Gammatella: Visualizing Program-Execution Data for Deployed Software*

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

Abstract

Software systems 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, some of the authors presented the GAMMA approach, whose goal is to improve software quality by augmenting software-engineering tasks with dynamic information collected from deployed software. The GAMMA approach enables analyses that (1) rely on actual field data instead of synthetic in-house data and (2) leverage the vast and heterogeneous resources of an entire user community instead of limited, and often homogeneous, in-house resources.

When monitoring a large number of deployed instances of a software product, however, a significant amount of data is collected. Such raw data are useless in the absence of suitable data-mining and visualization techniques that support 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. Finally, we show how the visualization capabilities of GAMMATELLA facilitate effective investigation of several kinds of execution-related information in an interactive fashion, and discuss our initial experience with a semi-public display of GAMMATELLA.

Keywords

GAMMA technology, software visualization, remote monitoring

1 Introduction

Developing reliable, safe, and efficient 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. We believe, and our previous research suggests, that software development can greatly benefit from augmenting analysis and measurement tasks performed in-house with *program-execution data*—information, such as coverage data, exception-related information, and performance related data that is collected from deployed software while it is used in the field. We call this approach the GAMMA approach.[1] The GAMMA approach aims to improve software quality through continuous monitoring and analysis of software after deployment [6, 15, 17].

Monitoring of a high number of deployed instances of a software product, however, can produce a huge amount of program-execution data. For example, in a preliminary study that involved 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, it is easy to 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

*An earlier version of the material presented in this paper appeared in the *Proceedings of the ACM Symposium on Software Visualization* (June, 2003) [16].

[1] We call the approach GAMMA because it can be seen as the next phase after α-testing, performed in-house, and *beta*-testing, performed in the field by a set of selected users.

does the size of the data grow, but also their complexity: different kinds of data may have intricate relationships and dependences that require such data to be analyzed together to be understood.

Such a huge and complex amount of data cannot be analyzed manually. To be able to extract meaningful information about program behavior from the raw data and exploit their potential, we need suitable data-mining and visualization techniques. In particular, visualization techniques can be effective in transforming program-execution data into visual information that can be explored and understood [2, 9, 20, 22, 24].

In this paper, we present a new visualization approach that can efficiently represent different kinds of program-execution data and that facilitates investigation of 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—exception and profiling analyses—and a feasibility study. In the study, we use GAMMATELLA, displayed on a semi-public display in our lab, to collect, store, visualize, and investigate program-execution data gathered from instances of a real software system 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 interactive fashion and get meaningful insights into the monitored program's behavior. Our initial experience with the semi-public display of GAMMATELLA is also discussed.

The main contributions of the paper are:

- a new visualization approach that facilitates visualization of various kinds of program-execution data and interactive study of 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.

In the next section, we describe the new visualization approach that we use to represent program-execution data. Section 3 describes the components of the GAMMATELLA toolset and discusses our setup for and initial experience with a semi-public display of GAMMATELLA. Section 4 presents two applications of our approach and a feasibility study performed using the tool. Section 5 discusses related work. Finally, Section 6 presents some conclusions and discusses future work.

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.

To minimize the interaction required by a user to see all dimensions of the display, we chose to focus our attention on two-dimensional visualization techniques rather than three-dimensional techniques. For example, in our approach, the user is not required to rotate the display to reveal obscured features. Obviously, a height dimension could be added to our visualization to let more variables be displayed.

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, the visualization represents source code, and each line of code is suitably colored (in cases where the information being represented does involve coloring). Figure 1 shows an example of a colored set of statements in this view. For example, if we are visualizing statement-coverage information, each line of code is colored based on whether it was covered by the considered execution(s) (e.g., gray if not covered and green if covered). The statement level is the level at which users can get the most detail about the code. However, directly viewing the code is not efficient for programs of non-trivial size. To alleviate this problem, our visualization approach provides representations at higher levels of abstraction.

File level. The representation at the file level provides a miniaturized view of the source code. This technique is similar to the one introduced by Eick and colleagues in the SeeSoft system [3, 8]: the technique maps 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, significant scrolling is still necessary to view the entire system. For example, the subject program for our feasibility study, which consists of about 60,000 lines of code, 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.

System level. The system level is the most abstracted level in our visualization. The representation at this level uses the treemap view developed by Shneiderman [21] as well as extensions to this view developed by Bruls and colleagues [7]. 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, treemaps can visualize, on average, programs of more than 4,000 files [21].

