UNIVERSIDAD DE CÁDIZ

TESIS DOCTORAL

Prueba de Mutación Evolutiva en Entornos Orientados a Objetos

Evolutionary Mutation Testing in Object-Oriented Environments

Autor:

Pedro Delgado Pérez

Directores:

Dra. Inmaculada Medina Bulo

Dr. Juan José Domínguez Jiménez

Escuela Superior de Ingeniería

Programa de Doctorado en Ingeniería y Arquitectura

Fecha: 21 de abril de 2017

Conformidad de los Directores

Dª María Inmaculada Medina Bulo, profesora del Departamento de Ingeniería Informática de la Universidad de Cádiz, y D. Juan José Domínguez Jiménez, profesor del Departamento de Ingeniería Informática de la Universidad de Cádiz, siendo Directores de la Tesis titulada *Prueba de Mutación Evolutiva en Entornos Orientados a Objetos*, realizada por D. Pedro Delgado Pérez y enmarcada en el Programa de Doctorado en Ingeniería y Arquitectura, para proceder a los trámites conducentes a la presentación y defensa de la tesis doctoral arriba indicada, en aplicación de la Normativa Reguladora de Estudios de Tercer Ciclo de la Universidad de Cádiz, informan que se autoriza la tramitación de la tesis.

Los directores de tesis

Inmaculada Medina Bulo

Juan José Domínguez Jiménez

Cádiz, a 21 de abril de 2017

Agradecimientos

Es casi imposible nombrar a la cantidad de personas que pusieron un granito de arena en el desarrollo de esta tesis doctoral. Os doy las gracias a todos, en especial a:

- *Inmaculada y Juanjo*, mis directores de tesis: nunca imaginé que aquella primera reunión en la sala de profesores de la antigua ESI nos llevaría hasta este punto. Ahí quedan miles de correos (alguno que otro a horas intempestivas) y de experiencias compartidas. Gracias por apostar por mí.
- *Mis padres y hermanos*, por su incomprensión. Sí, lo he dicho bien: sin comprender lo más mínimo de lo que hacía, me apoyaron y se interesaron a sabiendas de que no se enterarían de nada de lo que dijese por más que me esforzase. Ahí está la grandeza de su apoyo.
- *Mis amigos de siempre*, por cada hora que, quedando con vosotros, me quitasteis de avanzar en la tesis. Al día siguiente, me ayudaron a aprovechar cada hora por dos.
- *Mis amigos por el mundo* que pasan horas y horas discurriendo como yo. Sí, a los sufridores PhD, porque pensar en la próxima siempre fue un aliciente.
- *Antonio García*. Esta tesis apoya la hipótesis de que la paciencia es la madre de todas las ciencias (no se pudo localizar referencia bibliográfica).
- *A los miembros del grupo UCASE*, por hacerme sentir como un miembro más desde el minuto uno. La ayuda de cada uno de vosotros, en diferentes sentidos, ha sido fundamental.
- *Sergio Segura*, por ser casi como un segundo co-director y por aportarme sabiduría y pasión por la investigación.
- *Louis Rose y Ibrahim Habli*, por las enriquecedoras experiencias vividas en sendas estancias.

Thank you very much to all of you!

Agradecimientos Institucionales

Este trabajo fue financiado por la beca de investigación PU-EPIF-FPI-PPI-BC 2012-037 de la Universidad de Cádiz, por el proyecto DArDOS (TIN2015-65845-C3-3-R) del Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad del Ministerio de Economía y Competitividad, y por la Red de Excelencia SEBASENET (TIN2015-71841-REDT) del Programa Estatal de Fomento de la Investigación Científica y Técnica de Excelencia del Ministerio de Economía y Competitividad.

"Nosotros, los mortales, logramos la inmortalidad en las cosas que creamos en común y que quedan después de nosotros."

