Automatic 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 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 (useful to test dynamic tools). We demonstrate the relevance of our dataset for both static and dynamic bug localization tools with case studies on FINDBUGS and 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

To perform a scientific experimental investigation of software defects, we need bugs. Lots of them. Thousands only begins to cover it.
– Spacco, Hovemeyer, Pugh [24]

In the recent past, researchers have proposed a number of tools for automatic bug[1] localization [30, 8, 14, 2, 15, 16, 31, 4]. Given a program and a description of the failure, a bug localization tool pinpoints a set of statements most likely to contain the bug that caused the failure. Although all approaches try to solve the same problem, many papers use different datasets to evaluate the bug localization accuracy. This makes it difficult for researchers to compare new approaches with existing techniques.

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

The Software-Artifact Infrastructure Repository (SIR) [5] aims at providing a set of subject programs with known bugs that can be used as benchmarks for bug detection tools. Subjects from the SIR have already been used in a number of evaluations [30, 8, 14, 2, 15, 16, 31, 4]. Despite its success, the subjects currently present in the repository have several drawbacks. Most of the subjects are rather small and contain only a few known bugs. Another issue is that the majority of programs contains only bugs that were artificially seeded into the program. It is therefore difficult to argue that results obtained for these subjects can be transferred to real projects with real bugs. A reason why the SIR contains only few subjects with real bugs is that collecting this data is a tedious task.

We propose iBugs, a technique that automatically extracts benchmarks with real bugs from a project's history as available in software repositories and bug databases. Our approach searches log messages of code changes for references to bugs in the bug database. For example, a log message "Fixed bug 45298" indicates that the change contains a fix for bug 45298. We provide faulty versions for bugs by extracting snapshots of the program right before the fix was committed. For each version we try to build the project and execute the test suite. Syntactical analysis of the fixes allows us to provide a categorization of bugs and to identify tests that are associated with bugs.

We have applied our approach to 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 provide 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 *automatically extract bug localization benchmarks* from a project's history.
2. A *publicly available repository* containing a large open source project with 369 bugs, meta information about the bugs, and a test suite to run the program.
3. Two case studies that show that the *iBugs repository can be used to evaluate bug localization tools.*
4. A *step-by-step guide for researchers* that want to use our repository to evaluate their own tools.

In the remainder of the paper we discuss related work (Section 2), explain our approach and practical experiences (Section 3), present characteristics of the iBugs repository (Section 4) as well as the case studies (Section 5) and end the paper with concluding remarks and ideas for future work (Section 6).

2. RELATED WORK

We discuss the properties of existing benchmark suites, present a selection of bug localization approaches published in the recent past and what subjects were used for evaluation (see also Table 1), and summarize related work about bug categorization.

2.1 Existing Benchmark Suites

PEST. The National Institute of Standards and Technologies provides a small suite of programs for evaluating software testing tools and techniques (PEST). The current version contains two artificial C programs with each less than 20 seeded bugs. In contrast to the PEST suite, we aim at providing a set of real programs with bugs that actually occured in the program.

BugBench. Lu et al. [17] describe a benchmark suite with 17 C programs ranging from 2000 up to 1 million lines of code. The paper describes 19 bugs the authors localized in those projects, with more than two thirds being memory related bugs that can never occur in modern languages like JAVA or C#. We could not further investigate the benchmark since we could not find a released version.

Software-Artifact Infrastructure Repository (SIR). The publicly available Subject Infrastructure Repository to date provides 6 Java and 13 C-programs, including the well-known Siemens test suite [21, 19]. Each program comes in several different versions together with a set of known bugs and a test suite. Subjects from the repository have already been used in a number of evaluations. A drawback of the current subjects in the repository is that the average project in the repository is only 11 kLOC in size while most real projects are much larger. Another problem is that almost all subjects only have artificially seeded bugs which often represent only a small portion of the bugs that occur in real projects. Using our technique to mine bugs from source code repositories, we can provide subjects for the SIR with a large number of realistic bugs.

Marmoset. The group around Bill Pugh collected bugs made by students during programming projects. Their Marmoset project contains several hundred projects including test cases [25]. However, most student projects are small and not always representative for industrial development processes. In contrast to Marmoset, our iBugs project focuses on large open-source projects with industrial alike development processes.

2.2 Defect Localization Tools

Yang et al. [30] dynamically infer temporal properties (API rules) for method invocations from a set of training runs. The approach handles imperfect traces by allowing for a certain number of violations to a candidate rule. Violations of the rules in testing runs may point to bugs. Hangal et al. [8] tries to automatically deduce likely invariants from a set of passing runs. Invariants are used to flag deviating behavior right before the program crashes in a failing run. Li and Zhou [14] mines programming rules from a program's code. Violations of these rules are flagged as possible bug locations. The previously described approaches provide an *ad hoc evaluation* with subjects that are sometimes not available to the public (like the Windows Kernel). Most of them also report only bugs they were able to detect, but omit information about bugs they missed. This makes it difficult for other researchers to reproduce work by others and to assess the performance of their own approaches.

