# Mutation Testing

**Pedro Delgado-Pérez**
*University of Cádiz, Spain*

**Inmaculada Medina-Bulo**
*University of Cádiz, Spain*

**Juan José Domínguez-Jiménez**
*University of Cádiz, Spain*

## INTRODUCTION

*Mutation testing* is a suitable technique to determine the quality of test suites for a certain program. This testing technique is based on the creation of *mutants*, that is, versions of the original program with an intentionally introduced fault. These errors are inserted within the code through some defined rules called *mutation operators*. Mutation operators represent typical mistakes made by programmers when using a programming language and they produce a simple syntactic change in the program under test (PUT).

The mutation testing process starts with the generation of the mutants using the set of mutation operators. Then, those mutants are executed against the test suite created for the PUT in order to determine its quality. Test cases are supposed to produce the correct output when they are run on the original program. When the output of a mutant is different from the output of the original program, the mutant is classified as *dead*. Otherwise, the mutant is still *alive* and needs to be executed against the rest of the test cases to detect its modification.

A good set of test cases should be able to detect any changes generated affecting the program. Hence, if some mutants remain alive after the test suite execution, new test cases can be supplied to kill them. In this process, a *mutation score* is calculated to determine the test suite effectiveness distinguishing the mutants (see Equation 1 for the general calculation of mutation score); the goal is to increase it until all the mutants are killed. An equivalent mutant is produced when none of the test cases is able to kill it as the meaning of the program has not actually been modified. Equivalent mutants, test data generation and the expensive computational cost that this technique entails are the main drawbacks to a broader usage of mutation testing.

$$MS(P,C) = \frac{KM}{TM - EM} \tag{1}$$

*MS:* Mutation score
*P:* PUT
*C:* Test cases
*KM:* Killed mutants
*TM:* Total mutants
*EM:* Equivalent mutants

Mutation testing is a *white-box* testing technique, i.e., it tests a program at the source code level. Therefore, the set of mutation operators and the overall technique should be developed around each programming language in particular; the correct choice of the set is one of the keys to successful mutation testing. Thus, a great variety of research studies devoted to the definition of mutation operators for specific programming languages and tools automating the generation of mutants can be found.

The purpose of the article is to look in depth at the development and the current state of mutation testing in order to widely make known this technique in the computer science research field. Next sections deal with the related work, the way that mutants are killed, the steps to accomplish in the mutation testing process, the approaches to evaluate mutation operators and the suggested techniques to improve the aforementioned problems. Finally, the C++ programming language will be focused as an example of the development of mutation testing.

DOI: 10.4018/978-1-4666-5888-2.ch710

## BACKGROUND

Mutation testing was originally proposed by Hamlet (1977) and DeMillo, Lipton and Sayward (1978) and its development has taken place in parallel with the appearance of the different programming languages (Offutt & Untch, 2001). As a result, in the early years, most of the works centered on procedural programming languages: Agrawal et al. (1989) defined a set of 77 mutation operators for C, the tool *Mothra* was developed including 22 operators to apply mutation testing to Fortran (King & Offutt, 1991) and Offutt, Voas, and Payne (1996) composed a set of 65 operators for the Ada language. The mutation operators for these procedural languages are known as *traditional* operators.

However, recently, new languages and paradigms have drawn the attention as well as the research has expanded towards other domains (Jia & Harman, 2011). As an illustration, we can find testing tools for rather different languages like *SQLMutation* for SQL (Tuya, Suárez-Cabal & de la Riva, 2007), *GAmera* for WS-BPEL (Domínguez-Jiménez, Estero-Botaro, García-Domínguez & Medina-Bulo, 2009) or *AjMutator* for AspectJ (Delamare, Baudry & Le Traon, 2009). Besides, the attention to the object-oriented (OO) paradigm has risen and several papers and tools have appeared mainly around Java (Ahmed, Zahoor & Younas, 2010). The different tools have been enumerated by Jia and Harman (2011). Finally, new mutation frameworks have been also developed lately: *Mutpy* (n.d.) for Phyton 3.x, *Mutant* (n.d.) for Ruby or *PIT* (n.d.) for Java and other JVM languages.

All these languages, even though sharing part of the syntax, need a particularized study to define their set of mutation operators and tools to generate the mutants. For example, as exposed in Kim, Clark and McDermid (2000), the aforementioned traditional operators can be applied to test OO programs, but those operators that were developed in programming environments away from this paradigm, do not take into account some types of faults related to features of this kind of programs, so operators at the class-level are necessary. Simultaneously, mutation testing, usually performed on programs at the unit level, has also been applied at other levels. Hence, Delamaro, Maldonado, and Mathur (2001) studied the technique to be used for integration testing and Mateo, Usaola and Offutt (2012) even to test a complete system. Mutation testing has also been performed on technologies relating the SOA architecture (Bozkurt, Harman, & Hassoun, 2013). Furthermore, apart from the code, mutation testing has been used in other domains like the specification of models, such as Finite State Machines (Fabbri, Delamaro, Maldonado, & Masiero, 1994) or Petri Networks (Fabbri, Maldonado, Masiero, Delamaro, & Wong, 1996).

As the technique was evolving and it was applied to real-world and bigger applications, it became clearer the barriers that this technique involves, which are discussed by Offutt and Untch (2001): the high computational cost and the time that the user needs to spend, for example, to determine the equivalent mutants. Around these problems have emerged new fields of study so that the technique gets a higher degree of maturation (Usaola & Mateo, 2010; Grun, Schuler, & Zeller, 2009). Besides, apart from the mutation score, new calculations are being used to enhance the effectiveness of the technique (Estero-Botaro, Palomo-Lozano & Medina-Bulo, 2010).

The significance of mutation testing and its limitations has been analyzed in different studies. The empirical results in Offut, Pan, Tewary, and Zhang (1996) showed that "16% more faults can be detected using mutation adequate test sets than all-use test sets" (Jia & Harman, 2011). Besides, a program related to the civil nuclear field was used in an experiment comparing real faults and the faults modeled by mutants (Daran & Thévenod-Fosse, 1996); 85% of the errors simulated with mutants were produced by real faults as well. On the other hand, the mutants created with operators at the class-level in Ma, Kwon, and Kim (2009) were fewer than with traditional ones, but they produced a percentage of equivalence over 70%, unlike the 5-15% usually produced with traditional operators.

## MUTATION TESTING OVERVIEW

### Killing Mutants

As stated, a mutant is *dead* or a test case *kills* a mutant when the output of the original program and the one of the mutant program are different. As an illustration, we can consider testing a program with the next statement: