Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System

Pedro Delgado-Pérez[a,*], Inmaculada Medina-Bulo[a], F. Palomo-Lozano[a], A. García-Domínguez[a], J. J. Domínguez-Jiménez[a]

[a] *Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain*

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.

Keywords: Mutation testing, mutation system, C++, class mutation operators, object-oriented programming

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 motivated 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

*Corresponding author

Email addresses: pedro.delgado@uca.es (Pedro Delgado-Pérez), inmaculada.medina@uca.es (Inmaculada Medina-Bulo), francisco.palomo@uca.es (F. Palomo-Lozano), antonio.garciadominguez@uca.es (A. García-Domínguez), juanjose.dominguez@uca.es (J. J. Domínguez-Jiménez)

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 [21, 5, 7]. Furthermore, the first empirical studies on the effectiveness of class mutation operators have been accomplished recently [22, 6, 8]. 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 own C++ parser. Additionally, it is not possible to use introspection and secondary but complex issues must be addressed, like integrating the wide variety of C++ build environments for the SUTs into the mutation process. Despite these difficulties, it is important to apply mutation testing to C++ as it is an industrial-strength language with widespread adoption in key strategic areas, like the defense, aerospace and telecommunications industries. This language is placed in the third position of popularity in the TIOBE index.[1].

Traditional operators may not be sufficient to test OO applications because of the new structures and features added with this programming paradigm. Therefore, the OO characteristics tackled by class operators require their own research. The C++-specific features deserve special attention as they may provide new knowledge in this field.

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, in March 2016

2.3. Mutation Operator Quality

Mutants are classified depending on the output of their execution. Thus, evaluating the mutants produced by each operator is very important, as only some of the mutants may be useful in the assessment of a test suite. In particular, operators generating a high number of equivalent mutants are inadvisable because they represent false positives for missing test cases. Various papers have dealt with the concept of quality of mutation operator: several of them have used the mutation score (the ratio of killed mutants to non-equivalent mutants) in each particular mutation operator [27], while some others have considered new dimensions like the number and kinds of mutants generated [6, 28].

Moreover, the orthogonality of several class operators has been studied [22]. Cost reduction techniques, such as selective mutation, have been analyzed while trying to draw conclusions about the most effective operators [9]. In other studies, mutants that are difficult to kill have been valued over mutants killed by many test cases [28]. *Trivial mutants* fall within this category of easy-to-kill mutants, as they are killed by every test case exercising the mutation [16].

Estero-Botaro et al. [29] recently defined a more accurate metric of the quality of a mutant:

$$Q_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|, & m \in D \end{cases} \quad (1)$$

Where:

- M is the set of valid mutants.
- E is the set of equivalent mutants.
- D is the set of dead mutants.
- T is the test suite.
- K_m is the set of test cases that kill mutant m.
- C_t is the set of mutants killed by test case t.

Consequently, the quality of a set of dead mutants can be defined as:

$$Q_D = \frac{1}{|D|} \sum_{m \in D} Q_m \quad (2)$$

Mutant equivalence is one the major barriers to mutation testing. The analysis of the surviving mutants is a time-consuming and error-prone task that has to be done manually. In the case of class mutation operators, the results in different studies show that equivalence is a more relevant issue than when applying their traditional counterparts [30], so the usage of a technique to reduce equivalence is justified.

However, the approach in this paper is more in line with the ideas posed by Offutt et al. [7], where different situations always creating equivalent mutants are filtered out in MuJava to reduce the equivalence in various operators. Upon an in-depth analysis of the context of each operator, a mutation producing equivalence can be detected and, therefore, a new criterion can be used to avoid that type of mutants.

3. MuCPP Mutation System

In the following subsections we present the work undertaken to apply mutation testing to C++ programs: the construction of the C++ mutation system called *MuCPP*[2], its features and included operators.

3.1. Class Mutation Operators

The authors started out the process to apply mutation testing to C++ programs with the development of a set of mutation operators at the class level [10]. Research around other similar languages like Java and C# was surveyed in order to adapt existing class mutation operators to C++ as well as new operators were defined. The list of operators implemented in *MuCPP* can be seen in Table 1. We should note that some of the operators defined for C++ were not included in the system after performing a review of the class operators assessed in the literature as well as those available in other mature mutation tools like MuJava [7] or CREAM [23]. For instance, *MBC* and *MNC* were discarded from this study because they are too similar to traditional operators in other languages to be regarded as class operators. In addition, *SVR*, a traditional operator for FORTRAN in Mothra similar to *MBC*, was the most prolific operator in the experiments conducted by Offutt et al. [31]; however, the mutation score was still 100% in almost all the case studies when this operator was removed.

The definition of these operators can be found in [10], in addition to a complete comparison with the class operators for Java. Still, a description of the particular and outstanding characteristics about the operators in *MuCPP* and also new information is summed up below:

1. Inheritance **(I)**: Unlike other languages like Java, a class can have more than a single direct base class in C++; this fact is called *multiple inheritance*. Moreover, the scoping in C++ allows to reference the members of classes which are deeper in the hierarchy. Thus, the *ISI* operator can create two mutants in Figure 1, inserting a qualifier for the classes `A` and `B` before the member variable `n`, because that variable is present in both classes.

Original classes:

```
class A{
   ... ...
   int n;
};

class B: public A{
   ... ...
   int n;
};
```

```
class C: public B{
    ... ...
    int n;
    int m () {
        ... ...
        return n*2;
};
```

Mutant 1:

```
class C: public B{
   ... ...
   int n;
   int m () {
       ... ...
       return A::n*2;
   }
};
```

Mutant 2:

```
class C: public B{
    ... ...
    int n;
    int m () {
        ... ...
        return B::n*2;
    }
};
```

Figure 1: Example of mutants from the *ISI* operator in the same location.

2. Polymorphism and dynamic binding **(P)**: C++ resorts to a virtual method table (also known as *v-table*) to achieve a dynamic or late binding when calling a method (the method is then looked up at runtime). For this purpose, the method needs to be declared as *virtual* so that the *v-table* is consulted and the method is dispatched based on the runtime type of the invoking pointer to object. This polymorphic behavior is attained with the usage of pointers and also with references, which must be initialized and cannot reference another object once they are created. With respect to the type casting of objects, C++ provides specific casting operators apart from the generic form. The *dynamic_cast* conversion includes, in a safe way, the *downcasting* of pointers/references as well as the *upcasting*.
3. Method overloading **(O)**: Method or function overloading is the ability that allows to declare several methods on a context with the same name, provided that they differ in the type or number of parameters. The possibility of adding *default parameters* is a point to take into account as the same method can be invoked with different numbers of arguments.
4. Exception handling **(E)**: Throwing and catching exceptions is an important mechanism to control and handle errors in a uniform manner, which is closely related to object orientation. For instance, throwing exceptions is an alternative to ensure the correct construction of objects, which is a fundamental aspect in C++. Exception handling requires attention because different exceptions can be raised in a *try-catch* block.
5. Object and member replacement **(M)**: Class members can be confused when referenced, especially if they belong to the same semantic field. The same applies to similar classes in a class hierarchy. The operators in this block simulate this kind of faults.

[2] `https://ucase.uca.es/mucpp`

Table 1: Mutation Operators at the Class Level Included in *MuCPP*

Block	Oper.	Description
Inheritance	**IHD**	Hiding variable deletion
	IHI	Hiding variable insertion
	ISD	Base keyword deletion
	ISI	Base keyword insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
	IMR	Multiple inheritance replacement
Polymorphism and dynamic binding	**PVI**	*virtual* modifier insertion
	PCD	Type cast operator deletion
	PCI	Type cast operator insertion
	PCC	Cast type change
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PNC	New method call with child class type
	PRV	Reference assignment with other comparable variable
Method overloading	**OMD**	Overloading method deletion
	OMR	Overloading method contents replace
	OAN	Argument number change
	OAO	Argument order change
Exception handling	**EHC**	Exception handling change
	EHR	Exception handler removal
Object and member replacement	**MCO**	Member call from another object
	MCI	Member call from another inherited class
Miscellany	**CTD**	*this* keyword deletion
	CTI	*this* keyword insertion
	CID	Member variable initialization deletion
	CDC	Default constructor creation
	CDD	Destructor method deletion
	CCA	Copy constructor and assignment operator overloading deletion

6. Miscellany **(C)**: This block gathers some other operators checking different language features, mainly regarding the construction of objects. We can mention copy constructors and destructors, or the initialization of variable members through initialization lists in the constructors because a direct initialization cannot be performed. *CID* has been redefined so that it only considers the initialization list.

3.2. Mutation Operator Implementation

The technique used to introduce the mutations into the code and create correct mutants is a crucial step in the whole process of applying mutation testing to a language. In the case of C++, the determination of mutation locations (where mutation operators can be used) has been a major concern stemming from the complexity and flexibility in the language. Given the challenge presented by inserting simple changes in a program, the use of a comprehensive and full-fledged parser yields a much more robust detection of mutation locations. The selected parser is Clang, a front-end developed for the LLVM project which is dedicated to the C family of languages (including C++). This parser guarantees a complete coverage of the grammar and allows us to conceive operations on the code which would be unattainable otherwise.

The method followed to inject mutations resorts to the abstract syntax tree (AST) generated by Clang to analyze and transform the code. The approach of traversing the AST has also been taken in other mutation systems for mainstream languages, like Major for Java [32], whereas Clang has been used in CCMutator [25] for multithreaded C/C++ programs and MILU [33] for the C programming language. The main benefit of analyzing the AST is the consistent search of potential locations, omitting higher-level details. Mutation locations can be determined through the pattern-matching facilities in Clang. This is not a pattern matching based on the concrete syntax of the language as in other existing mutation tools, thus avoiding some practical issues that arise in those systems [34]. Class mutation operators deal with structures much more intricate than the ones mutated by traditional operators. Thus, this approach is very convenient to detect certain type of complex declarations or expressions beyond the most basic situations or elements.