Albert Einstein

UNIVERSIDAD DE CÁDIZ

Abstract

Escuela Superior de Ingeniería
Departmento de Ingeniería Informática

por Pedro Delgado Pérez

Mutation testing is acknowledged as a powerful method to evaluate the strength of test suites in detecting possible faults in the code. However, its application is expensive, which has traditionally been an obstacle for a broader use in the industry. While it is true that several techniques have shown to greatly reduce the cost without losing much effectiveness, it is also true that those techniques have been evaluated in limited contexts, especially in the scope of traditional operators for procedural programs. To illustrate this fact, Evolutionary Mutation Testing has only been applied to WS-BPEL compositions, despite the positive outcome when selecting a subset of mutants through an evolutionary algorithm with the aim of improving a test suite. As a result, it is unknown whether the same benefits can be extrapolated to other levels and domains.

In particular, we wonder in this thesis to what extent Evolutionary Mutation Testing is also useful to reduce the number of mutants generated by class mutation operators in object-oriented systems. More specifically, we focus on the C++ programming language, since the development of mutation testing with regard to this widely-used language is clearly immature judging from the lack of papers in the literature tackling this language. Given that C++ has been hardly addressed in research and practice, we deal with all the phases of mutation testing: from the definition and implementation of mutation operators in a mutation system to the evaluation of those operators and the application of Evolutionary Mutation Testing among other cost reduction techniques.

We define a set of class mutation operators for C++ and implement them in *MuCPP*, which allows us to perform experiments with real programs thanks to the facilities incorporated into this mutation tool. These mutation operators are automated following a set of guidelines so that they produce the expected mutations. In general, class-level operators generate far fewer mutants than traditional operators, a higher equivalence percentage and they are applied with varying frequency depending on the features of the tested program. Developing improvement rules in the implementation of several mutation operators help further reduce the number of mutants, avoiding the creation of uninteresting mutants. Another interesting finding is that the set of class mutants and the set of traditional mutants complement each other to help the tester design more effective test suites.

We also develop *GiGAn*, a new system to connect the mutation tool *MuCPP* and a genetic algorithm to apply Evolutionary Mutation Testing to C++ object-oriented systems. The genetic algorithm allows reducing the number of mutants that would be generated by *MuCPP* as it guides to the selection of those mutants that can induce the generation of new test cases (strong mutants). The performance of this technique shows to be better than the application of a random algorithm, both when trying to find different percentages of strong mutants and also when simulating the refinement of the test suite through the mutants selected by each of these techniques. The stability of EMT among different case studies and the good results of the simulation in the programs that lead to the largest set of mutants are additional observations.

Finally, we conduct an experiment to assess individually these mutation operators from a double perspective: how useful they are for the evaluation of the test suite (TSE) and its refinement (TSR). To that end, we rank the operators using two different metrics: degree of redundancy (TSE) and quality to guide on the generation of high-quality test cases (TSR). Based on these rankings, we perform a selective study taking into account that the less valuable operators are at the bottom of the classification. This selective approach reveals that an operator is not necessarily as useful for TSE as for TSR, and that these rankings are appropriate for a selective strategy when compared to other rankings or the selection of mutants randomly. However, favouring the generation of individual mutants from the best-valued operators is much better than discarding operators completely because each of the operators targets a particular object-oriented feature. Altogether, these evaluations about class operators suggest that their nature can limit the benefits of any cost reduction technique.

UNIVERSIDAD DE CÁDIZ

Resumen

Escuela Superior de Ingeniería
Departmento de Ingeniería Informática

por Pedro Delgado Pérez

La prueba de mutaciones es reconocida como un potente método para evaluar la fortaleza de un conjunto de casos de prueba en la detección de posibles fallos en el código. No obstante, la aplicación de esta técnica es costosa, lo cual ha supuesto normalmente un obstáculo para una mayor acogida de la misma por parte de la industria. Varias técnicas han mostrado ser capaces de reducir ampliamente su coste sin mucha pérdida de efectividad, pero también es cierto que estas técnicas solo han sido evaluadas en determinados contextos, especialmente en el ámbito de los operadores de mutación tradicionales para programas procedurales. Por ejemplo, la Prueba de Mutación Evolutiva ha sido aplicada únicamente a composiciones WS-BPEL, a pesar de que se obtuvo un resultado positivo al seleccionar un subconjunto de mutantes a través de un algoritmo evolutivo a fin de mejorar el conjunto de casos de prueba. Como resultado, se desconoce a día de hoy si los mismos beneficios pueden extrapolarse a otros niveles y dominios.

