Scalable Isolation of Failure-Inducing Changes via Version Comparison

Mohammadreza Ghanavati, Artur Andrzejak, Zhen Dong
Institute of Computer Science
Heidelberg University, Germany
{mohammadreza.ghanavati, artur.andrzejak, zhen.dong}@informatik.uni-heidelberg.de

Abstract—Despite of indisputable progress, automated debugging methods still face difficulties in terms of scalability and runtime efficiency. To reach large-scale projects, we propose an approach which reports small sets of suspicious code changes. Its essential strength is that size of these reports is proportional to the amount of changes between code commits, and not the total project size. In our method we combine version comparison and information on failed tests with static and dynamic analysis.

We evaluate our method on real bugs from Apache Hadoop, an open source project with over 2 million LOC\(^1\). In 2 out of 4 cases, the set of suspects produced by our approach contains exactly the location of the defective code (and no false positives). Another defect could be pinpointed by small approach extensions. Moreover, the time overhead of our approach is moderate, namely 3-4 times the duration of a failed software test.

Index Terms—Automated debugging, version comparison, failure-inducing changes, thin slicing, large-scale projects, software tests

I. INTRODUCTION

Debugging is an expensive and time-consuming task in the software development process. According to studies, half of the programming time of the developers is dedicated to investigate and correct bugs. The total cost of testing and debugging of the software development can easily range from 50 to 75 percent of the total development cost [22]. For these reasons, automated debugging has attracted a great deal of interest.

Techniques of automated debugging attempt to find the causes of a program failure without or with only minimal human involvement. In practical terms, after analysis of data obtained from testing results and code instrumentation, a programmer is supplied by a ranked list of suspicious code locations. By examining these locations, the programmer can possibly identify the code fragment bearing the true defect.

Despite of indisputable recent advances ([6], [8], [3], [1], [15], [4]), automated debugging is still facing significant challenges preventing its widespread adoption [13]. One of the essential ones is that while excelling at fault localization, they usually do a poor job in facilitating fault understanding. However, even knowing a (potential) fault location still requires the developer to find out what could happen there - a cognitively demanding task.

The second weakness of automated debugging is its limited scalability in terms of program size. Here even pinpointing the 0.1% of code which might contain the defect is not precise enough. Such level of specificity means that on a project with 100k Lines of Code (LOC), a developer still needs to examine 100 suspicious lines to find the defect. This is beyond the typical acceptance levels of programmers (most users do not inspect search results after the 1st page [13]), greatly reducing the utility of such techniques in large-scale projects.

We attempt to approach the latter problem by narrowing our focus to scenarios where software is developed through a series of minor modifications, with each intermediate version being tested thoroughly. This is a typical way how medium-size and large-size projects are developed, for example in a setup of continuous integration testing [22]. Consequently, this assumption is not a serious limitation, especially since our technique targets large-size projects.

Our basic idea is to use version comparison to localize newly introduced defects in the latest development version. Version comparison contrasts the behavior of two software versions under the same unit and/or integration tests. In more detail, using static and dynamic analysis we compare the sets of statements executed in the latest (faulty) version against those executed in a previous (base) version. We assume here that the faulty new version has caused a unit/integration test to fail, while the same test succeeded on the base version.

The key advantages of our approach is its precision and efficiency. Assuming that only the recently changed code contains the defect (this is not always but usually true), we can reduce the set of suspicious statements to a few LOC, see Section III. The good news here is that size of this set depends primarily on the size of changes (i.e. amount of differences between commits) and not on the total project size. Assuming frequent commits and test runs, our technique is thus likely to scale and present the developers a small set of suspicious statements - even for very large projects.

Secondly, we show in Section III-C that the overhead introduced by our approach is moderate. The essential additional execution cost comes from the requirement to execute a (crashing) test on each of the instrumented base and the instrumented faulty version. Due to very sparse instrumentation, the execution time of the instrumented versions is almost identical to the original versions. As shown in Table IV, the total time of executing our approach is about 2.5-3.5 times the duration of the failed test.

Finally, while not evaluated in this work, our approach is

---

\(^1\)On September 14, 2013, Ohloh (http://www.ohloh.net/p/Hadoop) was reporting that Apache Hadoop has 2,280,391 lines of code.
potentially capable to contribute to failure understanding. To
this aim, one can present the developer the contents of the
relevant variables as well as execution paths in both base
and faulty versions. While this requires future work, it is a
promising and low-overhead way to supporting a developer
in making sense what has changed during the execution of
suspicious code.

