Testing and Debugging
Project 1:
Code Coverage
Projects

http://www.st.cs.uni-saarland.de/edu/testingdebugging10/
Public Project

• If you are not yet registered, you should register and meanwhile you can use the project:
  • User name: 1234567
  • Password: tad10
<table>
<thead>
<tr>
<th>Forum Name/Description</th>
<th>Threads</th>
<th>Posts</th>
<th>Last Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmierung 2 Staff</td>
<td>1</td>
<td>3</td>
<td>Apr 21, 2010 12:31:55 PM</td>
</tr>
<tr>
<td>Algemeine Besprechungen</td>
<td>1</td>
<td>1</td>
<td>Apr 12, 2010 1:39:10 PM</td>
</tr>
<tr>
<td>OFF-Topic</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
<tr>
<td>Programmierung 2</td>
<td>6</td>
<td>26</td>
<td>Apr 21, 2010 7:28:01 AM</td>
</tr>
<tr>
<td>Allgemeine Fragen</td>
<td>31</td>
<td>77</td>
<td>Apr 21, 2010 5:58:12 AM</td>
</tr>
<tr>
<td>Projekt 1</td>
<td>16</td>
<td>36</td>
<td>Apr 21, 2010 10:20:20 AM</td>
</tr>
<tr>
<td>Übungsklausur 1</td>
<td>2</td>
<td>7</td>
<td>Apr 21, 2010 6:43:19 PM</td>
</tr>
<tr>
<td>OFF-Topic</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
<tr>
<td>Testing and Debugging</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
<tr>
<td>General Questions</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
<tr>
<td>Project 1 Code Coverage</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
<tr>
<td>OFF-Topic</td>
<td>0</td>
<td>0</td>
<td>No Posts</td>
</tr>
</tbody>
</table>
Forum

- Questions can be asked on the forum: https://forum.st.cs.uni-sb.de/boards/index
- Password and user name outside the university: prog2-2010
- Your personal user name and password is the same as for the project.
Task

(a) Write a tool that measures statement coverage.

(b) Create tests that satisfy a coverage goal.

(c) Implement the Tarantula fault localization technique.
Set up the project

• Check out the project from the given URL
• Use Maven to build it:
  • `mvn compile`
  • `mvn assembly:assembly`
  • `mvn eclipse:eclipse`
(a) Statement coverage

- We recommend Eclipse AST (Abstract Syntax Tree) to instrument the source code.
- AST uses visitor pattern - visit and endVisit methods.
- Instrumentation should log the covered statements and write the data to disk.
- See lecture: Advanced Coverage Criteria
Running Tests

- *ant check-coverage* runs integration tests.
- Produces 2 summary files: *check-coverage-passing.txt* and *check-coverage-failing.txt*. 
(b) Coverage Goal

• 100% Coverage has to be reached for two classes in commons-math.

• JUnit 4 Templates in the project have to be completed.

• Tests from the commons-math project are allowed.
Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique

James A. Jones and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
jones@cc.gatech.edu, harrold@cc.gatech.edu

Abstract
The high cost of locating faults in programs has motivated the development of techniques that assist in fault localization by automating part of the process of searching for faults. Empirical studies that compare these techniques have reported the relative effectiveness of four existing techniques on a set of subjects. These studies compare the rankings that the techniques compute for statements in the subject programs and the effectiveness of these rankings in locating the faults. However, it is unclear how these four techniques compare with Tarantula, another existing fault-localization technique, although this technique also provides a way to rank statements in terms of their suspiciousness. Thus, we performed a study to compare the Tarantula technique with the four techniques previously compared. This paper presents our study—it overviews the Tarantula technique along with the four other techniques studied, describes our experiment, and reports and discusses the results. Our studies show that, on the same set of subjects, the Tarantula technique consistently outperforms the other four techniques in terms of effectiveness in fault localization, and is comparable in efficiency to the least expensive of the other four techniques.