3.3. MuCPP Architecture

MuCPP, as most of the existing mutation systems, works in three distinct phases: analysis, mutant generation and test execution. These are described below and the entire process can be seen in Figure 2:

Analysis

MuCPP traverses the AST to study the code and determine where mutation operators can be applied. The set of class-level operators shown in Section 3.1 is available in the system and another set of operators can be added at any moment: the framework can be extended in a modular way to cover all the aspects of the language.

In this step, the tester provides one or more C++ files to the tool. Their ASTs are created and then visited for each of the operators. Note that all operators are executed at the same time, so each AST is only traversed once. This fact avoids introducing system overhead if the entire tree had to be visited as many times as operators were enabled.

Mutant generation

Mutants are created in this phase, each one only representing a single modification (first order mutation). Each mutant is a clone of the original program except for the modified files. Therefore, the files remaining unchanged are also stored in the clone. However, mutants are not created as new directories: each one is generated as a branch with a unique name in the Git version control system[3]. Thus, only the changes with respect to the original version occupy space on disk, allowing for a huge reduction of storage resources (see Section 3.4). This method allows

[3] `http://git-scm.com`

testing each mutant as a stand-alone program because the mutant contains all necessary files to build the program.

Test execution

Mutants are run against the tests defined for the SUT, reporting preset values when a mutant fails (1) or passes (0) a test. Then, these results are used to determine whether the mutants are dead or still alive after the test suite execution. *MuCPP* has been implemented in such a way that the tester is not limited to a specific testing framework. This is possible provided that the results of the test suite execution meet the output format that is expected by the tool. A timeout has been implemented, which can be configured depending on the tests run; the injected mutation can lead to unexpected behavior, so the timeout will stop the execution of the test when exceeding a reasonable time.

The testing process for the mutants derived from class mutation operators requires a test suite where objects belonging to the mutated classes are exercised. Thus, we have to differentiate test cases from test scenarios:

- *Test case*: It checks the state of one or more objects at a particular time. A test case is a single assertion to confirm that the response of concrete actions is the expected result.
- *Test scenario*: A scenario describes a particular logic where some objects work together. It may encompass different test cases, where a test scenario is tested by various test cases checking that the target of the scenario is achieved.

Thus, the tester creates several test scenarios including different kinds of test cases to check the correct operation of a set of classes and their members. Still, not only the test cases determine whether a mutant is killed or not, but a different behavior can be exhibited at any moment during the scenario execution, such as a runtime error or a timeout. Therefore, a test scenario may fail at any moment, even passing all the test cases.

3.4. Remarks

Dependency analysis: *MuCPP* requires the information handled by the build system of each SUT to correctly parse its source files. For instance, the tool should be aware about header file paths. This information and other configuration options can be found in the commands used to compile each source file in a project. This information can be provided through a *JSON compilation database* file[4], which can be automatically generated with *CMake*.

Header files: The AST contains the code of the headers included in the supplied files. The system is able to distinguish the user header files from the ones marked as system headers, only considering the former kind of headers for the insertion of mutations. Thus, if the user does not want a particular header file being mutated, this fact can be conveyed to the tool with the appropriate option. This is specially useful when working with third-party "lite libraries" provided by a single header placed within the project directory.

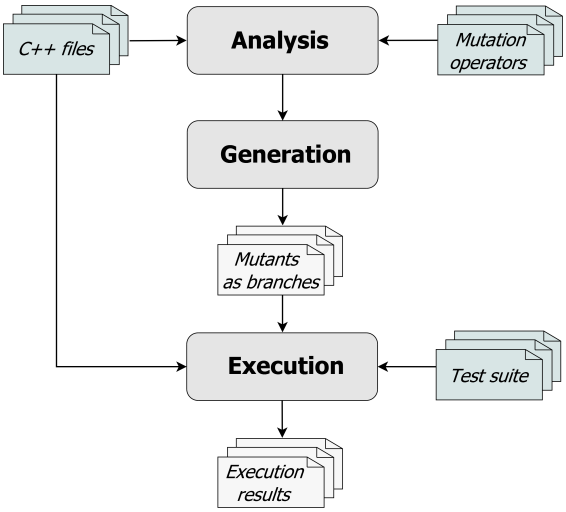

Figure 2: General *MuCPP* work-flow.

Git version control system: This is the first use of Git branches in a mutation tool as far as the authors know. Previously, the SVN version control system had been used to reduce space when storing mutants [23]. However, Git features make this version control system more efficient for mutation testing than SVN; Git is faster when switching between branches and when committing changes, as Git can be informed about the modified files. Although this usage of Git is unusual, the system has shown the ability to handle a large number of mutants without experiencing performance issues. As an illustration, during one of the case studies in Section 5 (*KatePart*), two sets of 2,127 and 54,984 mutants were generated: Git spent the same average time per mutant for both sets (0.174 seconds on a non-SSD hard disk). This shows that Git can scale to large numbers of mutants without problems.

Duplicate mutants: *MuCPP* has been designed to prevent the creation of duplicate mutants. As commented in the analysis phase, the system enables parsing various source files in the same execution, which are analyzed sequentially. Because of header files being contained in the AST and the same headers being included in different source files, a class could be analyzed more than once, leading to creation of the same mutants. Segura et al. [30] distinguishes the terms "generated" and "executed" mutants when carrying out the testing process in Java because of the existence of reusable classes. Thus, *MuCPP* saves a list of the locations mutated by each operator, ensuring that every mutant represents a different fault.

[4] `http://clang.llvm.org/docs/JSONCompilationDatabase.html`

4. Implementation Criteria for Operator Improvement

The main goal when implementing a mutation system is to obtain mutation operators which produce valid but also useful mutants. This section aims to address how imposing different rules or restrictions on the implementations of the operators can result in mutants of a higher quality. Although it is not mandatory in a basic implementation of an operator, reducing the number of mutants that do not help in evaluating the test suite may improve the effectiveness of the mutation testing technique, imposing less work on the tester and reducing the computational requirements of the approach.

In this regard, the idea is to avoid the generation of what we term *unproductive mutants*: those that are not useful for the assessment of the test suite. They include equivalent, invalid and trivial mutants (see Section 2.3). The generation of these mutants should be avoided as much as possible. However, in general, it is not possible to avoid every unproductive mutant: equivalence is an undecidable problem, and some restrictions can be very complex to implement.

Several studies exist identifying class-level mutants that should be avoided, by pointing to different situations in each concrete mutation operator [7, 22]. In this work, these situations are described in general instead of for each operator. Every case detected in this regard followed a systematic process:

1. When analyzing the generated mutants, a situation creating unproductive mutations could be located in a particular operator.
2. The detected case was then thoroughly studied to determine whether it was a one-time situation or it could be generalized as always creating unproductive mutants.
3. A feasibility study in terms of implementation was undertaken.
4. Then, the whole set of operators was processed to establish whether that situation could be extrapolated to other operators, creating an *improvement rule*, or it was an isolated occurrence. In the former case, the steps 2 and 3 were performed again in each of the concerned operators.

Bearing in mind that mutation testing is a white-box technique, most of the improvements are closely linked to C++ because they have been directly derived for the language under study. Still, some of these rules may apply to other OO programming languages. In addition to some particular cases implemented in specific operators, the following general improvement rules have been identified and carried out in the corresponding operators:

1. *Check for triviality*: In the operators related to constructors and destructors, deleting them would be useless if the compiler provides them with the same functionality by itself. This happens when the method is trivial: it has no initializers or the base classes are initialized with their default constructors, or the method has an empty definition.
2. *Member access control*: When replacing a reference to a member, if the member selected for the replacement belongs to the same class where it is referenced from, the access level is irrelevant. Otherwise, the access to the member needs to be checked to know whether that reference is allowed within that class.
3. *Declaration scopes*: Several operators replace a mention of a class to another class, but the new class may not have been declared yet at that point. Thus, it is necessary to check if the new class is available at the location to be mutated.
4. *Explicit invocation of constructors*: If a non-default constructor of a base class is invoked, this call cannot be removed if the base class does not have an explicit no-argument constructor. In this case, the class has to be always initialized explicitly as the compiler does not provide the default constructor.
5. *Check the invoked member*: A frequent reason for the existence of equivalent mutants is that the referenced member is still the same after the mutation. For instance, if a member of a base class is referenced with `Base::member` but the member has not been overridden in the child class, the mutant produced by *ISD* by removing `Base::` will be equivalent to the original program.
6. *Attributes marked as const*: Constant attributes require an explicit initialization. Hence, *CDC* for instance should not remove the default constructor if the class contains a constant attribute. Likewise, *IHI* will produce an invalid mutant when inserting an attribute marked as *const* in a child class: the variable would need to be initialized in the constructors. This also applies to reference type attributes.
7. *Default arguments*: When a method call is changed to invoke another method, default arguments must be taken into account: the list of parameters of a method needs to accept the arguments provided in the invocation.
8. *Pure virtual methods*: Several mutations result in pure virtual methods being called: these methods have no definition, resulting in invalid mutants. As an illustration, if *IOD* deletes an overriding method, the inherited method will be called instead when referenced. This will cause a compilation error if that method is marked as pure virtual.
9. *Infinite recursion*: Sometimes, the mutation can make a method calls itself indefinitely, as in *ISD* when deleting the base class qualifier within the overriding method (see Figure 3). This state leads to trivial mutants that would be killed by any test case covering the mutation.

Original:

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

Mutant:

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

Figure 3: Example of the "infinite recursion" rule for the *ISD* operator.

5. Experimentation with the Class Mutation Operators included in MuCPP

5.1. Research Questions

The general research goal of the experiments is to evaluate whether applying the class mutation operators implemented in *MuCPP* is useful and cost-effective. To this end, the following research questions will be answered:

RQ1: ***What is the impact of the reduction of unproductive mutants in the efficiency of the mutation system?*** We aim to know whether the reduction of unproductive mutants has a relevant effect in the cost of applying mutation testing.