In summary, this paper makes the following contributions:
• We propose and implement an approach to isolate test-
  failing defects which is based on comparing subsequent
  versions of software under development. It combines
  static analysis (backward slicing) and information on
  changed code to indicate which (small) code sections
  should be be instrumented. A subsequent dynamic anal-
  ysis (code coverage of a failing and passing version)
  reveals the statements which are likely to contain the
defects (Section II).
• We evaluate our approach on real defects from a large-
scale software project, namely Apache Hadoop (Sec-
tion III). The results show that it can pinpoint the de-
fective statements with high precision.
• We also compare the execution overhead of the images
  instrumented according to our approach against alter-
  native instrumentation schemes. Our results show that
  the overhead of the dynamic analysis proposed by us is
  negligible (Section III-C).

This paper is organized as follows: The approach and its
implementation is described in detail in Section II. The re-
result of the experiments and threats to validity are presented in
Section III. Section IV considers the related work, and finally
Section V concludes the paper.

II. THE VERSION COMPARISON APPROACH

This section describes the details of our approach. As indi-
cated in Section I, we assume that software under development
evolves as a series of versions, each one checked by executing
one or more test suites. We trigger our automated debugging
approach on the event that some test T has failed while
executing the latest software version. We denote this latest
version as \( v_f \) and call it a failing version. We also retrieve from
repository some previous software version \( v_p \), (called passing
version) which passed \( T \) successfully; usually it is a version
directly preceding \( v_f \).

A. Approach overview

The steps of our approach are explained below and illus-
trated in Algorithm 1.

As first we retrieve the set of changes between \( v_p \) and \( v_f \),
i.e. \( \text{Dif} = \Delta(v_p, v_f) \) and call it a difference set. This is
done with tools provided with popular software configuration
management systems like SVN, Git, CSV.

As a consequence of the failure of \( T \) on \( v_f \), the JVM (or
operating system) provides a stack trace which is analyzed
by our approach. We call the code location referenced by the
top-level entry within this trace (yet not devoted to exception
handling) a failure manifestation site or just failure site \( f_{site} \).

Algorithm 1 Steps of the proposed approach

**Step 1:** Find the differences of versions \( v_p \) and \( v_f \):
\[
\text{Dif} = \Delta(v_p, v_f)
\]

**Step 2:** Retrieve failure site \( f_{site} \) from the stack trace

**Step 3:** Compute backward slices for each version:
\[
\text{BSlice}_p = \text{BackwardSlice}(v_p, f_{site}),
\text{BSlice}_f = \text{BackwardSlice}(v_f, f_{site})
\]

**Step 4:** Compute the intersections \( IS_x = \text{BSlice}_x \cap \text{Dif} \) and
instrument versions:
\[
\text{cov}_p = \text{Instrument}(v_p, IS_p),
\text{cov}_f = \text{Instrument}(v_f, IS_f)
\]

**Step 5:** Execute test \( T \) on each \( v_p, \text{inst} \) and \( v_f, \text{inst} \) to get code
coverage profiles:
\[
\text{cov}_p = \text{Run}(v_p, \text{inst}),
\text{cov}_f = \text{Run}(v_f, \text{inst})
\]

**Step 6:** Get list of suspects by applying filtering lemmas:
\[
\text{SuspectSet} = \text{FilteringLemmas}(\text{cov}_p, \text{cov}_f)
\]

We use \( f_{site} \) as seed to compute (for each version \( v_p, v_f \)) the
backward slice \( \text{BSlice}_p, \text{BSlice}_f \) \( [18], [16] \). Essentially, it is
the set of code statements which could have affected variable
values at the failure site.

Subsequently, we compute (for each version \( v_p, v_f \)) the
intersection \( IS_x = \text{BSlice}_x \cap \text{Dif} \) and the
difference set \( \text{Dif} \) as \( IS_x = \text{Dif} \cap \text{BSlice}_x \ (x \in \{p, f\}) \).
This intersection gives us statements and method names which
are likely to contain the defect. In the next step, we instrument
the function calls within this intersection \( IS_x \) for both passing
\( v_p \) and failing \( v_f \) version.