En particular, en esta tesis nos preguntamos hasta qué punto la Prueba de Mutación Evolutiva es también útil para reducir el número de mutantes en sistemas orientados a objetos. Más específicamente, nos enfocamos en el lenguaje de programación C++, ya que la prueba de mutaciones casi no se ha desarrollado respecto a este popular lenguaje a juzgar por la falta de artículos de investigación en este campo que se dirigen este lenguaje. Dado que C++ ha sido apenas abordado en cuanto a investigación y en cuanto a la práctica, en esta tesis nos ocupamos de todas las fases de la prueba de mutaciones: desde la definición e implementación de operadores de mutación en un sistema de mutaciones, hasta la evaluación de esos operadores y la aplicación de la Prueba de Mutación Evolutiva entre otras técnicas de reducción del coste.

En esta tesis definimos e implementamos un conjunto de operadores de mutación de clase para C++ en *MuCPP*, herramienta de mutaciones que nos permite llevar a cabo experimentos con programas reales gracias a las características incorporadas a la misma. Estos operadores de mutación son automatizados siguiendo un conjunto de reglas para que produzcan los mutantes que se esperan de los mismos. En términos generales, los operadores de clase generan bastantes menos mutantes que los operadores tradicionales, un porcentaje mayor de mutantes equivalentes y se aplican con diversa frecuencia dependiendo de las características del programa analizado. El desarrollo de reglas de mejora en la implementación de los operadores permite reducir incluso más el número de mutantes, evitando generar mutantes que no son interesantes para el propósito de la prueba de mutaciones. Otro descubrimiento interesante es que el conjunto de mutantes de clase y el de mutantes tradicionales se complementan, ayudando a diseñar un conjunto de casos de prueba más efectivo.

También desarrollamos *GiGAn*, un nuevo sistema para conectar *MuCPP* y un algoritmo genético para aplicar la Prueba de Mutación Evolutiva a sistemas orientados a objetos en C++. El algoritmo genético permite reducir el número de mutantes que sería generado por *MuCPP* ya que guía la búsqueda a la selección de aquellos mutantes que pueden inducir a la generación de nuevos casos de prueba (mutantes fuertes). El rendimiento de esta técnica se muestra mejor que el de un algoritmo aleatorio, tanto cuando se buscan diferentes porcentajes de mutantes fuertes como cuando se simula el refinamiento del conjunto de casos de prueba mediante los mutantes seleccionados por ambas técnicas. La estabilidad de la Prueba de Mutación Evolutiva en los diferentes programas analizados y los buenos resultados en aquellos programas de los que se deriva un mayor número de mutantes son observaciones adicionales.

Finalmente, realizamos experimentos para evaluar de forma individual a estos operadores de mutación desde una doble perspectiva: cómo de útiles son para la evaluación (TSE) y para la mejora (TSR) de un conjunto de casos de prueba. Para ello clasificamos a los operadores usando dos métricas distintas: el grado de redundancia (TSE) y la calidad para guiar a la generación de casos de prueba de alta calidad (TSR). Siguiendo estas clasificaciones, ponemos en práctica un estudio selectivo teniendo en cuenta que los operadores menos valiosos están en las últimas posiciones. Este enfoque selectivo revela que los operadores no son necesariamente igual de útiles para TSE y TSR, y que estas clasificaciones son apropiadas para llevar a cabo una estrategia selectiva cuando lo comparamos con la aplicación de otras clasificaciones de operadores o la selección aleatoria de mutantes. Sin embargo, favorecer la generación de mutantes individuales a partir de los operadores mejor valorados es mucha mejor opción que descartar operadores al completo debido a que cada uno de estos operadores se centra en una característica concreta del paradigma de orientación a objetos. En conjunto, todas estas evaluaciones

en torno a estos operadores de clase sugieren que la naturaleza de los mismos puede limitar los beneficios de aplicar cualquier técnica de reducción del coste.

