Class Mutation Operators for C++ Object-Oriented Systems

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

the date of receipt and acceptance should be inserted later

Abstract Mutation testing is a fault injection testing technique around which a great variety of studies and tools for different programming languages have been developed. Nevertheless, the mutation testing research with respect to C++ is pending. This paper proposes a set of class mutation operators related to this language and its particular object-oriented (OO) features. In addition, an implementation technique to apply mutation testing based on the traversal of the abstract syntax tree (AST) is presented. Finally, an experiment is conducted to study the operator behaviour with different C++ programs, suggesting their usefulness in the creation of complete test suites. The analysis includes a Web Service (WS) library, one of the domains where this technique can prove useful, considering its challenging testing phase and that C++ is still a reference language for critical distributed systems WS.

Keywords Mutation testing, mutation operators, C++, object-oriented programming, abstract syntax tree, Web Services

1 Introduction

Mutation testing is a fault-based technique assessing the effectiveness of a test suite to detect faults within the code [13]. This technique involves inserting simple syntactic changes in the program under test using *mutation operators*, obtained from the analysis of the most common mistakes made by programmers. These modifications create new programs called *mutants*. For instance, $x > 1$ can be turned into $x < 1$ by a mutation operator replacing relational operators. A good test suite should be able to detect any changes affecting the program behaviour, i.e., making the mutant and the original program produce different outputs.

Mutation testing has been studied and successfully applied to several languages of diverse nature. However, its development around C++, an industrial-strength *object-oriented* (OO) language, is immature. As a result, mutation testing for C++ is underrepresented when compared to other programming languages [9]. The correct choice of the set of mutation operators is key to successful mutation testing and it must be specifically designed according to the unique features of each language. Regarding C++, only some typical faults concerning OO features have been enumerated [4]. Nevertheless, class mutation operators have been defined for other languages like Java [14] and C# [5].

C++ is a general-purpose language used in an assortment of application domains. Moreover, class mutation operators can be applied to any OO system. Thus, one of the areas where this technique can be really useful is Service-centric Systems (ScS) [2]. On one hand, mutation testing has been performed on Web Service (WS) compositions [1,6], black-box testing them at the interface level. The approach in this paper goes further by mutating the C++ code in order to leverage the benefits of applying mutation testing to the individual services as well. On the other hand, testability of this kind of systems is limited by many factors and is more challenging than in traditional systems. In this regard, the

Pedro Delgado-Pérez · Inmaculada Medina-Bulo · Juan José Domínguez-Jiménez · Antonio García-Domínguez · Francisco Palomo-Lozano
UCASE Software Engineering Group. Department of Computer Science and Engineering, University of Cádiz.
C/ Chile nº1, 11002 Cádiz, Spain

Pedro Delgado-Pérez
Phone: +34 956 015780
E-mail: pedro.delgado@uca.es

experiments in this paper have been conducted on several C++ programs, including a real C++ WS-library implementing the XML-RPC protocol.

Hence, this paper aims to introduce a set of mutation operators for C++ at the class level (by mutating OO features) and evaluate the kind of mutants generated with them. Section 2 looks in depth at the issue of mutation operators, the C++ characteristics and the need for testing WS. The next section deals with the defined set of mutation operators. Section 4 exposes how injecting faults into code can be accomplished with the help of its *abstract syntax tree* (AST), despite the difficulty of implementing mutation testing for this particular language. Section 5 shows the results from the experiments: firstly, a subset of operators related to object construction and destruction is explored to support their usefulness; secondly, the distribution of mutants generated for each program provides an approximation to the usefulness of operators when trying to create a good test suite to kill their mutants. Finally, the last section presents the conclusions and future research lines.

2 Background and Related Work

2.1 Mutation Operators

Mutation operators are associated with typical categories of errors arising when using a particular language. Several of these categories are common to many languages, but each language possesses certain features making a specific study necessary. Thus, many works have been devoted to define sets of operators for a variety of languages and, more important, some tools automating the mutant generation have been developed [9], such as *MuJava* for Java, *Proteum/IM 2.0* for C, and *SQLMutation* for SQL.

Mutation testing has originally focused on procedural programs, developing mutation operators for languages like C or Fortran. Nevertheless, these operators for procedural programs, known as traditional or standard operators, are insufficient to test OO programs: they own features like encapsulation, inheritance, and polymorphism, providing a new scope for potential faults. As the presence of the OO paradigm rose, mutation testing research regarding its characteristics increased as well. Most of the studies concerning this paradigm have been carried out around Java [14] and, in a smaller proportion, around C# [5]. In [4], Derezińska listed several common mistakes for the OO features of C++, but did not define a formal set of mutation operators. Hence, defining a set of class mutation operators for C++ is required.