In the next step we re-execute the test \( T \) on the both
instrumented versions of \( v_p \) and \( v_f \) and record coverage
information. The results are coverage profiles \( \text{cov}_p \) and \( \text{cov}_f \),
i.e. reports which code has been executed in the respective
version.

The last step involves comparison and filtering of the
coverage profiles using the following lemmas.

Filtering lemmas

The following statements are included in the set of suspects:
1) All statements added to or changed in \( v_f \) which are in
   the failing coverage profile \( \text{cov}_f \).
2) All statements deleted from \( v_p \) which are in the passing
   coverage profile \( \text{cov}_p \).

Finally, the resulting list of suspicious statements (suspects)
together with their locations is reported to the developer as
the potential root causes of a test failure.
We exemplify our approach on a real defect from the Apache Hadoop project, see Section III-A1.

B. Further aspects

In the following we discuss some secondary aspects of our approach.

As mentioned in Section II-A, we examine the stack trace of the test execution on $v_f$ to retrieve the failure site. However, usually we cannot take the top-most entry of the stack trace as this frequently points to logger code (or some exception-handling code). Therefore we use a heuristic and check the stack trace entries (from the topmost one) whether they point to “interesting” code, i.e. non-library and functional code. Figure 1 shows an example of the stack trace for issue Hadoop-3856 (Section III-A1), where the 2nd topmost entry points to assumed failure site.

The other aspect which requires explanation is slicing. For a given program $P$ and statement $s$ with a variable $v$ at the program location $\ell$, a backward slice computed from $s$ contains all of the statements in $P$ which can affect the value of $v$ [18]. It is obvious that if $\ell$ is the failure site, a backward slice includes all of the statements which might be the root cause of the failure at $\ell$.

However using traditional full slicing [18] results in a too large size of slice specially in real large-scale applications. To address this issue, we use thin slicing [16] which only contains statements that directly affect the value of the seed statement.

C. Implementation

Our approach is implemented on the top of WALA [17] which is a static analyzer developed by IBM. Slicing and instrumentation are the features of WALA which making it useful for our approach. Implementation of our approach is designed in two parts: static analysis and dynamic analysis. Finding version differences and computing backward slice are lied in the static analysis section. Slicing is implemented in WALA. For each application, WALA builds the corresponding call graph. Next it computes backward slice using this call graph and the failure manifestation point as a seed statement. For compatibility with our needs, we have modified some parts of WALA source code.

Dynamic profiling in our approach is implemented by Shrike which is a third-party library for instrumenting Java byte-code which is connected to WALA. It instruments the output of the static analysis by using the predefined instrumentation schema. Due to flexibility of instrumentation, we can easily exclude instrumenting of some parts of the code which is not necessary, e.g. Java libraries.

III. Experimental Evaluation

Our evaluation tries to answer the following research questions:

RQ1. How accurate is our approach to locate failure-inducing changes?

Here we want to evaluate whether the approach can find the true defect location (sensitivity), and how many false positives are contained in the final report (specificity). We show in Section III-A the results for 4 test cases used in this study (Table I). They demonstrate that in 2 of our 4 test cases we could find the correct root cause of the failure without false positives. Analysis of another case indicates that by small extensions of our method the defect location could be narrowed to 20 LOC. We also discuss the real bug fixes of each issue as stated in the Hadoop bug log.

RQ2. Are there any alternatives to our approach which are simpler yet have comparable accuracy?

This question targets the practicability of implementing our approach, and tries to answer whether a more simple variant of the method could yield similar results. Brief analysis in Section III-B indicates that all of the steps shown in Algorithm 1 are necessary to achieve this level of specificity.

RQ3. What is the time overhead of our approach, and how does it compare to alternatives?

We have collected for each of the test cases execution times and code size in various phases of our approach (Section III-C). This data is used to evaluate the overhead by two performance metrics, runtime slowdown and size overhead (Table III) and to compare them against alternative approaches. We also show the total time of our method (Table IV).

A. Experiment setup and case studies

All our experiments were run on a 2.9 GHz Intel Dual Core laptop with 8 GB physical memory (4 GB was allocated for the JVM), running Linux.

In this section we try to answer question RQ1. To this aim we use the Apache Hadoop as a real-world, complex and large-scale project. It deploys test cases and frequent versioning which fits our requirements. We use real bugs from Hadoop issue tracking\(^2\) system between 15th August 2012 and 27th July 2013. The selection of defects was done according to the following criteria:

1) The failure should be caused by a Hadoop component and manifest via a Hadoop test case. In other words, we do not consider library issues or other artifacts.