RQ2: ***What is the quantity and distribution of mutants generated with the class mutation operators?*** We aim to perform a quantitative analysis of the class-level mutants. Then, we want to compare the set of class operators with the set of traditional operators to know which of them gives rise to a higher computational cost.

RQ3: ***Are class mutation operators useful to reveal missing tests?*** The mutation scores produced after the execution of the class-level mutants will indicate whether class operators are useful to design test cases and scenarios that were missing.

RQ4: ***To which degree do class-level and traditional mutants subsume each other?*** There may be some test scenarios in the test suites which are only necessary to kill class-level mutants but not traditional mutants. We will also compare the minimal test suites that cover all operators (traditional and class operators) with those covering only the traditional operators. If the former require more tests than the latter, it will show that the class-level operators provide valuable information that is not available through the traditional operators.

5.2. Applications and Test Suites

We have selected several existing open-source libraries and applications to avoid the potential bias of using a single case study. They are briefly described below:

- *Matrix TCL Pro (version 2.2)*[5]: a library for performing matrix algebra calculations in C++ programs.
- *Tinyxml2*[6]: a lightweight and efficient XML parser that can be integrated into C++ applications, commonly used for data serialization.
- *XmlRpc++ (version 0.7)*[7]: an implementation of the XML-RPC protocol for client-server communication over HTTP with other C++ programs.
- *KMyMoney (version 4.6.4)*[8]: a KDE desktop application for personal finance management.
- *KatePart*[9]: a text editor component with many advanced features, common in the KDE desktop environment.

The selected libraries are reused in many other applications. For instance, *XmlRpc++* is used in *SIREMIS* (Open-Source Web Management Interface for SIP Routing Engines)[10] and *ROS* (Robot Operating System)[11]. In the case of *Tinyxml2*, we can mention *mFAST*[12], an efficient implementation of the FAST protocol. *KatePart* is used by various popular KDE applications, such as the *Kate* text editor, the *Konqueror* browser or the *KDevelop* IDE.

Different characteristics and measurements of these programs are collected in Table 2, providing an overall picture of their complexity[13]. We have also included the time that the original programs spend executing their test suites. Table 3 complements the information about the SUTs for the quantitative analysis by classifying their classes into four ranges according to the lines of code. For each program, the number of classes belonging to each group (C) and the overall percentage ($C\%$) are shown. The last column presents the total percentage of classes within each range.

For the experiments in this paper, the test suite distributed together with these applications has been used: a simple library and a script were developed so that the different programs reported the results to *MuCPP*. As a final remark, we selected a manageable set of all the mutants for the experiments in Section 6, in order to be able to check their results. These mutants come from the most representative parts of the SUT covered by the test suite. For the quantitative study to answer research question RQ2, however, we studied the whole code in the SUT.

[5] `http://www.techsoftpl.com/matrix/download.php`

[6] `http://github.com/leethomason/tinyxml2`

[7] `http://xmlrpcpp.sourceforge.net/`

[8] `http://sourceforge.net/projects/kmymoney2/`

[9] `https://kate-editor.org/about-katepart`

[10] `http://siremis.asipto.com`

[11] `http://www.ros.org`

[12] `http://sett.ociweb.com/sett/sett0ct2013.html`

[13] The name of these programs have been abbreviated in the tables throughout the paper: *Matrix TCL Pro* (TCL), *XmlRpc++* (RPC), *Tinyxml2* (TXM) and *KMyMoney* (KMY) and *KatePart* (KAP).

Table 2: Metrics About the SUT Used in the Experiments

Measure	**TCL**	**RPC**	**TXM**	**KMY**	**KAP**
Classes	9	13	20	68	365
Lines of code	3,228	2,194	2,620	29,094	57,833
Constructors (mean)	3.0	1.5	0.9	1.7	0.9
Methods (mean)	21.1	11.2	15.6	21.5	14.5
Attributes (mean)	2.6	3.8	2.9	4.8	5.3
Inheriting classes	0	5	8	27	135
Inherited members (mean)	0.0	6.6	41.1	18.9	20.9
Depth inheritance (max.)	0	1	1	2	2
Direct bases (max.)	0	1	1	3	14
Test suite (seconds)	0.5	0.8	1.7	4.0	141.1

Table 3: Number of Classes in the Analyzed SUT by Range of Lines of Code

	TCL		**RPC**		**TXM**		**KMY**		**KAP**		**Total**	
Range	C	C%	C	C%	C	C%	C	C%	C	C%	**C**	**C%**
0-100	7	77.8	7	53.9	13	65.0	38	55.9	245	67.1	310	65.3
101-300	1	11.1	3	23.0	3	15.0	12	17.6	67	18.4	86	18.1
301-500	0	0.0	2	15.4	3	15.0	6	8.8	25	6.8	36	7.6
+500	1	11.1	1	7.7	1	5.0	12	17.6	28	7.7	43	9.0

5.3. Experimental Procedure

To answer RQ1:

A basic version of the operators was prepared using the approach explained in Section 3.2. As a second step, an improved version of the operators was implemented after a thorough study of each operator. Hence, the rules described in Section 4 were automated in the corresponding operators. The mutants generated in both versions were compared to check if the improved version properly avoided unproductive mutants. The generation and execution times as well as the storage requirements were measured in both cases to calculate the enhancement in the efficiency of the mutation system. A tailored timeout for each SUT was set to stop a test scenario if it did not respond after a reasonable time.

To answer RQ2:

Different measurements were computed to study the distribution of the generated mutants across class operators on real programs. Traditional mutants were also generated to compare the number of mutants created with the two types of operators.

To answer RQ3:

The mutation score using class operators was calculated for each SUT. The surviving mutants were analyzed, and then we tried to add new test cases to kill the surviving non-equivalent mutants. Each test scenario is usually written with a goal according to the SUT. In this way, the augmentation of the test suite to kill the surviving mutants has been performed as follows (see Section 3.3):

- A new test case is inserted in an existing scenario when the test case needed is closely related to the logic of that scenario.
- A new test scenario is created when, in our view, there are no test scenarios checking a particular use of the program. This scenario may include some test cases. Thus, several test cases not needed to kill the targeted mutant were also inserted to complete the scenario and make it as general as possible.

To answer RQ4:

Two different experiments using the execution results of traditional and class mutants were prepared. In the first experiment, we created for each case study 30 test suites derived from the augmented test suite which were adequate for the set of class mutants. Then, we applied those test suites to the set of traditional mutants and calculated an average mutation score. We also prepared the reverse experiment by computing the mutation score of class mutants with adequate test suites for the set of traditional mutants.

In the second experiment, we defined a metric to know if class mutants added test cases with respect to traditional mutants. Let T be the test suite used, M_t the results of running each traditional mutant against each test scenario in T, and M_c the analogue of M_t for class mutants. The following procedure was carried out for each SUT:

1. Obtain M_t, M_c, and also $M_{t \cup c}$ as the combination of the results of M_t and M_c.
2. Minimize the test suite we are using for M_t, M_c and for $M_{t \cup c}$; we will refer to these minimal test suites as $TM(M_t)$, $TM(M_c)$ and $TM(M_{t \cup c})$ henceforth. When minimizing the test suite we obtain those test scenarios which are necessary to kill the same mutants as the complete test suite T, thus avoiding redundant test scenarios.
3. Compare the sets $TM(M_t)$ and $TM(M_{t \cup c})$. In this regard, the comparison can yield two results:
   - $|TM(M_{t \cup c})| = |TM(M_t)|$, that is, the size of the minimal test suite for the set of traditional mutants is not affected when adding the mutants at the class level.

- $|TM(M_{t\cup c})| > |TM(M_t)|$, that is, the size of the minimal test suite for the set of traditional mutants increases when considering the mutants at the class level.

4. Compute the metric T_d, defined as:

$$T_d = \frac{|TM(M_{t\cup c})| - |TM(M_t)|}{|TM(M_{t\cup c})|} \quad (3)$$

This metric will allow us to know what proportion of tests from the minimal test suite $TM(M_{t\cup c})$ appear when considering the class mutants in addition to the traditional ones. Notice that if $|TM(M_t)| = |TM(M_{t\cup c})|$, then $T_d = 0$. Also if $TM(M_t) = \emptyset$, then $T_d = 1$; therefore, $0 \leq T_d \leq 1$. We should note that T_d depends on T, as M_t and $M_{t\cup c}$ have been initially obtained from the complete test suite T.

We compared the metric Q_D (see Equation 2 in Section 2.3) for the set of traditional and class mutants to assess if the value of T_d is affected by the fact that the test suite was improved only inspecting the surviving class mutants.

5.4. Experimental Setup

As mentioned in Section 5.3, the class mutation operators have been compared with a set of traditional operators. Table 4 lists the traditional operators included in *MuCPP*. We have adapted a set of operators for structured languages (e.g. C or FORTRAN) that have been thoroughly studied in the literature [30, 9, 8]. Offutt et al. [35] found that focusing on replacing primitive operators sufficed to efficiently implement mutation testing for these languages. When implementing some of these operators, we can opt for (1) generating all possible mutations per mutation location, (2) producing a sufficient set of non-redundant mutations [36, 37] or (3) introducing just one mutation in order to further reduce the cost. *MuCPP* implements option (1) for most of its operators, except for *ARB*, *ROR*, *LOR* and *ASR*, in which option (3) is implemented. For those operators, the tool performs one replacement (for instance, *ROR* replaces each appearance of the relational operator >= only with >), following a similar approach to PITest[14].

