An OpenGL-based Eclipse Plug-in for Visual Debugging

André Riboira Rui Abreu Rui Rodrigues

Department of Informatics Engineering
University of Porto
Portugal
{andre.riboira, rma, rui.rodrigues}@fe.up.pt

ABSTRACT

Locating components which are responsible for observed failures is the most expensive, error-prone phase in the software development life cycle. Automated diagnosis of software faults (aka bugs) can improve the efficiency of the debugging process, and is therefore an important process for the development of dependable software. Although the output of current automatic fault localization techniques is deemed to be useful, the debugging potential has been limited by the lack of a visualization tool that provides intuitive feedback about the defect distribution over the code base, and easy access to the faulty locations. To help unleash that potential, we present an OpenGL-based Eclipse plug-in that explores two visualization techniques - viz. treemap and sunburst - aimed at aiding the developer to acquire a broad sense of the error distribution, and find faults quickly.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: testing and debugging; H.5.2 [**User Interfaces**]: Graphical user interfaces (GUI)

Keywords

diagnosis, visual techniques, eclipse plug-in.

1. INTRODUCTION

When unexpected behavior is observed, developers need to identify the root cause that makes the system deviate from its intended behavior. This task (also known as software debugging, fault localization, or fault diagnosis[1]) is the most time-intensive and expensive phase of the software development cycle [8], and is being performed since the beginning of computer history. As an indication of the downtime, debugging, and repair costs involved, a 2002 landmark study indicated that software bugs pose an annual $60 billion cost to the US economy alone [14].

[1]In this paper, the terms software debugging, fault localization and fault diagnosis are used interchangeably.

TOPI'11 May 28, 2011 - Waikiki, Honolulu, Hawai'i, USA

A traditional approach to fault localization is to insert *print* statements in the program to generate additional debugging information to help identifying the root cause of the observed failure. Essentially, the developer adds these statements to the program to get a glimpse of the runtime state, variable values, or to verify that the execution has reached a particular part of the program. Another common technique is the use of a symbolic debugger which supports additional features such as breakpoints, single stepping, and state modifying. Examples of symbolic debuggers are GDB [15], DBX [6], DDD [18], Exdams [4], and the debugger proposed by Agrawal, Demillo, and Spafford [3]. Symbolic debuggers are included in many integrated development environments (IDE) such as Eclipse[2], Microsoft Visual Studio[3], Xcode[4], and Delphi[5].

These traditional, manual fault localization approaches have a number of important limitations. The placement of print statements as well as the inspection of their output are unstructured and ad-hoc, and are typically based on the developer's intuition. In addition, developers tend to use only test cases that reveal the failure (i.e., failing test cases), and therefore do not use valuable information from passing test cases. Furthermore, the size of the program state at each point can be large, and there are many combinations of program executions that have to be examined. Hence, such techniques still require a detailed knowledge of the program, and also suffer from a substantial execution overhead in terms of execution time and space to store historical run-time data. Last, but not least, manual debugging is extremely expensive in terms of labor cost.

Aimed at drastic cost reduction, much research has been performed in developing automatic fault localization techniques/tools. Regardless, apart from [5, 11], little work has focused on developing a visual representation of the diagnostic report. In this paper, we present GZoltar, a visual debugger for Java programs. GZoltar is implemented as a plug-in for the IDE Eclipse using OpenGL technology for rendering the visualizations. In particular, two visualizations are implemented: (1) treemap and (2) sunburst [16].

The remainder of this paper is organized as follows. We start by introducing relevant concepts in the field of automatic debugging. Then, the GZoltar Eclipse plug-in is described. Finally, in section 4, we conclude and give directions for future work.

[2]`http://www.eclipse.org/`
[3]`http://msdn.microsoft.com/en-us/vstudio/`
[4]`http://developer.apple.com/tools/xcode/`
[5]`http://www.codegear.com/`

2. AUTOMATIC DEBUGGING

The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process, or as an input to automatic recovery mechanisms [12]. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches to fault localization can be classified as (1) statistical approaches or (2) reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report.