Several researcher improved on the lack of reproducibility by additionally testing their bug localization tools on publicly available benchmarks such as Gregg Rothermel's SIR. Cleve and Zeller [2] establish cause-effect chains for failures by applying Delta Debugging several times during a program run. Suspected bug locations are pin-pointed whenever the variable relevant for the failure changes. Liblit et al. [15] proposes a statistical approach that collects information about predicate evaluation from a large number of runs. Predicates that correlate with failure of the program are likely to be relevant for a bug. The SOBER tool by Liu et al. [16] calculates evaluation patterns for predicates from program executions. If a predicate has deviating evaluation patterns in passing and failing runs, it is considered bug relevant. Zhang et al. [31] automatically identify a (set of) predicate crucial for a failure. The suspected bug location is the dynamic slice of the crucial predicate(s). The AMPLE tool by Dallmeier et al. [4] captures the behavior of objects as call-sequence sets. Classes are ranked according to the degree of deviation between passing and failing runs.

2.3 Bug Classification

Several researchers investigated the phenomenon of bugs in the past. Ko and Myers proposed a methodology that describes the causes of software errors in terms of chains of cognitive breakdowns [12]. In their paper, they also summarized other studies that classify bugs. Defect classification has been also addressed by several other researchers: Williams and Hollingsworth manually inspected the bugs from the Apache web server and found that logic errors and missing checks for null pointers and return values were the most dominant bug categories [26, 27]. Xie and Engler demonstrated that many redundancies in source code are indicators for bugs [29]. Since such redundancies are easily caught by static analysis, this lead to an advent of static bug finding tools, such as FINDBUGS [9], JLint [11], and PMD [10] (for a comparison we refer to Rutar et al. [22]). Typically, such tools take rules, and search for their violations (roughly, every rule corresponds to a bug category). Recently, automatic bug classification techniques using natural language emerged: Anvik et al. used such techniques to assign bugs to developers [1] and Li et al. investigated whether bugs have changed nowadays [13].

3. HOW-TO CREATE A SUBJECT

Our goal is to exploit the history of a project to build a repository with realistic bugs that can be used to benchmark bug localization tools. We classify each bug by the characteristics of its fix, for example the size and the syntactical elements that were changed. For each bug we provide a compilable version with and without the bug as well as a means to run tests on the program.

The following steps are neccessary to prepare a subject for the iBugs repository. The sequence in which the steps are performed can vary, but some steps have to be performed before others (versions need to be extracted before they can be built):

1. Recognize fixes and bugs.
2. Extract versions from history.
3. Build and run tests.
4. Recognize tests associated with bugs.
5. Annotate bugs with size properties.
6. Annotate bugs with syntactic properties (*fingerprints*).
7. Assemble iBugs repository.

We first discuss the prerequisites for our approach and then present each step in detail. The number of bug candidates that we analyzed at the various stages are summarized in Table 2.

Approach	Language	Evaluation Type	Subjects
SOBER	C	Benchmark + Ad hoc	Siemens Test Suite (SIR), BC
AMPLE	Java	Benchmark + Ad hoc	Java Subject from SIR, 4 Bugs in AspectJ
Liblit05	C	Benchmark + Ad hoc	Siemens Test Suite (SIR)
Cause Transitions	C	Benchmark	Siemens Test Suite (SIR)
Predicate Switching	C	Benchmark	Siemens Test Suite (SIR)
Perracotta	Java, C	Ad hoc	JBOSS Transaction Module, Windows Kernel
PR-Miner	C	Ad hoc	Large C projects (Linux Kernel)
Diduce	Java	Ad hoc	Java SSE, MailManage, Joeq

Table 1: How are bug localization tools evaluated?

	Number
Candidates	
– 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
ASPECTJ dataset	
– bugs	369
– bugs with associated test cases	223

Table 2: Breakdown of the analyzed bug candidates.

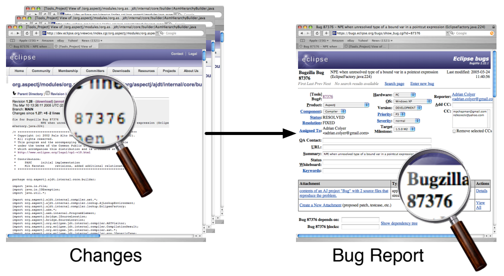

Figure 1: Linking bug reports and changes.

ASPECTJ: To illustrate how our approach works in practice, we describe our experiences with preparing the ASPECTJ compiler project as a subject.

3.1 Prerequisites

In order to be suitable for the iBugs repository, a project needs to meet the following prerequisites:

Source repository (required). The project must provide access to a system like CVS or SVN where the project history is stored. We use the repository to identify changes that fix a bug.

Bug tracker (optional). The availability of a bug tracking system like Bugzilla or Jira helps eliminate false positives in the detection of changes that fix a bug.

Test infrastructure (optional). If the project has a test infrastructure we can use it to provide runs of the program. If there is no test suite available, the subject can still be used to evaluate static bug detection tools.

Our experience with open-source projects shows that all successful projects meet these requirements. Organizations like the APACHE and ECLIPSE foundations use a standard infrastructure with source repositories and bug trackers for all of their projects.

