Mutation Testing Applied to Object-Oriented Languages

S

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 designed 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. Mutations are inserted within the code through some defined rules called *mutation operators*. The underlying idea is that a good set of test cases for the system under test (SUT) should be able to detect any changes generated affecting the behavior of the application.

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 for a test case, the mutation has been revealed and 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. Hence, if some mutants remain alive after the whole test suite execution, new test cases can be added in order to kill these surviving mutants. However, we classify a surviving mutant as equivalent when the meaning of the program has not actually been modified despite the injected mutation.

Mutation operators represent typical mistakes made when programming and they produce a simple syntactic change in the SUT. 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, we can find an assortment of research studies devoted to the definition of mutation operators for specific languages and tools automating the generation of mutants.

In the same sense, a set of mutation operators can be defined at different levels in each language. Mutation operators mainly dealing with variables, operators or constants were designed for some procedural programs in the early years of the technique. However, other mainstream languages as Java, C# or C++ also include object orientation and completely different mutation operators are needed to test the new structures in these languages. As an example, the operator *IHD* (Hiding Variable Deletion) deletes a variable member in a subclass which is hiding a variable in a parent class:

Original code:

```
class Base{     class Child: public
Base{
public:           public:
      …                 …
      int v;            int v;
};                };
```

Mutated code:

DOI: 10.4018/978-1-5225-2255-3.ch649

```
class Base{     class Child: public
Base{
public:           public:
      …                  …
      int v;            /*IHD*/
};                 };
```

The purpose of the chapter is to look in depth at the development and the current state of mutation testing, and more specifically, with regard to object-oriented programming languages, in order to widely make known this technique in the computer science research field. Next sections deal with the related work, the steps to accomplish in the mutation testing process, the approaches to evaluate mutation operators and the existing techniques to improve the problems of this technique: equivalent mutant detection, test data generation and the expensive computational cost. Finally, the definition and evaluation of mutation operators for object-oriented languages will be focused.

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 and Pan (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). The existing mutation tools have been enumerated by Jia and Harman (2011). Finally, new mutation frameworks have been also developed lately: *Mutpy* (Derezińska & Halas, 2014) for Phyton 3.x, *Mutant* (n.d.) for Ruby or *PIT* (Van Laeden, 2012) for Java and other JVM languages.

The attention to the object-oriented (OO) paradigm has also risen and several papers and tools have appeared, mainly around Java (Ahmed, Zahoor & Younas, 2010). The first definition of class operators for Java was accomplished by Kim, Clark and McDermid (2000). As exposed in that paper, 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 definitely necessary. Mutation tools including class mutation operators are *MuJava* (Ma, Offutt & Kwon, 2005) for Java, *CREAM* for C# (Derezińska & Szustek, 2009) and *MuCPP* for C++ (Delgado-Pérez, Medina-Bulo, Domínguez-Jiménez, García-Domínguez & Palomo-Lozano, 2015).

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. Mutation testing, usually performed on programs at the unit level, has also been applied at other levels in addition to the class level. 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