GZoltar: An Eclipse Plug-In for Testing and Debugging

José Campos André Riboira Alexandre Perez Rui Abreu

Department of Informatics Engineering
Faculty of Engineering, University of Porto
Portugal
{jose.carlos.campos, andre.riboira, alexandre.perez}@fe.up.pt; rui@computer.org

ABSTRACT

Testing and debugging is the most expensive, error-prone phase in the software development life cycle. Automated testing and diagnosis of software faults can drastically improve the efficiency of this phase, this way improving the overall quality of the software. In this paper we present a toolset for automatic testing and fault localization, dubbed GZoltar, which hosts techniques for (regression) test suite minimization and automatic fault diagnosis (namely, spectrum-based fault localization). The toolset provides the infrastructure to automatically instrument the source code of software programs to produce runtime data. Subsequently the data was analyzed to both minimize the test suite and return a ranked list of diagnosis candidates. The toolset is a plug-and-play plug-in for the Eclipse IDE to ease world-wide adoption.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Eclipse plug-in, Automatic Testing, Automatic Debugging, GZoltar, RZoltar

1. TESTING & DEBUGGING

Testing and Debugging is an important, yet the most expensive and tedious phase of the software development lifecycle. Although there are already off-the-shelf frameworks to ease these tasks, they still do not offer enough capabilities to fully automate this phase. Well known (unit) testing frameworks include JUnit, TestNG, and JTest, which automate the test execution but do not offer capabilities for, e.g., test suite minimization based on some criteria (such as coverage). Several debugging tools exist which are based on stepping through the execution of the program (e.g., GDB and DDD). 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, and therefore do not use valuable information from (the typically available) passing test cases.

Aimed at drastic cost reduction, much research has been performed in developing automatic testing and fault localization techniques and tools. As far as testing is concerned, several techniques have been proposed to minimize and prioritize test cases in order to reduce execution time and failure detection, while maintaining similar code coverage [15]. This paper presents a toolset, coined GZoltar, that provides a technique for test suite reduction and prioritization. The technique minimizes the original test suite using a novel constraint-based approach [4], while still guaranteeing the same code coverage. Furthermore, the technique allows the user to prioritize the minimized test suites by cardinality and execution time of the computed test suites.

As for debugging, one of the predominant techniques are those based on a black box statistics-based method [1] which takes a program and available test cases and returns the most probable location (component) that explains the observed failed test cases.

The GZoltar toolset implements a technique called spectrum-based fault localization (SFL [1]; in particular, the tool provides the Ochiai [1] incarnation of SFL, which is amongst the best for fault localization). SFL is based on instrumenting a program to keep track of executed parts. The instrumentation data is then analyzed to yield a list of source code locations ordered by the likelihood of it containing the fault. Furthermore, the toolset enables a program to be trained with expected behavior and to automatically detect an error if unexpected behavior is observed. The fact that no knowledge is needed of the program to acquire possible fault locations makes this set of tools a useful extension to currently applied methods of testing and debugging.

The GZoltar toolset, being developed at the University of Porto, aims at providing state-of-the-art techniques for (regression) test suite minimization and fault localization.

To obtain more information about the toolset, visit our Web site:

`http://www.gzoltar.com`

2. GZOLTAR TOOLSET

GZoltar is an Eclipse plug-in which produces accurate fault localization information using state-of-the-art spectrum-based fault localization algorithms, and provide the last studies in the field of regression testing [4]. It also creates intuitive and interactive diagnostic reports' visualizations, such as Treemap and Sunburst (see Fig. 3).

The integration with Eclipse (one of the most popular IDEs) is extremely useful. GZoltar uses Eclipse's standard features, such as detection of open projects in the workspace and their classes, to build the structure of System Under Test (SUT) to be used by the visualization view. Besides, GZoltar integrates well code editor as well as the standard Eclipse warnings generation with the provided visual diagnostic reports to facilitate the debugging process.

GZoltar aids developers finding faults faster, thus spending less time and resources in testing and debugging. This in turn leads to a higher software reliability level and/or to a decrease of its test period, thus reducing costs significantly.

2.1 GZoltar Architecture

The GZoltar is mainly written in Java and also uses third-party open source programs. The Eclipse's Workspace component is used to gather information it needs, such as open projects, their classes, and JUnit tests. ASM [3], a Java bytecode engineering library, is used to instrument the SUT in order to obtain coverage traces when executing the unit tests with JUnit. The Eclipse's Workbench component is used for generating the Eclipse User Interface (UI) tasks. This component has Standard Widget Toolkit (SWT) to create the GZoltar view, and provides a bridge to Abstract Windows Toolkit (required by the Java OpenGL (JOGL), the component that provides OpenGL bindings to Java). JOGL generates the OpenGL-based visualizations displayed on the GZoltar view.