ASPECTJ: The project builds a compiler that extends the JAVA language with aspect-oriented features. It provides access to a CVS repository with over 5 years of history and a bug tracking system with more than 1000 entries. With over 75000 lines of code excluding test code, it is among the larger open source projects.

3.2 Recognize fixes and bugs

The first step in the iBugs approach is to identify changes that correct bugs, in particular, bugs that were reported to bug databases such as Bugzilla. Typically, developers annotate every change with a message to describe the reason for that change. As sketched in Figure 1, we automatically search these messages for references to bug reports such as "Fixed 42233" or "bug #23444".[2] Basically every number is a potential reference to a bug report, however such references have a low trust at first. We increase the trust level when the message contains keywords such as "fixed" or "bug" or matches patterns like "# and a number". Since changes may span across several files, we combine all changes made by the same author, with the same messages and the same timestamp (with a fuzziness of 200 seconds) into a *transaction* [33]. Finally, every change with a reference to a bug report is assumed to be a fix and serves as a candidate for our bug dataset. Our approach for mapping code changes to bug reports is described in detail by Śliwerski et al. [23] and is similar to the approaches used by Fischer et al. [6] and by Čubranić et al. [3].

ASPECTJ: We were able to identify 890 transactions that fixed a bug. We removed all bugs that took more than one change to be fixed, since we cannot be sure which change was really neccessary to fix the bug. For similar reasons we did not consider changes that fix more than one bug. Altogether we found 489 bugs that were fixed only once in a transaction that fixed only one bug. A manual investigation of log messages revealed that 4 of them were actually false positives (the number in the log message accidentaly matched a bug id) and had to be removed.

3.3 Extract versions from repository

For each bug we extract two versions of the program (see Figure 2): The *pre-fix* version represents the state of the program right before the bug was fixed, while the *post-fix* version also includes the fix. We then compare these two versions and remove all fixes that don't change the program code. This is neccessary because some fixes don't affect the functionality of the program (like for example a

[2]The format of references to bug reports is project specific. It depends especially on the bug tracking system that is used.

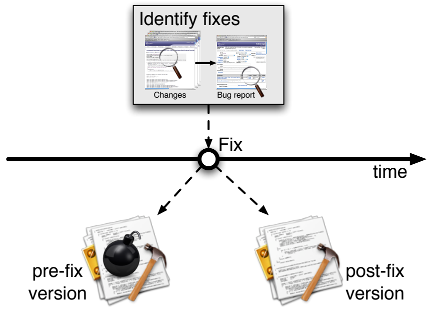

Figure 2: Extract pre-fix and post-fix versions.

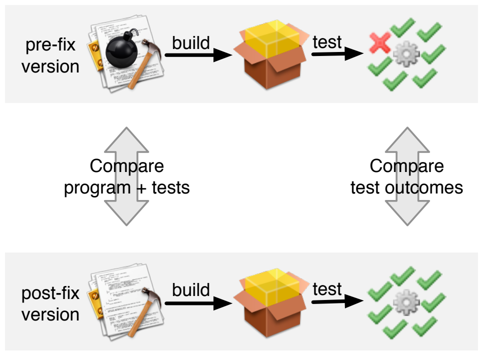

Figure 3: Building and testing pre-fix and post-fix versions.

misspelled dialog title in a resource file).

ASPECTJ: Altogether we found 67 bugs that did not change the source of the program and removed them from the iBugs repository.

3.4 Build and run tests

In the next step we prepare the pre- and post-fix versions of all bugs for execution (see also Figure 3). First we try to build each version. If the build process goes beyond a simple compile, most projects provide a build file. We identify the build file by examining the project and use it to run a build. Depending on the project, this may already include building and running a unit test suite. If this is not the case, we manually trigger the test suite and collect information about which tests were run and the outcome (pass or fail) of each test. After this step we remove all versions that fail to build.

ASPECTJ: The project provides a build file with seperate targets for building and running the program and its test suite. We first tried to build the program and found 12 versions that had compiler errors. We removed those versions and tried to build the test suite for all remaining versions.

Building and running the test suite for all versions required a lot more effort than building the project. This is due to some inconsistencies in the test system and the fact that the test process changed several times over the history of the project. This caused (amongst others) the following problems:

- For some versions the tests cannot be built without having all modules in an Eclipse workspace.
- In some cases the program built fine but the tests had compiler errors.
- The names of build targets and output files changed several times.

We analyzed the changes in the test system over time to fix as many problems as possible. For 37 bugs we could not build the test suite and therefore removed them. The remaining 369 bugs were included in the iBugs repository.

3.5 Recognize tests for bugs

Many dynamic bug localization tools [4, 2, 31] require a run that reproduces the failure and a passing run. While the project's test suite provides us with passing runs, it almost never contains failing runs for a previously unknown bug. This is because otherwise the bug would have been caught already by running the test suite and we assume that developers run the tests before realeasing the project.

To solve this problem we analyze the fixes for each bug and look for new tests that are committed together with a fix. The fact that a test is committed together with a fix is a strong indication that the test is related to the bug. Not all of these tests actually fail when executed since sometimes developers commit more than one test to check interesting cases that were discovered when fixing the bug. Bugs for which we can't find an associated test are not removed from the iBugs repository, as they may still be useful for static bug localization tools.