In our development of the system-level view, we considered other visualization techniques such as Stasko and Zhang's Sunburst visualization [23] or Lamping, Rao, and Pirolli's Hyberbolic Tree visualization [12]. These techniques, however, focus more on the hierarchical structure of the information they represent, and use a considerable amount of screen space to represent such structure. For our application, the hierarchical structure of the program modules is less important than representing as much information as possible at each level of the hierarchy. With treemaps, the entire screen space can be used to represent the color information for the hierarchical level being considered (e.g., a package or the classes in a package) without using valuable screen space to encode redundant information about nodes' parents. The hierarchical structure is used only to group nodes belonging to common branches of the tree.

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, where the area of each rectangle 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 entire 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 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.

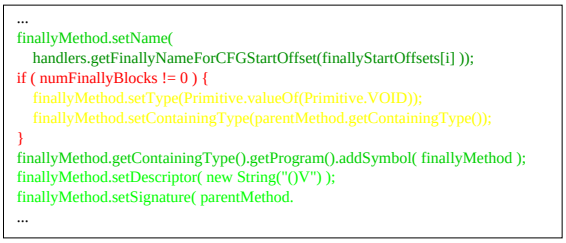


Figure 1: Example of statement-level view.

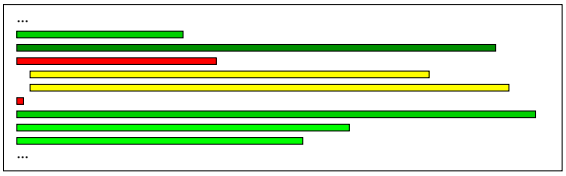


Figure 2: Example of file-level view.

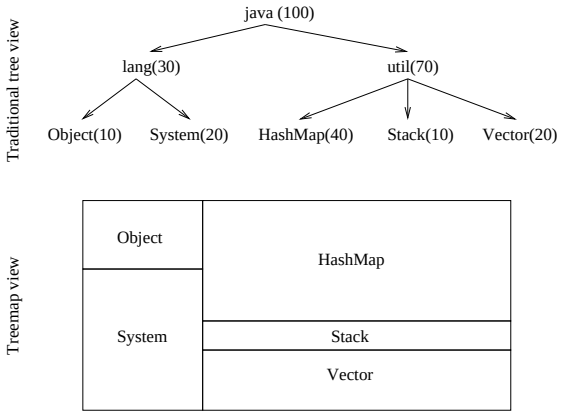


Figure 3: Example of treemap view applied to a sample hierarchy.

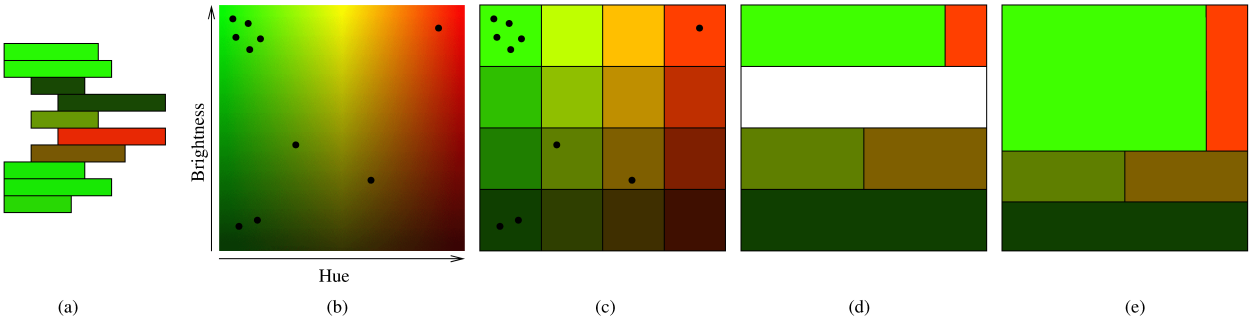


Figure 4: Example that illustrates the steps of the treemap node drawing.

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 defined by Jones, Harrold, and Stasko for fault-localization [11]. 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. If higher-dimensional information had to be visualized, the visualization approach could be suitably extended. For example, we could use saturation and textures to encode additional dimensions.

Hue component. For the hue component, our default color space is a continuous spectrum of colors from red to yellow to green. The range of hues considered is therefore one third of the color wheel. With this subset of colors, we convey notions analogous to 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 use other ranges of the color spectrum, our discussion here will be limited to the red-yellow-green sub-spectrum. The same ideas can be applied to other sub-spectra as well.

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 *min* to the minimum brightness and *max* to the maximum brightness. The values *min* and *max* are determined based on visual ranges that are perceptible to the human eye—typically, this means a value of about 0.3 for *min* and 1 for *max*. 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.

