ChangeDebugger: Automated Detection Of Fault-Inducing Changes Using Bytecode Analysis

Tianyi Zhang, Yamini Gotimukul
University of Texas at Austin
Austin TX, USA
{tianyizhang, yamini.gotimukul}@utexas.edu

ABSTRACT
Software evolves by modifying the existing programs for adding new features or fixing bugs, instead of building from scratch. However, this process is error-prone because small changes can have colossal non-local effects. This is especially conspicuous in object-oriented programming due to the extensive use of dynamic dispatching and subtyping. Although regression testing can expose unexpected behavior in a modified program, it still takes a significant amount of time and effort to manually debug and identify the root causes for failed tests. In this paper, we propose an approach that automatically pinpoints the responsible edits by analyzing the semantic impact of a set of program changes on bytecode level. To further reduce the effort in manually inspecting the responsible changes, we ranks the changes based on the suspiciousness factor. We have demonstrated our approach on a self-created java program and the result proves that our approach can effectively identify and rank changes responsible for failed test cases.

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

General Terms
Algorithms, Design, Experimentation, and Measurement

Keywords
Change impact analysis, regression test, program analysis

1. INTRODUCTION
In modern software development, developers rarely build software from scratch but make edits to existing source code to add new features, fix bugs or refactor code. However, software is such a delicate artifact that any minor modification could introduce unexpected exceptions or test failures. To ensure that the modified program behaves as intended, regression testing is performed. Conventionally, developers debug the program by manually inspecting the changes using program-differencing tools such as, SVN diff or Eclipse compare plug-in. This manual process is very tedious and laborious. In the worst case, developers revert all their modifications and start over again. Particularly, in object-oriented programming, the extensive use of subtyping and dynamic dispatching makes modifications unpredictable, as small edits can have nonlocal effect. For example, overriding a method in a class might affect the runtime behavior of virtual method invocation.

In the past decade, many researchers have contributed to improving the productivity of program debugging and fault localization by reasoning program changes. Delta debugging [16, 17] locates failure-inducing changes by partition and binary searching the initial large set of program edits. Tarantula [6] utilizes test coverage information for fault localization, including statements, branches and data dependency. Chianti [13] selects a subset of regression tests whose behavior might have been affected by program edits and identifies the affecting edits that are related to test failures. FaultTracer [18] combines change impact analysis with spectrum-based ranking to localize failure-inducing program edits. The results of these works show that it indeed reduces the time and effort for debugging and understanding program failures by analyzing change impact and isolating the responsible changes. However, the main limitation is that most of them [6,13,16] only identifies potential changes without measuring their suspiciousness, which may still look overwhelming to developers. Although FaultTracer [18] ranks suspicious changes based on test spectra, it totally relies on the JDT framework, especially Java AST parser, to analyze source code changes, which inherently generates lots of overhead.

Our approach is similar in spirit to previous work but analyzes program changes on the bytecode level. We presume the existence of test suite as well as the availability of the bytecode for original and edited program revisions. In this approach, the bytecode changes are analyzed to obtain a set of interdependent atomic changes [14], whose granularity is on the method level. For each failed test cases, a static call graph is constructed to identify the dangerous call sites where the corresponding methods are modified in the current revision. Then we locate the suspicious atomic changes responsible for these dangerous call sites. Finally, the suspicious changes are ranked based on how primarily a change is examined by failed test cases. To demonstrate our approach, we developed a prototype, ChangeDebugger and applied it on self-created java programs. The demo shows our approach can narrow down the debugging scope and help developers gain insights about why a test has suddenly failed after a long editing session. As a part of the future work, we plan to evaluate our approach on enterprise-scale data set and further improve the usability of ChangeDebugger.

The main contributions of this research are as follows:
• We propose an approach to identify fault inducing changes at a bytecode level and rank these changes based on
a suspiciousness factor to further improve the productivity of program debugging.

- We developed a prototype, ChangeDebugger for evaluating its effectiveness and usability and the result shows that this approach can effectively pinpoint the program changes responsible for failed test cases. It also helps developers better understand the root cause of test failures.

The rest of this paper is organized as follows. Section 2 describes our approach and implementations in three phases. Section 3 demonstrates our approach on a self-created java program. Section 4 describes the related work and Section 5 summarizes our work with contributions, limitations and future work.