The method to identify tests committed with fixes depends on the type of tests that are used in the project. However there is only a small number of testing frameworks used in practice and we can cover a lot of projects with techniques for the most popular ones.

ASPECTJ: The project uses two different types of tests. Unit tests are implemented using the JUNIT [7] framework, a popular testing framework for JAVA. Integration tests for the compiler (referred to as harness tests) are described in XML files. Our approach for identifying new JUNIT tests is straightforward: We examine all classes that were changed during the transaction that fixed the bug. A new test is found if a new subclass of `TestCase` was committed or a new test was added to an existing `TestCase`. New harness tests are found by analyzing the differences in the test description files. Altogether we found 223 bugs for which the fixing change added or touched at least one test case.

3.6 Annotate Bugs with Size Properties

Some bugs may not meet the assumptions and prerequisites of a specific bug localization tool. For instance a tool may pinpoint to exactly one code location. In this case, bugs that span across several files would never be recognized completely by the tool and should be treated separately in the evaluation. In order to provide an efficient selection mechanism for bugs we annotate them with *size properties* (discussed in this subsection) and *syntactic properties* (discussed in the next subsection). When computing these properties, we ignore changes to test files and classes, since they are not part of the actual correction.

For each bug, we list *size properties* of the corresponding fix.

- *files-churned:* the number of program files changed
- *java-files-churned:* the number of JAVA files changed
- *classes-churned:* the number of classes changed
- *methods-churned:* the number of methods changed

For computing the size of a fix in terms of lines, we parse the *hunks* returned by the GNU diff command. A hunk corresponds to a region changed between two versions. If the region is present in both versions, the hunk is called a *modification*, otherwise it is an *addition* (region is only present in the post-fix version) or *deletion* (region is only present in the pre-fix version). We use the line ranges of a region to compute the size of a hunk. Since for modification hunks the size may differ between pre-fix and post-fix region, we take the maximum in this case. In order to get the actual size of a fix, *lines-churned*, we aggregate the sizes of the hunks; we additionally break down the size to additions, deletions, and modifications.

- *hunks:* the number of hunks in a fix.
- *lines-added:* the total number of lines added.
- *lines-deleted:* the total number of lines deleted.
- *lines-modified:* the total number of lines modified.
- *lines-churned:* the total number of lines changed, i.e., the sum of *lines-added*, *lines-deleted*, and *lines-modified.*

From the bug report we extract *priority* and *severity* of a bug and include them as properties in our dataset. The priority of a bug describes it importance and ranges typically from P1 *(most important)* to P5 *(least important).* In contrast the severity describes the impact and is one of the following: *blocker, critical, major, minor, trivial,* or *enhancement.* A severe bug may be have low priority when only few users are affected by a bug. However, in most cases bugs with high severity have also a high priority.

In addition to the above properties, we annotate bugs that produce exceptions with *tags*. We obtain this information by parsing the short description of a bug for keywords: *null pointer exceptions* typically are indicated by the keywords "NPE" or "Null", while *other exceptions* are indicated by "Exception".

$\boxed{\textbf{ASPECTJ:}}$ We have included size properties for all bugs in the iBugs repository in the description file *repository.xml.*

3.7 Annotate Bugs with Syntactic Properties

In addition to size properties, we provide *syntactic properties* of changes. This supports the retrieval of bugs that were fixed in a certain way, say by changing a (single) method call or expression.

In order to express how a fix changed the program, we use the APFEL tool [32]. APFEL builds the abstract syntax trees of the pre-fix and post-fix version, flattens the trees into token sets and computes the difference between these sets (see Figure 4).[3] APFEL supports different types of tokens for method calls, expressions, keywords, operators, exceptions handling, and variable usage. The type of the token is encoded in a single capital letter (see Table 3).

We use the differences computed by APFEL to create two fingerprints of a change at different levels of detail: The *concise fingerprint* summarizes the most essential syntactic changes such as method calls, expressions, keywords, and exception handling. In contrast the *full fingerprint* additionally records changes in variable names and contains more detailed information about the affected tokens.

- The ***concise fingerprint*** shows whether a bug (more precisely, its fix) is related to keywords (presence of the "K" character), method calls ("M"), exception handling ("H"), or expressions ("Z"). In contrast to the full fingerprint, the concise fingerprint omits variable usage, operators, and literals, i.e., it is a subsequence of "KMHZ".
- The ***full fingerprint*** additionally shows variable usage ("V" and "T"), operators ("O"), and literals ("Y"). Furthermore, it specializes keywords (null, true, false, etc.), expression (if, while, for, cast, etc.) and operators (+, −, &&, etc.).

[3]Note that APFEL is insensitive to the order of tokens because it relies on sets. This means that certain types of changes are missed such as swapping two lines.

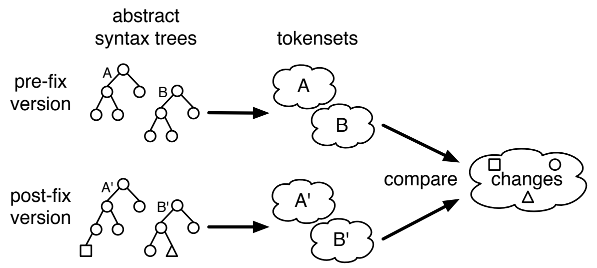

Figure 4: APFEL compares pre-fix and post-fix versions.