Integrated in the GZoltar plug-in, but written in C is the MINION constraint solver[1]. Finally, the TRIE [7] structure has a interface written in Java and implemented in C for efficiency. For a schematic view of these technological layers and their interactions, see Fig. 1.

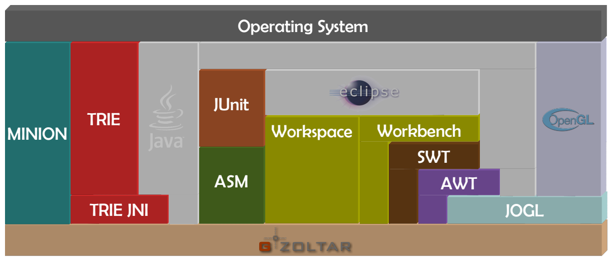


Figure 1: GZoltar Layers. Integration between GZoltar and other technologies.

2.2 GZoltar Flow

The GZoltar processing flow can be divided into eight main stages (see Fig. 2).

Initial Eclipse Integration: Eclipse makes it possible to automatically detect all open projects in the IDE. Once the

[1]MINION Homepage, `http://minion.sourceforge.net/`, 2012.

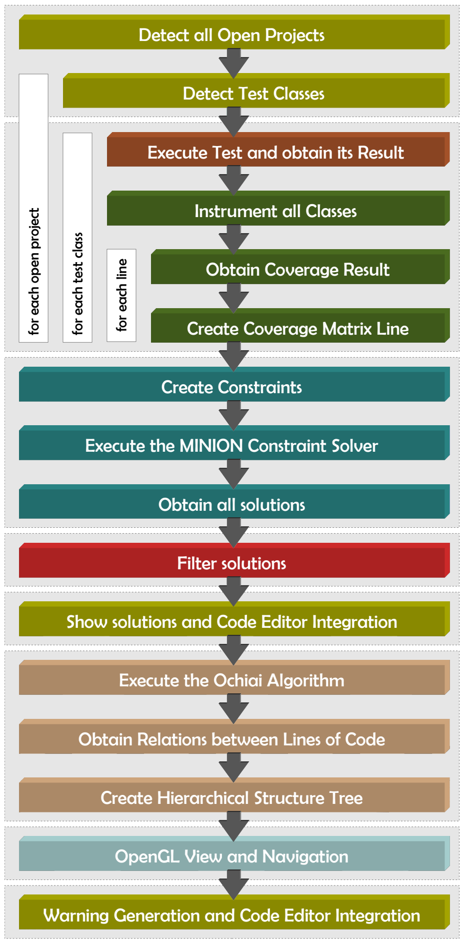

Detect all Open Projects
Detect Test Classes
for each open project
for each test class
for each line
Execute Test and obtain its Result
Instrument all Classes
Obtain Coverage Result
Create Coverage Matrix Line
Create Constraints
Execute the MINION Constraint Solver
Obtain all solutions
Filter solutions
Show solutions and Code Editor Integration
Execute the Ochiai Algorithm
Obtain Relations between Lines of Code
Create Hierarchical Structure Tree
OpenGL View and Navigation
Warning Generation and Code Editor Integration

Figure 2: Information flow.

set of the open projects is known, GZoltar search indexes all their classes and JUnit test classes (those who have test methods written in JUnit syntax, to be executed later). To avoid differences between source files and compiled classes, at this stage, GZoltar forces Eclipse to build the open projects, to guarantee that it is working with the latest version of the code.

JUnit and ASM: For each project there is a list of all test classes. For each class in the project all code is instrumented to allow the code coverage process. That process aims at detecting if a given line of code was executed or not. GZoltar uses ASM to instrument all open projects, thus being capable of debugging projects that call methods from other projects. Subsequently, test classes, implemented in the JUnit syntax, are executed automatically.

The information whether a test case has passed or failed is also stored to be used later by both the RZoltar (namely, to display if test fails) and GZoltar (namely, to compute the diagnostic report) views. The results are saved into a coverage matrix [4], a $N \times M$ binary matrix A, where N is the execution of a test case, M corresponds to different components of a software program, and a_{ij} is the coverage for component j when test i is executed. Once the code coverage matrix is gathered, RZoltar analyzes them to minimize the suite.

