The Journal of Systems and Software 82 (2009) 1819–1832

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Should software testers use mutation analysis to augment a test set?

Ben H. Smith *, Laurie Williams

Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, United States

ARTICLE INFO

Article history:
Available online 24 June 2009

Keywords:
Mutation testing
Empirical effectiveness
User study
Mutation analysis
Test adequacy
Web application
Open source
Unit testing
Mutation testing tool

ABSTRACT

Mutation testing has historically been used to assess the fault-finding effectiveness of a test suite or other verification technique. Mutation analysis, rather, entails augmenting a test suite to detect all killable mutants. Concerns about the time efficiency of mutation analysis may prohibit its widespread, practical use. *The goal of our research is to assess the effectiveness of the mutation analysis process when used by software testers to augment a test suite to obtain higher statement coverage scores.* We conducted two empirical studies and have shown that mutation analysis can be used by software testers to effectively produce new test cases and to improve statement coverage scores in a feasible amount of time. Additionally, we find that our user study participants view mutation analysis as an effective but relatively expensive technique for writing new test cases. Finally, we have shown that the choice of mutation tool and operator set can play an important role in determining how efficient mutation analysis is for producing new test cases.

1. Introduction

Mutation testing is a testing methodology in which a software tester executes two or more program mutations (mutants for short) against the same test suite to evaluate the ability of the test suite or other verification technique to detect these alterations (IEEE, 1990; Trakhtenbrot, 2007). A *mutant* (or mutation) is a computer program that has been purposely altered from its original version (DeMillo et al., 1988). Mutants are automatically created via a mutation testing tool using mutation operators. A mutation operator is a set of instructions for making a simple change to the source code (DeMillo et al., 1988). For example, one mutation operator changes one binary operator (e.g. &&) to another (e.g. ||) in an attempt to create a fault variant of the program.

Mutation testing has historically been used to assess the fault-finding effectiveness of a test suite or other verification technique. We use the term *mutation analysis* to denote the process of augmenting an existing test suite to make that test suite *mutation adequate*, meaning that the test suite detects all non-equivalent mutants (Murnane et al., 2001). After completing mutation analysis, the augmented test suite may reveal latent faults and may detect faults which might be introduced while the system under test is further developed (Andrews et al., 2005).

Mutation analysis is computationally expensive and inefficient (Frankl et al., 1997). Mutation operators are intended to produce mutants which demonstrate inadequacies in the test set – that is, the need for more test cases (Frankl et al., 1997). However, some mutation operators produce mutants which cannot be detected by a test suite, and the tester must manually determine these are *stubborn*, or "false positive" mutants. Stubborn mutants make a mutation adequate test suite difficult to achieve in practice. Additionally, when testers add a new test case, they frequently detect more mutants than was intended, which brings into question the necessity of multiple variations of the same mutated statement.

Using the Goal-Question-Metric (GQM) Basili (1992) approach, we formulated our research goal: The goal of our research is to assess the effectiveness of the mutation analysis process when used by software testers to augment a test suite to obtain higher statement coverage scores.

Using the goal templates provided by the GQM process, we can rephrase this goal as:

> Analyze the **mutation analysis process**
> for the purpose of **evaluation**
> with respect to **effectiveness**
> from the viewpoint of the **software tester**
> in the context of **test case augmentation**.

From this reformulation, we elicit four major questions which can help us achieve our goal:

Q1. What is the effect of mutation analysis on coverage scores?
Q2. How long does a developer spend on each new test case while performing mutation analysis?
Q3. Do software testers find mutation analysis useful for augmenting a test set?

* Corresponding author.
E-mail addresses: ben_smith@ncsu.edu (B.H. Smith), williams@csc.ncsu.edu (L. Williams).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.06.031

Q4. In terms of the set of operators, which operator(s) produces the most new tests?

We conducted two studies to provide insight into these questions. For our first study, we set out to answer Q1–3. We conducted a user study where we observed our participants as they performed mutation analysis for 60 min. For this study, our participants used the Jumble[1] mutation testing tool (Irvine et al., 2007). In our second study, we set out to answer Q4. We performed mutation analysis on the set of mutants created by the MuJava[2] tool, which employs more operators, to empirically determine which operators are the most effective at producing new tests (Offutt et al., 2004).

The remainder of this paper is organized as follows: Section 2 briefly explains mutation testing and summarizes other studies that have been conducted to evaluate its efficacy. Then, Section 3 presents our user study of mutation testing when used by software testers. Next, Section 4 discusses our empirical study on the behavior of mutants of a given operator set. Finally, Section 5 concludes.

2. Background and related work

Section 2.1 gives required background information on mutation testing. Section 2.2 analyzes several related works on the technique.