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 3: Token types in APFEL.

Figure 5 shows an example for a fix of a bug that caused a null pointer exception (NPE). The differences computed by APFEL show that a new *if* statement was inserted: several keywords (*if*, *null*, *else*, and *return*) and the operator *!-* were added exactly once; APFEL additionally reports the new usage of the variable *declaration*, its type *MethodDeclaration*, and the condition of the *if*-statement. For the concise fingerprint, we omit the variable, literal, and operator tokens and the names of the other tokens. This results in the fingerprint *"KZ"*, telling us that keyword(s) and expression(s) were changed. In contrast, the full fingerprint contains all tokens, but omits names, except for keywords and operators. In the example of Figure 5 it is *"K-else K-if K-null K-return O-!= T V Z-if"*.

We included fingerprints in our dataset to support researchers when retrieving a set of bugs that match certain syntactic properties. Say, a researcher is interested in bugs that are related to null pointer checks. In order to come up with a set of initial candidates, she can query for bugs containing *"K-null"* in their fingerprint.

$\boxed{\textbf{ASPECTJ:}}$ Fingerprints for all bugs in the iBugs repository are provided in the description file *repository.xml*.

3.8 Assemble IBugs repository

The iBugs repository may contain several hundreds of versions for a program. For a typical project the size of a checkout from the source repository can contain 50 MB or more of data. This yields

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

ASPECTJ	Size of code (latest revision)	75 kLOC
	Number of commits to CVS repository	7947
	Number of tests (latest revision)	1178
	Number of developers	13
	Number of bugs in iBugs repository	369
	Bugs with associated tests	223
	Size of iBugs repository	260 MB
	First bug report in iBugs repository	2002-07-03
	Last bug report in iBugs repository	2006-10-20

Table 4: Characteristics of the ASPECTJ dataset

a size of several gigabytes for the iBugs repository, which makes distribution difficult. We therefore create a new Subversion repository that stores the code for all versions. This greatly reduces the amount of space required to store the versions for the fixes included in the iBugs repository. Meta information about the fixes in the iBugs repository is stored in an xml file. For each bug we give information about the test suite, a pointer to the tests that were committed with the fix (if any), and the diffs for all files that were changed in the fix.

ASPECTJ: Snapshots of the project are approximately 60 MB in size. Although we have more than 700 versions (2 for each bug) in the iBugs repository, the resulting file size is only 260 MB. Figure 6 shows an excerpt of the description file *repository.xml*.

3.9 Summary

The ASPECTJ dataset provides 369 bugs for the evaluation of ***static*** *bug localization tools; 223 out of these have test cases associated and therefore can be used to evaluate* ***dynamic*** *bug localization tools.*

4. 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, the 13 developers regularly provide links to the bug database and include test cases in their commits.

4.1 Size of ASPECTJ

The ASPECTJ compiler consists of 75 kLOC and its test suite contains more than 1000 test cases. From its history we identified 369 bug reports that changed program code, for 223 we found associated test cases. The total size of the iBugs repository is 260MB (see Table 4).

```
<bug id="69459">
 <property name="files−churned" value="1"/>
 <property name="java−files−churned" value="1"/>
 <property name="classes−churned" value="1"/>
 <property name="methods−churned" value="1"/>
 <property name="hunks" value="3"/>
 <property name="lines−added" value="0"/>
 <property name="lines−deleted" value="0"/>
 <property name="lines−modified" value="11"/>
 <property name="lines−churned" value="11"/>
 <property name="priority" value="P3"/>
 <property name="severity" value="normal"/>
 <concisefingerprint>KMZ</concisefingerprint>
 <fullfingerprint>K−else K−if K−null M O−! O−&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; !child.isStatic()) {
...
   </file>
 </fixedFiles>
</bug>
```

Figure 6: XML content descriptor for bug 69459.

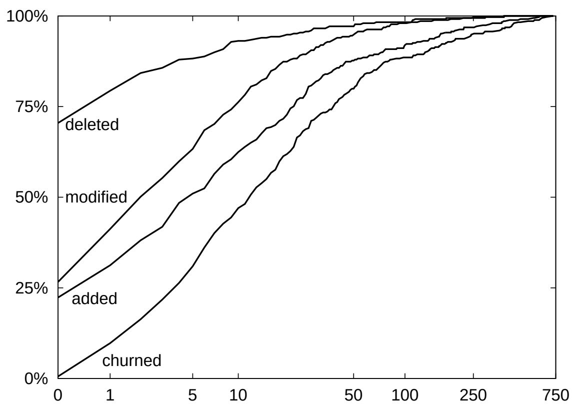

Figure 7: Most fixes churn only few lines of code.

