Information and Software Technology 000 (2016) 1–16

Contents lists available at ScienceDirect

Information and Software Technology

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

# Assessment of class mutation operators for C++ with the MuCPP mutation system

Pedro Delgado-Pérez*, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, Juan José Domínguez-Jiménez

*Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain*

ARTICLE INFO

*Article history:*
Received 31 July 2015
Revised 18 June 2016
Accepted 6 July 2016
Available online xxx

*Keywords:*
Mutation testing
Mutation system
C++
Class mutation operators
Object-oriented programming

ABSTRACT

**Context:** Mutation testing has been mainly analyzed regarding traditional mutation operators involving structured programming constructs common in mainstream languages, but mutations at the class level have not been assessed to the same extent. This fact is noteworthy in the case of C++, despite being one of the most relevant languages including object-oriented features. **Objective:** This paper provides a complete evaluation of class operators for the C++ programming language. *MuCPP*, a new system devoted to the application of mutation testing to this language, was developed to this end. This mutation system implements class mutation operators in a robust way, dealing with the inherent complexity of the language. **Method:** *MuCPP* generates the mutants by traversing the abstract syntax tree of each translation unit with the Clang API, and stores mutants as branches in the Git version control system. The tool is able to detect duplicate mutants, avoid system headers, and drive the compilation process. Then, *MuCPP* is used to conduct experiments with several open-source C++ programs. **Results:** The improvement rules listed in this paper to reduce unproductive class mutants have a significant impact in the computational cost of the technique. We also calculate the quantity and distribution of mutants generated with class operators, which generate far fewer mutants than their traditional counterparts. **Conclusions:** We show that the tests accompanying these programs cannot detect faults related to particular object-oriented features of C++. In order to increase the mutation score, we create new test scenarios to kill the surviving class mutants for all the applications. The results confirm that, while traditional mutation operators are still needed, class operators can complement them and help testers further improve the test suite.

## 1. Introduction

C++ is a popular industrial-strength multiparadigm programming language, supporting concepts from both structured programming and object-oriented (OO) programming. However, because of its advanced features and flexibility, it is not easy to learn. Inexperienced developers may misunderstand parts of the language, increasing the need for adequate testing. In this context, it is puzzling to see that not much attention has been paid to perform mutation testing on C++ programs. A survey of the overall state of mutation testing [1] lists many mutation systems for similar languages like Java or C#, but only a few commercial tools for C++ that only apply some simple mutations. Given the widespread use of C++, we can conclude that the large gap between the systems for other languages and C++ must originate from the specific challenges that C++ presents.

Mutation testing is a well-known fault-based technique which has been used since the late 1970s to evaluate and improve the quality of test suites designed for a system under test (SUT) [2]. This technique is based on the injection of simple changes into the code, following the rules prescribed by a set of mutation operators usually based on emulating real faults or promoting good coding practices. The new versions of the program are called *mutants*. Mutation testing is supported by the competent programmer hypothesis, which explains why most software faults have their origin in subtle defects. While Gopinath et al. [3] found that real faults tended to be more complex than most mutations, Just et al. [4] provided evidence that the simple errors introduced in the mutations were related to more complex ones. This is known as the coupling effect hypothesis.

Mutation testing has been applied to different domains as new technologies appeared. The popularity of OO programming moti-

* Corresponding author.
*E-mail addresses:* pedro.delgado@uca.es (P. Delgado-Pérez), inmaculada.medina@uca.es (I. Medina-Bulo), francisco.palomo@uca.es (F. Palomo-Lozano), antonio.garciadominguez@uca.es (A. García-Domínguez), juanjose.dominguez@uca.es (J.J. Domínguez-Jiménez).

http://dx.doi.org/10.1016/j.infsof.2016.07.002
0950-5849/© 2016 Elsevier B.V. All rights reserved.

vated the creation of class mutation operators for Java [5] and C# [6]. Nevertheless, existing tools for C++ do not tackle mutation operators at the class level, i.e., operators injecting mutations concerning OO features. We have not found any studies in the literature in this regard for C++ either, while several authors have evaluated sets of class mutation operators for Java [7,8] and C# [9].

This work aims to lower the barriers concerning the complex task of building an OO-aware C++ mutation tool by presenting the *MuCPP* system. *MuCPP* can produce useful data regarding novel features of C++. In concrete terms, this paper aims to evaluate the class-level mutation operators for their validation. To the best of our knowledge, a mutation tool for this purpose has not been developed so far. The abstract syntax tree (AST) produced by Clang, a widely known open-source compiler, is used to systematically inject the mutations in a robust and comprehensive way, taking into account the variety of issues that can arise when analyzing C++ programs. Several aspects of the tool are described: the class mutation operators included, the process to produce the mutants and the overall system architecture and functionalities.