For the C++ language, some commercial tools exist, such as PlexTest or Insure++, as shown in [9]. However, these products do not implement any class mutation operators, which are the focus of this paper. In contrast, they perform some simple mutations (for instance, PlexTest only removes instructions), using the technique in a selective way.

In order to define this set of operators, the most used features and common programming mistakes in C++ have been studied [8]. The *C++03* standard is taken as reference because it is soon to define operators for the *C++11* standard (it is not widely used yet). At the same time, contributions in other languages have been analysed, chiefly around Java [10,12,14], because this language has drawn the attention of multiple studies, and also C# [5]; this fact offers insight into the nature of the mistakes that programmers frequently make. These languages are syntactically similar, taking Java much of the C++ syntax but removing many of the low-level facilities (the main differences between these two languages are listed in [8]). On the other hand, C# basic syntax is influenced by C/C++ as well as Java in its object model.

2.2 C++ and Web Services

C++ is a multiparadigm language, which is backward compatible with a large fragment of C and includes features from the OO paradigm among other programming styles. As a result, this language is used in a wide range of applications and mutation testing can allow for an enhancement of the software quality in a variety of domains. Thus, the application of this technique to C++ WS can be fruitful, taking into account that errors in ScS are difficult to locate once deployed. Regarding the reliability measurement of these ScS, a considerable amount of research have been undertaken [2]. However, much of the testing work developed is specification-based as access to the service code is generally limited or simply non-existing. As an illustration, model checking has been used to test C++ WS, checking the high-level safety and liveness properties [15]. Concerning mutation testing, it has been applied to WSDL [1], as well as to WS-BPEL compositions [6].

C++ continues to preserve low-level facilities like pointers, omitted in other languages. Thus, it is important to differentiate between a pointer to an object and the object it actually points to. This aspect often causes mistakes when referring objects, especially if using dynamic allocation. Moreover, dynamic binding, which is used to introduce polymorphic behaviour, is not as simple as in other languages. These errors are also produced during object construction, entailing memory allocation and initialisation of every member, and object destruction (both managed by the programmer), which are well-known sources of faults [8]. The complex memory management model along with other characteristics, such as overriding members in the class hierarchy, method overloading, and exception handling, may confuse programmers coming from a procedural language background or even some programmers used to mainstream OO languages.

3 Description of Mutation Operators

3.1 Class Mutation Operators for C++

Mutation operators have been classified in seven categories according to the main OO characteristics; the usual sources of error, commented in Section 2.2, have been considered to define these groups and their mutation operators. Each category is identified by an upper case letter:

1. Access control: **A**
2. Inheritance: **I**
3. Method overloading: **O**
4. Polymorphism and dynamic binding: **P**
5. Exception handling: **E**
6. Object and member replacement: **M**
7. Miscellany: **C**

The naming convention to identify mutation operators is three upper case letters: the first one denotes the category, while the rest identify the operator within the category. Categories and operators are resumed in Table 1. Henceforth, several operators marked with '*' and used in Section 5 will be exposed, whereas the definition of the rest of operators can be found in the references provided in Section 3.2.

Information hiding or access control

Mutation operators in this group intend to confirm the correct accessibility.

– **AMC or Access modifier change**: *AMC* checks the correct access control to members. Access levels are determined by sections (*public*, *protected*, *private*) and as many sections of each level as desired can be added. This operator transfers the member to a block with a different access level.
– **AAC or Inheritance access modifier change**: When a class inherits from another one, it is possible to determine the access privileges by specifying an access modifier. This operator changes the access modifier when inheriting to ensure the assigned access is correct:

```
Example AAC:                  Mutant 1:
   class A{                      class B: protected A{
      ... ...                       ... ...
   };                            };
                               Mutant 2:
   class B: public A{            class B: private A{
      ... ...                       ... ...
   };                            };
```

Inheritance

Operators applied to inheritance, mainly with respect to the presence of overridden members, are included in this group.

– **ISI or Base keyword insertion:** *ISI* ensures the correct member is being referenced when a member in the subclass hides a variable or overrides a method of one of its ancestors. In the example below, it can be observed how the *scope resolution operator (::)* is employed to refer to a base class:

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

– **IPC or Explicit call of a parent's constructor deletion:** *IPC* removes the explicit call to a parent constructor so that the default constructor is used. The constructor of a parent class is invoked within the initialisation list of a constructor (see *CID* operator).
– **IOP or Overriding method calling position change:** This operator simulates the error that often occurs when calling a method of a base class, which is overridden in the child class, at the wrong time, producing an undesired state.
– **IOR or Overridden method rename:** This operator acts when an overriding method interacts with a parent's version (see example below). This situation can only occur when that method is declared *virtual*. In this way, the overriding method can be called from a method in a parent class when the binding is dynamic:

```
Example IOR:
        class A{
            ... ...
            virtual void m1() {//body of m1}
            void m2() {... m1(); ...}
        };
        class B: public A{
            ... ...
            void m1() {... ...}
        };
Mutant:
        class A{
            ... ...
            virtual void m1() {//body of m1}
            virtual void m3() {//body of m1}
            void m2() {... m3(); ...}
        };
        class B: public A{
            ... ...
            void m1() {... ...}
        };
```

