Correct Application of Mutation Testing to the C++ Language

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez

Department of Computer Science and Engineering
University of Cadiz, Spain
{pedro.delgado,inmaculada.medina,juanjose.dominguez}@uca.es

Abstract. Success of mutation testing greatly depends on the mutation operators defined. As a white-box technique, selecting specific mutants for each language addressed is necessary, but it should be accompanied by an implementation focused on the particular details of the language. Only then we will be able to undertake a correct application of the technique, obtaining exactly the mutants that should be generated. This paper shows different C++-specific features that a mutation tool for this language should take into account with a twofold goal: creating valid but also useful mutants. Refining the implementation may reduce the computational cost of mutation testing application and enhance the effectiveness of mutation operators.

Keywords: Mutation testing, mutation operators, C++

1 Introduction

Mutation testing allows us to determine the adequacy of a test suite revealing several syntactic faults in our program. This technique can also be used to improve the test suite by introducing new test cases that are able to distinguish these faulty versions from the original program [7,9]. This difference can be found in the results of their executions. The faults, commonly known as mutations, are introduced in the program via the *mutation operators* defined for a certain language. For example, a mutation operator replacing relational operators may transform $x > 1$ into $x < 1$ to create a *mutant*. If a test suite detects the mutation, we say that the mutant is dead; otherwise, the mutant is either alive or is equivalent to the program under test.

Most works in literature deal with mutation operators from a high-level perspective based on their definition. On the contrary, there are few papers studying in depth their implementation as this is not such an interesting matter to research for being considered a technical detail. However, we claim that the correct implementation of operators is of a great importance in the application of mutation testing. Being this a white-box technique, automating mutation operators tailored to the specifics of a language is a key factor to produce the mutants as expected.

In this paper, we aim to show how, in addition to a suitable operator definition, the particularities of a mainstream language like C++ may affect the number of mutants generated and the mutation operator effectiveness. Section 2 addresses the mutation operators which can be created for this language and the method to put them into practice to achieve a high capacity for analysis of the code. Section 3 lists different C++ features impacting the number of mutants generated and how attending to them we can avoid mutants not useful for the purpose of mutation testing. The last section presents the conclusions and the future work to accomplish.

2 Mutation Operators in C++

2.1 Definition of Mutation Operators

Mutation operators mainly represent typical faults made by programmers. They are therefore obtained from the analysis of the most common mistakes in the development of applications in a certain language, representing the faults that the technique will treat within the code. Several faults are common to many general purpose languages, but each language possesses certain features making a specific study necessary. Thus, different works have been prepared to define a set of operators for a great range of languages [4]. In any case, we deem that the entire development of the technique should follow the same path so that it is feasible to compare every contribution in this field.

Mutation testing has been applied at different levels of the language. The level that the user chooses depends on the program characteristics and the type of testing to accomplish [4]. As a mainstream language, a set of mutation operators can be defined for each of these levels:

- **Unit level**: It refers to the traditional mutation applied to a single function or method to test its functionality. These operators are usually known as *traditional operators*.
- **Class level**: This level deals with the mutation of object-oriented features, modifying both declarations and expressions relating a class.
- **Integration level**: Intermediate level between the unit and the system levels. Function invocations are tested, making changes from parameters to values returned by the functions.
- **Multi-class or system level**: The operators at this level are intended to test a complete program, checking from interactions among classes to the user interface [6].

The mutation operators for C++ at the unit level take as starting point the research around C, taking into account the similarity of both languages. We can make the appropriate modifications to adapt the operators to C++ [1]. As for the class level, a complete set of class mutation operators were defined [2]; we adapted several operators from Java and C#, and also defined new operators regarding C++ features which had not been studied yet. In addition to these levels, we can add new levels focused on specific properties of the language frequently used. Regarding C++, this level would mainly correspond to the C++ Standard Library, which provides a great amount of facilities (e.g. stream input/output). Hence, Kusano et al. [5] applied mutation testing for concurrency constructs in C++ applications.

