Extraction of Bug Localization Benchmarks from History

Valentin Dallmeier
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
dallmeier@cs.uni-sb.de

Thomas Zimmermann
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
tz@acm.org

ABSTRACT

Researchers have proposed a number of tools for automatic bug localization. Given a program and a description of the failure, such tools pinpoint a set of statements that are most likely to contain the bug. Evaluating bug localization tools is a difficult task because existing benchmarks are limited in size of subjects and number of bugs. In this paper we present iBUGS, an approach that semi-automatically extracts benchmarks for bug localization from the history of a project. For ASPECTJ, we extracted 369 bugs, 223 out of these had associated test cases. We demonstrate the relevance of our dataset with a case study on the bug localization tool AMPLE.

Categories and Subject Descriptors: D.2.5 [*Software Engineering*]: Testing and Debugging—*debugging aids, diagnostics, testing tools, tracing*; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—*corrections, version control*

General Terms: Management, Measurement, Reliability

1. INTRODUCTION

An objective evaluation of a bug[1] localization technique requires a set of programs with known bugs. The Software-Artifact Infrastructure Repository (SIR) aims at providing such a set of subject programs with known bugs that can be used as benchmarks for bug detection tools [4]. Subjects from the SIR have already been used in a number of evaluations [14, 5, 1, 6, 7, 15, 2].

Despite its success, using subjects from the SIR has several drawbacks. Most of them are rather small and contain only few known bugs. Also, the majority of bugs were artificially seeded and are not representative for realistic bugs. Therefore, it is difficult to argue that results obtained for SIR can be transferred to real projects. The absence of realistic bugs for large programs in SIR is because the collection is a tedious task—a task which we want automate with this work.

In this paper, we propose an approach called iBUGS that semi-automatically extracts benchmarks for bug detection with realistic bugs from a project's history. We applied our approach to the version archive and bug database of ASPECTJ, a large open-source project with more than 5 years of history. Using our technique we were able to extract faulty versions for 369 bugs. For 223 of these bugs, we also identified at least one associated test. We assembled this data in a repository called iBUGS and made it publicly available for other researchers.

The contributions of this paper are as follows:

1. A technique to *semi-automatically extract bug localization benchmarks* from a project's history (Section 2).
2. A *publicly available repository* containing a large open source project with *realistic* 369 bugs, meta information about the bugs, and a test suite to run the program (Section 3).

Using iBUGS, we re-evaluate the dynamic bug localization tool AMPLE with only 142 lines of JAVA code (Section 4). We then discuss related work (Section 5) and conclude the paper (Section 6).

2. THE IBUGS APPROACH

Our iBUGS approach extracts bugs from a project's history and assembles these bugs into a benchmark for bug localization tools. All we need is a version archive (such as CVS) to identify changes that fixed bugs. iBUGS also identifies regression tests that developers provided together with fixes. These tests are used by several bug localization tools to reproduce the corresponding failure [2, 15].

A subject for the iBUGS repository is created as follows:

1. Recognize fixes and bugs. We automatically identify fixes by searching log messages for references to bug reports such as "Fixed 42233" or "bug #23444" [11]. The underlying assumption is that the location of a bugs and its correction are the same.

2. Extract pre-fix and post-fix versions. For a bug, the *pre-fix* version of a program still contains the bug, while in the *post-fix* version the bug has been fixed. We ignore all fixes that do not change the program's functionality (like correcting a misspelled dialog title).

3. Build versions and run tests. We build the pre- and post-fix versions of all bugs and execute the test suites (if any) to collect test outcomes. In theory this step could be fully automated; however, in practice it sometimes requires manual interaction since software evolves and the names of build targets and output files change.

4. Recognize tests associated with bugs. Many developers commit test cases with fixes to prevent that previously fixed bugs re-emerge. When run on the pre-fix version, these tests are helpful for bug localization tools that need to reproduce the failure [2, 15]. Note that sometimes more than one test are committed for a bug and not all of them fail on the pre-fix version. Also, bugs without associated tests are still useful for static bug localization tools.

[1]We use the term bug to denote a defect in the code that causes a program to fail.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

```
  TypeX onType = rp.onType;
  if (onType == null) {
-     Member member = EclipseFactory.makeResolvedMember(declaration.binding);
-     onType = member.getDeclaringType();
+     if (declaration.binding != null) {
+     Member member = EclipseFactory.makeResolvedMember(declaration.binding);
+         onType = member.getDeclaringType();
+     } else {
+         return null;
+     }
  }
  ResolvedMember[] members = onType.getDeclaredPointcuts(world);
```

Tokens changes computed by APFEL [16]:
K-*else* (+1) K-*if* (+1) K-*null* (+1) K-*return* (+1)
O-*!=* (+1)
T-*MethodDeclaration* (+1) V-*declaration* (+1)
Z-if-*"declaration.binding != null"* (+1)

Concise fingerprint:
KZ

Full fingerprint:
K-else K-if K-null K-return O-!= T V Z-if

Figure 1: Fingerprints for Bug 87376 "NPE when unresolved type of a bound var in a pointcut expression (EclipseFactory.java:224)".

