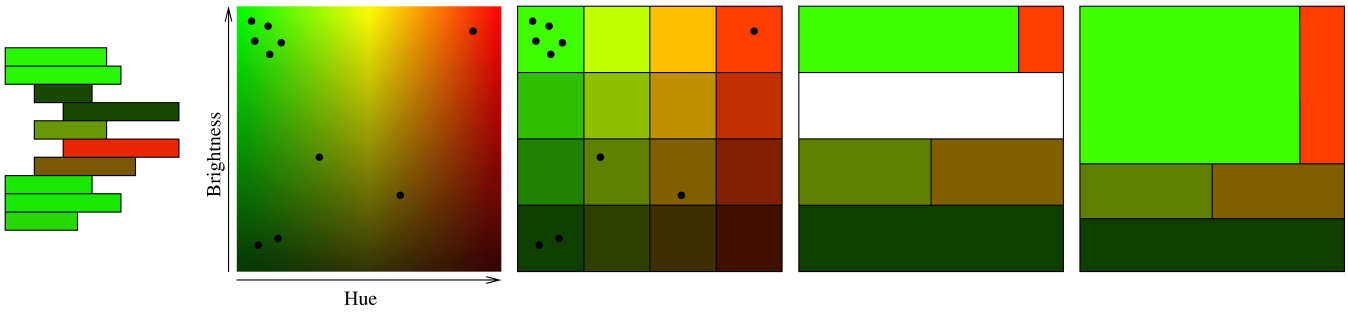


Figure 4: Example that illustrates the steps of the treemap node drawing.

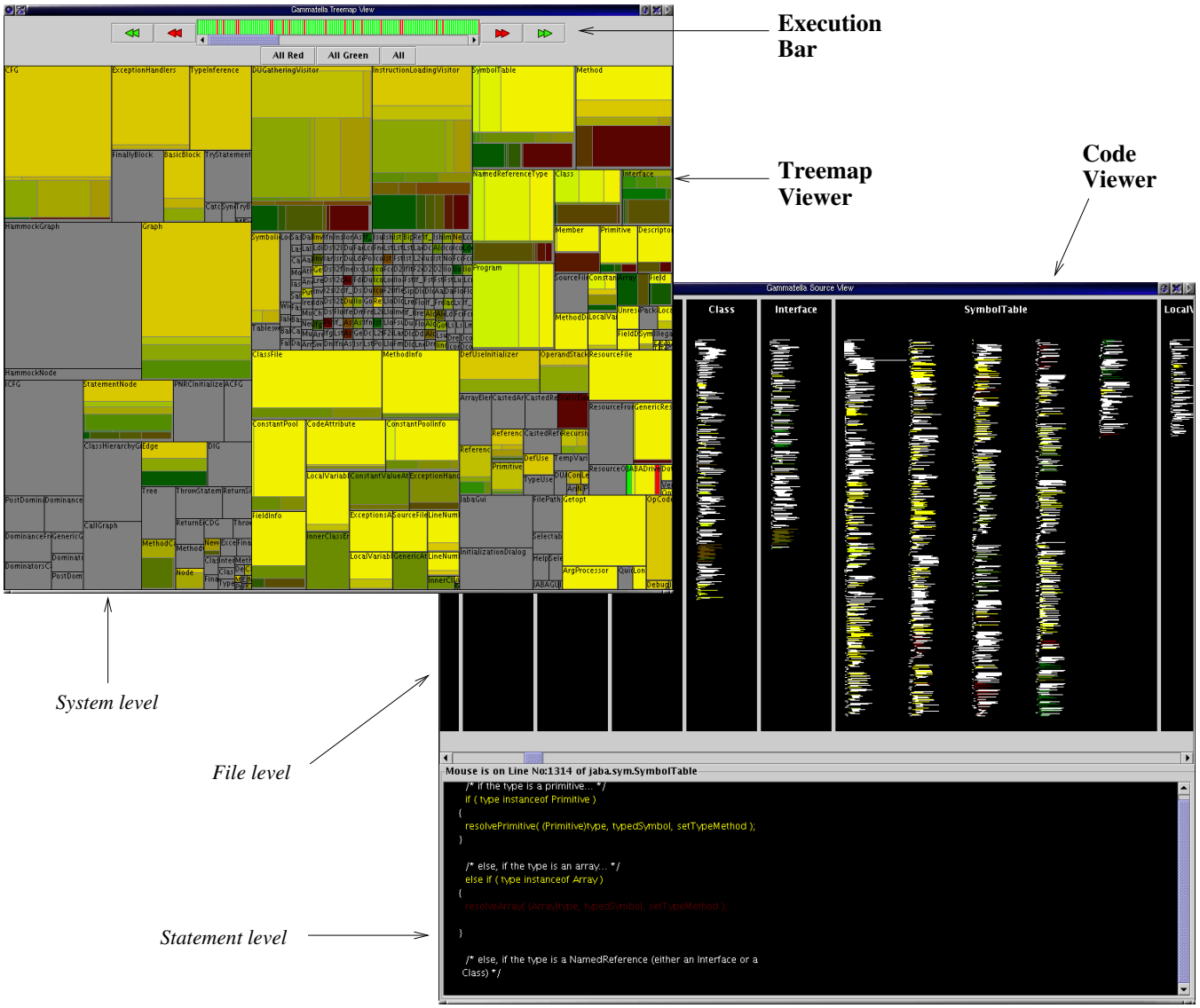


Figure 6: A screenshot of the GAMMATELLA Program Visualizer.