Each statement in the program is assigned a color depending on the task being performed, but the coloring applies differently to the different visual 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 the statements in a source file were colored bright red, and the other half were colored dark green, the treemap 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 nodes. For example, suppose 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 guaranteed. So, in node A, the bright red block may be placed in the upper-right corner, but in node B, which 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 appearance of the 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 file-level 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 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, the technique computes 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 example are shown in Figure 4(d). The technique assigns the leftmost block in the first row 5/6th of the total width of the node be-

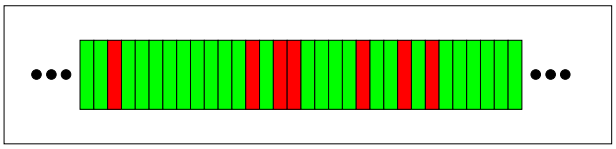


Figure 5: Example of execution bar.

cause 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 heights of the blocks for the example are 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.

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, our technique 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 includes 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	=	nXrPEQ7zq8w5JY9FAfThrFn
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 the user 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

$$(java.version \neq '1.3.0') \, and \, (os.name = 'Linux')$$

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.

Our language is simple and much less expressive than full-fledged query languages, such as SQL. For our purposes, however, our language is a good balance of functionality and usability. More powerful querying capabilities could be implemented, if required.

⟨*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 the user 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 classes in the executions with respect to the specified properties. 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 as the average color, in terms of both hue and brightness, among 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. Figure 6 shows a high-level view of GAMMATELLA and represents the flow of information throughout the various tools that are part of the toolset.

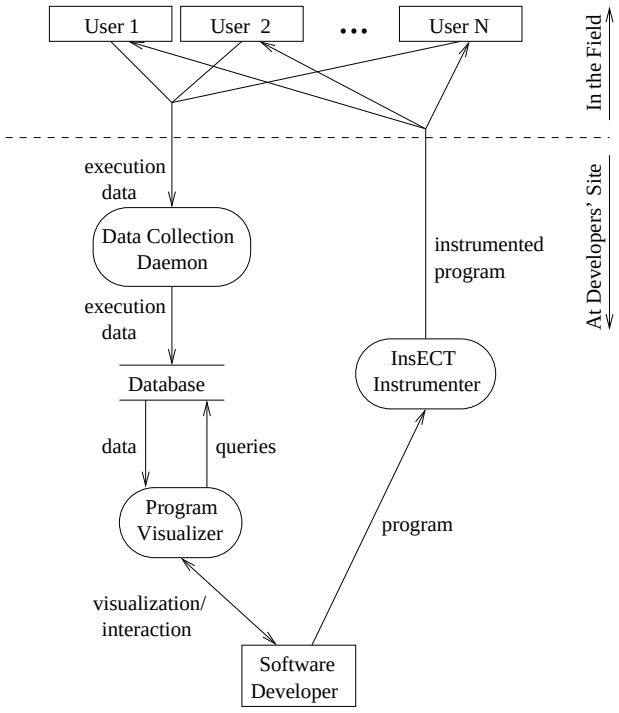


Figure 6: A high-level view of the Gammatella toolset.

GAMMATELLA is written in Java, supports the monitoring of Java programs, and consists of three main components: an Instrumentation, Execution, and Coverage Tool (called INSECT), 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 set N of entities with respect to a set of executions E is expressed in terms of the percentage of entities in N exercised by E. For example, statement coverage is expressed as the percentage 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 entities in a program have been exercised during one or more ex-

ecutions of the program. In general, profiling for an entity n with respect to a set of executions E is expressed in terms of how many times n has been exercised by E. Program profiles are often collected to identify where in the code a program spends its time.

Code coverage and profiling information can be gathered using two alternative approaches: instrumentation and access to the run-time system. *Instrumentation* works by (1) inserting probes in specific parts of the code prior to execution so as to report when they are executed, and (2) adding a monitoring mechanism to the code to record the information provided by the probes. For example, for statement profiling, a probe can be inserted for each statement so that, as the program executes, the probes report to the monitoring mechanism that keeps track of the number of times each statement is executed. *Access to the run-time system* can be used as an alternative to instrumentation when the run-time system provides an interface for gathering dynamic information during execution. For example, Java Virtual Machines usually provide an interface called JVMPI[2] (Java Virtual Machine Profiling Interface). *JVMPI* can be used to get notifications of various events, such as heap allocations or method calls, at run-time.

In Gammatella, we chose to use instrumentation instead of access to the run-time system for three main reasons. First, instrumentation gives us complete control over the kind of information collected. Second, we found instrumentation to be more efficient than JVMPI. Third, to take advantage of JVMPI, the user must launch the program using some specific parameters, thus complicating the use of JVMPI for deployed software.

Before developing InsECT, we considered existing instrumentation tools, such as Gretel [18] and a few commercial tools. None of the considered tools, however, provided the kind of flexibility and customizability that we needed with our approach. Moreover, none of the tools provided exception-coverage information, which is required for one of the applications that we consider.