A statistical approach to spectrum-based fault localization (SFL) will now be described. A program under analysis comprises a set of M components (e.g., functions, statements) c_j where $j \in \{1, \ldots, M\}$, and can have multiple faults, the number being denoted C (fault cardinality). A *diagnostic report* $D =< \ldots, d_k, \ldots >$ is an ordered set of diagnosis candidates d_k ordered in terms of likelihood to be the true diagnosis. Statistical approaches yield a single-fault diagnostic report with the M components ordered in terms of statistical similarity (e.g., $< \{3\}, \{1\}, \ldots >$, in terms of the indices j of the components c_j).

Program (component) activity is recorded in terms of program spectra [9]. This data is collected at run-time, and typically consists of a number of counters or flags for the different components of a program. In this paper we use the so-called hit spectra, which indicate whether a component was involved in a (test) run or not. Both spectra and program pass/fail (test) information is input to SFL. The program spectra are expressed in terms of the $N \times M$ *activity matrix* A. An element a_{ij} is equal to 1 if component j was observed to be involved in the execution of run i, and 0 otherwise. For $j \leq M$, the row A_{i*} indicates whether a component was executed in run i, whereas the column A_{*j} indicates in which runs component j was involved. The pass/fail information is stored in a vector e, the *error vector*, where e_i represents whether run i has *passed* ($e_i = 0$) or *failed* ($e_i = 1$). Note that the pair (A, e) is the only input to SFL.

In SFL one measures the statistical similarity between the error vector e and the activity profile column A_{*j} for each component c_j. This similarity is quantified by a *similarity coefficient*, expressed in terms of four counters $n_{pq}(j)$ that count the number of elements in which A_{*j} and e contain respective values p and q, i.e, for $p, q \in \{0, 1\}$, we define

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}|$$

An example of a well-known similarity coefficient is the Ochiai coefficient, which is among the best for fault localization [1, 2]

$$s(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{10}(j)) \cdot (n_{11}(j) + n_{01}(j))}} \qquad (1)$$

Due to space limitation we do not illustrate how Ochiai works; for detailed information, refer to [1]. Ochiai is also implemented in the Zoltar toolset [10].

3. GZOLTAR

Current visual (automatic) debugging tools are either standalone applications [10, 11] or lack a meaningful data visualization [5]. GZoltar is an effort to overcome these shortcomings, by adding powerful visualization features to SFL's core processing. GZoltar[6] aims at integrating debugging processing: executes automatically unit tests to generate the activity matrix; obtains diagnostic report; processes data dependency between lines of code; and provides a powerful interactive debugging data visualization and navigation interface. All these features are integrated in a plug-in for Eclipse, one of the most popular IDE's [7].

GZoltar's execution is split into two main phases:

- Data Processing
- Visualization and Interaction

The first phase, data processing, is only executed at startup, and later upon user request (e.g., after modifying the source code of the project under analysis). In this phase, GZoltar detects Eclipse's open projects and their contents (e.g, classes, methods, etc.). The activity matrix is generated based on the execution of a series of unit tests (these have to be provided in the project following a naming convention as described in [13]). Code coverage information is gathered during execution and the failure probability for each module is then computed using SFL.

The second phase, visualization and interaction, is executed in loop until exit or user request. The pre-processed tree has the needed data to render powerful OpenGL visualizations. User can interact with GZoltar by navigating through the visualization, by expanding or collapsing visualization tree nodes, zooming, panning, and changing the visualization tree root. Because there is no visualization that offers the best result in all circumstances, GZoltar offers two popular tree structured data visualizations, treemap and sunburst [16] (see Figure 1). Many other visualizations can easily be added in the future. The user can swap between visualizations, maintaining the same debugging data tree as well as navigation data.