Contents

List of Figures

List of Tables

Abbreviations

AST	**A**bstract **S**intax **T**ree
EMT	**E**volutionary **M**utation **T**esting
FOM	**F**irst **O**rder **M**utation
HOM	**H**igher **O**rder **M**utation
TSE	**T**est **S**uite **E**valuation
TSR	**T**est **S**uite **R**efinement

Dedicado a todos aquellos que sacrifican horas de ocio, sueño y dedicación a la familia y amigos con la convicción de que deben emplear su tiempo y esfuerzo en otras labores no tan gratificantes pero igual de importantes.

Chapter 1

Introduction

In this chapter, we describe the motivation behind the preparation of this doctoral thesis and we list the main goals and contributions derived from the research period. Finally, we show the structure and the purpose of each chapter in this dissertation.

1.1 Motivation

Testing is an important activity in the verification and validation of software development. The *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [19], which establishes a baseline for the current knowledge in Software Engineering, defines *Software Testing* among the ten knowledge areas. According to this guide, "software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.", where:

Dynamic means that testing always implies executing the program on selected inputs.

Finite denotes that testing is conducted on a subset of all possible tests because a complete set of tests can generally be considered infinite.

Selected refers to how the test suite is selected.

Expected indicates that it must be possible to decide whether the observed outcomes are acceptable or not.

Mutation testing is a testing technique widely studied by researchers in the last decades as a method to estimate the robustness of test suites [68]. According to Naik and Tripathy [96], "Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases). The original intention behind mutation testing was to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases, and the actual testing of program units is an added benefit."

The research studies on mutation testing have consistently produced evidence of its usefulness to evaluate and improve the quality of test suites, where test quality is measured in terms of the effectiveness at finding faults in the code. This is a stricter requirement than the one imposed by code coverage criteria, which simply validate that all parts of the code have been exercised. In order to improve test quality by means of mutation testing, several faulty versions of the system under test are intentionally produced following some predefined rules, known as *mutation operators*. For example, a mutation operator replacing relational operators may transform $x > 1$ into $x < 1$. Each of these new versions, called *mutant*, contains a simple syntactic change which should be detected if sufficient testing has been performed. These mutants stress the fault detection capability of the test suite: the test suite should be able to show that there is a difference in the outputs of the original and the mutated program. When a test case executed against a mutant reveals its mutation, we say that the mutant has been *killed* or is *dead*. Otherwise, the mutant remains undetected or is *alive*.

SWEBOK [19] states that mutation testing was "originally conceived as a technique to evaluate test sets", but "mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants." In summary, the goals when a system undergoes a mutation testing process are:

1. **Test suite evaluation (TSE)**: Evaluate to what extent the test suite is able to identify faults within the code.
2. **Test suite refinement (TSR)**: Improve the test suite with new test cases based on the inspection of alive mutants.

However, because of the large number of mutants that can be generated even in small-sized programs, mutation testing represents overhead to the testing activity. From an industrial perspective, it is not easy to justify additional expenses. As a consequence, in spite of the efforts in this regard to facilitate the application of mutation testing, the cost has hindered its adoption by practitioners in the past. Several techniques have been suggested to ease the cost of applying mutation testing [100, 125]. While most of them

are useful for TSE, *Evolutionary Mutation Testing* (EMT) [43] was recently presented with a focus on TSR.

