Chapter VII

Test-Case Mutation

Macario Polo, University of Castilla - La Mancha, Spain

Mario Piattini, University of Castilla - La Mancha, Spain

Abstract

This chapter presents a new testing technique called “test-case mutation.” The idea is to apply a set of specific mutation operators to test cases for object-oriented software, which produces different versions of the original test cases. Then, the results of the original test case and of its corresponding mutants are compared; if they are very similar, the technique highlights the possible presence of a fault in the class under test. The technique seems useful for testing the correctness of strongly constrained classes. The authors have implemented a supporting tool that is also described in the chapter.

Introduction

Source-code mutation is a testing technique whose main goal is to check the quality of test cases used to test programs. Basically, a program mutant is a copy of the program under test, but with a small change in its source code, such as the substitution of “+” by “-.” Thus, this small change simulates a fault in the program, the program mutant therefore being a faulty version of the program under test. If a test case is executed both...
on the program being tested and on a mutant and their outputs are different, then it is said that the mutant is “killed.” This means that the test case has found the fault introduced in the original program and, therefore, the test case is “good.”

Changes in the source code are seeded by mutation operators that, in many cases, are language-dependent (i.e., there are mutation operators specifically designed for Java, C++, etc.).

Although powerful, source-code mutation is computationally a very expensive testing technique (Baudry, Fleurey, Jézéquel, & Traon, in press; Choi, Mathur, & Pattison, 1989; Duncan, 1993; Mresa & Bottaci, 1999; Weiss & Fleyshgakker, 1993). In fact, source-code mutation has several very costly steps:

- **Mutant generation:** Offut, Rothermel, Untch, and Zapf (1996) reported on an experiment that, from a suite of 10 Fortran-77 programs ranging from 10 to 48 executable statements, between 183 and 3010 mutants were obtained. Mresa and Bottaci (1999) showed a set of 11 programs with a mean of 43.7 lines of code that produced 3211 mutants. The Mujava tool (Ma et al., 2004), when applied to a Java version of the triangle-type program with 37 lines of code, produces 469 mutants.

- **Mutant execution:** According to Ma, Offutt, and Kwon (2004), research in this line proposes the use of nonstandard computer architectures (i.e., Krauser, Mathur, & Rego, 1991) and weak mutation. In weak mutation, the state of the mutant is examined immediately after the execution of the modified statement, considering that the mutant is killed even though the incorrect state is not propagated until the end of the program. Weak mutation was initially introduced by Howden. Offut and Lee (1994) concluded that weak mutation is a cost-effective alternative to strong mutation for unit testing of noncritical applications.

- **Result analysis:** Besides the study of mutants, both killed and alive, this step also involves the discovery of functionally equivalent mutants that, for Mresa & Bottaci (1999), “is the activity that consumes the most time.”

In order to reduce the number of mutants generated, Mathur (1991) proposed “selective mutation.” This line has also been worked by other authors: Mresa and Bottaci (1999), Offut et al. (1996), and Wong and Mathur (1995) conducted experiments to find a set of sufficient mutant operators that decreases the number of mutants generated without information loss. In Mresa and Bottaci (1999) and Wong and Mathur (1995), the respective authors also investigate the power of randomly selected mutants and compare it to selective mutation. In Hirayama, Yamamoto, Okayasu, Mizuno, and Kikuno (2002), the authors proposed a new testing process starting with a prioritization of program functions from several viewpoints; according to these authors, this ordination reduces the number of test cases generated without decreasing the results.

In this very same line, Kim, Clark, and McDermid (2001) analyze the effectiveness of several strategies for test-case generation with the goal of finding out which one gets kills more mutants.

Offut and Pan (1997) demonstrated that it is possible to automatically detect functionally equivalent mutants. A mutant is equivalent to the original program if it is impossible to
Related Content

HMM-Based Vietnamese Speech Synthesis
[www.igi-global.com/article/hmm-based-vietnamese-speech-synthesis/133113?camid=4v1a](www.igi-global.com/article/hmm-based-vietnamese-speech-synthesis/133113?camid=4v1a)

Architecture-Centered Integrated Verification
[www.igi-global.com/chapter/architecture-centered-integrated-verification/51970?camid=4v1a](www.igi-global.com/chapter/architecture-centered-integrated-verification/51970?camid=4v1a)

Transaction Level Model Automation for Multicore Systems
[www.igi-global.com/chapter/transaction-level-model-automation-multicore/36346?camid=4v1a](www.igi-global.com/chapter/transaction-level-model-automation-multicore/36346?camid=4v1a)

OPAC Usability Problems of Archives: A Case Study of the Hong Kong Film Archive
[www.igi-global.com/article/opac-usability-problems-of-archives/153171?camid=4v1a](www.igi-global.com/article/opac-usability-problems-of-archives/153171?camid=4v1a)