– **IMR or Multiple inheritance replacement:** Multiple inheritance is supported in C++. When a derived class inherits from two or more classes, it may occur that those base classes have member variables with the same name or methods with the same signature. Thus, the programmer can be mistaken when referencing a certain inherited member. That is the fault modelled by *IMR*:

```
Example IMR:                      Mutant:
    class A{                          class A{
        ... ...                           ... ...
        int a;                            int a;
    };                                };
    class B{                          class B{
        ... ...                           ... ...
        int a;                            int a;
    };                                };
    class C: public A,                class C: public A,
             public B {                        public B {
        ... ...                           ... ...
        void m () {                       void m () {
            ... ...                           ... ...
            b = A::a + 1;                     b = B::a + 1;
        }                                 }
    };                                };
```

Method overloading

Operators in this group ensure that a method calling invokes the correct method when overloading is employed.

– **OAN or Argument number change:** This operator mutating the number of arguments in method invocations should consider the possibility of using *default parameters*. If a method has just another overloaded method, as in the example below, then only one mutant can be generated; but if the overloaded method has a default parameter, a further one can be created:

```
Without default parameter:
   class A{
       ... ...
       void m (int a) {... ...}
       void m (int a, float b) {... ...}
   };

   Example:                 Mutant:
       a.m(0,0);                a.m(0);

With default parameter:
   class A{
       ... ...
       void m (int a = 1) {... ...}
       void m (int a, float b) {... ...}
   };
   Example OAN:             Mutant 1:
       a.m(0,0);                a.m(0);
                             Mutant 2:
                                a.m(); (1)
   (1) a.m(1) is actually invoked.
```

Polymorphism and dynamic binding

This block is composed of operators checking that the polymorphic mechanism is used in the right way.

– **PCI or Type cast operator insertion:** The role of this operator is to cast an object reference, turning its actual type into the parent or child of the original declared type. In a case like the example below, the invoked method may be non-virtual, although the base class needs to be polymorphic to allow the *downcasting*:

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

Example PCI:          Mutant:
B b;                  B b;
A *pa = &b;           A *pa = &b;
pa->m();(1)           (dynamic_cast<B*>(pa))->m();(2)

(1) A::m() is invoked
(2) B::m() is invoked
```

– **PPD or Parameter variable declaration with child class type:** This operator, which targets the parameters of a method, changes the declared type of an object reference to a parent class type.

– **PVI or *virtual* modifier insertion:** Whenever a method in a class is intended to have a polymorphic behaviour, the programmer must indicate it by adding the *virtual* modifier. Forgetting to insert the *virtual* keyword is contemplated with this operator.

Table 1 Summary of categories and mutation operators at the class level

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

*Legend: Operators marked with * are original or have been changed with respect to their original definition or implementation in other languages.*

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

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

Exception handling

Improper handling of exceptions (although exceptions are not unique to this paradigm, they are closely related to it) is treated in this block.

– **EHC or Exception handling change:** *EHC* removes the exception handling statement. The exception will not be caught within the method, but it will be propagated to the nearest handler. This case can be achieved through a relaunch of the exception so that it is caught and handled, hopefully, at a higher level:

Example EHC:
```
int f(){
    try{
        ... ...
    }catch(Handler1){
        ... ...
    };
}
```
Mutant:
```
int f(){
    try{
        ... ...
    }catch(Handler1){
        throw;
    };
}
```

Object and member replacement

Operators in this category are dedicated to the replacement of the object invoking a member or to the change of the member invoked, by a compatible object or member respectively.

– **MCO or Member call from another object:** When an object member variable is referenced and calls a method, *MCO* replaces that reference to the object by another variable of the same class type (the invoked method is not changed):

Example MCO:
```
class A{          class B{
  ...               ...
  void method();    A a1;
};                  A a2;
                    void m(){...a1.method();...}
                  };
```
Mutant:
```
class A{          class B{
  ...               ...
  void method();    A a1;
};                  A a2;
                    void m(){...a2.method();...}
                  };
```

– **MCI or Member call from another inherited class:** This operator is similar to *MCO* in the sense that it makes that a method is invoked from a different object, but now the objects are from different class types, both having the same base class.

– **MBC or Member changed:** *MBC* accesses a different instance variable (with the same type) when a member variable is referred.

Miscellany

This block contains operators related to different specific C++ characteristics.

– **CID or Member variable initialisation deletion:** *CID* removes the initial value given to member variables, checking thereby that the proposed initialisation is correct. Initial values are assigned in the constructors, either in the body or using *initialisation lists*. Thus, if the initialisation is within the body of the constructor, the assignment statement is deleted, while, if it is within the initialisation list, then that element is removed from the list:

```
Example CID:    A::A(): a(0) { b = 1;}
Mutant 1:       A::A(): a(0) {}
Mutant 2:       A::A() {b = 1;}
```

– **CCA or Copy constructor and assignment operator overloading deletion:** The task of copying objects is accomplished through the definition of a copy constructor and, usually, the assignment operator overloading too (when they are not defined, the compiler provides them automatically). This operator deletes the defined copy constructor or the assignment operator overloading, checking they are correctly implemented:

Example CCA:
```
    class A{
        ... ...
        A(const A& copy){... ...}
        A& operator =(const A& copy){... ...}
    };
```
Mutant 1:
```
    class A{
        ... ...
      // A(const A& copy){... ...}
        A& operator =(const A& copy){... ...}
    };
```
Mutant 2:
```
    class A{
        ... ...
        A(const A& copy){... ...}
      // A& operator =(const A& copy){... ...}
    };
```

– **CDD or Destructor method deletion:** C++ allows the programmer to define not only how the objects are constructed, but also how they are destroyed. If a destructor is not specified, the compiler automatically provides one. *CDD* deletes the destructor checking its correct implementation:

Example CDD:
```
class A{
    ... ...
    ~A(){... ...};
};
```
Mutant:
```
class A{
    ... ...
  // ~A(){... ...};
};
```

3.2 Comparison with other Languages

Summarising, a set of 37 operators has been conformed for C++ at the class level, distributed in seven categories. This number of operators is higher than the one presented in [14] for Java (29) and the same as the one shown in [5] for C# (37 operators without counting the invalid ones).

The operator *AMC* [3,10,14] is different in C++ because the access level is specified by sections and not individually as in Java. Most operators in the *inheritance* category have been defined for Java [3,10,14] and taken in C# [5]. However, *ISI, ISD, IPC* and *IOP* change with respect to Java as they are related to the *super* keyword, which does not exist in C++ because of the multiple inheritance. Thus, not only a single direct base class has to be considered, but every inherited class. Regarding *method overloading, OMR*, *OMD*, *OAO* and *OAN* are based on [12] and *OPO* is based on *POC* from [10], adapting its name to the established convention. All operators from [14] in the *polymorphism* group have been considered with a similar meaning but different implementation (as commented in Section 2, the usage of pointers and references to dynamically bind the objects is necessary).

Concerning *exception handling*, a definition of *EHR* and *EHC* can be found in [10] and *EXS* in [5]. In Java, the function of *EHC* is achieved using a *throws* declaration instead of the *try-catch* statement. Besides, in C++ the *finally* clause is not used and the exception could be ultimately captured in the `main` function instead of the class `Object`. *MCO*, *MCI*, *MNC* and *MBC* are all named in [5] and the fault that they simulate is shown in [4] (in *Object* and *Member* blocks). For this category, only an explicit definition for *MNC* has been found in [3]. Finally, some operators in *miscellany* take as reference the *Java-specific* group in [14]. Regarding *CID*, an initial value cannot be assigned directly to members as in Java, but in the constructors.

4 Applying Mutation Operators

4.1 Approach

Parsing C++ code to determine where the operators can be used is one of the main obstacles in the construction of a mutation tool. Firstly, some important techniques used to develop mutation tools for Java and C# are not available in C++, such as the *reflection* mechanism (which allows us to examine the type and state of objects at runtime) or the insertion of faults in the bytecode. Thus, mutations have to be performed on the code. Secondly, the complexity of C++ does not ease the task of parsing source code for detecting where the operators can be applied. At the same time, ensuring that an ad-hoc parser (e.g., based on pattern-matching) can cover every syntactic construct is really difficult.

These considerations lead the authors to *reuse the AST internally produced by a compiler to analyse and transform the code* according to the criteria defined in the mutation operators. The AST is an *intermediate form* that a compiler generates to represent the source code; it provides a simplified and clear structure of the code focusing on the essential aspects, facilitating the traversal of the tree as nodes containing the language elements are processed.

Hence, the AST traversal makes the harsh labour of detecting the locations where a change can be introduced easier. In addition, 100% of the features of the language covered by the compiler are guaranteed to be manageable with this approach.

4.2 Implementation Process of a Mutation Operator

The *Clang* compiler[1] has been chosen to process the AST. Unlike others, this compiler has a well-designed API, allowing us to reuse its libraries to parse C++ code. These libraries have been used to implement the mutation operators, automating the process of traversing the tree and finding the nodes that conform to the rules defined in the operator. The whole process follows these steps:

1. Using a domain-specific language (DSL) to create predicates in *Clang*, a pattern is created to search for nodes of the tree which could be mutated. For instance, a pattern for the operator *CID* looks for members that are initialised in the initialisation list.
2. Then, selected nodes are analysed to avoid cases in which the mutation would produce a mutant that could not be compiled. As for *CID*, it needs the position of the initialisation within the list to know if a comma or the colon preceding the list has to be deleted.
3. Now, mutation can be applied. In the case of *CID*, the node and the appropriate characters (as found in the previous step) are deleted.
4. Finally, the tree is translated back into source code form through the *Clang* libraries.