Evolutionary Mutation Testing (EMT) aims at generating a reduced set of mutants by means of an evolutionary algorithm. That subset should contain a high proportion of the mutants that may provide the tester with the possibility of adding new test cases to the set, called *strong mutants*. Two types of mutants are considered to be strong mutants: *potentially equivalent*, which are not detected by the test suite under evaluation, and *difficult to kill* mutants, detected by one test case only killing that mutant. EMT was successfully put into practice regarding web services compositions in WS-BPEL [43] with the aid of the *GAmera* system [42], which implements the genetic algorithm. However, this technique has not been assessed in other domains after that. As a result, its applicability to other contexts is an open question. Gaining broader knowledge about the benefits that EMT can provide is the first motivation for this thesis.

At the same time, most of the studies in the literature covering issues related to the cost of mutation testing have been carried out with *traditional* operators. Also known as *standard* operators, they were defined for procedural programs such as C or FORTRAN in the early years of the technique. In general, traditional operators have been widely assessed in comparison with other kinds of mutation operators. The definition of **class-based mutation operators** (or simply class operators) started in 1999 thanks to the increasing popularity of the object-oriented programming. Standard operators, developed in programming environments away from the object-oriented paradigm, do not take into account some types of faults related to object-oriented features. The evaluation of class-based mutation operators has experienced a growth in the last years, but they have not been assessed to the same extent as traditional operators. Therefore, it remains unclear whether the good performance reported when using some cost reduction techniques also apply to operators at the class level. In fact, studies on class operators have shown that they exhibit different features when compared to traditional operators [88, 117]. For instance, class operators generate fewer mutants but a higher equivalence percentage. Consequently, it is interesting to explore the reduction achieved when applying EMT in an object-oriented environment.

On reviewing the sets of class mutation operators defined for object-oriented programming languages, we found that the development of mutation testing with respect to **C++** was underrepresented when compared to other languages (such as Java or C#). As a matter of fact, only several faults regarding object-oriented characteristics of C++ had been listed without defining a formal set of mutation operators [33]. C++ is an industrial-strength multiparadigm language, supporting concepts from both structured

and object-oriented programming. It is also one of the most used programming languages all over the world in a wide range of applications, with object orientation as the most prominent feature. Nevertheless, because of its advanced features and flexibility, programming in this language without mastering its key concepts can be error prone; inexperienced developers may misuse some of its characteristics due to wrong expectations. This clearly motivates the need for adequate testing, so it is puzzling to find that not much attention has been paid to mutation testing in C++ applications.

Consequently, we set as our **final goal applying EMT with class mutation operators for C++**. However, since the development of mutation testing around C++ has been postponed and not even a set of operators has been formally defined, a complete study from the beginning is required to lay the groundwork for the application of mutation testing to C++ programs. As a result, in this thesis we address the three main categories of research regarding mutation testing:

1. Definition and implementation of mutation operators.
2. Evaluation of the utility of mutation operators.
3. Reduction of the cost of mutation testing.

1.2 Aim

The main goals to achieve during the development of this thesis are:

1. **To propose a set of class mutation operators related to C++** and its particular object-oriented features. In order to define this set of operators, the most used and unique features in C++ will be studied. At the same time, contributions in other object-oriented languages will be analysed, mainly in Java and C#, as this fact will offer insight into the nature of the mistakes that programmers frequently make.
2. **To develop a mutation system for C++** in order to systematically analyse source code files written in C++, inject class mutations into the code and execute a test suite against those mutants. An implementation technique to automate the set of class operators in a robust way will be required as well as suitable means to deal with the specific challenges that this language presents.
3. **To analyse the usefulness of the set of mutation operators** in the evaluation (TSE) and refinement (TSR) of test suites for object-oriented systems. This assessment will comprise several aspects:

- Number and distribution of mutants generated.
- Test deficiencies that these operators help identify.
- Comparison with traditional operators.

4. **To develop a system to put into practice Evolutionary Mutation Testing with class mutation operators**. This new system will connect the mutation system for C++ and the genetic algorithm proposed to implement the evolutionary approach in an object-oriented environment.