InsECT is a modular, extensible, and customizable instrumenter and coverage analyzer that we developed in Java. InsECT inputs a Java program and outputs an instrumented version of the program that contains the probes for reporting executed entities along with a set of monitor classes that collect the information at runtime. InsECT works at the bytecode level and can instrument the whole program or only parts of it. For example, for a program that consists of multiple components, InsECT can instrument only a subset of the components (e.g., the ones developed in house or the most critical ones) so that as the instrumented program executes, it collects execution data only for those components.

Within Gammatella, InsECT instruments for statement coverage, branch coverage, call coverage, exception coverage, and statement and branch profiling. The instrumented program is then deployed to the customers for their use in the field. For exception coverage, InsECT inserts probes in the instrumented program so that, as the program executes in the field, it reports information for each exception thrown, the type of the exception, the `throw` statement responsible for throwing the exception, and the `catch` block that caught the exception (if any). The instrumented program also reports uncaught exceptions because, during instrumentation, InsECT suitably wraps the program. In addition, the instrumented program reports, for each execution, various kinds of 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.

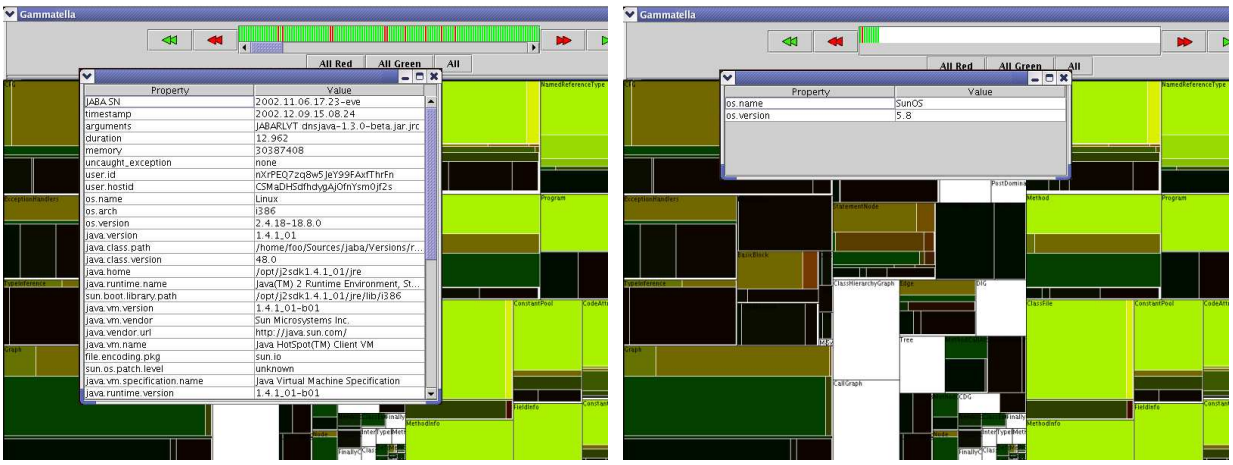
The information reported by the probes is collected during the execution by the monitor classes that are called by the probes inserted in the code. At the end of the execution, or at given time intervals (e.g., in the case of continuously running applications), the information is dumped, compressed, 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, the information can be stored locally and sent when a connection is available.

We use the Simple Mail Transfer Protocol (SMTP [19]) 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* is a simple tool written in Java that runs as a daemon process on a server on which we store the execution data. Each instance of the tool monitors for execution data from all instances of a specific version of a specific program, provided to the tool in the form of the corresponding

[2] `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`



(a) (b)

Figure 7: Windows that show execution properties.

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 [25]).

For each message retrieved, the daemon parses the subject of the message to check whether (1) the message contains coverage information (i.e., the subject contains the coverage label), and (2) the information is coming from the correct program and version (i.e., the ID provided to the daemon matches the one in the subject). If both conditions are satisfied, the daemon 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* is the module of GAMMATELLA that implements the visualization technique described in Section 2. The visualizer is divided into JavaBeans components written in Java using the graphical capabilities of the Swing toolkit. The visualizer retrieves and queries the coverage data stored by the Data Collection Daemon. These data are used to update all appropriate views. The Program Visualizer, shown in Figure 11, 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).

It is worthwhile to note that each window pane can be dynamically resized so that the user can choose the appropriate proportion of the total window real estate allocated to each component. We describe each component and widget in detail.

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 GAMMATELLA 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 immediately 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. This window 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 shows only the common properties of those executions (Figure 7(b)).