This process is implemented through the *Visitor* pattern. As a full-fledged compiler, *Clang* provides an AST with all the information that the mutation operators may need. The AST generated by *Clang* is easily understandable, does not implicitly simplify the code

[1] http://clang.llvm.org

and saves information about every token, among other advantages.

5 A Discussion of Mutation Operators

5.1 Experiment Setup

An experiment was prepared to analyse several C++ programs using the obtained class mutation operators. The goal in Section 5.2 is to illustrate, with concrete cases, different situations where various operators can be useful. In Section 5.3 and 5.4, the objective is to examine the distribution of mutants for OO systems and measure the effectiveness of their tests.

Two programs were chosen from the *LLVM-3.2 test-suite*[2], containing pieces of code written in C/C++ handling the diverse language constructs:

- *Garage*, with a class modelling a parking where two kind of vehicles are parked and released.
- *Family*, with three classes simulating the hierarchy "grandfather-father-son", sharing some attributes.

Two known open-source libraries were used to apply the operators in real applications, including a library for WS:

- *Tinyxml2*[3] to parse XML documents.
- *XmlRpc++ (ver. 0.7)*[4], to incorporate XML-RPC client and server support into C++ applications.

Both programs could work together in a side of a client-server communication, the former parsing XML files and the latter sending/receiving them via HTTP protocol. Different measures of their characteristics are shown in Table 2.

In Section 5.2, the mutants generated with a subset of operators in the aforementioned programs were analysed to evaluate their quality in the composition of a test suite. For the second and main part, the two real applications were considered, classifying the kind of mutants with all the operators presented. The faults modelled by the operators were introduced into the code resorting to the procedure explained in Section 4.2. Then, each mutant was independently run against the test suite to see the response of its execution. If the mutant produces a different output from the original program, for at least one test case, it is classified as *killed* or *dead* (in the opposite case, the mutant is still *alive*). Some mutants always produce the same output as the original program for any input: these are said to be *equivalent*. Finally, if the mutant gives a compilation error, it is considered as *invalid*.

Table 2 Size statistics of *Tinyxml2* and *XmlRpc++*

Measure	*Tinyxml2*	*XmlRpc++*
LoC[a]	2,620	2,194
Classes	18	13
Methods (mean)	17.8	11.9
Attributes (mean)	3.1	4.5
Inheriting classes	8	5
Inherited members (mean)	32.4	6

[a]Counted with *c_count* 7.14 (`http://invisible-island.net/c_count`)

Regarding the test suite, several test scenarios (a series of test cases using objects of one or more classes) were created for *garage* and *family* testing their main functionalities. In the case of *tinyxml2* and *xmlrpc++*, the diverse test scenarios that were distributed with the programs were employed to yield the results. The execution of the tests in *xmlrpc++* was not done in the conventional manner as both server and client had to be run so that communication was possible. Equivalent mutant determination is an undecidable problem, so they were manually identified.

5.2 Usefulness of Operators

The operators *CDD, CCA* and *CID*, related with the construction and destruction of objects, are studied in this section; they refer to language elements that have some distinguishing features compared to the rest of methods and are much used as they are always invoked whenever an object is built or destroyed respectively. Moreover, they are error prone as commented in Section 2.

CDD mutants are mostly "potentially" equivalent because the destructor is usually invoked just to release memory. The word "potentially" is used because an anomalous behaviour concerning the memory can only be detected when memory is a limited resource. In this case, the mutants can be killed if the memory is not released properly. Nevertheless, this experiment shows that a destructor also performs other actions that can be tested in new scenarios. In *tinyxml2*, two mutants were killed because the destructor is used to unlink a pointer; as the pointer is not handled in the destructor of the mutant, the change can be detected by a test case checking the pointer. In the case of *xmlrpc++*, the deletion of the destructor also affects the execution of a mutant because a pointer to a boolean is not given the appropriate value.

CCA mutants are usually equivalent as well because the copy constructors are often similar to the default

[2] http://llvm.org/releases/3.2/docs/TestingGuide.html
[3] https://github.com/leethomason/tinyxml2 (Last access: 03/2014)
[4] http://xmlrpcpp.sourceforge.net/ (Last access: 03/2014)

one. Nonetheless, this operator can suggest the inclusion of new scenarios performing a copy of objects when this constructor is somewhat different. Regarding the *family* program, a specific scenario copying an object of class `Parent` was included, and the mutant was killed when the destructor was executed. The original version reserves a new block of memory for the copied object. In the mutant, both objects involved in the copy pointed to the same address, producing an error when trying to free the same block of memory twice.

The *CID operator* tends to create many mutants. Some mutants in *tinyxml2* were easily killed because a member pointer was not initialised in the constructor. Additionally, when the variable `_allocated` of the class `DynArray` was not initialised properly, a problem with memory allocation could be detected: the value assigned to a member by the compiler can be unexpected. Sometimes, this undefined behaviour will be captured in the execution, but in other situations, a new scenario with a timer can be introduced checking how long it lasts. For instance, in *garage,* when the variable `maxVehicles` of the class `Garage` is not initialised, the *for loop* below takes a different time depending on the value assigned to that variable:

```
Garage::Garage(int max) {
    // CID initialisation deletion: maxVehicles = max;
    parked = new Vehicle* [maxVehicles];
    for (int bay = 0; bay < maxVehicles; ++bay)
        parked[bay] = NULL;
}
```

As an interesting fact, the case of some mutants of *CID* in *tinyxml2* can be mentioned; this program was executed against 112 test cases, but 3 of the mutants generated with this operator were killed by a single test case (different in each case). This information demonstrates that some faults can be difficult to locate and enforces the need of a complete test suite and, therefore, the usefulness of this approach. The situations explained above prove that these operators have potential in revealing faults not covered by other operators, often requiring particular test cases.

5.3 Experimental Results

For the analysis of the distribution of mutants, the applications *tinyxml2* and *xmlrpc++* were mutated. Regarding *tinyxml2,* all the mutants generated were considered; in contrast for *xmlrpc++*, the mutants related with *log* and *error* reports were discarded for being secondary functionalities as well as the classes uncovered by the test suite were excluded (the terms *discarded* and *uncovered* are taken from [16]). The classification of the mutants obtained is depicted in Table 3 and 4 for *tinyxml2* and *xmlrpc++* respectively. "Generated" counts the number of mutants including invalid ones, which are deducted in "Total". "Dead" indicates the amount of mutants resulting dead. The remaining alive mutants (fourth column) were studied, calculating then how many were equivalent. The last column shows the *mutation score*: the ratio between the killed and non-equivalent mutants.

However, not every non-equivalent mutant can be said to be killable; the alive *CDD* mutants can only be killed under certain conditions (see Section 5.2); in the case of *EXS*, an exception not considered in the program is needed to be thrown, being this an open question. Besides, the alive mutants in *AAC* will be invalidated with adequate tests. The equivalence and mutation score was not studied in *MBC* and *MNC*; they are supposed not to suffer from equivalence unless two methods in a class perform the same action or two variables have the same value occasionally, but these cases are not significant (they completely depend on the program implementation).

Concerning the WS-library *xmlrpc++*, mutants usually died after producing not very enlightening errors or blocking indefinitely while waiting for the other side of the communication. To catch this second situation, a timeout was set to stop execution after a reasonable time. Hence, from a client side, the origin of these problems is difficult to determine, supporting the idea that intensive testing is needed in this kind of applications.

5.4 Discussion and Related Studies

Firstly, it should be noted that the amount of mutants stemming from class mutation operators is much lower than for traditional statement-level operators, as stated in [14] and also in [16], where the number of class-level mutants produced was 60% lower than their traditional counterparts; the language constructs modified by these operators are not as much used as arithmetic or logical operators. Thus, ten operators in the set did not create any mutant in this experiment. Therefore, the results obtained with traditional operators for other languages should not be compared with the results presented here, but with studies around class mutants, where similar numbers can be found [11,16]. In this sense, the practical application of the technique is more affordable.

Other common conclusion in related studies is the high equivalence percentage: while traditional operators generate between 5-15% of equivalence, this percentage was largely increased using class operators in the same programs [14,16]. However, the statistics calculated are

Table 3 Mutant classification in *Tinyxml2*

Oper.	**Gen**	**Tot**	**Dead**	**Alive**	**Equiv**	**MS**
AAC	16	0	-	-	-	-
AMC	738	443	0	443	428(96.6%)	0%
IHI	48	47	12	35	32(68.1%)	80%
ISD	1	1	1	0	0(0%)	100%
ISI	1	0	-	-	-	-
IPC	6	0	-	-	-	-
IOD	47	25	21	4	1(4%)	87.5%
IOP	8	8	8	0	0(0%)	100%
IOR	10	10	7	3	1(10%)	77.8%
OMD	68	31	9	22	16(51.6%)	60%
OPO	46	23	4	19	13(56%)	40%
PCD	12	6	4	2	2(100%)	100%
PCI	774	533	393	140	62(11.6%)	83.4%
PCC	5	5	0	5	1(20%)	0%
PPD	29	6	3	3	3(50%)	100%
MCO	19	19	13	6	0(0%)	68.4%
MCI	39	39	11	28	0(0%)	28.2%
MNC	399	331	189	142	-	-
MBC	469	469	279	190	-	-
CID	62	62	43	19	0(0%)	69.4%
CDC	4	4	3	1	0(0%)	75%
CDD	14	14	2	12	8(57.1%)	-
CCA	4	4	0	4	4(100%)	-