2. IMPLEMENTATION

ChangeDebugger is implemented as an eclipse plug-in, integrated with the JUnit framework in Eclipse. It leverages ASM library\(^1\), a bytecode analysis framework. Figure 1 gives an overview of the architecture of ChangeDebugger. It takes as input the bytecode of two program revisions as well as a regression test suite. Firstly, it identifies atomic changes between these two revisions and determines the dependencies among the changes based on a set of prescribed atomic change dependency rules [13,14]. Then ChangeDebugger automatically runs the regression test suite and captures the failed test cases. For each failed test cases, it constructs the call graph and determines the suspicious call sites which might induce the test failure. Finally, it ranks the atomic changes involved in these suspicious call sites based on how primarily a change is examined by failed test cases than passed ones. The following subsections give a detailed description of each phase.

2.1 Bytecode Differencing

Given the original and edited programs, ChangeDebugger first differences their bytecode and converts the bytecode changes to atomic changes. There are a number of existing techniques [1,4,9] to compute either the textual or syntactic differences in source code, however, few research has contributed to bytecode differencing. In this paper, we presents a bytecode differencing algorithm to capture syntactic differences in bytecode. It first loads the .class files, transforming bytecode to ASM ClassNode. For each method

\(^1\)http://asm.ow2.org/. It is similar to BCEL framework but provides more community support. The ASM APIs also generate less overhead in comparison to BCEL and JDT AST parser.

Figure 1: ChangeDebugger architecture.

Figure 2: Bytecode differencing procedure.

2.2 Atomic Changes

Our change impact analysis relies on the computation of a set of atomic changes that capture all bytecode modifications at a semantic level. It is currently accomplished with a coarse grain model at method level. To keep our approach simple, we have adapted 4 atomic changes, including added methods(AM), deleted methods(DM), changed methods(CM) and lookup change(i.e., dynamic dispatch)(LC). We also analyze the syntactic dependencies among atomic changes to further reduce the effort in fault localization. Intuitively, an atomic change A\(_1\) is dependent on another atomic change A\(_2\) if applying A\(_1\) without A\(_2\) will lead to syntactical error. For example, a method m1 is modified to call a new method m2 in the new version. Our approach considers the atomic change, CM on m1, dependent on the atomic change, AM on m2, as shown in Figure 4(where we present the source code change instead of bytecode change for illustrating here).

Moreover, in object-oriented language, minor edits in one location can result in unexpected run-time behaviors somewhere else, due to subtyping and dynamic dispatch. ChangeDebugger models these non-local change impact as the lookup atomic change(LC). In order to capture LC changes in bytecode, ChangeDebugger needs to generate the class hierarchy and then create the lookup table for both the original and modified programs, which is slightly different with prior works. Because ASM library doesn’t provide the binding information for each method invocation, we have to enable our analyzer to generate the class hierarchy and obtain binding
information by analyzing the class hierarchy. The lookup table records the real methods invoked by an object of a specific type, similar to virtual method table (VMT) in C++. Figure 4 monitors an example of lookup atomic changes. The addition of method B.foo overrides the method A.foo in the new revision. As a result of this change, a call to A.foo on an object of type B will dispatch to B.foo in the edited program, whereas it used to dispatch to A.foo in the original program. Therefore, the LC change here models the fact that a call to method A.foo on the object of type B invokes a different method now.

### 2.3 Identifying suspicious changes

Before identifying suspicious changes, ChangeDebugger first automatically runs the regression test suite and captures the failed tests. In order to automate the test execution process we have integrated the Junit testing framework in the prototype. For each failed tests, ChangeDebugger constructs its call graph by statically analyzing the bytecode. In fact, prior work [13] generates the dynamic call graph rather than static call graph, by tracing the execution of a test. The dynamic analysis indeed has advantage of handling dynamic dispatch and capturing run-time behaviors, but it also generates overhead by monitoring the stack trace. In our static bytecode analysis, the biggest challenge is that ASM library doesn’t provide binding information for variables and method invocations. As described before, we have already obtained the binding information for method invocations by creating the lookup table. Here, in order to identify the real type of the object that invokes a method, ChangeDebugger applies data flow analysis and creates a global variable table. The global variable table records the real type for each variable that get accessed when executing a test case. Figure 5 illustrates the idea of global variable table with a simple example. In this example, although variable a and b are declared as type A, both of them refers to a subtype of A. Therefore, the variable a is binded with type C while b is binded with B. During the execution of test1, it passes the variable b to the method equals. So the parameter obj is binded with b. In this way, ChangeDebugger can obtain the real type of a object that a method get invoked on. Then by querying the lookup table, it can recognize the dispatched method in run-time execution.