All the experiments were carried out on a server equipped with an Intel Xeon 2.60 GHz CPU and 16GB RAM running Ubuntu 14.04. The total execution time has been measured using the standard Unix utility *time*, while the execution, compilation and Git times were measured using the C++ standard library *chrono*. Concerning RQ4, we used an ad-hoc algorithm to produce a non-redundant and minimal test suite. This algorithm computes the first minimal test suite with respect to the execution order of the test scenarios, as there may be multiple minimal test suites of the same size. This allows us to properly compare $TM(M_t)$ and $TM(M_{t\cup c})$.

[14] `http://pitest.org/quickstart/mutators/`

Table 4: Traditional Mutation Operators included in *MuCPP*

Oper.	Description
ARB	Arithmetic Operator Replacement (Binary:+,−,∗,/,%)
ARU	Arithmetic Operator Replacement (Unary:+,−)
ARS	Arithmetic Operator Replacement (Short-cut:++,−−)
AIU	Arithmetic Operator Insertion (Unary:−)
AIS	Arithmetic Operator Insertion (Short-cut:++,−−)
ADS	Arithmetic Operator Deletion (Short-cut:++,−−)
ROR	Relational Operator Replacement (<,<=,>,>=,==,!=,*not_eq*)
COR	Conditional Operator Replacement (&&,*and*,||,*or*)
COD	Conditional Operator Deletion (!,*not*)
COI	Conditional Operator Insertion (!,*not*)
LOR	Logical Operator Replacement (&,|,^)
ASR	Short-Cut Assignment Operator Replacement (−=,+=,∗=,/=,%=)

Table 5: Reduction of Mutants for Improved Class Operators Generating Fewer Mutants in the Analyzed SUT

Oper.	**Basic**	**Improved**	**Reduction**	**Red.%**
IHI	223	152	71	31.8
ISD	16	2	14	87.5
ISI	98	8	90	91.8
IOD	201	43	158	78.6
IPC	67	35	32	47.8
IMR	3	0	3	100.0
PCD	38	13	25	65.8
PCI	2,324	901	1,423	61.2
PCC	5	0	5	100.0
PMD	458	453	5	1.1
PPD	334	261	73	21.9
OMD	340	199	141	41.5
OAN	33	27	6	18.2
CID	323	300	23	7.1
CDC	23	15	8	34.8
CDD	74	28	46	62.2
Total	4,560	2,437	2,123	46.6

6. Results and Discussion

6.1. Evaluating the Reduction of Unproductive Mutants

The reduction of mutants achieved by the improvement rules listed in Section 4 has been computed for every SUT. Table 5 shows how many mutants were produced in the original and improved versions, for those operators that produced fewer mutants after their improvement. The difference between the basic and the improved version (*Reduction*) and the percentage of reduction in the mutants obtained (*Red.%*) are also presented.

The total percentage of mutants excluded by the improvement rules in 16 mutation operators was 46.6%. This percentage should be taken with caution because of the few mutants produced in various operators and the varying reductions among the operators. In relation to the complete set of class operators, the reduction represents 32.1% of the total number of mutants. As for the individual results, the removal of all the mutants from *IMR* and *PCC* is remarkable. In contrast, the number of mutants was not reduced for other operators due to the characteristics of the SUTs.

Table 6 shows a complete list of times measured when applying mutation testing to each SUT. Results have been calculated for the basic and the improved version of the operators, as in Table 5. The time for each version has been

Table 6: Times for the Generation of Mutants and Test Suite Execution in the Analyzed SUT with the Basic and the Improved Version of Class Operators

	Basic version								**Improved version**						
		Generation		**Test suite execution**						**Generation**		**Test suite execution**			
SUT	$\|M\|$	Git	Total	Git	Comp.	Exec.	Total	—	$\|M\|$	Git	Total	Git	Comp.	Exec.	Total
TCL	172	5.7	11.2	0.00	0.13	0.02	0.17		137	4.7	9.3	0.00	0.11	0.02	0.13
RPC	244	7.5	23.2	0.01	0.32	0.06	0.39		191	5.7	19.0	0.01	0.27	0.06	0.35
TXM	1,140	37.6	39.5	0.01	0.22	0.17	0.41		614	20.2	22.2	0.01	0.10	0.12	0.23
KMY	2,289	100.6	163.8	0.14	9.86	0.84	10.84		1,421	62.5	123.5	0.09	7.17	0.75	8.01
KAP	2,768	481.6	526.1	0.82	24.87	47.10	72.80		2,127	371.3	413.5	0.68	20.50	44.28	65.46

Generation times measured in seconds; Test suite execution times measured in hours.

Table 7: Storage Resources for Class Mutants in the Analyzed SUT

SUT	$\|M\|$	**Original**	**Git**	**Mean**
TCL	137	0.3	6.9	0.05
RPC	191	2.5	12.0	0.05
TXM	614	0.9	24.0	0.04
KMY	1,421	96.0	167.0	0.05
KAP	2,127	520.0	754.0	0.11

Disk space measured in MB.

Table 8: Distribution of Class Mutants Generated by SUT and Operator, Divided by the Categories in Table 1

Oper.	**TCL**	**RPC**	**TXM**	**KMY**	**KAP**	**Total**
IHD	0	0	0	1	1	2
IHI	0	4	48	42	762	856
ISD	0	1	0	2	2	5
ISI	0	3	0	6	18	27
IOD	0	3	25	48	98	174
IOP	0	0	8	6	15	29
IOR	0	15	11	31	347	404
IPC	0	1	0	37	78	116
IMR	0	0	0	0	0	0
PVI	0	0	0	3	1	4
PCD	0	0	0	12	116	128
PCI	0	8	324	493	3,988	4,813
PCC	0	0	0	0	32	32
PMD	0	2	11	62	1,269	1,344
PPD	0	4	21	361	370	756
PNC	0	0	0	0	2	2
PRV	0	0	0	0	0	0
OMD	46	19	61	92	77	295
OMR	36	15	0	75	65	191
OAN	0	0	0	14	75	89
OAO	0	0	0	0	0	0
MCO	3	88	19	677	7,369	8,156
MCI	0	0	39	0	108	147
EHC	0	2	0	27	0	29
EHR	0	0	0	0	0	0
CTD	0	0	0	0	0	0
CTI	0	0	0	0	15	15
CID	40	17	34	152	832	1,075
CDC	0	2	3	7	29	41
CDD	2	5	6	8	84	105
CCA	10	2	4	4	10	30
Total	137	191	614	2,160	15,763	18,865
Mean	15.2	14.7	30.7	31.8	43.2	39.7

divided according to the two last phases in Section 3.3, mutant generation and test execution. *Total* in *Generation* measures the time needed to analyze the source files and produce the mutants. The time used by Git has been calculated separately, including creation of new branches and changes in the corresponding files. Regarding the test suite execution, the compilation and the execution times have been measured. The time taken by Git has also been computed, encompassing switches between branches and storage of the execution results.

As it can be seen, execution is the critical operation when compared with the phase of mutant generation. The compilation and the execution times are almost entirely dependent on the compilation system and the duration of the tests, respectively (see Table 2). The operations performed by Git are the least time-consuming in the execution phase. In contrast, Git takes most of the time in the generation phase. However, this result is not unexpected, taking into account that Git performs output operations which imply writing files.

When comparing the times in the basic and the improved version, we can see from Table 6 that the highest reduction is achieved in the compilation time. Many of the mutants avoided are invalid, which only increase the compilation time but not the execution time. The other mutants that are also discarded help reduce compilation time further. Test suite execution times were lowered thanks to the improvement rules. While these rules may require spending more time when detecting mutation locations, the final time is nonetheless lower than generating all the mutants in the basic version.

Regarding the storage requirements, Table 7 shows the size of each original SUT (*Original*) and the disk space occupied (*Git*) after creating the mutants in each SUT with the improved version of the operators ($|M|$). The size of the Git repository barely depends on the size of the SUT, but mainly on the number of mutants; we calculated the average of storage resources needed by each mutant (*Mean*) and the result is similar for every case study. This fact supports that Git just needs to save the mutation when generating a mutant, as commented in Section 3.3. Because of Git, the difference between the basic and the improved version of the operators with regard to the storage needed is not such an important matter as the