Total	2806	2080	1002	1078	571(34.7%)	66.9%

Legend: Gen: Generated; Tot: Total; Equiv: Equivalent;
MS: Mutation score; "-": indicates a statistic not calculated

Table 4 Mutant classification in *XmlRpc++*

Oper.	**Gen**	**Tot**	**Dead**	**Alive**	**Equiv**	**MS**
AAC	10	4	0	4	-	-
AMC	436	210	0	210	210(100%)	-
IHI	13	13	2	11	11(84.6%)	100%
ISD	2	2	0	2	2(100%)	-
ISI	3	3	0	3	3(100%)	-
IPC	3	1	1	0	0(0%)	100%
IOD	8	3	0	3	3(100%)	-
IOR	15	15	0	15	15(100%)	-
OMD	30	9	6	3	0(0%)	66.7%
OMR	15	10	6	4	0(0%)	60%
OAN	7	7	0	7	0(0%)	0%
OPO	2	1	0	1	0(0%)	0%
PCD	3	0	-	-	-	-
PCI	49	5	4	1	1(20%)	100%
PPD	7	1	0	1	1(100%)	-
EHC	2	2	0	2	0(0%)	0%
EXS	2	2	0	2	-	-
MCO	48	48	20	28	0(0%)	41.6%
MNC	436	370	215	155	-	-
MBC	200	200	85	115	-	-
CID	16	15	10	5	0(0%)	66.7%
CDC	2	2	0	2	0(0%)	0%
CDD	8	8	1	7	3(37.5%)	-
CCA	2	2	2	0	0(0%)	100%
Total	1320	933	352	581	249(39,1%)	52.9%

Legend: Gen: Generated; Tot: Total; Equiv: Equivalent;
MS: Mutation score; "-": indicates a statistic not calculated

rather different depending on the operators and applications analysed: from 45,4% [16] to 86.45% [11] in recent works. This study reports an average percentage of 36,9%, which is lower than the aforementioned results, but corroborates that equivalence is a prominent issue. Nevertheless, the determination of equivalence was not such a difficult task in several operators because some classes contained common structures, but it is challenging in others like *MNC* or *MBC*.

The study brings out the likeliest operators to be applied in OO systems as well as the often most prolific operators. For instance, the operators in *object and member replacement* seem the most frequently used. As a result, the operators in this category spawned the 41.25% and 66.2% of valid mutants in *tinyxml2* and *xmlrpc++* respectively. However, in general, class mutation operators are applied with varying frequency depending on the language features used in each application, e.g., *PCI* created 774 mutants in *tinyxml2* but only 49 in *xmlrpc++*.

The experiment yields an average mutation score of 59.9%, which means that 40.1% of the mutants need new test cases to be killed. This shows how mutation testing can help find missing tests in real scenarios. The operators with a low mutation score are the most promising because their mutants seem to be harder to detect, like *OAN*, *PCC* and *EHC*, with a score of 0%. In contrast, *ISD*, *IPC*, *IOP*, *PCD*, *PPD* and *CCA* obtained 100%. However, it would be useful to confirm these results by extending the study with additional programs and test suites. As a final remark, the *AMC* operator might need to be reconsidered in future studies, because its mutants have been either equivalent, invalid or killable with the same test cases as *OMD*, confirming the original observations by Offutt et al. for Java [14]. Similarly, *AAC* may require further refinements, as it did not produce any killable mutants.

5.5 Threats to Validity

One of the main challenges in mutation testing is checking if a mutant is equivalent or not: since it is an undecidable problem, it must be done by hand and there is an inherent possibility of error. Another threat is the reliance of mutation testing on the test suites: they will determine the time consumed and the reliability of the results. A test suite making an exhaustive use of every class and their members would broadly reduce the number of alive mutants. For instance, if no object copy were performed in any test case, all the mutants produced with *CCA* would survive.

As for the limitations in the conducted studies, only a subset of all the operators have been analysed in depth in Section 5.2. Therefore, the conclusions cannot be generalised to the entire set of operators. In the same way, not every operator produced mutants from the two applications in Section 5.4, though results about those operators will be obtained in further research. Various simple as well as real and complex applications were chosen to avoid the partial perspective of the individual programs, but the performance of the operators is quite different in each application. Hence, it is not easy to assure that the population studied is representative, so the figures shown should be treated as estimations. The different results reported in other similar analyses prove that the usage of OO features varies greatly from one application to another [16].

6 Conclusions and Future Work