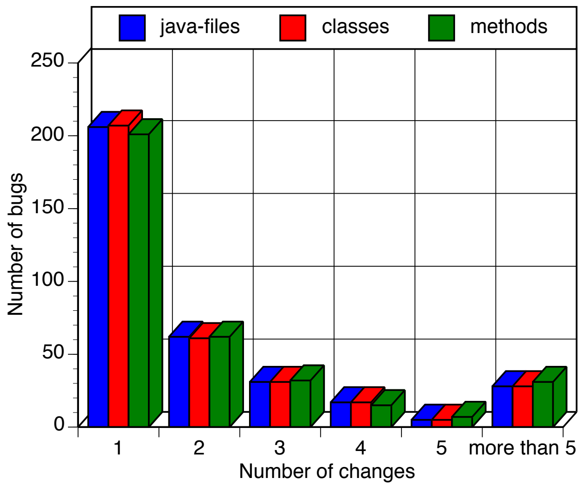

Figure 8: Most fixes affect only few files, classes, and methods.

Fingerprint	Small fixes	All fixes	Examples in Figure 11
empty	6	33	Bug 132130
HK		2	
HKM	1	4	
HKMZ		32	
K	5	10	Bug 151182
KM	7	24	Bug 43194
KMZ	13	192	Bug 67774
KZ	20	31	Bug 123695
M	12	18	Bug 80916
MZ	10	16	Bug 42539
Z	5	7	Bug 69011, Bug 161217
Total	79	369	

Table 5: Number of bugs per fingerprint in ASPECTJ.

4.2 Size of fixes

The histogram in Figure 8 shows the number of bugs that were fixed in one, two, three, four, five, or more than five Java files (blue, left bars), classes (red, middle bars), and methods (green, right bars), respectively. 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.

Figure 7 shows the distribution of churned lines of code. There are many small fixes for ASPECTJ, 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 for ASPECTJ are consistent with the ones observed by Purushothaman and Perry [18].

4.3 Fingerprints

In Table 5 we show the distribution of concise fingerprints for *small fixes* (i.e., five lines or less churned within one method) and *all fixes* of the ASPECTJ dataset. The most dominant fingerprint is "KMZ" indicating that most fixes are of complex nature. Several fixes change only literals and variable names and therefore have an empty fingerprint. Exception handling (fingerprint with substring "H") is exclusive to larger fixes, likely, because adding the skeleton of *try/catch* already takes four lines.

Simple fingerprints are most dominant for small fixes: 12 fixes changed only method calls ("M"), 5 fixes changed only keywords ("K"), and 5 fixes changed only expressions. The fingerprint "KZ" typically points to the addition of null pointer checks, that consist of a keyword (either *if* or *null* and an expression that checks for *null*).

We inspected all small fixes and observed mainly three categories: (1) fixes that change expressions, mostly checks for null pointers (presence of "Z" in the fingerprint), (2) fixes that add or change method calls (presence of "M" and absence of "Z"), and (3) other fixes (for instance empty fingerprint). In future work we plan to classify fixes automatically. For examples of fingerprints and characteristic fixes, we refer to Figure 11.

5. CASE STUDIES

We have conducted two case studies to verify that the ASPECTJ dataset can indeed be used to easily evaluate both static and dynamic bug localization tools.

5.1 FindBugs

FINDBUGS [9] is a static bug pattern detection tool for JAVA. The current version has a catalogue of 183 bug patterns, which are organized in categories like *Correctness, Bad Practice,* and *Performance*. FINDBUGS takes a set of jar files as input and looks for instances of the patterns in its catalogue. It outputs the number of bugs found for each class that was analyzed.

5.1.1 Experimental Setup

For our experiments we used the command-line client of FINDBUGS 1.1.3. We ran FINDBUGS using the default configuration on each of the 369 bugs in the iBugs repository.

5.1.2 Running the Experiment

Running the FINDBUGS experiment requires only two steps. First we checkout and build the buggy version for each build using the scripts provided with the iBugs repository (see Steps 2 and 3 of Figure 10). Then we run FINDBUGS on each version that we built and store its output.

5.1.3 Results

The aim of our study was to investigate if FINDBUGS reports warnings or errors close to the actual location of the fix for the bug. We therefore used a simple approach to evaluate FINDBUGS precision: we consider a bug caught by FINDBUGS if at least one error was reported in a class that was changed when the bug was fixed.

Although FINDBUGS reported many warnings and errors, we found no case where a potential bug was reported in a class that was changed when the bug was fixed. This may be due to the fact that the ASPECTJ developers use ECLIPSE, which provides many static checks of the code that look for similar types of bugs like FINDBUGS. Another reason might be that the bugs in the iBugs repository are too complex and FINDBUGS warnings are better suited to point to less complex bugs.

5.2 Ample

Our second case study evaluates AMPLE [4], a dynamic bug localization tool for JAVA. AMPLE works on a hypothesis first stated by Tom Reps et al. [20]: 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 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.

In previous work [4] we have evaluated AMPLE using NANOXML, one of the subjects from the Software-Artifact Infrastructure Repository and four bugs from the ASPECTJ compiler. Back then finding those four bugs required manually searching the bug database and source repository of ASPECTJ. By using our iBugs repository we were able to repeat our evaluation with a much larger number of bugs in much less time.