Token type	Description
Z–*expression*	Expressions that are used in casts, conditional statements, exception handling, loops, and variable declarations.
K–*keyword*	Keyword such as *for, if, else, new,* etc.
M–*method-name*	Method calls.
H–*exception-name*	Catch blocks for exceptions.
V–*variable-name*	Names of variables.
T–*variable-type*	Types of variables.
Y–*literal*	Literals such as numbers or strings)
O–*operator-name*	Operators such as +, −, *&&,* etc.

Table 1: Token types used for iBUGS.

ASPECTJ	Number
Candidate bug reports	
– retrieved from CVS and BUGZILLA	489
– after removing false positives	485
– that change source code	418
– for which pre-fix and post-fix versions compile	406
– for which test suites compile	369
Final dataset	
– bugs	369
– bugs with associated test cases	223

Table 2: Breakdown of the analyzed bug candidates.

5. Annotate bugs with meta information. Some bugs may not meet the assumptions and prerequisites of a specific bug localization tool. In order to provide an efficient selection mechanism, we automatically annotate bugs with *size properties* of their corresponding fix, such as the number of changed methods and classes, as well as the number of churned lines. In addition, we provide *syntactic properties* that describe what syntactic tokens the fix changed (see Table 1). The *concise fingerprint* summarizes the most essential syntactic changes such as method calls, expressions, keywords, and exception handling. In addition, the *full fingerprint* records changes in variable names and contains more detailed information about the affected tokens. Figure 1 shows a sample fingerprint.

6. Assemble iBUGS repository. We collect all versions of a subject in a Subversion repository (to reduce space). The meta information is stored in an XML file (see Figure 3). Also, we provide for each bug the files that were fixed and, if available, associated test cases.

3. THE ASPECTJ DATASET

In this section we present several characteristics of the dataset that we created from the ASPECTJ project. ASPECTJ is an aspect-oriented extension to the Java programming language and includes among other tools a compiler. Its history is well-maintained, developers regularly link fixes to the bug database and include test cases when they fix bugs.

3.1 Size of ASPECTJ

The ASPECTJ compiler consists of 75 kLOC and its test suite contains 1,184 test cases. We analyzed all 7,947 commits by the 13 developers between July 2002 and October 2006 (each commit can be understood as a version of ASPECTJ). From the bug database, we identified 369 bug reports that changed program code, for 223 we found associated test cases (see Table 2). The total size of the ASPECTJ dataset in the iBUGS repository is 260 MB.

3.2 Size of fixes in ASPECTJ

The majority of bugs in ASPECTJ (201 out of 369) was corrected in exactly one method. This suggests that most bugs are local, spanning across only few methods. Also, many fixes in ASPECTJ are small: 44.4% of all fixes churned ten lines or less; almost 10% of all fixes are one-line fixes, i.e., churned exactly one line. Only few fixes deleted code (about one third), most fixes modifies existing code (e.g., wrong expressions) or added new code (e.g., null pointer checks). The percentages of small fixes that we observed are consistent with the ones observed by Purushothaman and Perry [9].

3.3 Fingerprints in ASPECTJ

While the majority of fixes changed different kinds of token (for instance 205 fixes with fingerprint "KMZ"), several fixes changed *exclusively* one token type: 30 fixes changed only method calls ("M"), 15 fixes only keywords ("K"), and 12 fixes only expressions ("Z"). Only literals and variable names were changed by 39 fixes changed (empty fingerprint). For examples of fingerprints and characteristic fixes, we refer to our technical report [3].

4. THE AMPLE CASE STUDY

With iBUGS we re-evaluated AMPLE, our dynamic bug localization tool for JAVA. AMPLE was previously evaluated on NANOXML (about eight kLOC) from the Software-Artifact Infrastructure Repository, and four bugs from the ASPECTJ compiler [2]. Back then finding those four bugs required manually searching the bug database and version archive of ASPECTJ. By using our iBUGS repository, we were able to repeat our evaluation with a much larger number of bugs and less manual effort.