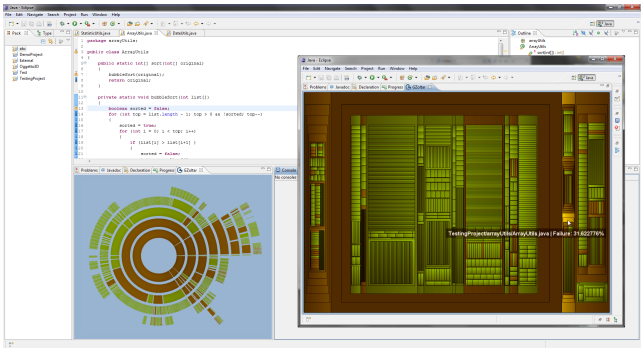

Figure 1: GZoltar view in the Eclipse IDE. In the background, sunburst in a common view. In the foreground, treemap in a maximized view.

GZoltar integrates seamlessly into Eclipse. The user can open the standard Eclipse code editor directly from the visualization, in the line of code that is being analyzed. More-

[6]GZoltar official website `http://www.gzoltar.org/`

over, standard Eclipse warnings with each line's failure probability are also generated. Like other Eclipse warning, these are also listed in the "Problems" area, and in the tooltips of the code editor near the line of code as well as in the editor's scrollbar. This aims at reducing the learning curve, through the use of standard Eclipse features (see Figure 2).

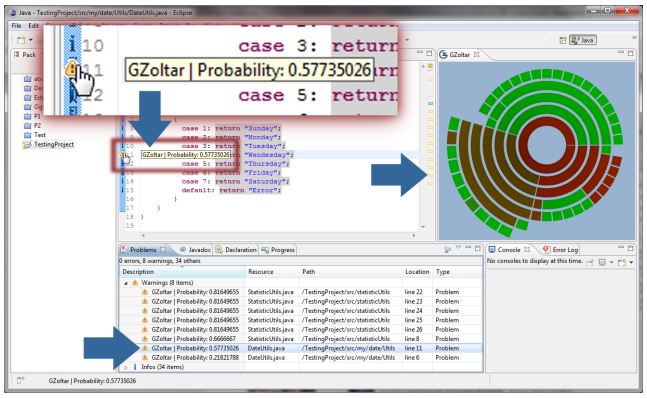

Figure 2: GZoltar produces standard Eclipse warnings revealing each component failure probability.

OpenGL integration and code coverage operations require additional plug-ins. Since Eclipse does not support OpenGL natively, GZoltar uses Java OpenGL library (JOGL) [17], to create a bridge between native OpenGL system libraries and AWT[7], the toolkit that grants window abstraction. Eclipse uses SWT[8] to process its Workbench, but it also has a bridge between SWT and AWT. Because of this, it is possible to use OpenGL technology inside a standard Eclipse view. Eclipse also offers connections to its Workspace, allowing GZoltar to detect open projects and create standard warnings. Code coverage analysis of the test executions is done with JaCoCo[9], a Java Code Coverage API from EclEmma[10] development team. Relations between different technologies used by GZoltar are depicted in Figure 3.

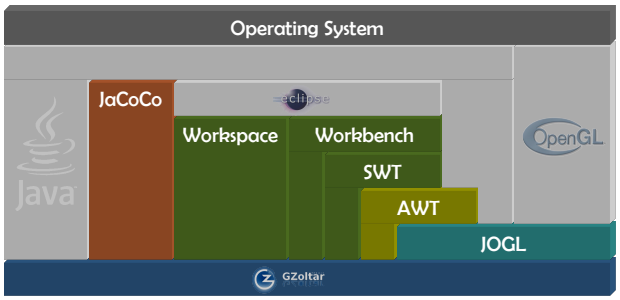

Figure 3: GZoltar's technological layers.

GZoltar's architecture is very modular. It is simple to improve GZoltar by modifying only a certain module, such as replacing the diagnostic algorithm, or creating new visualizations. Each module works as a service provider, and can be upgraded in the future without much effort. Installing GZoltar is straightforward: like any traditional Eclipse plug-in, it can be installed directly by Eclipse's "Install new software" feature. GZoltar is multi-platform, compatible with 32 and 64 bit CPU architectures and Microsoft Windows, Apple Mac OS X, and Linux operating systems. After installation, GZoltar view is available in Eclipse and can be used just as any other Eclipse view. The default visualization is the sunburst, which focuses more on the hierarchical data structure, and the second visualization is the treemap, which focuses more on the tree leafs (see Figure 4). Each one of these visualization concepts has advantages and disadvantages, and their efficiency varies much with the data tree arrangement [16]. As such, it is important to have multiple visualizations to help the user to achieve his/her goals.