Gathering code coverage information of the SUT for each test case consists of three steps: (1) the code is instrumented to register what statements where touched by an execution, (2) the test case is executed, and (3) a coverage trace of what statements were executed is computed, and appended to the coverage matrix.

In summary, at this stage, all code from open projects are instrumented and built, test classes are executed, and code coverage matrix (plus any other relevant information) is stored.

MINION: After collecting the coverage of the SUT, into the coverage matrix explained before, the coverage information is passed to the constraint solver and it returns at least one minimum set that cover the entire software program such as original set.

Filtering out Solutions: The results provided by the constraint solver are then filtered out using a TRIE data structure. Essentially, this stage is to discard non-minimal test suites from the collection presented to the user.

Show Solutions: At this stage the user can select a reduced subset to re-execute, or sort all subsets by: their cardinality or execution time of each subset.

Run Ochiai: At this stage, GZOLTAR executes the SFL algorithm Ochiai, known to be amongst the best performing techniques for fault localization [1]. Based on all JUnit test results, the Ochiai similarity coefficient is calculated for every element of the system.

Graphical Visualization: To perform powerful and efficient visualizations of SUT, GZOLTAR uses OpenGL technology in order to take advantage of the Graphics Processing Unit (GPU).

Warning Generation and Final Eclipse Integration: The GZOLTAR plug-in also generates warnings that are integrate with the code editor, marking the lines that have a high probability of being faulty.

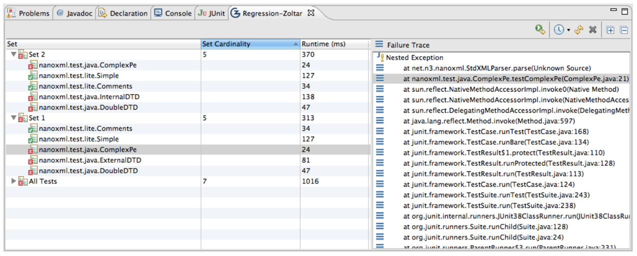
After all these stages, the user can inspect the failed unit tests and faulty lines (if they exist). This is a recursive process, so until all faults are not fixed, user can select a minimum set to re-execute (testing and saving time at the same time), fix and re-execute again.

2.3 Eclipse Views

By default, GZOLTAR offers two Eclipse views integrated into the IDE: GZOLTAR (Fig. 3) and RZOLTAR (Fig. 4).

While analyzing the SUT, the user can click on the visual representation of a line of code on the GZOLTAR view, and jump directly to that line in the Eclipse's code editor. An Eclipse code editor is opened with the text cursor placed on the line selected in the GZOLTAR view. Furthermore, GZOLTAR also generates a list of markers on the code editor's vertical ruler, which indicates the fault probability of the respective line when hovering the mouse over the marker. These markers can be of three different types: (1) red for the top third statements most likely to contain a fault, (2) yellow for the middle third statements, and (3) green for the bottom third statements. Every marker also has an embedded ColorADD[2] symbol, in order to help colorblind people distinguish between markers. These annotation markers are also displayed on the Eclipse "Problems" view.

GZOLTAR view also provides two visualizations [14] Treemap and Sunburst, as shown in Fig. 3. With this seamless integration, the user can easily analyze the SUT structure and localize the root cause of observed failures. Thus, GZOLTAR provides an easy way to access directly to the source code in order to fix faults.



Set	Set Cardinality	Runtime (ms)
Set 2	5	370
nanoxml.test.java.ComplexPe		24
nanoxml.test.lite.Simple		127
nanoxml.test.lite.Comments		34
nanoxml.test.java.InternalDTD		138
nanoxml.test.java.DoubleDTD		47
Set 1	5	313
nanoxml.test.lite.Comments		34
nanoxml.test.lite.Simple		127
nanoxml.test.java.ComplexPe		24
nanoxml.test.java.ExternalDTD		81
nanoxml.test.java.DoubleDTD		47
All Tests	7	1016

Failure Trace

```
Nested Exception
    at net.n3.nanoxml.StdXMLParser.parse(Unknown Source)
    at nanoxml.test.java.ComplexPe.testComplexPe(ComplexPe.java:21)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMeth
    at java.lang.reflect.Method.invoke(Method.java:597)
    at junit.framework.TestCase.runTest(TestCase.java:168)
    at junit.framework.TestCase.runBare(TestCase.java:134)
    at junit.framework.TestResult$1.protect(TestResult.java:110)
    at junit.framework.TestResult.runProtected(TestResult.java:128)
    at junit.framework.TestResult.run(TestResult.java:113)
    at junit.framework.TestCase.run(TestCase.java:124)
    at junit.framework.TestSuite.runTest(TestSuite.java:243)
    at junit.framework.TestSuite.run(TestSuite.java:238)
    at org.junit.internal.runners.JUnit38ClassRunner.run(JUnit38ClassRu
    at org.junit.runners.Suite.runChild(Suite.java:128)
    at org.junit.runners.Suite.runChild(Suite.java:24)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:231)
```

Figure 4: RZoltar interface [4].