\(^2\)http://hadoop.apache.org/issue_tracking.html
In class: org.apache.hadoop.hdfs.server.namenode.NameNode

```java
@@ -511,9 +511,13 @@
 private void startTrashEmptier(Configuration conf) throws IOException {
     - long trashInterval = conf.getLong(
             - CommonConfigurationKeys.FS_TRASH_INTERVAL_KEY,
             - CommonConfigurationKeys.FS_TRASH_INTERVAL_DEFAULT);
     + long trashInterval = namesystem.getServerDefaults()
     + .getTrashInterval();
     if (trashInterval == 0) {
         return;
     } else if (trashInterval < 0) {
 ...
```

Figure 2. Excerpt of code changes for issue Hadoop-8689 (simplified)

2) The bug should be well documented, and it should be clear which update or patch caused the test to fail. We require this information to validate the result of our approach.

3) There should exist a passing version, i.e. a (previous) version which passed the test which failed in a subsequent (failed) version.

1) Issue HDFS-3856: The first issue is an attempted solution to a new feature request HADOOP-8689. We explain it in more detail as a showcase for the approach. Figure 2 shows a subset of changes making up the (erroneous) solution. A part of this patch provides separate trash cleanup intervals (fs.trash.interval) for client side versus the server side. However, this change causes the test TestHDFSServerPorts to fail due to new call getServerDefaults() in the startTrashEmptier() function. The failure of this test is reported in Hadoop bug HDFS-3856.

In Step 1 of our approach (Algorithm 1) we retrieve a passing \(v_p\) as well as a failing \(v_f\) code version (passing or failing for test TestHDFSServerPorts). As shown in Table II, the difference set \(D\) between \(v_p\) and \(v_f\) is very large, amounting to more than 109 KLOC.

Step 2 needs the failure stack trace of the test TestHDFSServerPorts to identify the failure site. As shown in Figure 1, code location FSNamesystem.java:759 is the assumed failure site.

In Steps 3 and 4 we obtain the respective backward slices for \(v_p\) and \(v_f\) (922 and 759 JVM-bytecode statements), instrument the intersections \(I_S_p\) and \(I_S_f\) of \(D\) and backward slices (544 and 445 statements). In the subsequent Step 5 we obtain the code coverage of executing TestHDFSServerPorts on each of the two instrumented versions \(v_p,\text{inst}\) and \(v_f,\text{inst}\) (sizes 189 and 166 statements).

Finally, we can apply the filtering lemmas in Step 6. They yield the final report which is shown in Figure 3. In the future, we will present to the developer not only this report but also the differences of code coverage to support failure understanding.

![Figure 3. Final report for using our approach for Hadoop-3856](image)

Suspicious failure-inducing changes:

In class: org.apache.hadoop.hdfs.server.namenode.NameNode:514

```java
long trashInterval =
    namesystem.getServerDefaults().getTrashInterval();
```

We also compared our suspect against the fix of this problem, which is committed in SVN revision 1377934. We found that indeed our hypothesis was correct. In the fix, the previously added faulty change is replaced with a new statement (shown in bold):
passing the conditional statements with diverging condition values.

Even assuming that the while-statement would be included in the final report, it is not the precise root cause of the failure. However, this statement is within the method run() in the inner class of ReplicationMonitor in the BlockManager class. The method run() (about 20 LOC) indeed contains the defect. Thus, by small extension we can at least indicate the method with the actual defect.

An investigation of the actual fix which is committed in SVN revision 1501841 shows that (because the defect is outside the changes), merely deleting the added changes from the new version does not solve the problem. In fact, to fix this bug, a check statement (shown in bold in following) was added to the conditional branch in the code:

```java
if (!namesystem.isRunning()) {
    ...
    break;
} else if (isNSRunning && t instanceof InterruptedException) {
    LOG.info("Stopping ReplicationMonitor for testing.");
    break;
} else {
    LOG.fatal("ReplicationMonitor thread received runtime exception. ", t);
    terminate(t, t);
}
```

3) Issue HDFS-4282: Bug issue HDFS-4282 reports the failing of TestEditLog.testFuzzSequence test. Comments in the issue show that this test is broken by HADOOP-9103. Due to errors in decoding Unicode characters, in HADOOP-9103 a patch is created which has modified code of UTF8 class in Hadoop Common project. However, this patch caused a failure of TESTEditLog test.

