ABSTRACT

Many academic studies in the field of software testing rely on mutation testing to use as their comparison criteria. However, recent studies have shown that redundant mutants have a significant effect on the accuracy of their results. One solution to this problem is to use mutant subsumption to detect redundant mutants. Therefore, in order to facilitate research in this field, a mutation testing tool that is capable of detecting redundant mutants is needed. In this paper, we describe how we improved our tool, LittleDarwin, to fulfill this requirement.

CCS CONCEPTS

- Software and its engineering → Software testing and debugging;

KEYWORDS

Software Testing, Mutation Testing, Mutant Subsumption, Dynamic Mutant Subsumption

ACM Reference Format:


https://doi.org/10.1145/3121245.3121249

1 INTRODUCTION

Many academic studies on fault detection need to assess the quality of their technique using seeded faults. One of the widely-used systematic ways to seed simulated faults into the programs is mutation testing [5]. Mutation testing is the process of injecting faults into software (i.e. creating a mutant), and counting the number of these faults that make at least one test fail (i.e. kill the mutant). The process of creating a mutant consists of applying a predefined transformation on the code (i.e. mutation operator) that converts a version of the code under test into a faulty version. It has been shown that mutation testing is an appropriate method to simulate real faults and perform comparative analysis on testing techniques [2, 3, 10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

A-TEST’17, September 4-5, 2017, Paderborn, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5155-3/17/09 ... $15.00
https://doi.org/10.1145/3121245.3121249

2 BACKGROUND

The idea of mutation testing was first mentioned by Lipton, and later developed by DeMillo, Lipton and Sayward [5]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [4]. Mutation testing is performed as follows: First, a faulty
version of the software is created by introducing faults into the system (Mutation). This is done by applying a known transformation (Mutation Operator) on a certain part of the code. After generating the faulty version of the software (Mutant), it is passed onto the test suite. If there is an error or failure during the execution of the test suite, the mutant is marked as killed (Killed Mutant). If all tests pass, it means that the test suite could not catch the fault, and the mutant has survived (Survived Mutant) \([9]\).

If the output of a mutant for all possible input values is the same as the original program, it is called an equivalent mutant. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant is indistinguishable from the original program. This makes the creation of equivalent mutants undesirable, and leads to false positives during mutation testing. In general, detection of equivalent mutants is undecidable due to the halting problem \([15]\). Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical in real projects due to the amount of work it requires. Therefore, the common practice within today’s state-of-the-art is to take precautions to generate as few equivalent mutants as possible, and accept equivalent mutants as a threat to validity (accepting a false positive is less costly than removing a true positive by mistake \([6]\)).

\[
\text{Mutation Coverage} = \frac{\text{Number of killed mutants}}{\text{Number of all non-equivalent mutants}} \tag{1}
\]

Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage (see Equation (1)) \([9]\). A test suite is said to achieve full mutation test adequacy whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a mutation-adequate test suite.

3 STATE OF THE ART

Mutant subsumption is defined as the relationship between two non-equivalent mutants \(A\) and \(B\) in which \(A\) subsumes \(B\) if and only if all inputs that kill \(A\) is guaranteed to kill \(B\) \([12]\). The subsumption relationship for faults has been defined by Kuhn in 1999 \([11]\), but its use for mutation testing has been popularized by Jia et al. for creating hard to kill higher-order mutants \([8]\). Later on, Ammann et al. tackled the theoretical side of mutant subsumption \([1]\). In their paper, Ammann et al. define dynamic mutant subsumption, which redefines the relationship using test cases. Mutant \(A\) dynamically

![Figure 1: An Example Mutated Method](image)

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Range of (a)</th>
<th>Range of (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>(-\infty, \infty)</td>
<td>(-\infty, \infty)</td>
</tr>
<tr>
<td>M1</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M2</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M3</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M4</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M5</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M6</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
<tr>
<td>M7</td>
<td>(-\infty, \infty)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

4 DYNAMIC MUTANT SUBSUMPTION ANALYSIS WITH LITTLEDARWIN

Figure 3 shows the input and output of LittleDarwin’s dynamic mutant subsumption (DMS) component. To facilitate dynamic mutant subsumption analysis in LittleDarwin, we retain all the output provided by the build system for each mutant. As a result, we can parse this output and extract useful information, e.g. which test cases kill a particular mutant. LittleDarwin’s DMS component can then use this information to determine dynamic subsumption relation between each mutant pair. This component then outputs the results in two different ways: (i) the dynamic mutant subsumption.
Dynamic Mutant Subsumption Analysis using LittleDarwin