In order to detect the atomic changes responsible test failures, we assume that a change is suspicious if it is examined by the failed test cases. Because any modification to the execution path may induce test failure. Basically, ChangeDebugger checks each call site and groups all atomic changes involved in any invoked methods in a call graph of a failed test case. This subset of atomic changes are considered as the root cause for the test failure.

### 2.4 Ranking Suspicious Changes

The traditional change impact analysis only identifies a group of changes that may be responsible for test failures. However, the number of the suspicious changes in practice may still be overwhelming for manual inspection. So in the final step of our approach, we ranks the suspicious changes based on how primarily it is examined by the failed test cases rather than the passed ones. To achieve this, ChangeDebugger first generates call graphs for all test cases in the test suite, no matter it is passed or failed. Then it gathers the test coverage information for each method in the program in a matrix, as shown in Table 1. The rows of the matrix represents methods while the columns represent test cases. If a method is invoked by a particular test case, the corresponding item is updated as true. The suspiciousness of an atomic change is measured as a ratio of the number of failed test cases examining this change to the number of all test cases examining this change. Finally, all atomic changes are ranked based on the suspiciousness.

![Figure 4: The example of lookup atomic changes.](image)

![Figure 5: The example of global variable table.](image)

#### Table 1: The example of test coverage matrix

<table>
<thead>
<tr>
<th></th>
<th>test1</th>
<th>test2</th>
<th>test3</th>
<th>...</th>
<th>testn</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td>m2</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td>m3</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>...</td>
<td>false</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mn</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>...</td>
<td>false</td>
</tr>
</tbody>
</table>

3. DEMONSTRATION

This section demonstrates ChangeDebugger on a self-created java program. Figure 6 shows the UML diagram for the original and edited program revision. A developer made four changes between these two revisions, (1) changing coordinates x and y to be double, (2) changing the parameter types for the constructor of Circle, (3) overriding the method equals and (4) replacing the constant value, 3.14159276 with Math.PI. There are five test cases in the regression test suite, as shown in Table 2. Because the added equals method overrides the equals method in superclass Shape, invoking equals method on an object of type Circle is dispatched to Circle.equals rather than Shape.equals. Moreover, the added equals method not only compares the radius of two circles, but also checks the consistency between the coordinates, x and y of two circles. Therefore, test1 fails after modification. ChangeDebugger can help the developer to detect the suspicious changes that may be responsible for the test failure. It begins with setting the configuration file with the paths to both the program revisions and the test suite. Then click the button to run ChangeDebugger. It takes only 202 milliseconds (the average time after running 10 times) for the computation. Figure 7 shows the result. The edits in the
In the past decade, many researchers have contributed to the related fields of change impact analysis and regression test selection. Interestingly, there are many similarities and differences between our approach and the past works.

Identifying Responsible Changes. In Chianti [13], the author identify the changes in the program version by analyzing and differencing using call graphs. Our approach is similar to this work in terms of using call graphs for identifying the responsible changes but, we analyze at a byte code level, where as they analyze at a higher abstraction of source code level. Many other approaches employ control flow graphs. In comparison with this technique of test selection, our approach can be more effective in selecting test cases because it is based on call graphs traversal rather than control flow graph analysis.

Ranking Responsible Changes. When compared to prior works [10] [13], our approach gives a good insight about the responsible fault inducing changes, as we rank the changes based on a suspiciousness factor similar to [18]. None of the prior works in the field of change impact analysis employ the ranking approach while analyzing the change impacts.

5. SUMMARY

In this paper, we investigated the change impact analysis techniques with bytecode analysis and developed a prototype, ChangeDebugger. The demonstration shows that ChangeDebugger can effectively identify the suspicious changes for test failures, which helps narrow down the debugging scope in comparison to manually debugging. Moreover, it ranks these detected changes based on the suspiciousness factor, which can reduce the labor of inspecting all changes.
and help developers better understand the change impact. Although our approach of utilizing the bytecode in change impact analysis is novel, we have adapted only four atomic changes for the sake of simplicity. Besides, currently we only demonstrated the tool against a self-created java program without substantial evaluation. Therefore, the usefulness and scalability of the tool cannot be accessed. In future work, we need to further investigate this approach on large-scale data to evaluate its efficiency and accuracy, including precision and recall. We also plan to conduct a user study with ten to twelve participants to further assess its usability, in comparison with existing techniques such as Chianti and FaultTracer.

6. REFERENCES