Categories and Subject Descriptors
D.2 [Software]: Software Engineering; D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Reliability, Experimentation

Keywords
Fault localization, automated debugging, program analysis, empirical study

1. INTRODUCTION
Debugging software is an expensive and mostly manual process. Of all debugging activities, locating the faults, or fault localization, is the most expensive [14]. This expense occurs in both the time and the cost required to find the fault. Because of this high cost, any improvement in the process of finding faults can greatly decrease the cost of debugging.

In practice, software developers locate faults in their programs using a highly involved, manual process. This process usually begins when the developers run the program with a test case (or test suite) and observe failures in the program. The developers then choose a particular failing test case to run, and iteratively place breakpoints using a symbolic debugger, observe the state until an erroneous state is reached, and backtrack until the fault is found. This process can be quite time-consuming.

To reduce the time required to locate faults, and thus the expense of debugging, researchers have investigated ways of helping to automate this process of searching for faults. Some existing techniques use coverage information provided by test suites to compute likely faulty statements (e.g., [1, 4, 5, 10, 12]). Other techniques perform a binary search of the heap data using one failing test case and one passing test case to find likely faulty statements (e.g., [2, 15]). Still other techniques are based on the remote monitoring and statistical sampling of programs after they are deployed (e.g., [3, 7, 8, 9]). Most papers reporting these techniques also report empirical studies that evaluate the presented technique in terms of its effectiveness and efficiency. However, because of the difficulty in comparing techniques developed on different platforms for different languages and using different programs and test suites, few empirical studies have reported comparisons of existing techniques.

Although comparing the fault localization of techniques is difficult, several recent studies have compared four existing techniques in terms of their ability to localize faults. Beiner and Reis [12] presented their technique, called Nearest Neighbor, and compared it to two techniques that use set union and intersection operations on coverage data (similar to those presented in [7] and [11]). Their studies show that, on a given set of subjects, the Nearest-Neighbor technique performs more effectively than the two set-based approaches. Lauer, Clev, and Zeller [2] presented their technique, called Causal Provenance, compared it to the Nearest Neighbor technique, and found that, on the same set of subjects, Causal Provenance consistently performs better than Nearest Neighbor (and thus set union and intersection tech-
(c) Fault Localization

• Implement the Tarantula technique. (presented in the next lecture)

• Paper can be obtained from our web site. (Password and user name: tad10)

• ant check-fault-localization runs an integration test.
Grading

• Version in svn, of 13th May will be graded.
• Public tests have to be passed.
• Secret tests are used for grading.
Grading for lecture

Grading

To pass, you need

- 60% of exam points and
- 60% of project points

Exam 50%
Project 1 12%
Project 2 8%
Project 3 15%
Project 4 15%
### Grading Scheme for Project 1

<table>
<thead>
<tr>
<th>Grade</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>at least 90% of public and 50% of secret tests</td>
</tr>
<tr>
<td>3</td>
<td>100% of public and at least 50% of secret tests</td>
</tr>
<tr>
<td>2</td>
<td>100% of public and at least 90% of secret tests</td>
</tr>
<tr>
<td>1</td>
<td>100% of public and 100% of secret tests</td>
</tr>
</tbody>
</table>
• Use `svn status`.

• Do a fresh checkout and check whether project compiles and passes tests.
Specification is incomplete

- We try to test only for specified behavior.
- Tests will be made publicly available.
- You can write tests that are run against the reference implementations.
Example

1. for(int i = 0;
2.     i < 2;
3.     i++;) {

• We only expect the start (line 1) to be coverable.
• See Test: 
  AnotherLoopCoverageTest
Constructors

1. public Foo()
2. super();
3. this.val = 1;

Only line 3 is expected to be coverable.

- *this* and *super* constructor calls are not expected to be covered
- “The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass (§8.8.7.1).”
You are allowed to use 3rd party libraries.

If you want to use a library not listed in pom.xml, contact us.

pom.xml and build.xml and src/test will be overwritten for secret tests.