5. **To evaluate the performance of Evolutionary Mutation Testing when applied to object-oriented systems**. Namely:

   - To measure the reduction of mutants achieved.
   - To compare this technique with the random selection of mutants.
   - To analyse how the genetic algorithm helps improve the test suite.

6. **To study different techniques for the reduction of the number of mutants**:

   - Improvement rules to discard uninteresting mutants in the implementation of operators.
   - Degree of redundancy of mutation operators in TSE.
   - Quality metric of mutation operators in TSR.
   - Selective mutation following an operator-based (selection of a subset of operators) and a mutant-based (selection of a subset of mutants) approach.

1.3 Contributions

The main contributions derived from this thesis are enumerated in this section.

1. **A set of class mutation operators for the C++ programming language.** Despite the importance of this multiparadigm programming language, a set of operators for C++ had not been defined previously as far as we know. This set includes both operators adapted from other languages and new C++-specific operators.

2. **A mutation system for the C++ programming language.** This mutation system, called *MuCPP*, automatically analyses C++ programs, generate the mutants according to the set of mutation operators implemented and execute the test suite against those mutants. We also describe the main features incorporated into the tool to facilitate mutation testing for this language. To the best of our

knowledge, this is the first system devoted to C++ mutation testing implementing a set of mutation operators at the class level.

3. **A method to implement mutation operators and a set of guidelines for the generation of the appropriate mutants**. We describe a robust and comprehensive method to inject mutations into the code through the abstract syntax tree, which avoids practical issues in systems based on the concrete syntax of the language. We also provide a list of requirements that testers should take into account when implementing mutation operators so that they generate the mutants that are expected from them.

4. **A qualitative and quantitative evaluation of the set of class mutation operators.** We show that object-oriented mutation testing can help detect test deficiencies related to particular object-oriented features of C++. We calculate the quantity and distribution of mutants generated with class operators. We also provide a general list of improvement rules for the implementation of class operators, which reduces unproductive class mutants and has a significant impact on the computational cost of the technique.

5. **A comparison between traditional and class-based operators**. Class operators are developed because they address object-oriented features, which were not examined by research studies regarding conventional programming languages. The results confirm that class operators can complement traditional operators and can help testers further improve the test suite.

6. **A system to apply EMT to C++ object-oriented systems**. This system, called *GiGAn*, connects the mutation system *MuCPP* and the genetic algorithm implemented in *GAmera*, allowing the application of EMT to object-oriented programs.

7. **An evaluation of EMT when applied to object-oriented systems**. We go beyond previous experiments by conducting an experimental procedure to assess the improvement of the test suite when applying EMT. This study supports previous studies about EMT when compared to random mutant selection, reinforcing its use for the goal of improving the fault detection capability of the test suite but at a lower cost.

8. **A double assessment of C++ class operators based on their influence during the evaluation (TSE) and the refinement (TSR) of the test suite respectively.** This is the first work assessing mutation operators from this double perspective as far as we know. We apply two different metrics to evaluate the

effectiveness of mutation operators for TSE and TSR, which leads to different classifications of mutation operators associated with these two goals.

9. **A comparison between operator-based and mutant-based selective mutation**. We compare the performance of these two selective strategies, based on the selection of operators and mutants respectively, in the scope of object-oriented mutation testing. The results show that selecting individual mutants from all the operators is more convenient than discarding operators at the class level.