2.2 Technique to Inject Mutations in C++

One of the traditional approaches when analyzing the code to inject mutations is by means of a pattern matching based on regular expressions over the code. However, this method presents some limitations and may not be sufficient to achieve the expected result. As a simple example, when using this approach we have to check that each mutation location is not within a line or block comment before applying the mutation. In addition, the analysis of C++ code is not easy, especially because of its huge grammar. In case of similar languages to C++ like Java, mutations have been inserted directly on the bytecode or introspection has been also used. However, we have to discard these options in the case of C++ as they are not available for the user.

Thus, the most robust and comprehensive approach to apply mutation testing to C++ is resorting to the abstract syntax tree (AST) generated with a full-fledged compiler for this language [2]. This tree structures the code and determines the relations among the elements. Therefore, the AST allows us to properly analyze and solve complex situations in this language, as the ones listed in Section 3.1. Moreover, we will be able to undertake a much more fine-grained control of the potential mutation locations as well as to create mutants complying with the grammar rules.

3 Implementation Criteria

3.1 Creating Tailored Mutants to C++ Particularities

The operator implementation mode explained in Section 2.2 is definitely challenging, as stated by Derezińska [3]: it is necessary to analyze the AST and establish a margin for each mutation operator, which sets the different situations where an operator can be applied or not. Notwithstanding, the insertion of the faults can be controlled in an accurate manner as we can grasp from the AST well-defined elements.

Nevertheless, we need to take into account several aspects related with the C++ characteristics so that mutations meet expectations. These features have a direct effect on the number and kind of mutants created. An illustrative enumeration of various considerations are described below:

- **Class and structure**: For compatibility with C, classes were introduced as an extension of the structures so that, in addition to data members, function members and operators could be added as well. However, members in a class are hidden by default while structures are visible from outside. This is, in fact, the only difference between the keywords *struct* and *class*. Therefore, the set of operators applicable to classes can also perform on structures when possible.

– **Operator overloading property**: Most operators of this language can be redefined, giving them different semantics depending on the operand types. This feature needs to be considered in mutation testing as it implies the possibility of using operators of the language with user-defined types instead of defining new standard methods. At the same time, for instance, the traditional mutation operator replacing arithmetic operators may have changed its usual mathematical meaning when applied to user-defined objects. Nonetheless, the AST represents the invocations to an overloaded operator with a special kind of node, which is different from the one for traditional method callings. Hence, this aspect can be solved when implementing mutation operators related with this matter.
– **Typedefs and namespaces:** Evaluating types when searching for mutation locations in the code is an essential task. For instance, if a mutation operator replaces variables of the same type, a comparison of the types of the involved variables should be undertaken previously. The *typedef* keyword allows us to give a new name to a concrete type. Therefore, if we do not take this feature into account, some mutants may be overlooked.
  Also in this regard, several declarations can be grouped together in a *namespace*. We can declare similar elements in the code provided that they are in different namespaces. Thus, we need to properly qualify the references to declarations in namespaces to avoid confusion.
– **Definition and declaration:** Several elements in C++, as functions and methods, can be declared at any moment, but defined further in the code. This distinction is important in two aspects. Firstly, both the declaration and the definition should be modified if a mutation operator changes the signature or the value returned of a function or method. Secondly, we can invoke an element which has not been defined yet but has been declared previously.

3.2 Creating Useful Mutants

The operator implementation is usually a complex task that requires follow-up work. This implementation becomes more difficult when some conditions are imposed on the operators to prevent the creation of uninteresting mutants, i. e., mutants which do not help us assess the adequacy level of a test suite. However, this fact could allow for a reduction of mutants and, consequently, of the computational cost of the technique, which is a major concern when using mutation testing. Specific rules for several operators to cut out unnecessary mutants has been shown for Java [8].