After applying our approach to this test case, the results point that the changes created in HADOOP-9103 are not faulty. However, the only difference in the execution profiles of passing and failing runs is a call of the method toString() in UTF8 class of the io package of Hadoop Common project. Also here Table II (bottom rows) give the sizes of code over all phases of our approach.

By investigating the execution profiles, we created a hypothesis that a solution to this bug is to add methods related to the toString() function. The real fix to this bug (committed in SVN revision 1418214) confirms our hypothesis. A new method toStringChecked() (with IOException) is added to the UTF8 class which now can throw an IOException for invalid UTF8 characters4.

4) Issue Yarn-960: Bug reported in issue Yarn-960 manifests in failure of tests TestBinaryTokens and TestMRCredentials. The reported reasons are changes committed for issue Yarn-701. Unfortunately, when applying our approach to this bug, we can not find the root cause of the error.

As we mentioned in the Section II (Step 2 in Algorithm 1), our approach needs a failure manifestation site in the failing version. However, in the stack trace of the failing executions

---

### Table II

<table>
<thead>
<tr>
<th>Issue</th>
<th>Version</th>
<th>Size</th>
<th>BSlice</th>
<th>IS</th>
<th>Cov</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS-3856</td>
<td>passing</td>
<td>109207</td>
<td>922</td>
<td>544</td>
<td>189</td>
<td>1</td>
</tr>
<tr>
<td>HDFS-3887</td>
<td>passing</td>
<td>1030</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>passing</td>
<td>1325</td>
<td>795</td>
<td>367</td>
<td>88</td>
<td>1</td>
</tr>
</tbody>
</table>

---

4. [Link](https://issues.apache.org/jira/browse/HDFS-4282)
Table III
OVERHEADS OF OUR APPROACH COMPARED TO FULL INSTRUMENTATION (FULL INSTR.) AND INSTRUMENTING ONLY THE CODE IN BSlice (BSlice instr.); IN “X/Y”, X IS THE RUN-TIME SLOWDOWN (A FACTOR) AND Y SIZE OVERHEAD OF INSTRUMENTING (A FACTOR); P = PASSING VERSION, F = FAILING VERSION, “-” = INSTRUMENTATION NOT POSSIBLE

<table>
<thead>
<tr>
<th>Issue</th>
<th>Ver.</th>
<th>Original version</th>
<th>Full instr.</th>
<th>BSlice instr.</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run time (s)</td>
<td>Size (MB)</td>
<td>Run time (s)</td>
<td>Size (MB)</td>
<td>Run time (s)</td>
</tr>
<tr>
<td>HDFS-3856</td>
<td>p</td>
<td>6</td>
<td>4.8</td>
<td>- / 4.00</td>
<td>1.20 / 1.04</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>6</td>
<td>4.1</td>
<td>- / 3.40</td>
<td>1.00 / 1.02</td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>p</td>
<td>9</td>
<td>4.8</td>
<td>1.00 / 4.00</td>
<td>1.00 / 1.00</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>9</td>
<td>4.8</td>
<td>1.00 / 4.00</td>
<td>1.00 / 1.00</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>p</td>
<td>30</td>
<td>4.4</td>
<td>3.00 / 4.00</td>
<td>1.00 / 1.00</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>30</td>
<td>4.4</td>
<td>3.00 / 4.00</td>
<td>1.00 / 1.00</td>
</tr>
</tbody>
</table>

Table IV
RUNNING TIMES OF A FAILING TEST AND TIMES FOR VARIOUS PHASES OF OUR APPROACH (TIMES IN SECONDS); TOTAL / TEST IS THE RATIO OF TOTAL APPROACH TIME TO TEST TIME

<table>
<thead>
<tr>
<th>Issue</th>
<th>App. time for passing + failing version</th>
<th>Test time</th>
<th>Total / Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slicing</td>
<td>Instr.</td>
<td>Run</td>
</tr>
<tr>
<td>HDFS-3856</td>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>4</td>
<td>6</td>
<td>64</td>
</tr>
</tbody>
</table>

Even so, our approach beats the alternatives, having negligible overheads due to instrumentation.