Table 9: Quantitative Statistics by SUT and Operator

	TCL			RPC			TXM			KMY			KAP			Mean	
Operator	C	C%	M	C	C%	M	C	C%	M	C	C%	M	C	C%	M	C%	M
IHD	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	1	1.5	0.0	1	0.3	0.0	0.4	0.0
IHI	0	0.0	0.0	2	15.4	0.3	6	30.0	2.4	9	13.2	0.6	70	19.2	2.1	15.6	1.1
ISD	0	0.0	0.0	1	7.7	0.1	0	0.0	0.0	1	1.5	0.0	1	0.3	0.0	1.9	0.0
ISI	0	0.0	0.0	1	7.7	0.2	0	0.0	0.0	2	2.9	0.1	6	1.6	0.0	2.4	0.1
IOD	0	0.0	0.0	3	23.1	0.2	7	35.0	1.3	18	26.5	0.7	34	9.3	0.3	18.8	0.5
IOP	0	0.0	0.0	0	0.0	0.0	1	5.0	0.4	2	2.9	0.1	4	1.1	0.0	1.8	0.1
IOR	0	0.0	0.0	3	23.1	1.2	2	10.0	0.6	1	1.5	0.5	29	7.9	1.0	8.5	0.7
IPC	0	0.0	0.0	1	7.7	0.1	0	0.0	0.0	22	32.4	0.5	76	20.8	0.2	12.2	0.2
PVI	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	1	1.5	0.0	1	0.3	0.0	0.4	0.0
PCD	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	2	2.9	0.2	46	12.6	0.3	3.1	0.1
PCI	0	0.0	0.0	6	46.2	0.6	9	45.0	16.2	20	29.4	7.3	128	35.1	10.9	31.1	7.0
PCC	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	5	1.4	0.1	0.3	0.0
PMD	0	0.0	0.0	2	15.4	0.2	7	35.0	0.6	7	10.3	0.9	86	23.6	3.5	16.9	1.0
PPD	0	0.0	0.0	4	30.8	0.3	8	40.0	1.0	27	39.7	5.3	42	11.5	1.0	24.4	1.5
PNC	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	2	0.5	0.0	0.1	0.0
OMD	7	77.8	5.1	2	15.4	1.5	13	65.0	3.1	18	26.5	1.4	32	8.8	0.2	38.7	2.3
OMR	9	100.0	4.0	3	23.1	1.2	0	0.0	0.0	27	39.7	1.1	36	9.9	0.2	34.5	1.3
OAN	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	7	10.3	0.2	14	3.8	0.2	2.8	0.1
MCO	1	11.1	0.3	2	15.4	6.8	3	15.0	0.9	16	23.5	10.0	87	23.8	20.2	17.8	7.6
MCI	0	0.0	0.0	0	0.0	0.0	2	10.0	1.9	0	0.0	0.0	3	0.8	0.3	2.2	0.4
EHC	0	0.0	0.0	1	7.7	0.2	0	0.0	0.0	7	10.3	0.4	0	0.0	0.0	3.6	0.1
CTI	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	0	0.0	0.0	4	1.1	0.0	0.2	0.0
CID	8	88.8	4.4	4	30.8	1.3	8	40.0	1.7	24	35.3	2.2	153	41.9	2.3	47.4	2.4
CDC	0	0.0	0.0	2	15.4	0.2	3	15.0	0.1	7	10.3	0.1	29	7.9	0.1	9.7	0.1
CDD	2	22.2	0.2	5	38.5	0.4	6	30.0	0.3	8	11.8	0.1	84	23.0	0.2	25.1	0.2
CCA	8	88.8	1.1	1	7.7	0.2	2	10.0	0.2	3	4.4	0.1	6	1.6	0.0	22.5	0.3
Mean	5.8	64.8	2.5	2.5	19.5	0.9	5.5	27.5	2.0	10.5	15.3	1.4	39.2	10.7	1.7	27.6	1.7

C: Number of mutated classes – C%: Percentage of mutated classes – M: Average mutants per class

time expenses.

Answer to RQ1: The improvement rules avoided the generation of 46.6% of the mutants across 16 class operators, as presented in Table 5, which represents a total reduction of 32.1% mutants for the analyzed SUT. As a result, the computational cost is reduced in all the cases (see Table 6) both when generating and executing the mutants (mainly with regard to the compilation time), improving the efficiency of *MuCPP*.

6.2. Quantitative Study

Table 8 depicts the number of mutants created with *MuCPP* in the analyzed SUTs. The number of mutants is shown per operator and the total is added up at the end of the table. In addition, the average number of mutants produced by class in each application is calculated (this mean only considers the operators producing at least one mutant). Table 9 includes, for each program and operator, the number (C) and percentage ($C\%$) of all the classes that were mutated, and the average number of mutants that were generated per class (M) (operators producing no mutants are not shown in this table). As an example of the meaning of M, the operator *OMR* produced 36 mutants in *Matrix TCL Pro* (see Table 8); the number of classes in this SUT is 9 (see Table 2), so the value of M in Table 9 is 4 (36/9).

PCI and *MCO* produce a considerable number of mutants, so they may increase the cost of the technique. These two operators have a great influence in the data shown in Table 8 as they produce almost 69% of the total

Table 10: Distribution of Traditional Mutants Generated by SUT and Operator (see Table 4)

Oper.	TCL	RPC	TXM	KMY	KAP	Total
ARB	1,252	64	58	232	2,068	3,674
ARU	12	14	5	162	747	940
ARS	896	40	104	348	1,680	3,068
AIU	3,841	348	288	1,475	10,649	16,601
AIS	11,304	828	620	2,096	23,668	38,516
ADS	63	12	44	141	412	672
ROR	612	155	97	589	3,593	5,046
COR	28	53	88	425	2,023	2,617
COI	533	277	229	1,442	8,172	10,653
COD	20	48	74	482	1,501	2,125
LOR	1	4	17	10	170	202
ASR	172	12	18	22	301	525
Total	18,734	1,855	1,642	7,424	54,984	84,639

of mutants. In order to keep the cost of mutation testing manageable, testers could decide to manually disable *MCO* based on their knowledge about the program (i.e. they know that the members of the analyzed classes are not prone to cause confusion). However, the decision of excluding some operators could introduce a bias in the testing process. One option would be investigating if this decision could be automated in some form by the tool (e.g. by comparing member names according to a heuristic): this would merit additional studies.

While *PCI* and *MCO* are also the operators injecting the highest number of mutations per class (7.0 and 7.6 respectively), *CID* (47.4%), *OMD* (38.7%) and *OMR* (34.5%) are the operators mutating more classes as a percentage. This is partially explained by the fact that they mutate constructors.

On the contrary, other operators do not generate any mutants or only introduce few mutations. This is the case of *IMR*, *PRV*, *OAO*, *EHR* and *CTD*. This is largely because the language features they address are not so used. In other cases, the restrictions imposed on the operators prevent several mutants from appearing. For instance, the basic version of *IMR* generates various invalid mutants in *KMyMoney* (see Table 5) because the rule about pure virtual methods (number 8 in Section 4) is disabled. Despite not generating any mutants for the SUTs in this work, these operators can be valuable because the features they address may receive less attention due to their rare use. Therefore, we do not recommend discarding them.

A key factor in the creation of class mutants is the existence of inheritance relationships among classes. The absence of inheritance will not only impact the "inheritance" operators, but also the operations in the "polymorphism and dynamic binding" group. This is also the case of *MCI*, including over half of the class-level operators. This fact has been purposely explored with the inclusion of *Matrix TCL Pro*, encompassing nine classes with no inheritance relations among them.

Table 11: Mutation Score in *Matrix TCL Pro*

Operator	**Mutants**	**Dead**	**Alive**	**Equivalent**	**MS**
OMD	46	31	15	8	0.82
OMR	34	25	9	1	0.76
MCO	3	0	3	0	0.00
CID	40	31	9	2	0.82
CDD	2	0	2	2	-
CCA	10	3	7	7	1.00
Total	135	90	45	20	0.78

Table 12: Mutation Score in *XmlRpc++*

Operator	**Mutants**	**Dead**	**Alive**	**Equivalent**	**MS**
IHI	4	2	2	2	1.00
ISD	1	0	1	0	0.00
ISI	3	0	3	1	0.00
IOD	3	0	3	2	0.00
IOR	15	0	15	15	-
IPC	1	1	0	0	1.00
PCI	3	2	1	1	1.00
PPD	1	0	1	1	-
OMD	10	7	3	1	0.78
OMR	10	7	3	0	0.70
MCO	48	19	29	10	0.50
EHC	2	0	2	1	0.00
CID	17	10	7	3	0.71
CDC	2	0	2	0	0.00
CDD	5	1	4	3	0.50
CCA	2	2	0	0	1.00
Total	127	51	76	40	0.59

Table 13: Mutation Score in *Tinyxml2*

Operator	**Mutants**	**Dead**	**Alive**	**Equivalent**	**MS**
IHI	47	30	17	6	0.73
IOD	25	21	4	1	0.87
IOP	8	8	0	-	1.00
IOR	11	8	3	1	0.80
PCI	190	133	57	20	0.78
PMD	3	0	3	3	-
PPD	7	4	3	3	1.00
OMD	37	15	22	14	0.65
MCO	19	17	2	1	0.94
MCI	39	11	28	26	0.85
CID	34	21	13	10	0.87
CDC	3	3	0	-	1.00
CDD	6	3	3	3	1.00
CCA	4	0	4	4	-
Total	433	274	159	92	0.80

Table 14: Mutation Score in *KMyMoney*

Operator	**Mutants**	**Dead**	**Alive**	**Equivalent**	**MS**
IHD	1	1	0	-	1.00
IHI	23	6	17	15	0.75
ISI	3	0	3	3	-
IOD	1	0	1	0	0.00
IPC	18	9	9	6	0.75
PCI	15	14	1	1	1.00
PMD	1	0	1	1	-
PPD	18	4	14	14	1.00
OMD	13	4	9	4	0.44
OMR	32	28	4	0	0.87
OAN	7	3	4	4	1.00
MCO	87	31	56	7	0.39
EHC	6	1	5	5	1.00
CID	48	15	33	22	0.58
CDC	5	4	1	1	1.00
CDD	4	2	2	2	1.00
CCA	2	0	2	2	-
Total	284	122	162	87	0.62

Table 15: Mutation Score in *KatePart*

Operator	**Mutants**	**Dead**	**Alive**	**Equivalent**	**MS**
IHI	51	4	47	28	0.17
ISI	2	0	2	2	-
IOD	4	1	3	2	0.50
IOR	5	0	5	5	-
IPC	5	0	5	0	0.00
PCD	1	0	1	0	0.00
PCI	53	12	41	29	0.50
PMD	1	0	1	1	-
PPD	11	0	11	11	-
OMD	5	1	4	1	0.25
OMR	8	4	4	3	0.75
OAN	16	2	14	0	0.12
MCO	46	0	46	0	0.00
MCI	15	0	15	0	0.00
CTI	2	2	0	-	1.00
CID	54	21	33	26	0.75
CDC	5	1	4	2	0.33
CDD	10	8	2	2	1.00
CCA	6	2	4	4	1.00
Total	300	58	242	116	0.32

Table 10 shows how many traditional mutants were generated for each SUT. This table reports that the number of traditional mutants from only 12 operators (84,658) is far higher than their class-level counterparts (18,865): over four times as many, in fact. There are fewer mutants at the class level in every SUT, especially in the case of *Matrix TCL Pro* which heavily uses arithmetic operations (137 class-level and 18,734 traditional mutants). We should note that we have spent much time improving class operators, but traditional operators have also been refined to prevent unproductive mutants. As a conclusion, with regard to the number of mutants generated, most class mutation operators tend to require less computational resources than traditional operators.