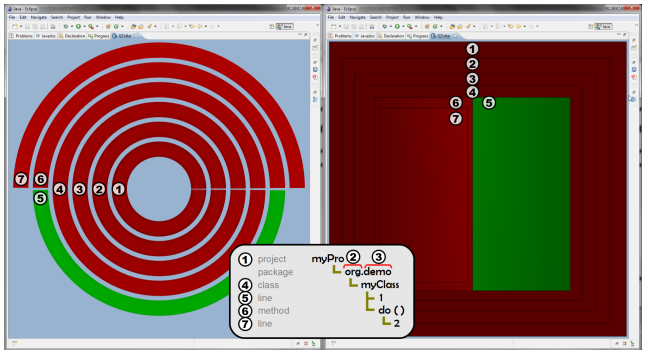

Figure 4: Sunburst and treemap visualizations, side-by-side, using the same debugging data as input.

Users can interact with GZoltar in multiple ways. Initially only the first level of the data tree is displayed, and users can click on any component to expand it. If (s)he clicks on an opened component, it will be collapsed. This way, (s)he can analyze only the desired part of the system. If the user wants to have a glimpse of the entire system, (s)he can expand all components by pressing the "space" key. The user can also zoom and pan into a specific area of any visualization to obtain more detail (see Figure 5). Because system visualizations can get overly complex, GZoltar also offers the possibility of analyzing a sub-tree, by choosing any inner node to be the new root of the tree (ignoring siblings). A new visualization is rendered with that chosen sub-tree. (S)he can raise the visible tree nodes by selecting any upper level. This is an important feature to analyze complex systems, because the user can focus only on the desired area, and visualize it as an independent system. The user can also get information for each component by placing the mouse cursor over its graphical representation, causing a label with that component name, location and failure probability to be rendered.

Each component color represents its failure probability, as given by the diagnostic algorithm. Component colors vary from red (maximum failure probability) to green (zero failure probability). When using sunburst visualization, the user can have extra information, by hovering the mouse cursor over a leaf node component. In that case, the component colors are changed to reveal relations between lines of code. Leaf node component colors will vary from the color of the selected component to gray, considering the depth of relation between that component and the selected one (see Figure 6).

[7] http://java.sun.com/products/jdk/awt/
[8] http://www.eclipse.org/swt/
[9] http://www.eclemma.org/jacoco/
[10] http://www.eclemma.org/

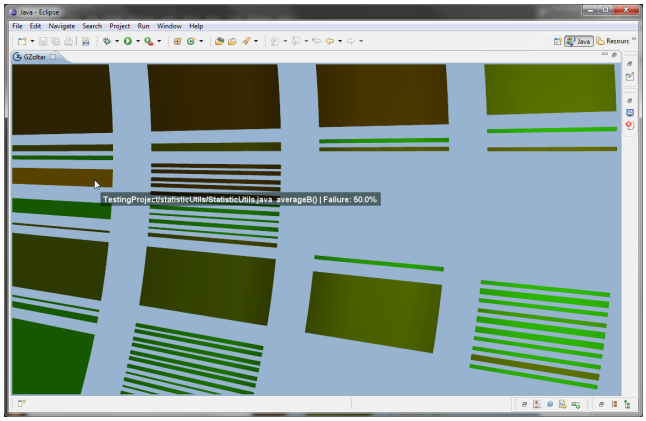

Figure 5: Sunburst visualization zoomed into a specific system area.

If two components have exactly the same color, it means that they were executed in the same unit tests execution (same execution pattern). On the other hand, if a component has a gray color, it means that there is no (execution) relation between that component and the selected one.