5.2.1 Experimental setup

We used the same experimental setup as in our previous evaluation [4]: for each bug we compare the sequences of one failing run and one or more passing runs. We use the tests that were committed together with the fix as failing runs and randomly choose three passing tests from the regression test suite. In order to be able to use the same evaluation method, we restrict our experiments to bugs that were fixed in a single class. We use a value of 5 for the length of call sequences as this produced the best results in our previous evaluation.

5.2.2 Running the Experiment

With the help of the meta information in the iBugs repository we easily identified 74 bugs that fix problems in one class of the compiler. We then wrote a small JAVA program (142 lines of code) that calls the scripts provided by the iBugs repository to build and run the passing and failing tests. As mentioned in Section (3.5), not all test cases committed together with a fix actually fail. An evaluation of the test output revealed that this was the case for 30 bugs. We removed those bugs and restricted our evaluation to the remaining 44 bugs.

5.2.3 Results

For each bug we get a ranking of all classes that were executed during the failing run. This ranking is a recommendation in which order a programmer should search the classes when looking for the bug. We express the quality of a ranking as the *search length*: the number of classes that are ranked higher than the class where the bug was fixed. A low search length means that a programmer has to check only a small portion of the code before he gets to the buggy class.

Figure 9 shows a cumulative plot of the relative search length for all 44 bugs. The plot shows that if a developer is willing to investigate the top 10% of the ranked classes, she would have found 40% of all bugs. A comparison to the results of our previous evaluation shows that AMPLE works well in many cases but also produces bad results in some. We now have a much broader range of bugs and can take a closer look at what types of bugs produce bad rankings. This is a good example of how the data in the iBugs repository can help improve existing and upcoming bug localization approaches.

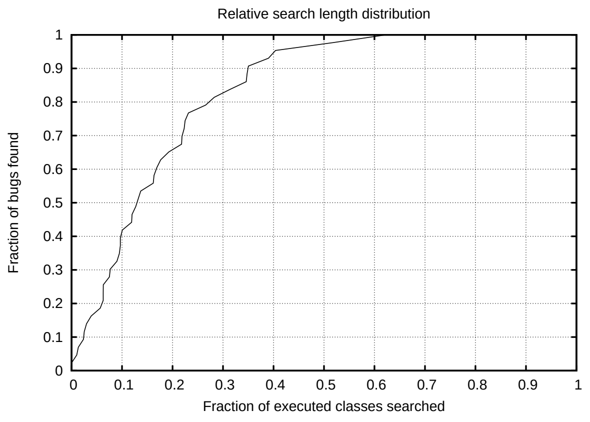

Figure 9: 40% of all bugs are found by searching at most 10% of the executed code.

5.3 Running your own evaluation

The prerequisites for using the iBugs repository are low: you need a copy of Ant (a popular build tool available at `ant.apache.org`) and the Java Development Kit, version 1.4. We tested iBugs under Linux, Mac OS X, and Microsoft Windows XP.

The case studies from Section 5 illustrate that it is easy to use the iBugs repository to evaluate both static and dynamic bug localization tools. Figure 10 contains a step-by-step guide how to conduct an evaluation. We encourage other researchers to evaluate their bug localization tools by using the iBugs repository and our guide.

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 (useful for static tools), 223 out of these had associated test cases useful for dynamic tools). We demonstrated the relevance of our dataset with case studies on FINDBUGS (a static tool) and AMPLE (a dynamic tool). To summarize, our contributions are as follows:

Automatic extraction. Our iBugs approach automatically extracts benchmarks for bug localization from the history of a project (using its version archive and bug database).

Realistic bugs. The bugs collected by iBugs are real bugs as they occur in real projects. Therefore, results obtained by using our ASPECTJ benchmark are more likely to transfer to real projects.

Publicly available. Our benchmark is publicly available (see below). In addition to the bugs themselves, we provide a fully-fledged infrastructure for reconstructing, building, and testing the versions with and without bugs (see the step-by-step guide in Figure 10).

The ASPECTJ dataset is a first step towards the "huge collection of software defects" that was demanded by Spacco et al. [24] 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 fur iBugs will consist of the following:

Extend the repository. Obviously, we plan to add more subjects to the iBugs repository. In particular, we want to add projects that are multi-threaded and provide a graphical user interface.

Classification of bugs. Our tags and fingerprint provide an initial classification of bugs. We plan to further improve this classification by using automated techniques from data mining. This will greatly improve the value of our datasets, because researchers can test for which kinds of bugs their tools perform best.

Score measure. In order to measure the success of bug localization tools, Renieris and Reiss introduced a *score* [19] that indicates the fraction of the code that can be ignored when searching for a bug. In future releases of our dataset, we want to provide a tool that computes this score. This will hopefully unify the assessment of results.

To support ongoing work on bug localization, we made iBugs publicly accessible, including the underlying infrastructure. For ongoing information on the project, log on to

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

Acknowledgments. Thomas Zimmermann is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". David A. Wheeler made the SLOCCount tool available. David Schuler provided valuable comments on earlier revisions of this paper.

7. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM Press.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of 27th International Conference on Software Engineering (ICSE)*, pages 342–351. ACM, 2005.

[3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In A. P. Black, editor, *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 528–550. Springer, 2005.