Answer to RQ2: Absolute and relative counts for the

Table 16: Mutation Score Obtained After Improving the Test Suite for the Analyzed SUTs with Respect to Surviving Non-Equivalent Class Mutants

	Original		Added			Augmented		
SUT	$\|S\|$	$\|C\|$	$\|M\|$	$\|S\|$	$\|C\|$	$\|S\|$	$\|C\|$	MS
TCL	17	87	3	7	35	24	122	1.00
RPC	26	61	5	8	36	34	97	1.00
TXM	57	111	3	5	32	62	143	0.91
KMY	241	2,281	10	7	67	248	2,348	0.98
KAP	158	1,843	1	16	56	174	1,899	0.57

class-level mutants are shown in Tables 8 and 9. On average, the class operators create mutants for 27.6% of the classes, inserting 39.7 changes in each one (1.7 per operator). Each of these means is obtained considering only the operators that created mutants in its case study. On the other hand, there are many more traditional mutants for the same SUT than class mutants (see Table 10), which require more time to evaluate.

6.3. Test Suite Improvement

We have categorized the valid mutants of the classes of each SUT. Tables 11–15 include the mutants produced by operator (*Mutants*), how many are killed (*Dead*), how many remain alive (*Alive*), how many are found to be equivalent (*Equivalent*), and the mutation score (*MS*). As shown in the results of Sections 6.1 and 6.2, *MuCPP* prevents creating several mutants because of the improvement rules, and class operators generally produce fewer mutants than traditional operators. Thus, the low quantity of mutants when compared to other evaluations of operators in the literature corresponds to the column M in Table 9, where most operators inject less than one mutation per class on average.

As it can be observed, the mutation score is far from 100% in all the programs, especially for *XmlRpc++* (51%) and *KatePart* (32%). Therefore, in these cases, we can say that the test suite is not able to detect the different variations mimicked by the class operators: the test suite does not ensure a minimum coverage of class mutations.

Even after reducing the number of equivalent mutants, 27.9% of the valid class mutants across all SUTs are still considered equivalent (357 out of 1,279). We have to note that some mutants are classified as equivalent because we could not find a way to reach the mutation. For instance, in the case of *EHC* in *KMyMoney*, we were unable to throw an exception that reached the catch block. There are other equivalent mutants related to memory management, as in some mutants from *CDD*, which might be only killable under certain memory restrictions. All the valid mutants from the operator *PMD* are considered as equivalent; among the class operators creating at least one mutant for the analyzed SUTs, *PMD* is the only one producing no useful mutants.

The mutation scores in these tables show that mutation testing can help improve the test suite. We have analyzed the surviving non-equivalent mutants, and included new

```
1  bool XmlRpcServerConnection::executeMulticall(
2        const std::string& methodName, XmlRpcValue& params,
3        XmlRpcValue& result){
4    ...
5    try{
6     if( !executeMethod(methodName, methodParams, resultValue[0])
7      && !executeMulticall(methodName, params, resultValue[0]) ){
8        ...
9     }
10   }catch(const XmlRpcException& fault){
11       ...
12   }
13 }
```

Figure 4: Method "executeMulticall" in *XmlRpc++*

test cases or scenarios following the procedure explained in Section 5.3. We have added new tests within our possibilities, due to the demanding nature of the work involved in creating new tests to kill mutants of third-party libraries.

Table 16 shows the original size of the test suite, the additions made, and the size of the final augmented test suite. $|S|$ is the number of test scenarios, $|C|$ is the number of test cases, and $|M|$ is the number of modified scenarios. This table also shows the mutation score (*MS*) computed with the final test suite. We have achieved a class-adequate test suite for the programs *Matrix TCL Pro* and *XmlRpc++*.

In addition to accomplishing the main goal of mutation testing, we have found some defects in the analyzed code thanks to the technique:

- While removing the `SetAttribute(float)` method implemented in the `XMLAttribute` class, we detected that it was not reachable by clients of the `XMLElement` class. `XMLElement` only has a `double` variant for its `SetAttribute(const char*, type)` methods, so only `XMLAttribute::SetAttribute(double)` is reachable from it. This is also a problem when performing shallow clones in `XMLElement`, since it reuses the 2-argument `SetAttribute` methods. In short, this forces all floating-point attributes to use `double` values.

- While trying to design a test case that threw an exception in line 7 of Figure 4, we detected a case of infinite recursion in *XmlRpc++*. The code seems to have been designed to allow executing multiple invocations by iterating through a data structure, but it is not correctly implemented. The method calls itself without changing the value of `params`, resulting in infinite recursion and eventually a segmentation fault.

Answer to RQ3: According to the mutation scores in Tables 11-15, the test suites distributed with the SUTs cannot completely detect the mutations proposed by the class operators. After inspecting the surviving mutants, we designed new tests (see Table 16) to form better test suites, which are used to answer the next research question.

6.4. Usefulness of Class Mutation Operators

Table 17 has been included to answer if class mutants or traditional mutants can subsume the other in some way.

Table 17: Average Mutation Scores for Traditional and Class Mutants over 30 Class-Adequate and 30 Test-Adequate Test Suites

SUT	**Class-Adequate Traditional MS**	**Traditional-Adequate Class MS**
TCL	0.84	0.97
RPC	0.90	0.98
TXM	0.93	0.94
KMY	0.52	0.90
KAP	0.81	0.87
Mean	0.80	0.93
SD	0.16	0.05

The column *Class-Adequate Traditional MS* contains, for each SUT, the average mutation score associated to traditional mutants when applying different adequate test suites for the set of class mutants. Conversely, the column *Traditional-Adequate Class MS* shows the average scores for class mutants using adequate test suites with respect to traditional ones. As it can be seen, the results favor traditional operators because, with a mean score of 80% in the SUTs, class mutants do not cover traditional mutants. However, traditional operators are not able to completely subsume class mutants either. While these results suggest that only few class mutants cannot be killed through traditional mutants, it is also true that there are far fewer class mutants than traditional ones (see Section 6.2).

In a second experiment, we have calculated the metric T_d (defined in Section 5.3) after obtaining minimal test suites for each of the programs. The results are shown in Table 18, where D is the result of $|TM(M_{t\cup c})|-|TM(M_t)|$ and N is the number of scenarios in D that were modified or added to improve the test suite in Section 6.3.

Table 18: Calculation of Metric T_d with the Improved Tests for the Analyzed SUT

SUT	$\lvert TM(M_c)\rvert$	$\lvert TM(M_t)\rvert$	$\lvert TM(M_{t\cup c})\rvert$	**D**	**N**	T_d
TCL	15	21	24	3	3	0.12
RPC	15	22	23	1	1	0.04
TXM	15	31	37	6	4	0.16
KMY	36	77	90	13	7	0.14
KAP	24	49	56	7	6	0.12

In every SUT, the set of class-level mutants provides at least a test scenario to $TM(M_{t\cup c})$ which is not included in $TM(M_t)$, despite being $|TM(M_t)| > |TM(M_c)|$ in all the cases. It is also interesting to observe that, in the case of *Matrix TCL Pro*, the minimal test suite matches with the complete test suite (24 test scenarios) only when considering both the class-level and the traditional mutants. This fact illustrates that the two types of mutation operators complement each other when improving a test suite.

The value of T_d is quite similar for all the analyzed programs, except for *XmlRpc++*. These values can be seen as low, but we have to remark that T_d indicates the proportion of test scenarios that appear in the minimal test suite exclusively when considering the class-level mutants. The remaining scenarios in $TM(M_c)$ are already included in $TM(M_{t\cup c})$ because of the traditional mutants.

As commented in Section 5.3, when improving the test suite, we tried to include test scenarios which were not completely specific to kill a concrete class mutant. The new test scenarios appearing in $TM(M_t)$ reflect this fact: several new tests are included in the minimal test suite for the traditional mutants. For instance, we created 8 scenarios for *XmlRpc++* (see Table 16), but 7 of them are also in $TM(M_t)$. In the same line, we computed the column N to check if there were any tests in D that belonged to the original test suite. This happens with three programs, most notably with *KMyMoney*, where 6 of the 13 scenarios in D are from the original test suite.

Table 19: Calculation of Metric Q_D for the Set of Killed Class and Traditional Mutants from the Analyzed SUT and the Improved Test Suite

SUT	$\lvert K_c\rvert$	$\lvert K_t\rvert$	Q_{DC}	Q_{DT}	**Dif**
TCL	115	10,402	0.95	0.96	−0.01
RPC	87	1,064	0.94	0.90	0.04
TXM	310	1,017	0.88	0.91	−0.03
KMY	193	1,744	0.99	0.99	0.00
KAP	104	1,595	0.99	0.97	0.02

Table 19 shows the calculation of the metric Q_D for the killed class mutants (K_c) and the killed traditional mutants (K_t), obtaining Q_{DC} and Q_{DT} respectively. *Dif* is the result of $Q_{DC} - Q_{DT}$. When $Dif > 0$, then the results for K_c are of higher quality than K_t; in contrast, when $Dif < 0$, then the results for K_t are of higher quality. With this metric, we have checked that there is not a significant difference in any case study after improving the test suite using the class mutants that remained alive. Indeed, Q_{DT} is even higher than Q_{DC} in *Matrix TCL Pro* and *Tinyxml2*. This shows that T_d values are not due to the design of new tests only with regard to the surviving class mutants.

Answer to RQ4: On one hand, in all the analyzed SUTs, some scenarios only appeared in the minimal test suite when the class mutants were considered (see Table 18). On the other hand, the minimal test suites for the traditional mutants have been consistently larger than the minimal test suites for the class-level mutants. This suggests that while there is some overlap between class-level and traditional mutants, they do not subsume each other fully, but rather complement each other.