We consider that the following kinds of mutants should be prevented as much as possible. Each of these kinds are accompanied by examples closely related to the C++ features:

- **Invalid mutants:** The executable programs created from the mutated code cannot be compiled or linked. An example of mutation that always produces an invalid mutant is deleting the initialization of reference type variables, as they must be initialized when declared.
- **Equivalent mutants:** There is not any input which is able to detect a difference between the original program and the mutant. For instance, the *PVI* operator [2] inserts the *virtual* keyword in methods which are not marked with this modifier. The method however may be already virtual although it is not marked as virtual. This happens when the method is overriding a virtual method in a base class. This mutation would produce an equivalent mutant (see Figure 1).
- **Trivial mutants:** The difference between the original and the mutant version is found by any input exercising the mutation. The usage of *templates* sometimes produces this type of mutants. For instance, some errors in a class template do not emerge until an object of the class template is created (whatever the type used), such as a constant data member which had not been initialized in a constructor.

Original classes:

```
class A{                    class B: public A{   class C: public B{
  ... ...                     ... ...                ... ...
  virtual void m(){...}       void m(){...}          void m(){...}
};                          };                   };
```

Mutant:

```
class B: public A{
  ... ...
  virtual void m(){...}
};
```

Equivalent:

```
The method 'm' in class B is virtual
with or without including the virtual
 keyword because the method 'm' in A
is already marked as virtual.
```

Fig. 1. Example of equivalent mutant in the *PVI* operator

4 Conclusion and Future Work

A correct implementation of the mutation operators beyond their narrow definitions is a decisive step towards a fruitful application of mutation testing. In this regard, the specifics of each language should be handled because the grammar rules may influence the kind of mutants generated with a mutation operator. The AST analysis is really useful to fulfil with the restrictions imposed on the application of the mutation operators. These restrictions, mainly based on the particular features of the language, may avoid generating mutants which are not useful, thereby reducing the high computational cost of the technique.

As future work, we intend to obtain a complete list of C++ features which can affect the mutation operators at the unit and the class level. This collection can be useful to develop mutation tools addressing this language so that the study of the results of different tools is somewhat comparable. Likewise, we aim to implement different mutation operators using rules to reduce equivalent, invalid and trivial mutants and then evaluate the impact of such improvement in the cost and effectiveness of the technique.

Acknowledgement: This paper was funded by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cadiz.

References

1. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J.: Analysis of the development process of a mutation testing tool for the C++ language. In: The Ninth International Multi-Conference on Computing in the Global Information Technology, ICCGI 2014. Seville, Spain (2014)
2. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A., Palomo-Lozano, F.: Class mutation operators for C++ object-oriented systems. Annals of telecommunications 70(3-4), 137–148 (2015)
3. Derezinska, A., Kowalski, K.: Object-oriented mutation applied in common intermediate language programs originated from C#. In: Software Testing, Verification and Validation Workshops, IEEE 4th International Conference on. pp. 342–350 (2011)
4. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on 37(5), 649 –678 (Oct 2011)
5. Kusano, M., Wang, C.: Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. pp. 722–725. IEEE (2013)
6. Mateo, P.R., Usaola, M.P., Offutt, J.: Mutation at the multi-class and system levels. Science of Computer Programming 78(4), 364–387 (2013), special section on Mutation Testing and Analysis (Mutation 2010) and Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing
7. Offutt, J.: A mutation carol: Past, present and future. Information and Software Technology 53(10), 1098–1107 (2011), http://www.sciencedirect.com/science/article/pii/S0950584911000838, special Section on Mutation Testing
8. Offutt, J., Ma, Y.S., Kwon, Y.R.: The class-level mutants of MuJava. In: Proceedings of the 2006 International Workshop on Automation of Software Test. pp. 78–84. AST '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1138929.1138945
9. Woodward, M.R.: Mutation testing - its origin and evolution. Information and Software Technology 35(3), 163–169 (Mar 1993)