The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

Finding the Same Bug Twice

Harry J. Mills

Version 0.1, 2012-February-20

Supervisor: Iain Bate

Number of words = 0, as counted by `wc -w`.
This includes the body of the report only.

Abstract

Testing is an incredibly important part of software development and automated testing tools can greatly decrease the cost of testing and reduce the burden on the programmer. One problem faced when using automated testing tools that test methods and functions by generating random inputs is that it can be hard to know whether two of the bugs found are caused by the same fault in the code or two unrelated faults that happen to manifest in similar ways or at the same location. In this case two bugs are considered to be the same if they result from the same cause, i.e. the same mistake in the code. This project investigates a classification based approach to differentiating between bugs found using the YETI automated testing program.

To all students everywhere

Acknowledgements

I would like to thank my goldfish for all the help it gave me writing this document.

As usual, my boss was an inspiring source of sagacious advice.

Contents

1 Introduction

Testing is widely recognised as being of great importance when developing software at any scale. The York Extensible Testing Infrastructure (YETI) is a tool that performs automated testing for a variety of languages including Java, .NET languages, and others. To do this it uses random testing techniques in which randomly generated inputs are used to test the different componants of the program under test.

When performing random testing it can be hard to determine if two bugs found during the same test session are the same bug. (For the purposes of this document two bugs are the same if they both arise from the same cause). They may occur at the same place in the code (and determining where a bug occurs isn't always straight forward) but be triggered by different inputs. For example, an erroneous implementation of the sine function could potentially cause a divide by zero error for some inputs and a value outside of the range allowed by the function (from -1.0 to 1.0) for other inputs. These could be classed as two different bugs despite occuring at the same place in the code. In this example the divide by zero error may be caused by a mistake when sanitising the input data whereas the value outside the range may be a logic error in the code. These are different bugs as the programmer would have to change the code in multiple places or ways to fix them both.

This project aims to explore the concept of similarity as it applies to bugs in software, compare the relative benefits and disadvantages of different approaches to detecting similarity and then implement a classifier for the YETI automated testing system. To develop the classifier approaches applied in mutation testing will be used to generate buggy mutants of known working software. These introduced bugs will be used to train the classifier and then a second set of mutants will be used to evaluate its performance.

2 Background

The purpose of this section is to give readers sufficient information on the area of research this project covers so they will be able to better understand the work and see how the work described later in this document fits in to the field.

2.1 Testing

In his book, the Art of Software Testing, Glenford Myers defines testing as "the process of executing a program with the intent of finding errors." [**?**]. This can be done in a variety of ways using different techniques that each have different advantages and disadvantages, all with the end goal of testing the system as exhaustively as possible.

Exhaustive testing (in which all the possible inputs are tested to ensure they give the correct output) can be performed but it is only feasible when testing functions with a small number of input combinations and states. As the number of inputs increases the time required to exhaustively test a function rapidly increases, for example, a program that takes two 32 bit integers as input (such as a simple addition function) has 18,446,744,065,119,617,025 possible inputs.

Because exhaustive testing is rarely feasible other ways of measuring test coverage have been devised. Statement coverage is one of these methods. A test suite that obtains 100% statement coverage is one that ensures every statement is executed by the tests. The problem with this is that there are still paths through the code that may not be tested despite every statement being tested. An example of this is shown in listing 2.1, where testing the function with an input greater than or equal to five would ensure every line of code is tested but would not test whether the code works when the condition on line 3 is false and the value of b is not set.

As b is not initialised to a default value it is likely that if the if condition is false the function will not work as desired.

Branch coverage is another method that has been proposed as a way to measure the completeness of a test suite. Instead of executing every statement the test suite must ensure every possible branch is taken, for example, in the code shown in listing 2.1 testing the function with the inputs a=2 and a=5 would achieve 100% branch coverage. Branch coverage is not perfect however, for example it will not reveal some errors in the logic at branch points as it only requires both true and false values to be shown for the condition as a whole, not for individual variables within those conditions.