1.4 Thesis Structure

In this section, we briefly comment the content of the chapters that comprise this thesis. The structure of this document is as follows:

- The current chapter, Chapter 1, presents the motivation of the work undertaken, enumerates the goals of this thesis and summarises the main contributions achieved.
- Chapter 2 describes the fundamental aspects and concepts of mutation testing and the state of the art in this research field. We review the works about mutation testing in general and about mutation operators and cost reduction techniques in particular.
- Chapter 3 addresses the first step in mutation testing: the definition of mutation operators. In our case, we define a set of mutation operators at the class level for C++, explaining their main purpose. We also compare this set with existing class mutation operators for other object-oriented programming languages.
- The approach to implement the mutation operators defined in the previous chapter is described in detail in Chapter 4. This chapter looks in depth at the requirements that mutation operators should meet to generate appropriate mutants, defines a list of general rules to improve their effectiveness and also contains an example of the implementation of an operator. Finally, this chapter shows the structure and features of the developed mutation system for C++, which implements the set of class mutation operators.
- Chapter 5 studies class mutation operators from a double perspective: *quantitative analysis* (reduction in the number of mutants through the implemented improvement rules, distribution of mutants and mutation score and test suite improvement) and *qualitative analysis* (utility of these operators, comparison with traditional mutants and detection of coding errors).

- Chapter 6 focuses on Evolutionary Mutation Testing, where this technique is analysed in detail. The system that implements EMT for C++ object-oriented systems is presented. This system is used to evaluate the performance of this cost reduction technique in diverse experiments.

- Chapter 7 continues the analysis of mutation operators at the class level following a selective mutation approach. Mutation operators are studied regarding their contribution to TSE and TSR separately, obtaining different rankings of mutation operators for each of these two goals pursued when using mutation testing. The selective approach is additionally divided into operator-based selection and mutant-based selection based on these rankings, which allows us to analyse whether it is better to discard mutation operators or individual mutants from different operators. The selective study measures the trade-off between the loss of effectiveness and the reduction in the number of mutants.

- Chapter 8 summarises the main results reported (and discussion about them) in the conducted experiments throughout this thesis. Threats to validity of the results are also exposed.

- Chapter 9 collects the conclusions drawn from this research period, and also presents several future research lines. This chapter ends with a list of contributions (journals, book chapters and conferences).

- Appendix A shows relevant features of the case studies used in the different experimental procedures in this thesis.

- Appendix B defines several useful concepts related to the execution of mutants and test suites that are used in this dissertation.

Chapter 2

Concepts and State of the Art

The first purpose of the chapter is to present the main concepts related to mutation testing. The second goal is to look in depth at the development and the current state of this testing technique, mainly in relation with the content of this thesis. We also mention and describe terms that will be used later on in this dissertation.

2.1 Fundamentals of Mutation Testing

Mutation testing is a testing technique based on the injection of simple syntactic changes into the code, following the rules prescribed by a set of *mutation operators*. These mutation operators usually emulate real faults or promote good coding practices. The rationale behind mutation testing is that a good test suite should be able to detect all the changes that are introduced into the code. It is important to note that a test suite is used to find faults within a program, whereas mutation testing is used to find deficiencies in that test suite. There are three main stages when applying mutation testing: *mutant generation*, *test suite execution* and *mutant analysis*. We will explain the fundamental aspects of mutation testing while describing each of these phases.

Mutant generation

In this stage, the source code of the program under test is analysed with respect to the set of mutation operators in order to determine the locations where a fault can be injected.

The original code is then modified to generate faulty versions of the program according to these locations. The new versions of the program are called *mutants*.

As a running example, consider a program with the next statement:

```
Original:    a = b * c;
```

The mutation operator "arithmetic operator replacement" could generate the following four mutants in that statement:

```
Mutant 1:    a = b + c;
Mutant 2:    a = b - c;
Mutant 3:    a = b / c;
Mutant 4:    a = b % c;
```

Each mutant is usually a clone of the original program except for a simple syntactic change: the injected fault. A mutant should represent a valid fault. As such, the resulting code should comply with the language rules. However, mutation operators sometimes generate *invalid mutants*, which are malformed because of the injected fault and infringe the language rules.