The Execution Bar contains also three buttons:

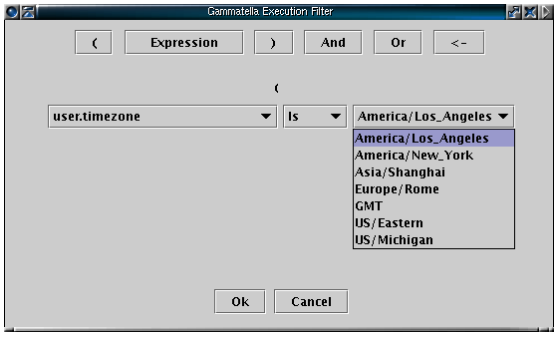


Figure 8: Dialog window for the definition and editing of filtering rules.

Execution filters, *Execution summaries*, and *Reset execution bar* (see Figure 11). Clicking on button *Execution filters* displays a dialog window in which the user can construct filtering rules to be applied to the executions set or modify existing filters. Figure 8 shows a screenshot of a window in which a filter is being defined. Analogously, clicking on button *Execution summarizers* displays a window in which the user can define new summarizers or modify existing ones. Button *Reset execution bar* provides a convenient way to remove all filters and summarizers at once and show all the executions.

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. This menu 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 Program Visualizer 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. Moving the mouse cursor over a node causes a tool tip to appear. This tool tip describes 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 [5] 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 [7] to present more visible nodes. Initially, the Treemap Viewer displays the system as described in Section 2, where each node represents an individual source file. Additionally, the user can collapse all files in a package into a node representing the color distribution for all of the files in that package. This can be done iteratively until eventually the entire treemap contains one node that represents the color distribution for the entire system. Likewise, the user can dissect the package-level treemap nodes to their respective components and eventually to their source files. Package-level treemaps can be collapsed and dissected both one at a time and all at once. The user can also zoom into any particular area of the treemap to provide a more detailed view of the corresponding nodes.

Additional Widgets. In addition to the three major components, the Program Visualizer contains some components for convenience and informational purposes: an interactive color legend, a statistics pane, a color slider, and color-space control menus.

The *interactive color legend* is located in the lower right part of the Program Visualizer (see Figure 11). The color legend is drawn as a two-dimensional plane with hue varying on the horizontal axis and brightness varying on the vertical axis. The legend contains a black dot at each position in the color space occupied by a source-code statement. By rubberbanding a rectangle around some points in this region, the viewer can modify (filter) the main display area, showing and coloring only statements having the selected color and brightness values.

The *statistics pane* is located above the interactive color legend. This pane shows information about the last statement for which the mouse cursor was moved over in the file-level or source-level views. For example, in Figure 11, the mouse was last placed over line 456 of file jaba/graph/icfg/ICFGImpl.java. The statistics pane shows that (1) this line was executed by 40 of the 707 executions, (2) of the 707 executions, 695 terminated normally and 12 terminated with an exception, (3) 28 of the normally-terminating executions executed this line, and (4) all of the 12 exceptionally-terminating executions executed this line.

The *color slider* is located in the upper left corner of Program Visualizer. This widget is a slider that controls the brightness of the gray color used to color lines, such as comments, unexecuted lines, and filtered lines, that are not being drawn using the red-yellow-green mapping. The slider can be used, for

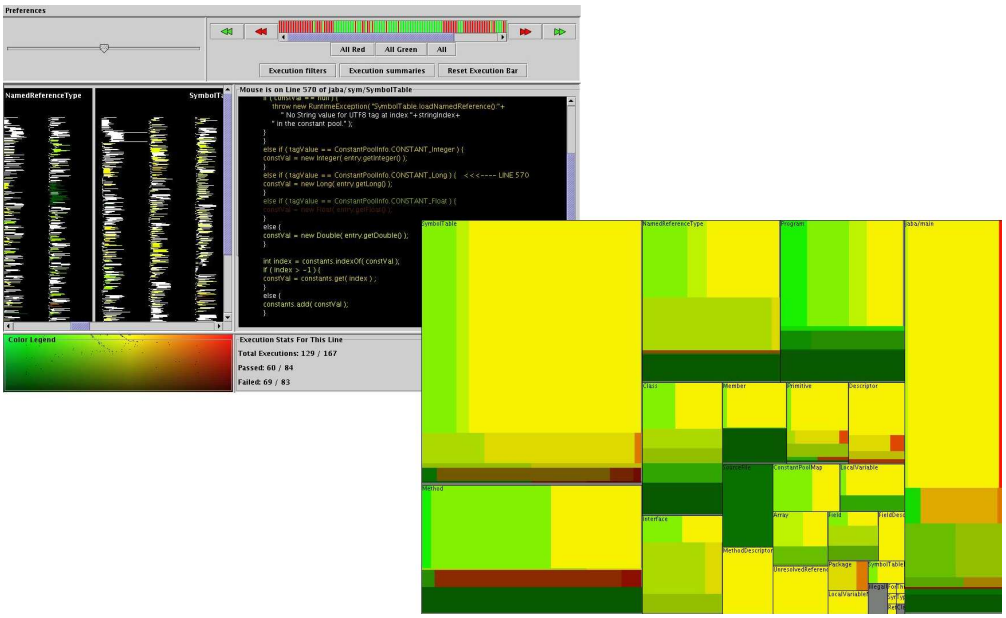


Figure 9: GAMMATELLA Program Visualizer in two-window modality.