6.5. Threats to Validity

The different behavior of the operators in each application makes it difficult to provide final conclusions about the obtained results. The OO features of the language are used with varying frequency, so several operators require further research as they did not produce a significant number of mutants. Moreover, many of the improvement rules to enhance the effectiveness of the operators were not applicable to the SUTs. Subsequently, the implemented rules could produce major improvements in operator efficiency in some cases, while having no effect in others. Since the

equivalence problem is undecidable, the identification of these mutants is a manual task, and the mutation scores shown may be inaccurate.

As for the construct validity, the restrictions to avoid invalid, equivalent and trivial mutants have been studied for the set of operators, identifying several of the unproductive mutations in a subset of them. However, further reduction of unproductive mutants may still be possible by establishing new rules, which could be detected applying the technique to other programs with different features that foster the appearance of new situations.

The mutation score may greatly vary depending on the operators because of the few mutants inserted within a class, as commented in Section 6.3. We spent considerable effort in creating valuable mutants through the improvement rules; several trivial mutants were avoided, which may have reduced this score. Not all the mutants generated in the applications were later studied because of the time-consuming and hard task of analyzing class mutants and designing new test scenarios.

As for the experiment to answer RQ4, we have to remark that, being T_d dependent on the test suite, the results may change with different test suites. In this regard, the results yielded by this experiment could have been different if the test suites were adequate both for class-level and traditional mutants. In the same line, the results are also dependent on the tests added to increase the mutation score, although we designed the tests to be as general as possible to reduce this kind of threat. Finally, we have to note that Q_D should be calculated with an adequate test suite, but when the test suite does not meet with this requirement, this metric gives us an approximation to the quality of the dead mutants with that test suite.

7. Related Work

There have been many studies on the utility and significance of mutation testing. In this regard, we can mention several: Offutt et al. [38] pointed out that "16% more faults can be detected using mutation adequate test sets than all-use test sets", and Daran et al. [39] stated that 85% of the errors simulated with mutants in an industrial program were produced by real mistakes. Likewise, Andrews et al. [40] concluded, from empirical results using four types of mutation operators in C, that the mutations inserted were similar to real faults but different from the manually seeded ones, which were harder to detect than real faults. These facts underpin the importance of the technique and the main underlying idea behind mutation operators: to detect common mistakes when developing software applications.

As mentioned in Section 2.2, no broad empirical analyses of the application of class-level mutation testing to C++ had been conducted so far. Existing studies in this vein have used Java and C# [23]. One of the first analyses was made by Lee et al. [22], studying the orthogonality of the class mutation operators compiled by Ma et al. for Java [5], and also the distribution of the mutants stemming from large programs. Experimental results show that class operators could reveal many faults while producing few mutants in comparison to traditional operators in procedural programs. Ma et al. found that, for the same applications, the traditional operators produced about twice as many mutants as the class-level operators [8]. In our experiments, the gap between the mutants generated by both sets of operators is even wider. Over 70% of the class-level mutants in [8] were equivalent, whereas the traditional operators usually generated 5-15% of equivalent mutants. The analyzed programs regarding class operators in Section 6 reported that 27.9% of the valid mutants from several operators were equivalent.

In any case, the results for unit testing of procedural languages are broader and even more conclusive than the ones for OO languages: Segura et al. [30] reported that 45.4% of the generated mutants were equivalent using their class operators for Java, which is quite different from the aforementioned results of Ma et al. [8] and our experiments, finding other contradictory results in related literature. These mixed data can be explained by the different behavior of the operators in the tested programs; class mutation operators are less frequently used and depend on the program characteristics. Thus, some operators with high percentage of equivalence in the study by Ma et al. [8] did not produce any mutants in the experiments conducted by Segura et al. [30] and in this paper.

The impact of equivalence has been widely studied, for example by Grun et al. [41]. Hence, a variety of works have tried to alleviate this issue: from the first heuristics for detecting equivalence [42], to the use of a coevolutionary algorithm to discard equivalent mutants in the process through a fitness function [43]. The first hints on the automation of rules to avoid class mutants which are known to be equivalent were proposed for three class operators [22]. Equivalence conditions were then extended by Offutt et al. [7] for sixteen operators; almost 75% of equivalent mutants on average were identified and their generation was prevented for sixteen operators. The report with the definitions of the mutation operators for Ada [16] mentions several decisions made to avoid trivial mutants in their implementation. Our paper provides a collection of rules to detect equivalent, invalid and trivial mutants, putting forward some general ideas not only for C++, but for all class operators in different languages. In our study, 32.1% of mutants from the analyzed SUTs were not generated because of the implemented restrictions.

Several techniques have been proposed to generate fewer mutants, including mutant sampling [44], high-order mutation [45] and mutant clustering [46]. Another common technique for mutant reduction is selective mutation, which has been mainly applied to procedural languages like FORTRAN [27] or Ada [38]. More recently, Derezińska and Rudnik [9] studied different selective strategies regarding traditional and class-level operators for C#, finding that 93% of the original mutation score could be obtained

with a reduction in the number of mutants from 18 class operators (74%) and the number of tests (14%). Still, the results of selective mutation were better when using 8 traditional operators. Bluemke and Kulesza [47] applied a selective reduction of mutants by operator to Java, also including traditional and class operators. Their experiments, which only included 8 classes, also pointed to acceptable results regarding both mutation score and code coverage.

Other mutation systems implement dynamic optimizations, such as the analysis of infection and propagation states in Major [48] or the use of code coverage information in Javalanche [49] or Bacterio [50]. The authors of Major reported an average reduction of 40% in mutation analysis times, and Bacterio also showed a significant reduction in the number of executions. In the case of Bacterio, only the class operators that did not change the structure of the program could leverage this information, due to their use of mutant schemata. *MuCPP* does not use mutant schemata, but rather creates one separate program for each mutation, and therefore could adopt a similar approach without running into that limitation.

In our previous work, an initial approach on different aspects to consider for the development of this mutation tool was provided [11], and a general set of class operators were defined and applied to two libraries [10]. In this paper, however, the features of the mutation system have been presented in depth, reporting data about its performance. We have also shown a detailed distribution of the generated class mutants. Additionally, we have compared the mutants produced with class and traditional operators, providing evidence that class operators add valuable information to mutation testing.

8. Conclusion

In this paper, the fundamentals for the application of mutation testing to the C++ programming language have been developed, allowing us to accomplish the construction of the mutation testing system called *MuCPP* for this language. Firstly, different categories of mutation operators at the class level have been introduced; these operators have been defined and implemented according to the specific characteristics of this language. Secondly, the work presented here removes the obstacles regarding the complex task of automating mutation in C++ by developing a feasible and comprehensive solution for any SUT through the traversal of the AST generated with Clang. *MuCPP* has been devised to handle the intrinsic complexity of the language. Among the most remarkable issues, the system avoids the generation of duplicate mutants, allows the use of a compilation database containing the commands to compile the different source code files, and saves space on disk by creating the mutants as branches in the Git version control system.

The correct definition and implementation of mutation operators that provide valid and useful mutants is key to successful mutation testing. Operator quality has been enhanced by establishing a specific scope for the application of each operator, which discards unproductive mutants. With this approach, fewer equivalent mutants are generated, and the effectiveness and efficiency of the mutation operators can be improved. In the conducted experiments, mutants were reduced by 32% on average with the improved operators, and equivalent mutants dropped to 27.9%. Computational costs were lowered with these class-level operators, as they produced far fewer mutants than traditional operators.

A quantitative study of the mutants generated in 475 classes belonging to five well-known libraries and programs of different sizes was performed. The figures reveal the most prolific operators (*PCI* or *MCO*), which should be used cautiously. In contrast, the operators that seem to emerge infrequently depend on the features used by the SUTs. Hence, a priori, we do not recommend to eliminate them from the set of operators. The experiment also points out that *CID*, *OMR*, and *OMD* are the operators with the highest rate of mutated classes, and that inheritance is a determining factor in the generation of class mutants.

The class-level operators showed their usefulness, helping find missing and important test cases in the test suite distributed with the SUTs. We were able to augment the test suite using the surviving class mutants for every SUT, which constitutes an important contribution for further studies. We also found some defects in the code along the way thanks to these mutants. The results indicate that the OO systems should be tested considering the particularities of the OO paradigm. The experiments carried out provide evidence that these operators not only motivate the tester to create new tests, but also that they add valuable information with respect to traditional operators. As a conclusion, traditional and class mutants can complement each other when designing more complete test suites.

As for further research, new improvement rules could be detected in the operators. However, the compilation and execution times shown in the largest SUTs reveal that other mutant reduction techniques could be needed to alleviate the high computational cost. In this regard, evolutionary algorithms, parallel computation and mutant schemata are techniques to explore. Moreover, coverage analysis of the test suite could help prevent unnecessary executions. We also plan to evaluate the quality of each of the class-level operators separately, using metrics such as those proposed by Estero-Botaro et al. [29]: the results could allow us to obtain a sufficient set of class-level mutation operators for C++. Finally, C++ is quickly evolving and new standards should be studied for creating new class-level operators or adapting existing ones to new semantics or syntax.

Acknowledgment

This paper was funded by the Spanish Ministry of Economy and Competitiveness (National Program for Research, Development and Innovation), through the project DArDOS (TIN2015-65845-C3-3-R) and the Excellence Network SEBASENet (TIN2015-71841-REDT), and by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cádiz.