In our previous work [10], we showed an initial version of a set of class-level mutation operators for C++, conducted two case studies to evaluate how mutants were distributed across operators, and carried out a qualitative study on three specific class-level operators. The tool was first outlined in another previous work [11]. This paper extends the evaluation of the usefulness of the class-level operators with new case studies and compares them with traditional operators using *MuCPP*, which is presented in more depth. The relevant contributions of this paper are:

1. **A collection of restrictions on the generation of mutants** (several of them are C++-specific). These *improvement rules* reduce the number of *unproductive mutants*: those mutants which do not help the purpose of mutation testing as they do not provide interesting information for the assessment of a test suite. The conducted experiment, which evaluates these situations creating unproductive mutants, shows that these rules enhance mutant effectiveness and the efficiency of the system.
2. **A set of solutions for several technical challenges involved in C++mutation testing of real-world programs**, such as the detection of duplicate mutants, system headers and the full commands to compile the source files analyzed. These solutions allowed *MuCPP* to perform the experiments in this paper. Generating mutants as Git branches has been especially helpful to simplify implementation and save space without impacting scalability.
3. **A quantitative evaluation of the distribution of the mutants across five open-source programs**, showing the number of mutants generated by each operator and various statistics about the mutations. The experiment reveals that since the class-level operators generate fewer mutants than traditional ones, using these operators takes less time overall.
4. **An assessment of the usefulness of class operators and a comparison with traditional operators**. Mutation scores show that the tests distributed together with these SUTs did not handle some of the OO details. The class operators are shown to be useful in suggesting key missing test scenarios and helping find defects in the analyzed programs. The experiments provide evidence that the scenarios needed to kill certain class mutants may not be derivable from just using the traditional operators.

The paper is structured as follows. Section 2 describes the evolution of mutation testing in general, the existing research and issues around C++ mutation testing, and selects a metric for assessing operator quality. The next section introduces the *MuCPP* C++ mutation system, the implemented mutation operators and its approach across the different phases of the technique. Section 4 lists various restrictions imposed to improve operator effectiveness. Section 5 provides research questions and Section 6 answers them by discussing the results obtained in the conducted experiments. Section 7 explores related work, and the final Section 8 presents the conclusions and future research lines.

## 2. Background

### 2.1. Mutation testing evolution

Mutation testing research dates back to the 1970s from the ideas posed by Hamlet [12] and DeMillo et al. in 1978 [13]. In its early years, this technique was developed for a limited number of procedural languages such as FORTRAN, Ada or C, creating sets of mutation operators for those languages commonly known as standard or traditional operators. Some of these early landmarks are:

- Agrawal et al. [14] defined in 1989 a set of 77 mutation operators for C, divided into four categories (statement, operator, variable and constant mutations). This collection constitutes a base for the composition of sets of mutation operators for different programming languages afterwards.
- King et al. [15] developed the tool Mothra including 22 operators to apply mutation testing to FORTRAN.
- Offutt et al. [16] composed a set of 65 Ada operators.

Woodward [17] collected all the research on mutation testing from these first years. However, the appearance of new languages boosted research in the late 1990s and shifted the focus to other kinds of languages and domains [1]. Hence, in a short period, the technique has been applied to languages of diverse nature, and has also been used to detect faults in some technologies related to Web Services [18] or in the specification of models like Petri Nets [19].

The number of languages that have been tackled with this technique has definitely expanded, including OO languages. Although the OO paradigm became widely used in the early 90s, research regarding mutation testing started in 1999 with the definition of the first class operators for Java [20]. The class-level mutation operators for Java were refined and increased later in [5,7,21]. Furthermore, the first empirical studies on the effectiveness of class mutation operators have been accomplished recently [6,8,22]. Nonetheless, we can find an assortment of tools to test Java programs since then, such as MuJava [7] or CREAM [23].

### 2.2. Challenges to address in C++

When it comes to C++, it is no wonder that other languages have drawn more attention regarding object orientation because of the difference in complexity. Regarding the catalog of mutation operators, a rough approximation was made by Derezińska [24], but no operators were formally defined. The definition of a set of operators at the class level was carried out recently [10]. In the case of mutation systems for C++ [1], state-of-the-art commercial software adopting mutation testing within their testing techniques, like Insure++ and PlexTest, do not cover mutations at the class level, but only some standard operations (e.g., the removal of expressions and subexpressions in PlexTest). As for open-source systems, CCMutator [25] is a mutation generator for concurrency constructs in C or C++ applications.

The intricate structures involved in the analysis at the class level and the variety of alternatives provided by the language require thorough and arduous work. Indeed, the compilers for C++ are more complex than compilers for other languages because of the size of the grammar and the ambiguities (the meaning of a token depends on the context). At the same time, the compilers for this language have to deal with overgeneration during parsing [26], which would be a problem to solve if we consider developing our