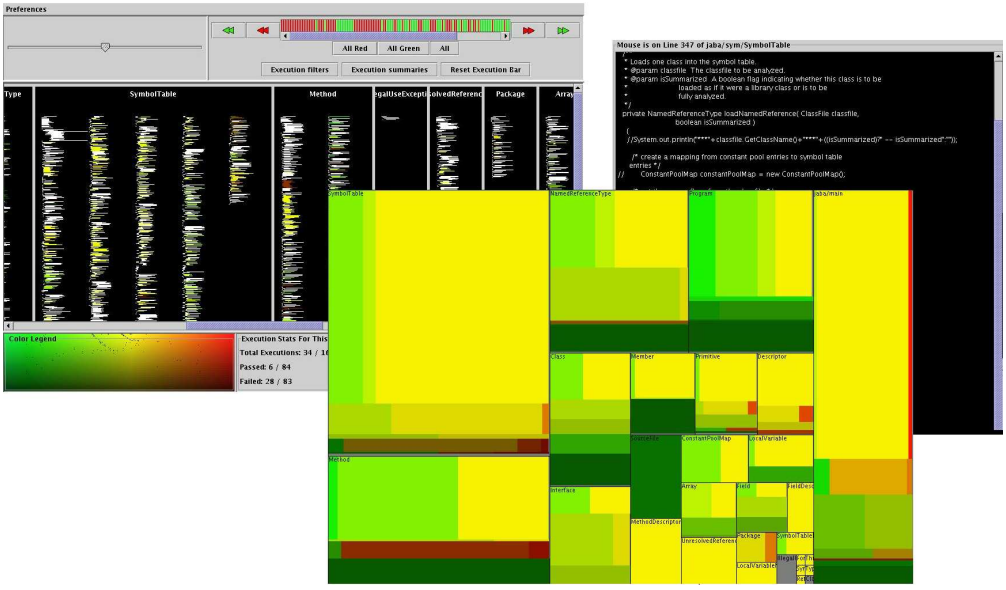


Figure 10: GAMMATELLA Program Visualizer in three-window modality.

instance, to make all gray statements almost disappear and let the user focus only on the colored ones, while inspecting the execution data. In Figure 11, the slider is positioned to color those statements using light gray. In cases in which users interact with the slider rarely, they can resize the slider's pane and make it disappear, so as to have more screen real estate available to the other components, as discussed in Section 3.3.

Finally, the *color-space control menus* are located under the menu named *Preferences*. These menus are controls to change the color-space used. In our current implementation, this menus let the user switch from the red-yellow-green spectrum to an alternative spectrum that is suitable for color-blind users.

Figure 11: A screenshot of the GAMMATELLA Program Visualizer.

Figure 12: Developers' semi-public display of GAMMATELLA.

The Program Visualizer can be displayed in three alternative configurations, or modalities. The first modality is the one shown in Figure 11, in which all three main components are aggregated in one window. The second modality is based on a dual-window representation, in which one window contains the Execution Bar and the Code Viewer, and a second window contains the Treemap Viewer. Finally, in the third modality, all three major components are displayed in separate windows. Figures 9 and 10 show snapshots of Gammatella in the second and third modality, respectively.

The different modalities are defined to accommodate different configurations of the visualization hardware. In particular, we defined the modalities to accommodate the presence of multiple monitors. For example, for our semi-public display of Gammatella we used the dual-window modality on a machine with two monitors, as discussed in the next section. In the two- and three-windowed modes, we chose to place the system-level view in its own window to have the most screen real estate for the view that represents the most code. In the three-windowed mode, we chose to place the code-level view in its own window because we did not want to use a small window-pane for the view that contains the most detailed information (i.e., the actual code). However, due to the modular nature of the system, the configuration of the components in the different modalities can be changed easily.

The visual components in the Program Visualizer 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 little effort.

The Program Visualizer dynamically updates the information displayed to reflect the latest data: As the Data Collection Daemon receives additional executions from the field, all visual components 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.