[1] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, Software Engineering, IEEE Transactions on 37 (5) (2011) 649–678.
URL http://dx.doi.org/10.1109/TSE.2010.62

[2] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, Encyclopedia of Information Science and Technology, Third Edition, IGI Global, 2014, Ch. Mutation Testing, pp. 7212–7221.
URL http://dx.doi.org/10.4018/978-1-4666-5888-2.ch710

[3] R. Gopinath, C. Jensen, A. Groce, Mutations: How close are they to real faults?, in: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, 2014, pp. 189–200.
URL http://dx.doi.org/10.1109/ISSRE.2014.40

[4] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, ACM, New York, NY, USA, 2014, pp. 654–665.
URL http://dx.doi.org/10.1145/2635868.2635929

[5] Y. S. Ma, Y. R. Kwon, J. Offutt, Inter-class mutation operators for Java, in: S. Kawada (Ed.), Proceedings of XIII International Symposium on Software Reliability Engineering, IEEE Computer Society, Annapolis (Maryland), 2002, pp. 352–363.
URL http://dx.doi.org/10.1109/ISSRE.2002.1173287

[6] A. Derezińska, Quality assessment of mutation operators dedicated for C# programs, in: P. Kellenberger (Ed.), Proceedings of VI International Conference on Quality Software, IEEE Computer Society, Beijing (China), 2006, pp. 227–234, ISSN 1550-6002.
URL http://dx.doi.org/10.1109/QSIC.2006.51

[7] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of MuJava, in: Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06, ACM, New York, NY, USA, 2006, pp. 78–84.
URL http://dx.doi.org/10.1145/1138929.1138945

[8] Y.-S. Ma, Y. R. Kwon, S.-W. Kim, Statistical investigation on class mutation operators, ETRI Journal 31 (2) (2009) 140–150.
URL http://dx.doi.org/10.4218/etrij.09.0108.0356

[9] A. Derezińska, M. Rudnik, Quality evaluation of object-oriented and standard mutation operators applied to C# programs, in: C. Furia, S. Nanz (Eds.), Objects, Models, Components, Patterns, Vol. 7304 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 42–57.
URL http://dx.doi.org/10.1007/978-3-642-30561-0_5

[10] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, F. Palomo-Lozano, Class mutation operators for C++ object-oriented systems, Annals of telecommunications - annales des télécommunications 70 (3-4) (2015) 137–148.
URL http://dx.doi.org/10.1007/s12243-014-0445-4

[11] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, 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, pp. 151–156.

[12] R. G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering 3 (4) (1977) 279–290.

[13] R. DeMillo, R. Lipton, F. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41.

[14] H. Agrawal, R. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, E. Spafford, Design of mutant operators for the C programming language, Tech. rep., Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana (Mar. 1989).

[15] K. N. King, A. J. Offutt, A FORTRAN language system for mutation-based software testing, Software: Practice and Experience 21 (7) (1991) 685–718.
URL http://dx.doi.org/10.1002/spe.4380210704

[16] A. J. Offutt, J. Voas, J. Payne, Mutation operators for Ada, Tech. Rep. ISSE-TR-96-09, George Mason University, Fairfax, Virginia, Information and Software Systems Engineering, George Mason University (1996).

[17] M. R. Woodward, Mutation testing - its origin and evolution, Information and Software Technology 35 (3) (1993) 163–169.
URL http://dx.doi.org/10.1016/0950-5849(93)90053-6

[18] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo, GAmera: an automatic mutant generation system for WS-BPEL compositions, in: R. Eshuis, P. Grefen, G. A. Papadopoulos (Eds.), Proceedings of the 7th IEEE European Conference on Web Services, IEEE Computer Society Press, Eindhoven, The Netherlands, 2009, pp. 97–106.

[19] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, E. Wong, Mutation testing applied to validate specifications based on Petri Nets, in: Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII, Chapman & Hall, Ltd., London, UK, 1996, pp. 329–337.
URL http://dx.doi.org/10.1007/978-0-387-34945-9_24

[20] S. Kim, J. A. Clark, J. A. McDermid, The rigorous generation of Java mutation operators using HAZOP, in: Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99), Paris, France, 1999.

[21] S. Kim, J. Clark, J. McDermid, Class mutation: Mutation testing for object-oriented programs, in: Proc. Net.ObjectDays, 2000, pp. 9–12.

[22] H.-J. Lee, Y.-S. Ma, Y.-R. Kwon, Empirical evaluation of orthogonality of class mutation operators, in: Software Engineering Conference, 2004. 11th Asia-Pacific, 2004, pp. 512–518.
URL http://dx.doi.org/10.1109/APSEC.2004.49

[23] A. Derezińska, A. Szustek, Object-oriented testing capabilities and performance evaluation of the C# mutation system, in: Advances in Software Engineering Techniques, Springer, 2012, pp. 229–242.
URL http://dx.doi.org/10.1007/978-3-642-28038-2_18

[24] A. Derezińska, Object-oriented mutation to assess the quality of tests, in: EUROMICRO Conference, 2003. Proceedings. 29th, IEEE Computer Society, Belek, Turkey, 2003, pp. 417–420.
URL http://dx.doi.org/10.1109/EURMIC.2003.1231626

[25] M. Kusano, C. Wang, CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications, in: 28th International Conference on Automated Software Engineering (ASE), 2013 IEEE/ACM, IEEE, 2013, pp. 722–725.
URL http://dx.doi.org/10.1109/ASE.2013.6693142

[26] D. Lin, Principle-based parsing without overgeneration, in: Proceedings of the 31st Annual Meeting on Association for Computational Linguistics, ACL '93, Association for Computational Linguistics, Stroudsburg, PA, USA, 1993, pp. 112–120.
URL http://dx.doi.org/10.3115/981574.981590

[27] E. S. Mresa, L. Bottaci, Efficiency of mutation operators and selective mutation strategies: an empirical study, Software Testing, Verification and Reliability 9 (4) (1999) 205–232.
URL http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X

[28] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Quantitative evaluation of mutation operators for WS-BPEL compositions, in: Third International Conference on Software Test-

ing, Verification, and Validation Workshops (ICSTW), 2010, pp. 142–150.
URL http://dx.doi.org/10.1109/ICSTW.2010.36

[29] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, Quality metrics for mutation testing with applications to WS-BPEL compositions, Software Testing, Verification and Reliability.
URL http://dx.doi.org/10.1002/stvr.1528

[30] S. Segura, R. M. Hierons, D. Benavides, A. Ruiz-Cortés, Mutation testing on an object-oriented framework: An experience report, Information and Software Technology 53 (10) (2011) 1124–1136, special Section on Mutation Testing.
URL http://dx.doi.org/10.1016/j.infsof.2011.03.006

[31] A. J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of 15th International Conference on Software Engineering, 1993, 1993, pp. 100–107.
URL http://dx.doi.org/10.1109/ICSE.1993.346062

[32] R. Just, F. Schweiggert, G. Kapfhammer, MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler, in: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 612–615.
URL http://dx.doi.org/10.1109/ASE.2011.6100138

[33] Y. Jia, M. Harman, MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language, in: Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference, 2008, pp. 94–98.
URL http://dx.doi.org/10.1109/TAIC-PART.2008.18

[34] D. Spinellis, Global analysis and transformations in preprocessed languages, IEEE Transactions on Software Engineering 29 (11) (2003) 1019–1030.
URL http://dx.doi.org/10.1109/TSE.2003.1245303

[35] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 99–118.
URL http://dx.doi.org/10.1145/227607.227610

[36] R. Just, F. Schweiggert, Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators, Software Testing, Verification and Reliability 25 (5-7) (2015) 490–507.
URL http://dx.doi.org/10.1002/stvr.1561

[37] G. Kaminski, P. Ammann, J. Offutt, Improving logic-based testing, Journal of Systems and Software 86 (8) (2013) 2002–2012.
URL http://dx.doi.org/10.1016/j.jss.2012.08.024

[38] A. J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Software Practice and Experience 26 (2) (1996) 165–176.
URL http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K

[39] M. Daran, P. Thévenod-Fosse, Software error analysis: A real case study involving real faults and mutations, SIGSOFT Software Engineering Notes 21 (3) (1996) 158–171.
URL http://dx.doi.org/10.1145/226295.226313

[40] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, 2005, pp. 402–411.
URL http://dx.doi.org/10.1145/1062455.1062530

[41] B. J. M. Grün, D. Schuler, A. Zeller, The impact of equivalent mutants, in: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 192–199.
URL http://dx.doi.org/10.1109/ICSTW.2009.37

[42] D. Baldwin, F. Sayward, Heuristics for Determining Equivalence of Program Mutations, Department of Computer Science: Research report, Yale University, Department of Computer Science, 1979.

[43] K. Adamopoulos, M. Harman, R. M. Hierons, How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, in: GECCO (2)'04, 2004, pp. 1338–1349.
URL http://dx.doi.org/10.1007/978-3-540-24855-2_155

[44] T. A. Budd, Mutation analysis of program test data, Ph.D. thesis, Yale University (1980).

[45] Y. Jia, M. Harman, Higher order mutation testing, Information and Software Technology 51 (10) (2009) 1379–1393.
URL http://dx.doi.org/10.1016/j.infsof.2009.04.016

[46] S. Hussain, Mutation clustering, Master's thesis, King's College London (2008).

[47] I. Bluemke, K. Kulesza, Reduction in mutation testing of Java classes, in: 9th International Conference on Software Engineering and Applications (ICSOFT-EA), 2014, 2014, pp. 297–304.
URL http://dx.doi.org/10.5220/0004992102970304

[48] R. Just, M. D. Ernst, G. Fraser, Efficient mutation analysis by propagating and partitioning infected execution states, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 315–326.
URL http://dx.doi.org/10.1145/2610384.2610388

[49] D. Schuler, A. Zeller, Javalanche: Efficient mutation testing for Java, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 297–298.
URL http://dx.doi.org/10.1145/1595696.1595750

[50] P. R. Mateo, M. P. Usaola, Reducing mutation costs through uncovered mutants, Software Testing, Verification and Reliability 25 (5-7) (2015) 464–489.
URL http://dx.doi.org/10.1002/stvr.1534