AMPLE works on a hypothesis first stated by Reps et al. [10]: *bugs correlate with differences in traces between a passing and a failing run.* AMPLE captures the control-flow of a program as sequences of method calls issued by the program's classes. A class

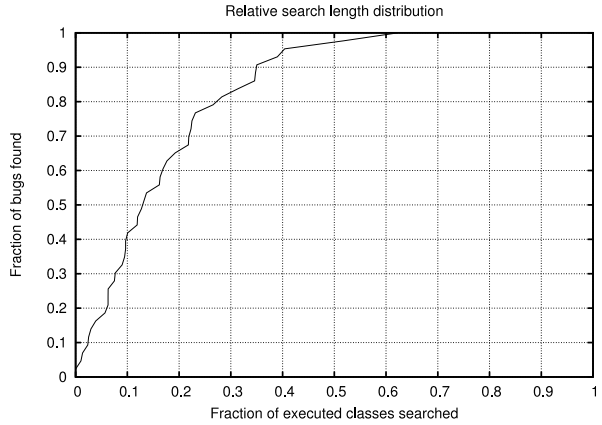

Figure 2: Using the rankings of AMPLE, 40% of all bugs are located by searching at most 10% of the executed code.

that produces substantially different call sequences in failing and passing runs is more likely to contain the bug than a class that behaves the same in all runs. The output of AMPLE is a ranking of classes that puts the class with the strongest deviations on top.

Experimental setup.

In order to be able to use our previous evaluation method [2], we restricted our experiments to bugs that were fixed in a single class (74 bugs). Additionally, we ignored bugs for which associated tests did not fail on the corresponding pre-fix version (44 bugs remaining). Failing tests were used as failing runs and we randomly chose three passing tests from the regression test suite as passing runs. For AMPLE's parameter *length of call sequences*, we used a value of 5 which produced the best results in the previous evaluation. The entire evaluation is controlled by a short JAVA program (142 LOC) that reuses the ANT scripts provided by the iBUGS repository to build versions and run the passing and failing tests.

Results.

For each bug, AMPLE returns a ranking of classes that were executed during the failing run. This ranking is a recommendation to a programmer in which order she should search the classes when locating the bug. The usefulness is measured by the *search length*, i.e., the number of classes that are ranked higher than the class with the bug. A low search length means that a programmer has to check only a small portion of the code before she locates the bug.

Figure 2 shows a cumulative plot of the search length relative to the number of executed classes: a developer who investigates the top 10% of AMPLE's rankings, would locate the bug in 40%. While AMPLE is pretty effective in most cases, the plot also indicates that for only few rankings more than half of the executed code has to be searched. We encourage other researchers to evaluate their bug localization tools by using the iBUGS repository (see the step-by-step guide in Figure 4).

5. RELATED WORK

The publicly available *Subject-Artifact Infrastructure Repository (SIR)* provides six JAVA and thirteen C-programs [4]. Each program comes in several different versions together with a set of known bugs and a test suite. There are two drawbacks of SIR: First, most projects are rather small, the average project size is only 11 kLOC. Second, almost all bugs are artificially seeded; however, realistic bugs are likely to be different. With iBUGS, we will provide SIR with subjects that come with a large number of realistic bugs.

```
<bug id="69459">
 <property name="files-churned" value="1"/>
 <property name="classes-churned" value="1"/>
 ...
 <concisefingerprint>KMZ</concisefingerprint>
 <fullfingerprint>K-else K-if K-null M O-! O-&amp;amp;amp;
  O-+ T V Y Z-if</fullfingerprint>
 <pre-fix-testcases failing="105" passing="1203"/>
 <post-fix-testcases failing="105" passing="1204"/>
 <testsforfix type="new">
  <file location="ajcTests.xml">
    <test name="Hiding_of_Instance_Methods"/>
  </file>
 </testsforfix>
 <fixedFiles>
  <file name="ResolvedTypex.java" revision="1.27">
...
1194c1194,1202
&lt;
---
&gt; if (parent.isStatic()
&gt; &amp;amp;amp; !child.isStatic()) {
...</file>
 </fixedFiles>
</bug>
```

Figure 3: XML content descriptor for bug 69459.

Spacco et al. collected bugs made by students during programming projects in their *Marmoset* project [13]. The Marmoset data contains several hundred projects including test cases. Most student projects are small, however, and not always representative for industrial software development. In contrast, iBUGS focuses on large open-source projects with industrial alike development processes.

In their *BugBench* benchmark suite, Lu et al. manually collected 19 bugs from 17 C-programs, most bugs being memory related [8]. The size of the subjects ranged from two kLOC to one mLOC. In contrast to iBUGS, BugBench is not publicly available.

6. CONCLUSION

The version history of a project collects all past successes and failures. In this paper, we presented iBUGS, an approach that leverages the history of a project to automatically extract benchmarks for bug localization tools. These benchmarks are useful for both static and dynamic bug localization tools: for ASPECTJ, we extracted 369 bugs and their fixes, 223 out of these had associated test cases. To summarize, our contributions are as follows:

1. iBUGS *automatically extracts bug localization benchmarks* from version archives and bug databases; only when the build process changes, manual interaction is necessary.
2. iBUGS collects a *large number of realistic bugs* as they actually occurred in the development history. Therefore, results obtained on iBUGS are more likely to transfer to real projects.
3. iBUGS is *publicly available* and comes with a fully-fledged infrastructure for reconstructing, building, and testing the versions with and without bugs (step-by-step guide in Figure 4).