3.4 Semi-public Display of Gammatella

To investigate the usefulness of Gammatella from a team of developer's perspective, we have placed a semi-public display running the tool in our lab. Semi-public displays [10] are public displays for small groups of people to make certain information visible to the environment and to promote collaboration. The semi-public display of Gammatella is placed in the workplace of the software engineers that have developed and are maintaining the deployed, monitored system. The goal is to promote awareness of the system's behavior in the field and to promote interaction among the developers in regard to this information. We provide more details about the monitored program and its users in Section 4.3.

In our current setup, the semi-public display of Gammatella runs, in dual-window mode, on a machine installed in the common area of our lab, and continuously visualizes the exception-analysis information described in Section 4.1. The machine is connected to two 23-inch flat-panel monitors, each one displaying one of the two Gammatella windows. Users can interact with Gammatella through a traditional keyboard and mouse. Figure 12 shows a photograph of the semi-public display area. In the next section, we describe our initial experience with the semi-public display of Gammatella.

4 Applications

To investigate the feasibility 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 exception investigation that involved the collection of data from a real program deployed to a set of real users. The study also involved the installation of a semi-public display of Gammatella in our lab, where the information collected from the deployed program is continuously shown. In the rest of this section, we describe the two applications, present the feasibility study, and discuss our initial experience with the semi-public display of Gammatella.

4.1 Exceptions Analysis

We applied our technique to the visualization of exception-related information. To this end, we used the approach defined by Jones, Harrold, and Stasko for fault localization [11]. The idea is to assign a color to each statement in the program to represent the likelihood that the statement is responsible for the behavior that led to the exception being thrown. 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 (call this F), and a set of executions that do not result in an uncaught exception (call this 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, s is assigned a greener hue to represent some confidence in its correctness. Conversely, if f is larger than p for s, a redder 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 precisely, we use the larger of the two percentages f and p. Reference [11] 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 targeting parts of code for optimization, determining feature-usage, aiding in the reduction of software-bloat, and aiding the guidance of future enhancements.

For profiling analysis, 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. The goal of our feasibility study was to show that our framework could be applied to these tasks, as well as to show that we could do this for a real, deployed system with real users in the field. Jaba (Java Architecture for Bytecode Analysis [1]) is a framework for analyzing Java programs developed in Java within our research group that consists of 550 classes, 2,800 methods, and approximately 60,000 lines of code. 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. In our internal tests, the instrumentation caused an average overhead of 24% in terms of execution time. It is worth noting that this is the overhead for the fully instrumented code and with unoptimized instrumentation; better engineering would likely reduce this figure. We distributed the first release of the instrumented Jaba to nine users, who used it for two months. This first release helped us tune the approach in terms of instrumentation, data collection, and interaction with the user's platform [16].

Using the information we obtained from this first release, we created a second instrumented version of Jaba, and distributed it to 14 users. The studies reported in this paper are based on the data collected using the second release of our tool. Five of the 14 users had already used Jaba for their work (and were part of the first data collection experiment), whereas the other nine users had just started projects that involved the use of Jaba.

Seven of the 14 users involved in the studies are working in our lab: four are part of our group and use Jaba for their research; another two are students working in our department who use Jaba for their graduate-level projects; the last one is a Ph.D. student who is using a regression testing tool built on Jaba. The remaining seven users are four researchers and three students working in three different universities across three countries.

After releasing the instrumented version of Jaba, we started the Data Collection Daemon on a dedicated machine in our lab. While users used Jaba for their work and the program-execution data were sent

to the collection server, the Data Collection Daemon retrieved and stored the data, and the Program Visualizer visualized the corresponding information on a semi-public display, as described in Section 3.4.

In a period of 12 weeks, we collected about 1,500 executions. Using GAMMATELLA, we have been able to save the information about the executions automatically and visualize them. 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 JABA package responsible for performing dominance analysis was never utilized. The treemap view provided by GAMMATELLA let us immediately spot the large uncovered parts and identify the corresponding parts in the code. Such a situation occurred for both releases of JABA to external users and motivated our decision to build a trimmed-down version of JABA—one in which the unused parts are released as an additional, optional package.

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, thereby reducing the amount of spurious information.

Yet another 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 user 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 performed 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. Although this problem could have been discovered in-house, during testing, such a discovery would have required testing the software in that specific configuration. This is an ideal example of the kind of problems for which the Gamma approach was defined: in general, it is very difficult to test adequately, in-house, software that must function in many different environments and configurations. For this kind of software, feedback from the field can provide invaluable information.

As far as the semi-public display is concerned, we have begun our experimentation with it and have started to collect feedback. This feedback has let us assess how the tool is perceived and identified characteristics of the tool that could be improved.