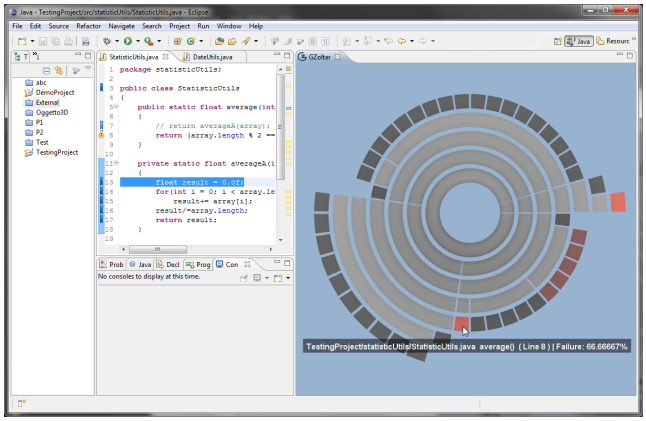

Figure 6: Relations between leaf node components on sunburst visualization.

For a detailed overview on GZoltar operation, installation and user manual, please refer to [13]. A video showing how to install and use GZoltar can be found at `http://www.gzoltar.org/`.

4. CONCLUSIONS & FUTURE WORK

Software faults are responsible for a significant part of the costs of software development. Thus, reducing them is of utmost importance, leading to a high interest in the search for effective tools for (automatic) fault localization. However, the tools currently available are limited in the localization information they provide, in the ways that they present that information, or relate to the main development tools and environments. Any of these factors hampers the effectiveness of such debugging tools. The tool presented in this paper - dubbed GZoltar - aims to overcome these limitations. It uses a state-of-the-art SFL algorithm [1] to achieve high-quality fault localization. It provides powerful interactive graphical visualizations of the SFL data - sunbursts and treemaps - to allow fast comprehension of the fault distribution and interactive data analysis. It was developed as a plugin for one of the most used IDE's [7], thus allowing seamless integration and interaction between fault visualization, code, and IDE's reporting facilities.

The tool has been built in a modular and extendable way, thus facilitating future improvements. A thorough usability study is being conducted, to evaluate the tool and provide ideas for such improvements. Some ideas that are already being considered include: the integration of JUnit with GZoltar (to handle unit tests); the exploration of new tree-structured data visualization concepts; extensions to the navigation tools, such as the creation of a mini-map to be displayed on zoom-in, or a spectrum color bar to be used as reference to aid users in the component color evaluation process; and integration with new interaction paradigms, like multi-touch input devices, to facilitate user interaction.

5. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems & Software (JSS)*, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *Proc. of the International Conference on Automated Software Engineering (ASE'09)*. IEEE Computer Society, 2009.

[3] H. Agrawal, R. de Millo, and E. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, 1991.

[4] R. M. Balzer. EXDAMS: Extendible debugging and monitoring system. In *Proc. of the AFIPS Spring Joint Conference*. AFIPS Press, 1969.

[5] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. Ezunit: A framework for associating failed unit tests with potential programming errors. In *Proc. of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP'07)*. Springer, 2007.

[6] DBX. Debugging tools – DBX, SunOS 4.1.1 ed., 1990. SUN MICROSYSTEMS, INC.

[7] D. Geer. Eclipse becomes the dominant Java IDE. *Computer*, 2005.

[8] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 2002.

[9] M. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 1998.

[10] T. Janssen, R. Abreu, and A. J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proc. of the Int'l Conference on Automated Software Engineering (ASE'09) - Tools Track*. IEEE Computer Society, 2009.

[11] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In M. Young and J. Magee, editors, *Proc. of the International Conference on Software Engineering (ICSE'02)*. ACM Press, 2002.

[12] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, University of California at Berkeley, 2002.

[13] A. Riboira. GZoltar: A graphical debugger interface. Master's thesis, University of Porto, 2011.

[14] RTI. Planning report 02-3: The economic impacts of inadequate infrastructure for software testing. Planning report, National Institute of Standards and Technology, 2002.

[15] R. Stallman. Debugging with GDB – The GNU source level debugger, 1994. Free Software Foundation.

[16] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 2000.

[17] Z. Xu, Y. Yan, and J. Chen. Opengl programming in java. *Computing in Science Engineering*, 2005.

[18] A. Zeller and D. Lütkehaus. DDD – A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 1996.