2.1. Mutation testing

Mutation testing is conducted in two phases. In the first phase, the code is altered into several instances, called mutants, which are then compiled. Mutation generation and compiling can be done automatically, using a mutation engine, or by hand. Each mutant is a copy of the original program with the exception of one atomic change. The atomic change is made based upon a specification embodied in a mutation operator. The use of atomic changes in mutation testing is based on two ideas: the Competent Programmer Hypothesis and the Coupling Effect. The Competent Programmer Hypothesis states that developers are generally likely to create a program that is close to being correct (DeMillo et al., 1978). The Coupling Effect assumes "test cases that distinguish programs with minor differences from each other are so sensitive that they can distinguish programs with more complex differences" (DeMillo et al., 1978).

Mutation operators are classified by the language constructs they are created to alter. Traditionally, the scope of operators was limited to statements within the body of a single procedure (Alexander et al., 2002). Operators of this type are referred to as traditional, or method-level, mutants. For example, one traditional mutation operator changes one binary operator (e.g. &&) to another (e.g. ||) in an attempt to create a fault variant of the program. Recently, class-level operators, or operators that test at the object level, have been developed (Alexander et al., 2002). Certain class-level operators in the Java programming language, for instance, replace method calls within source code with a similar call to a different method. Class-level operators take advantage of the object-oriented features of a given language. They are employed to expand the range of possible mutation to include specifications for a given class and inter-class execution.

In the second phase of mutation testing, a test suite is executed against a mutant and pass/fail results are recorded. If the test results of a mutant are different than the original's, the mutant is said to be *killed* (Alexander et al., 2002), meaning at least one test case was adequate to catch the mutation performed. If the test results of a mutant are the same as the original's, then the mutant is said to *live* or to be *living* (Alexander et al., 2002) indicating that the change represented by the mutant escaped the test cases. *Stubborn*[3] mutants are mutants that cannot be killed due to logical equivalence with the original code or due to language constructs (Hierons et al., 1999). A mutation score is calculated by dividing the number of killed mutants by the total number of mutants. A mutation score of 100% is considered to indicate that the test suite is adequate (Offutt et al., 1996). However, the inevitability of stubborn mutants may make a mutation score of 100% unachievable. In practice, mutation analysis entails creating a test set which will kill all mutants that can be killed (i.e. are not stubborn).

In an earlier study, we have shown that mutation analysis is a viable technique to guide test case creation (Smith and Williams, 2009). To leverage mutation analysis for this purpose, the ideal is for every mutant to be detectable and to in fact produce a new test case. The effectiveness of the mutation analysis process, then, can be viewed as the number of new test cases a mutant set produces.

2.2. Related studies

Offutt et al. contend, "Research in mutation testing can be classified into four types of activities: (1) defining mutation operators, (2) developing mutation systems, (3) inventing ways to reduce the cost of mutation analysis, and (4) experimentation with mutation" (Offutt et al., 2006). In this sub-section, we summarize the research related to the last item, experimentation with mutation, the body of knowledge to which our research adds.

Several researchers have investigated the efficacy of mutation testing. Andrews et al. (2005) chose eight popular C programs to compare hand-seeded faults to those generated by automated mutation engines. The authors found the faults seeded by experienced developers were harder to catch. The authors also found that faults conceived by automated mutant generation were more representative of real world faults, whereas the faults inserted by hand underestimate the efficacy of a test suite by emulating faults that would most likely never happen.

Some researchers have extended the use of mutation testing to include specification analysis. Rather than mutating the source code of a program, specification-based mutation analysis changes the inputs and outputs of a given executable unit. Murnane et al. (2001) illustrate that mutation analysis must be verified for efficacy against more traditional black box techniques which employ this technique, such as boundary value and equivalence class partitioning. The authors completed test suites for a data-vetting and a statistical analysis program using equivalence class and boundary value analysis testing techniques. The resulting test cases for these techniques were then compared to the resulting test cases from mutation analysis to identify redundant tests and to assess the value of any additional tests that may have been generated. The case study revealed that there was only 14–18% equivalence between the test cases revealed by traditional specification analysis techniques and those generated by mutation analysis. This result indicates that performing mutation analysis will reveal many pertinent test cases that traditional specification techniques will not.

Frankl et al. (1997) compare analysis testing to all-uses testing using a set of common C programs, which contained naturally-occurring faults. All-uses testing entails generating a test suite to cause and expect outcomes from every possible path through the call graph of a given system. The authors concede that for some

[1] http://sourceforge.net/projects/Jumble.
[2] http://cs.gmu.edu/~offutt/mujava/.
[3] *Stubborn* mutants are more clearly defined as those living mutants that may or may not be *equivalent* to the original source code. Sometimes, a mutant remains alive and yet cannot be feasibly proven equivalent through formal analysis (Hierons et al., 1999).