As far as the general perception of the tool is concerned, the users found it interesting and informative. The developers of the system can see that the program is being used and get a first assessment of how it is being used. For example, they can keep track of users that are having problems with the program (those with executions that are terminating with an exception). In addition, usage measures of the various components of the tool can be assessed to provide feedback on which features of the program are most useful. In terms of areas for improvement, the main complaint from the users concerns the speed of the tool. GAMMATELLA is still a prototypal implementation, and no effort has yet been made to optimize it. Some users also provided interesting suggestions on ways to improve the interface of GAMMATELLA in general. Based on the initial feedback for the semi-public display, we are currently investigating ways to improve the efficiency of the tool. In the meanwhile, we will continue the semi-public display study and the collection of users' feedback.

5 Related Work

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

Eick and colleagues developed the SeeSoft system [8], 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 by applying the visualization to a new domain.

Shneiderman developed the treemap visualization [21] for visualizing hierarchical data in a space-filling manner. Bruls and colleagues developed an approach [7] 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 of 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 [2], 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, Podgurski, and White, in their work on observation-based testing, describe some uses of multivariate visualization [13] applied to execution profiles. They use multivariate visualization to project many-dimensional profiling information onto a specific visualization, a two-dimensional scatter plot. Their approach is related to ours because they too visualize information about multiple executions. However, their approach is targeted to a specific goal, namely, clustering of similar executions according to some criteria, whereas our goal is to provide a generic visualization framework that can be instantiated for different tasks.

Reiss and Renieris developed the Bloom system [20], 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 also be implemented leveraging the Bloom framework.

Storey and colleagues developed the SHriMP Views system [4, 24], 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 relationship to the program.

Jones, Harrold, and Stasko developed the Tarantula [11] 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 can be considered a specific instance of the approach described in this paper.

There is also related work in the area of collecting information from deployed software. Liblit, Aiken, and colleagues [14] developed a lightweight instrumentation infrastructure based on statistical sampling for gathering information from users' executions and used it to localize faults. Bowring, Orso, and Harrold introduce the concept of software tomography [6], which enables lightweight collection of runtime information from deployed software based on sparse sampling. Both those approaches are not concerned with visualization and are complementary to this work: we could leverage lightweight instrumentation approaches to reduce the overhead of our data-collection phase.

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 enable the representation of different kinds of data, and to allow 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 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 such discoveries are preliminary, they provide evidence of the usefulness of the approach. We also discussed our initial experience with a semi-public display of Gammatella, in our lab.

The feasibility study and the initial semi-public display experience 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 or by automatically excluding some kinds of exceptions) 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. We will select a num-

ber of these tasks, apply our visualization approach to them, and evaluate the results. 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, we will continue our semi-public display experience, which already provided some initial, useful feedback.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-0306372, CCR-0205422, CCR-9988294, CCR-0209322, and SBE-0123532 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 co-developed INSECT and helped with its integration in GAMMATELLA. Jim McPherson aided in the development and integration of the tool. John Stasko provided many useful suggestions and comments that helped improve the work and the paper. The anonymous reviewers provided comments that helped improve the paper's presentation.

References

[1] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`, 2003.

[2] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(2):119–133, 1995.

[3] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[4] Casey Best, Margaret-Anne D. Storey, and Jeff Michaud. SHriMP views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[5] Christophe Bouthier. TreeMap Java Library, 2002. `http://treemap.sourceforge.net/`.

[6] Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, pages 2–8, Nov 2002.

[7] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. Squarified treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, 2000.

[8] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft – a tool for visualizing line oriented software. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov 1992.

[9] Jim Gray, Donald Slutz, Alexander Szalay, Ani Thakar, Jan vandenBerg, Peter Kunszt, and Chris Stoughton. Data Mining the SDSS SkyServer Database. Technical Report MSR-TR-2002-01, Microsoft Research, January 2002.

[10] Elaine M. Huang and Elizabeth D. Mynatt. Semi-public displays for small, co-located groups. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 49–56, 2003.

[11] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'01)*, pages 467–477, May 2001.

[12] John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 401–408, 1995.

[13] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22th International Conference on Software Engineering (ICSE'00)*, pages 116–125, June 2000.

[14] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[15] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003.

[16] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *Proc. of the ACM Symposium on Software Visualization*, pages 67–76, Jun 2003.

[17] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 65–69, Jul 2002.

[18] Christina Pavlopoulou and Michael Young. Residual test coverage monitoring. In *Proceedings of the International Conference on Software Engineering*, pages 277–284, 1999.

[19] Jonathan B. Postel. RFC821: Simple Mail Transfer Protocol, 1982. `http://www.ietf.org/rfc/rfc0821.txt`.

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

[21] Ben Shneiderman. Tree visualization with treemaps: A 2-D space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

[22] John Stasko, John Domingue, Marc Brown, and Blaine Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[23] John Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 57–65, 2000.

[24] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)*.

[25] University of Washington. The IMAP Connection, 2002. `http://www.imap.org/`.