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

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[7] E. Gamma. JUnit 3.8.1 GPL, 2007.

1. **Select bugs.** Use the meta information provided in the file *repository.xml* to select relevant bugs.

   Example: In order to select all bugs that raised a NullPointerException, use the XPath [28] 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 10: Step-by-step guide to your own evaluation.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[10] InfoEther. PMD. http://pmd.sourceforge.net/.

[11] K. Knizhnik and C. Artho. Jlint–Find bugs in Java programs. http://jlint.sourceforge.net/.

[12] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1-2):41–84, 2005.

[13] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM Press.

[14] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM Press.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[16] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 286–295, New York, NY, USA, 2005. ACM Press.

[17] 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.

[18] 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.

[19] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39. IEEE Computer Society, 2003.

[20] 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 M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, Sept. 1997.

[21] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, 1998.

[22] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[23] 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.

[24] 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.

[25] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: an automated programming project snapshot and testing system. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press. See also: http://marmoset.cs.umd.edu/.

[26] C. Williams and J. K. Hollingsworth. Bug driven bug finders. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 70–74, Edinburgh, Scotland, UK, May 2004.

[27] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[28] World Wide Web Consortium. "XML Path Language (XPath)". http://www.w3c.org/TR/xpath/.

[29] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, 2003.

[30] 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, New York, NY, USA, 2006. ACM Press.

[31] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM Press.

[32] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, New York, NY, USA, October 2006. ACM Press.

[33] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Edinburgh, Scotland, UK, May 2004.

```
    for (int i = types.length - 1; i >= 0; i--) {
-   if (typePattern.matchesExactly(types[i])) return true;
+   if (typePattern.matchesStatically(types[i])) return true;
    }
    return false;
```

Bug 42539: “throw derivative pointcuts not advised.”

Figerprint: M Z-if

```
    ResolvedTypeX[] parameterTypes = searchStart.getWorld().resolve(...);

-   arguments = arguments.resolveReferences(bindings);
+   TypePatternList arguments = this.arguments.resolveReferences(bindings);

    IntMap newBindings = new IntMap();
```

Bug 43194: “java.lang.VerifyError in generated code”

Fingerprint: K-this M

```
    if (getKind().isEnclosingKind()) {
        return getSignature();
+   } else if (getKind() == Shadow.PreInitialization) {
+   // PreInit doesn't enclose code but its signature
+   // is correctly the signature of the ctor.
+   return getSignature();
    } else if (enclosingShadow == null) {
        return getEnclosingMethod().getMemberView();
```

Bug 67774: “Nullpointer-exception in pointcuts using withincode() clause”

Fingerprint: K-else K-if K-return M O-== Z-if

```
    String packageName = StructureUtil.getPackageDeclarationFromFile(inputFile);

-   if (packageName != null ) {
+   if (packageName != null && packageName != "") {
        writer.println( "package␣" + packageName + ";" );
    }
```

Bug 69011: “ajdoc fails when using default package”

Fingerprint: O-!= O-&& T V Y Z-if

```
    if (shadow.getSourceLocation() == null
        || checker.getSourceLocation() == null) return;

+   // Ensure a node for the target exists
+   IProgramElement targetNode = getNode(...);
+
    String sourceHandle = ProgramElement.createHandleIdentifier(
        checker.getSourceLocation().getSourceFile(),
```

Bug 80916: “In some cases the structure model doesn’t contain the ”matches declare” relationship”

Fingerprint: M T V

```
    // matched by the typePattern.
    ResolvedType[] annTypes = annotated.getAnnotationTypes();
-   if (annTypes.length!=0) {
+   if (annTypes!=null && annTypes.length!=0) {
        for (int i = 0; i < annTypes.length; i++) {
```

Bug 123695: “Internal nullptr exception with complex declare annotation statement that affects injected methods”

Fingerprint: K-null O-!= O-&& T V Z-if

```
            }
        }
-   if (it.hasNext()) sb.append(",␣");
+   if (it.hasNext()) sb.append(",");
    }
    sb.append(')');
```

Bug 132130: “Missing relationship for declare @method when annotating a co-located method”

Fingerprint: Y

```
    try {
+   synchronized (loader) {
            WeavingAdaptor weavingAdaptor = WeaverContainer.getWeaver(...);
            if (weavingAdaptor == null) {
                if (trace.isTraceEnabled()) trace.exit("preProcess",bytes);
                return bytes;
            }
            return weavingAdaptor.weaveClass(className, bytes);
+   }
    } catch (Exception t) {
        trace.error("preProcess",t);
```

Bug 151182: “NPE in BcelWeaver using LTW”

Fingerprint: K-synchronized T V

```
    // at the moment it only deals with 'declared exception is not thrown'
    if (!shadow.getWorld().isIgnoringUnusedDeclaredThrownException()
-   && !thrownExceptions.isEmpty()) {
+   && !getThrownExceptions().isEmpty()) {
        Member member = shadow.getSignature();
        if (member instanceof BcelMethod) {
```

Bug 161217: “NPE in BcelAdvice”

Fingerprint: Z-if

Figure 11: Examples for different bugs with fingerprints.