A set of class level mutation operators for the C++ programming language has been introduced. Mutation testing is a language-dependent technique, at least in terms of mutation operators, which presents particular difficulties in the case of C++. In this sense, the work presented here is an important contribution because it defines a set of C++ mutation operators for the first time and it develops a feasible and comprehensive solution to automate mutation in this context. This solution is based on the traversal of the AST internally generated by the *Clang* compiler.

These operators have been analysed through a carefully designed experiment with different OO programs to illustrate the applicability of the technique. In brief, several special operators related to object construction and destruction have been studied in detail, showing that specific tests are usually needed to kill their mutants. The distribution of mutants obtained confirms several results observed in the literature: class mutation operators generate fewer mutants than traditional operators, equivalent mutants are also a relevant issue in C++, and the generalisation of results is difficult because of the dependency on the subjects under study. The results promote the benefits of the technique as the test suites only killed around 60% of the mutants, being even more valuable when developing WS for high-availability systems to avoid unusual errors where the communication was not successfully achieved.

The study of equivalent and invalid mutants can lead to identify different situations always creating non-desirable mutants, which may be prevented in the future. Moreover, it will be interesting to analyse the *Evolutionary Mutation Testing* (EMT) technique [7], which could allow for a reduction of the number of mutants without a significant loss of effectiveness. In addition, the possibility of introducing new operators according to features not covered yet will be explored.

References

1. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: WS-TAXI: A WSDL-based testing tool for web services. In: ICST, pp. 326–335 (2009)
2. Bozkurt, M., Harman, M., Hassoun, Y.: Testing and verification in service-oriented architecture: a survey. Software Testing, Verification and Reliability **23**(4), 261–313 (2013)
3. Chevalley, P.: Applying mutation analysis for object-oriented programs using a reflective approach. In: Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific, pp. 267 – 270 (2001)
4. Derezińska, A.: Object-oriented mutation to assess the quality of tests. In: Proceedings of the 29th Conference on EUROMICRO, pp. 417–420. IEEE Computer Society, Belek, Turkey (2003)
5. Derezińska, A.: Quality assessment of mutation operators dedicated for C# programs. In: P. Kellenberger (ed.) Proceedings of VI International Conference on Quality Software, pp. 227–234. IEEE Computer Society, Beijing (China) (2006). ISSN 1550-6002
6. Domínguez-Jiménez, J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I.: 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, pp. 97–106. IEEE Computer Society Press, Eindhoven, The Netherlands (2009)
7. Domínguez-Jiménez, J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I.: Evolutionary mutation testing. Information and Software Technology **53**(10), 1108–1123 (2011)
8. Horstmann, C., Budd, T.: Big C++, 2nd Edition. Wiley (2009)
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on **37**(5), 649–678 (2011)
10. Kim, S., Clark, J., McDermid, J.: Class mutation: Mutation testing for object-oriented programs. In: Proc. Net. ObjectDays, pp. 9–12 (2000)
11. Ma, Y.S., Kwon, Y.R., Kim, S.W.: Statistical investigation on class mutation operators. ETRI Journal **31**(2), 140–150 (2009)
12. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators for Java. In: S. Kawada (ed.) Proceedings of XIII International Symposium on Software Reliability Engineering, pp. 352–363. IEEE Computer Society, Annapolis (Maryland) (2002)
13. Offutt, A.J., Untch, R.H.: Mutation testing for the new century. chap. Mutation 2000: uniting the orthogonal, pp. 34–44. Kluwer Academic Publishers, Norwell, MA, USA (2001)
14. Offutt, J., Ma, Y.S., Kwon, Y.R.: The class-level mutants of MuJava. In: K. Anderson (ed.) Proceedings of the 2006 International Workshop on Automation of Software Test, pp. 78–84. ACM, Shanghai (China) (2006)
15. Qi, Z., Liu, L., Zhang, F., Guan, H., Wang, H., Chen, Y.: FLTL-MC: Online high level program analysis for web services. In: SERVICES I, pp. 171–178 (2009)
16. Segura, S., Hierons, R.M., Benavides, D., Ruiz-Cortés, A.: Mutation testing on an object-oriented framework: An experience report. Information and Software Technology **53**(10), 1124 – 1136 (2011)