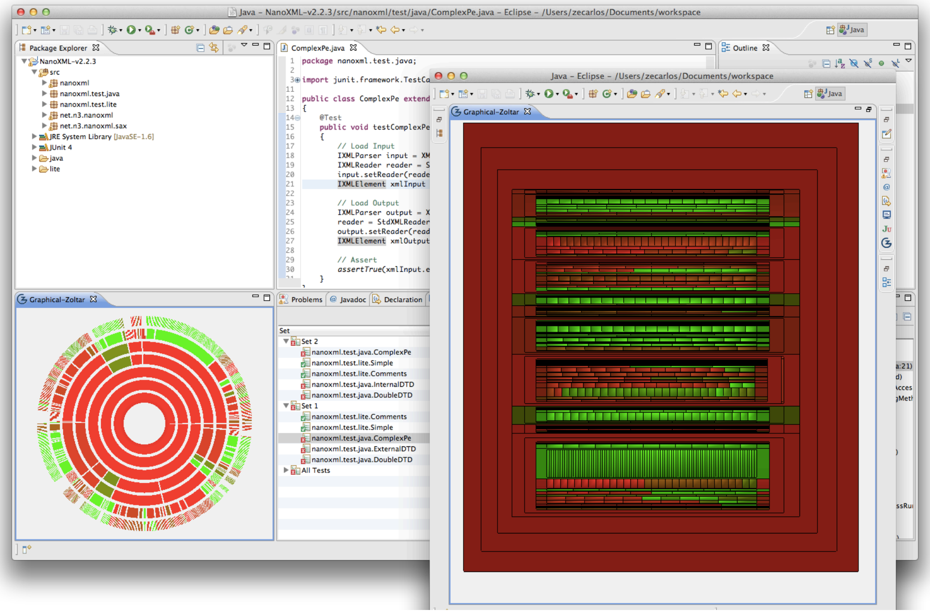
The RZOLTAR view (Fig. 4) is divided in two layers. On the left layer, the user can access the list of minimum set coverage (including the set with all test cases, just in case the user decides to re-execute the original test suite) and check the result of test cases (pass, fail, or error) by the color of icon. If a test fails or returns an error, the error message is displayed in "Trace" window. In all layers, the user can always double-click on a test case and jump to the test case file, or at failure trace, double-click goes to line (presents in that layer). This is similar to the functionalities offered by JUnit. The RZOLTAR view offers two criteria to prioritize test suites: *Cardinality of Set* and *Runtime*. The latter prioritization orders the minimum sets found (test suites of reduced size) using the take it takes to re-execute the suite, whereas the former orders the sets by the number of tests cases (set cardinality).

All in all, GZOLTAR provides an excellent ecosystem for regression testing and automatic debugging. This toolset is also straightforward to understand because we use familiar interface features (e.g., icons similar to the ones used in JUnit).

3. RELATED WORK

Nowadays most of the IDEs in market only offer a limited and manual debugging tool, such as breakpoints, conditional breakpoints, or the possibility to execute the software in a step-by-step. To the best of our knowledge, the most well known automatic debugging tools is Tarantula [12]. This independent tool is based on the the code coverage of multiple test executions of a given system. Although, Tarantula has not integration with any IDE, and do not support unit tests. Zoltar [11] is another available automatic debugging tool. It uses similarity coefficients to predict the failure probability of each line of code. Currently Zoltar runs only on Linux systems and works only with projects written in C language. Other relevant tools for automatic debugging are: Vida [8] an Eclipse plug-in based in Tarantula approach; EzUnit4 [2]

[2]ColorADD color identification system, `http://coloradd.net/`, 2012.