Our ASPECTJ dataset is a first step towards the "huge collection of software defects" that by Spacco et al. [12] at the Bugs workshop at PLDI 2005. The history of open source projects offer a huge number of collector's bugs which wait to be discovered by researchers. Therefore, our future work for iBUGS will concentrate on adding more subjects. For ongoing information on the project and an extended technical report [3], log on to

`http://www.st.cs.uni-sb.de/ibugs/`

Acknowledgments. Thomas Zimmermann is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". LOC numbers were generated using David A. Wheeler's "SLOCCount". Christian Lindig, Rahul Premraj, and David Schuler provided valuable comments on earlier revisions of this paper.

7. REFERENCES

[1] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of 27th International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[2] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proc. of 19th Europ. Conf. on Object-Oriented Prog. (ECOOP)*, pages 528–550, 2005.

[3] V. Dallmeier and T. Zimmermann. Automatic extraction of bug localization benchmarks from history. Technical report, Saarland University, Saarbrücken, Germany, June 2007.

[4] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[5] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 15–26, 2005.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proc. of European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 286–295, 2005.

[8] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *PLDI Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[9] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[10] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. European Software Engineering Conference (ESEC/FSE)*, pages 432–449, Sept. 1997.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On Fridays. In *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, U.S., May 2005.

[12] J. Spacco, D. Hovemeyer, and W. Pugh. Bugbench: Benchmarks for evaluating bug detection tools. In *PLDI Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[13] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: an automated programming project snapshot and testing system. In *Proc. International Workshop on Mining Software Repositories (MSR)*, 2005. See also: http://marmoset.cs.umd.edu/.

[14] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 282–291, 2006.

[15] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the 28th Int. Conf. on Software Eng.*, pages 272–281, 2006.

[16] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, October 2006.

1. **Select bugs.** Use the meta information provided in the file *repository.xml* (see Figure 3) to select relevant bugs.

   Example: In order to select all bugs that raised a NullPointerException, use the XPath expression

   /repository/bug[tag="null pointer exception"]

2. **Extract versions.** Use the ant task *checkoutversion*.

   Example: In order to checkout the pre-fix and post-fix versions for Bug 4711, type

   ```
   ant -DfixId=4711 checkoutversion
   ```

   The results are placed in the directory *"versions/4711/"*.

3. **Build the program.** Use the ant task *buildversion*.

   Example: Build the pre-fix version of Bug 4711 with

   ```
   ant -DfixId=4711 -Dtag=pre-fix buildversion
   ```

   If the build succeeds, you find the Jar files in the directory *"…/pre-fix/org.aspectj/modules/aj-build/dist/tools/lib/"*

   Note: Static tools can analyze the Jars in this directory, while dynamic tools that execute tests need to instrument the Jars created in the next step.

4. **Build tests (dynamic tools).** Use the ant task *buildtests*.

   Example: In order to build the tests for the pre-fix version of Bug 4711, type

   ```
   ant -DfixId=4711 -Dtag=pre-fix buildtests
   ```

   This creates a Jar file that includes the ASPECTJ compiler and all resources needed for testing in the directory *"versions/4711/prefix/org.aspectj/modules/aj-build/jars/"*.

5. **Run test suites (dynamic tools).** Use the ant tasks *runharnesstests* for the integration test suite and *runjunittests* for the unit test suite of ASPECTJ, respectively.

   Example: Run unit tests for the pre-fix version of Bug 4711

   ```
   ant -DfixId=4711 -Dtag=pre-fix runjunittests
   ```

6. **Run specific tests (dynamic tools).** Generate scripts by using the ant task *gentestscript* and execute them.

   Example: In order to execute test *"SUID: thisJoinPoint"* described in file *"org.aspectj/modules/tests/ajcTests.xml"* generate a script with

   ```
   ant -DfixId=4711 -Dtag=pre-fix \
     -DtestFileName="org.aspectj.modules/\
                     tests/ajcTests.xml"\
     -DtestName="SUID: thisJoinPoint".
   ```

   This creates a new ant script in the directory *"4711/pre-fix/org.aspectj/modules/tests/"*. Execute this file to run test *"SUID: thisJoinPoint"*.

   Hint: All tests executed by the test suite are described in the file *"4711/pre-fix/testresults.xml"*.

7. **Assess your tool.** Compare the predicted bug location against the location changed in the fix (see *repository.xml*).

   Note: Static bug localization tools typically integrate with Step 3 and 4. Dynamic tools need to run programs and therefore integrate with Step 4, 5, and 6.

Figure 4: Step-by-step guide to your own evaluation.