A-TEST'17, September 4-5, 2017, Paderborn, Germany

Figure 2: Dynamic Mutant Subsumption Graph for JTerminal

Figure 3: Dynamic Mutant Subsumption Component I/O

Figure 4: Mutants 45 and 56 of JTerminal

Figure 5: The Test that Kills Mutant 45, but Not Mutant 56

graph, to visualize the subsumption relation, and (ii) a detailed report is generated in CSV\(^2\) format that contains all the information processed by the DMS component. For each mutant, mutant ID, mutant path, source path, mutated line number, whether it is a subsuming mutant, number of failed tests, the mutants it subsumes, the mutants that it is subsumed by, and the mutants that are mutually subsuming with it are provided in this report. Since LittleDarwin is a Java mutation testing framework, the application of the DMS component is also restricted to Java programs.

To showcase the ability of LittleDarwin in performing dynamic mutant subsumption analysis, we use JTerminal\(^3\) as a subject. We improved the test suite of JTerminal by automatically generating test cases using EvoSuite [7]. The information about characteristics of JTerminal is shown in Table 2. The DMSG for JTerminal is depicted in Figure 2. In this figure, each number represents a single killed mutant, each node represents a group of mutants that are killed by exactly the same set of test cases, and each edge shows the dynamic subsumption relationship between each node where the node at the end is subsumed by the node at the start. The survived mutants are not shown in this figure. The double-circled nodes contain the subsuming mutant groups. In order to remove the redundant mutants, one only needs to keep one mutant from each subsuming mutant group and discard the rest.

Take M45 and M56 as an example. According to the DMSG, M56 subsumes Mutant M45. Using the CSV report, we can locate the actual mutation of the source code (Figure 4). Both M45 and M56 belong to method parse of class AnsiControlSequenceParser, and mutate the same statement on line 99. M45 acts as a negation of the conditional statement. This means that any input character (except -1) that used to trigger "else if" and "else", now trigger this branch. Since this branch contains a "break" statement, it avoids the rest of the iteration of the loop to be executed. If the input is -1, the "else" branch would be executed, which wrongfully appends -1 to "text". M56, however, changes only two special cases. If the input is +1, the "if" branch would be executed, and the current iteration breaks. If the input is -1, the same thing as M45 happens. For any other input, the program executes as it should. This means that M56 truly subsumes M45. Figure 5 shows the test case that kills M45, but not M56. The input value here is a single control sequence, which is neither -1 or +1, and therefore cannot kill M56. However, since it should have been handled by "else if" branch and M45 does not allow that, it kills M45. Hence, in Figure 2 (on the left side), we can see that M56 dynamically subsumes M45. Analysis such as this

\(^2\)Comma-separated Values

\(^3\)https://www.grahamedgecombe.com/projects/jterminal

```java
int nextCharacter = reader.read();
if (nextCharacter == -1) {
    // M56: nextCharacter == -1
    // M45: nextCharacter != -1
    buffer.append((char) nextCharacter);
    break;
} else if (nextCharacter == MULTI_CSI[i]) {
    introducedControlSequence = true;
} else {
    text.append((char) character);
    text.append((char) nextCharacter);
}
```
Table 2: JTerminal Software Information

<table>
<thead>
<tr>
<th>Project</th>
<th>Ver.</th>
<th>Size (LoC)</th>
<th>Prod. Test</th>
<th>#C</th>
<th>TS</th>
<th>SC</th>
<th>BC</th>
<th>MC</th>
<th>#M</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTerminal</td>
<td>1.0.1</td>
<td>687</td>
<td>428</td>
<td>8</td>
<td>2</td>
<td>66%</td>
<td>56%</td>
<td>60.0%</td>
<td>160</td>
</tr>
</tbody>
</table>

Acronyms: Version (Ver.), Line of code (LoC), Production code (Prod.), Number of commits (#C), Team size (TS), Statement coverage (SC), Branch coverage (BC), Mutation coverage (MC), Number of Mutants (#M)

allows researchers to understand the relations between the mutants and reduce the effects of redundant mutants on their results.

5 CONCLUSION

Many academic studies in the field of software testing rely on mutation testing to use as their comparison criteria, and the existence of redundant mutants is a significant threat to their validity. We developed a component for our mutation testing tool, LittleDarwin, to facilitate the detection of redundant mutants using dynamic mutant subsumption analysis. We performed dynamic mutant subsumption analysis on a small, real-world project to demonstrate the capabilities of our tool. Using our tool, it is possible to detect and filter out redundant mutants, and help in increasing the confidence in results of experiments using mutation testing as a comparison criteria.

REFERENCES