Java – NanoXML-v2.2.3/src/nanoxml/test/java/ComplexPe.java – Eclipse – /Users/zecarlos/Documents/workspace
Java – Eclipse – /Users/zecarlos/Documents/workspace
Package Explorer
ComplexPe.java
Outline
Graphical-Zoltar
Problems
Javadoc
Declaration
Java

```
NanoXML-v2.2.3
  src
    nanoxml
    nanoxml.test.java
    nanoxml.test.lite
    net.n3.nanoxml
    net.n3.nanoxml.sax
  JRE System Library [JavaSE-1.6]
  JUnit 4
  java
  lite
```

```
package nanoxml.test.java;

import junit.framework.TestCa

public class ComplexPe extend
{
    @Test
    public void testComplexPe
    {
        // Load Input
        IXMLParser input = XM
        IXMLReader reader = S
        input.setReader(reade
        IXMLElement xmlInput

        // Load Output
        IXMLParser output = X
        reader = StdXMLReader
        output.setReader(read
        IXMLElement xmlOutput

        // Assert
        assertTrue(xmlInput.e
    }
```

```
Set
  Set 2
    nanoxml.test.java.ComplexPe
    nanoxml.test.lite.Simple
    nanoxml.test.lite.Comments
    nanoxml.test.java.InternalDTD
    nanoxml.test.java.DoubleDTD
  Set 1
    nanoxml.test.lite.Comments
    nanoxml.test.lite.Simple
    nanoxml.test.java.ComplexPe
    nanoxml.test.java.ExternalDTD
    nanoxml.test.java.DoubleDTD
  All Tests
```

Figure 3: GZoltar's visualizations: Treemap and Sunburst [14].

is also an Eclipse plug-in that uses statistical analysis to determine the failure probability of every tested method.

Regression testing has been the field of several research studies in last years. Similar to RZOLTAR, the following tools are the best known tools for regression testing: MINTS [10] which uses an Integer Linear Programming (ILP) solver with multi-criteria test minimization; TestTube [5] partitions the SUT in several program entities, and follow the execution of test cases to analyze the relation between tests and program entities. Other techniques for regression testing include Greedy heuristic [6] and Program Slicing [9].

4. CONCLUSIONS

Testing and debugging is tedious and cumbersome phase in the software development life-cycle. Aimed at aiding the developer to test the software application and, if needed, pinpoint the source of observed failures, this paper describes GZOLTAR, an Eclipse plug-in which has a GZOLTAR view to deal with tasks about debugging, and RZOLTAR view for regression testing purpose. The toolset as well as a tutorial can be obtained from `http://www.gzoltar.com`.

Future work include the following. We plan to provide more techniques for minimizing test suites (e.g., greedy, MINTS) and add more visualizations of the diagnostic reports. Furthermore, there are plans to add the capability of dynamically instrumenting the source code, this way reducing the overhead imposed to collect information. Finally, we intend to combine other approaches, such as model-based diagnosis, to refine the rankings yielded by spectrum-based fault localization [13].

5. ACKNOWLEDGMENTS

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project reference PTDC/EIA-CCO/116796/2010.

6. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, Nov. 2009.

[2] 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 XP' 07*.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a Code Manipulation Tool to Implement Adaptable Systems. In *Proc. of the ASF Journées Composants*, 2002.

[4] J. Campos. Regression testing with GZoltar: Techniques for test suite minimization, selection, and prioritization. Master's thesis, University of Porto, Portugal, 2012.

[5] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: a system for selective regression testing. In *Proc. of the ICSE '94*, pages 211–220, Los Alamitos, CA, USA.

[6] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 1979.

[7] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, 1960.

[8] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. VIDA: Visual interactive debugging. In *Proc. of the ICSE '09*, pages 583–586, Washington, DC, USA.

[9] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2:270–285, July 1993.

[10] H.-Y. Hsu and A. Orso. MINTS: A General Framework and Tool for Supporting Test-suite Minimization. In *Proc. of the ICSE '09*, pages 419–429, Washington, DC, USA.

[11] T. Janssen, R. Abreu, and A. J. C. v. Gemund. Zoltar: A Toolset for Automatic Fault Localization. In *Proc. of the ASE '09*, pages 662–664, Washington, DC, USA.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization for Fault Localization. In *Proc. of the WSV '01*, Canada.

[13] W. Mayer, R. Abreu, M. Stumptner, and A. J. van Gemund. Prioritizing Model-Based Debugging Diagnostic Reports. In *Proc. of the DX' 08*, pages 127–134.

[14] A. Riboira. GZoltar: A Graphical Debugger Interface. Master's thesis, University of Porto, Portugal, 2011.

[15] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.