Listing 2.1: Statement coverage isn't always sufficient

```
public static int foo(int a) {
   int b;
   if (a >= 5) {
      b = 10;
   }
   return b * a;
}
```

There are other metrics used to measure the coverage of a test suite that try to improve on branch coverage. Condition coverage requires that each variable in a condition be shown to be both true and false during tests. This doesn't subsume branch coverage, as testing with a = true, b = false and a = false, b = true achieves condition coverage but not branch coverage for the code shown in listing 2.2. Testing with both a and b set to true as one test and a and b set to false as another would achieve both condition and branch coverage.

Listing 2.2: Condition coverage does not subsume branch coverage

```
if (a || b) {
   // do something
}
```

Decision coverage is another metric, similar to branch coverage but with the requirement that all outcomes are tested at each branch applied to any point at which a decision is made. This includes conditionals in

assignments as shown in listing 2.3.

Listing 2.3: Conditional assignments are tested by decision coverage but not branch coverage

```
boolean foo = bar || baz;
```

2.2 Automated Test Generation

Rather than developing a suite of tests that can be run to test the system manually automated test generation techniques can generate and run the tests automatically. This reduces the work required by the programmer as they no longer have to write tests for the software they develop but the tests are only as good as the automated test generation techniques that are used.

The automated test generation process can take several forms. One method involves creating a model of the system under test and then running the test generation software against the model. This produces a test suite that can then be run against the program to be tested. Examples of this method of test generation are given in [**?**] and [**?**].

Random testing is another method for automated test generation in which the functions or methods under test are executed with randomly generated inputs in an attempt to locate bugs. Purely random testing is far from ideal as it is unlikely to test key inputs, for example in the code shown in listing **??** it is very unlikely that the then branch of the if condition will ever get executed using random testing (with a 32 bit integer input the chance is $\frac{1}{2^{32}}$). When tests are automatically generated as in random testing the problem becomes one of choosing the necessary inputs to test with rather than creating tests to run.

Various research has been done in to better ways to generate inputs based on static code analysis. DART is one implementation of this technique where constraints are built up as the code is run from the branching conditions within the code and these constraints are then used to formulate test inputs that will test different branches in the code in an attempt to get 100% branch coverage.

2.3 Mutation Testing

Mutation testing is a method used to judge the performance and coverage of a test suite. The original code, when found to be correct according to the test suite, is altered using minor mutation operators designed to emulate mistakes that may have been made by the programmer. The test suite is then run on the mutated code. If the altered version of the code (called a 'mutant') is found to contain bugs according to the test suite then the mutant is considered 'killed'.

For example, if a mutation operator that changed the increment operator (++) to the decrement operator (–), was applied to the code in listing 2.4 it would result in the code in listing 2.5. This code would never terminate and a well written test suite should detect this, thus killing the mutant.

Listing 2.4: Code before mutation

```
int i = 0;
while (i < 10) {
   System.out.println(i);
   i++;
}
```

Listing 2.5: Code when mutated

```
int i = 0;
while (i < 10) {
   System.out.println(i);
   i--;
}
```

If all the mutants created are buggy the test suite should kill all of them. A mutant that is not killed means that the test suite has not detected the change in the code. This may be because the test suite is incomplete and unable to detect the bug that was introduced, alternatively it may be that the mutation does not actually change the behaviour of the program, for example the code shown in listings 2.3 and 2.4 are logically equivalent and thus the test suite will be unable to detect a difference in the outputs.

Listing 2.6: Code using the greater than or equal to operator

```
int numb = 20;
if (numb >= 15) {
   return numb;
}
```

Listing 2.7: Different but equivalent code

```
int numb = 20;
if (numb > 15) {
   return numb;
}
```

Mutation testing consists of three main phases, mutant generation, test selection, and mutant detection. In the mutant generation phase the working code is mutated using various mutation operators. This can be done in several different ways which each have different benefits and disadvantages. Firstly either the source code can be mutated and recompiled or the compiled byte code can be mutated. If the source code is mutated it is normally easier for mutations to copy the mistakes a programmer may make. The alternative is to mutate the byte code, this is normally faster than mutating the source code, partly because it doesn't require a recompilation step. The disadvantage is that the mutations are harder to develop and may not be equivalent to the mistakes a programmer may have made during development.

2.4 Classification