Even more interesting is Table IV which contrasts the running time of the failed test (Test time) against the total time needed to execute our approach (Total). As shown in the column Total / Test, the total time of our approach requires at most 3.3 times the duration of the failed test, and the latter is just one of many tests executed within a test suite.

D. Threats to Validity

Like in any empirical study, there are some threats to validity in our evaluation. First, we use only four bugs. This is not a sufficient sample size to draw general conclusions, especially in respect to applications. Note that the reason of using a small set of bugs in this study is that finding the real bugs and reproducing them is a time-consuming job.

Secondly, we focus on finding failure-inducing changes which are caused by changes in the faulty version \(v_f\). However, it is be possible that the actual bug has been introduced already in an older version \(v_p\) or before) but manifests only with changes yielding \(v_f\).

IV. RELATED WORK

Recently automated debugging has attracted a great deal of interest. Here, we focus on works which are closely related to our approach.

With increasing size of software, debugging software versions before committing them to the repository is an important step. Approaches in [14] and [2] generate an alternative input (which differs from failing input in the control flow behavior) and then compare their executions to find the root cause of the failure. [2] combines slicing and symbolic execution to locate the faulty code in a modified version of a program. However, these approaches incur large overheads.

One other interesting approach closely related to our work is delta debugging introduced in [20] and extended in [21]. These methods have significant limitations, namely high false positive ratio and large amount of tests. This encouraged others to improve on delta debugging, see [12] and [19]. Delta debugging differs from our version comparison as the former narrows down the search space (to locate suspicious changes) gradually, by applying changes to the application iteratively. This requires lot of test repetitions, creating large time overhead.

Program slicing is another debugging technique which is introduced in [18]. Due to large size of the slice, different approaches ([11], [7], [23], [24]) have been proposed to reduce the size of slice efficiently. Despite of these improvements, the sizes of slices are still fairly large in real-world applications. To address this issue, thin slicing has proposed in [16]. We use this technique for computing the backward slice in our approach (see Section II-B).

In [5], program slicing is used in combination with delta debugging to narrow down the search space of failure. While [5] is focused on the changes in the input, our approach concentrates on the changes in the source code.

Another approach in debugging is spectrum-based techniques which collect the execution information of faulty and correct runs and compare them to find the root cause of the failure [6], [4]. This technique differs from our approach in that we use just one failing and one passing run.

Statistical techniques are used in [9], [10] and [3] for bug isolation. Statistical debugging analyzes a large amount of executions gathered from running the instrumented application to rank the suspicious predicates which are highly relevant to the bug. Contrary to our approach, the requirements on a large amount of executions create significant time and resource overheads.

V. CONCLUSIONS AND FUTURE WORK

We have presented a scalable approach to isolating failure-inducing changes which exploits version differences together with static and dynamic code analysis. Our method has two essential strengths. First, the size of the set of suspicious statements is proportional to the size of recent code changes, making it potentially applicable to very large projects. Secondly, the additional runtime overhead is on the order of executing a test triggering bug search. This allows for integrating our approach in a traditional testing process in order to enhance test outcomes with locations of potential defects. Our preliminary evaluation on a large-scale project (Apache Hadoop) shows that results are promising, and the approach could locate a part of the true defects with high accuracy.

Our approach has also some limitations. The most important one is the property that we can isolate only these defects which cause a failure of a software test. As pointed out by an anonymous reviewer, this ignores the fact that some defects
remain latent for long periods, until the are discovered by end users.

In our future work, we will target the following challenges. **Extended evaluation.** We will evaluate the approach on more bugs (also artificial defects) and other applications. **More runtime facts.** Dynamic analysis will consider not only code coverage, but also other runtime information, like value of conditional expressions. **Adaptivity.** We will experiment with modifying the steps of our method depending on the sizes of intermediate results. **Supporting failure understanding.** We will study whether contrasting runtime data collected during the passing and failing run can help the programmer to understand the causes of the failure. **Enhancing tools for continuous integration.** As a long-term project, we plan to integrate these methods in wide-spread testing tools like Jenkins to support wide-spread adoption of these techniques.

**ACKNOWLEDGMENTS**

We would like to thank the members of PVS group of Heidelberg University, Felix Langner and Lutz Büch for their useful suggestions. This work is supported in part by the grant AN 405/2-1 entitled Automated, minimal-invasive Identification and Elimination of Defects in Complex Software Systems financed by the Deutsche Forschungsgemeinschaft (DFG).

**REFERENCES**