Test suite execution

Once generated, the mutants are run on the test suite developed to test the program with the aim of evaluating its fault detection capability. A test case sets the state of the program inducing a particular execution. Consider a single test case, *test case 1*, exercising the statement of our example:

```
Test case 1:    b = 2,   c = 1
```

The outputs after the execution of the mutants allow knowing whether the test suite is able to reveal those possible faults in the code. When the outputs of the original artefact and a mutant differ in at least one test case, the test suite uncovers the mutation and the mutant is *killed*. On the contrary, when the outputs are the same for all the test cases, the mutant is *alive*. Thus, the mutants in our example are classified as follows:

```
Original:    a = 2 * 1;   →   a = 2

Mutant 1:    a = 2 + 1;   →   a = 3    Killed   (a ≠ 2)
Mutant 2:    a = 2 - 1;   →   a = 1    Killed   (a ≠ 2)
Mutant 3:    a = 2 / 1;   →   a = 2    Alive    (a = 2)
Mutant 4:    a = 2 % 1;   →   a = 0    Killed   (a ≠ 2)
```

Three mutants have been killed and one remains alive (*mutant 3*) because the value of the variable "a" is different from the value of this variable in the original program (a = 2).

Mutant analysis

When some of the mutants remain alive because the current test suite is not able to detect the fault that they model, it is the turn for the tester to manually review those surviving mutants. Sometimes the functionality of a mutant and the original program is exactly the same. In that case, we find an *equivalent mutant* and no input data can detect the mutation. Otherwise, the test suite designed for the tested system has failed in detecting faults within the code, that is, the analysis of the mutants reveals some deficiencies in the test suite.

The *mutation adequacy score* is a well-known metric to estimate the fault-revealing power of a test suite. The mutation score is the ratio of the number of dead mutants over the total of non-equivalent mutants:

$$Mutation\ score(P,T) = \frac{K}{M-E} \times 100 \tag{2.1}$$

Where:

- P is the program under test.
- T is the test suite.
- K is the number of killed mutants.
- M is the total number of mutants.
- E is the number of equivalent mutants.

We say that the test suite is *mutant adequate* when the mutation score is 100% (i.e., when it is able to kill the full set of non-equivalent mutants). The higher the mutation score, the higher the test suite quality and therefore its ability to reveal coding errors. Returning to our example, three out of four mutants have been killed and *mutant 3* is not equivalent. Therefore, $K = 3$, $M = 4$ and $E = 0$, so the mutation score associated with the test suite is:

$$\frac{3}{4-0} \times 100 = 75\%$$

A new test case (*test case 2*) can be added to the test suite to kill *mutant 3*:

```
Test case 1 :    b = 2,   c = 1
Test case 2 :    b = 2,   c = 2
```

The execution of *test case 2* against *mutant 3* effectively kills the mutant as shown below:

```
Original :     a = 2 * 2;   →   a = 4

Mutant 3 :     a = 2 / 2;   →   a = 1    Killed    (a ≠ 4)
```

However, we have to note that just checking the state of a variable after the execution of the mutation does not ensure that those mutants are actually killed. Ammann and Offutt [5] proposed the *RIP model*, which establishes the three conditions that a test case needs to meet to kill a mutant:

1. **Reachability**: the test case covers the mutant, that is, the mutated statement is reached or exercised by the test case.
2. **Infection**: the execution of the mutation causes a difference in the internal state of the program.
3. **Propagation**: the infection is not masked after the mutated statement and the difference is also reflected in the output.

In our example, we have seen how a test case achieves the infection. The following fragment illustrates a situation in which the mutated statement is not reached by any of the test cases in our example (reachability):

```
if(c > b){
  a = b * c;
}
```

Likewise, even when the test case is reached and infected, as in the following fragment, the change might not be propagated to the end of the program (propagation):

```
if(b > 0){
  a = b * c;
}
```

```
if(a >= 0){
  a = 1;
}
```

As it can be seen, even though the value of "a" changes because of the mutation, the output is 1 in the end, exactly as in the original program.

2.2 Mutation Testing in the Literature

Mutation testing research dates back to the 1970s from the ideas posed by Hamlet [54] and DeMillo et al. in 1978 [30]. Therefore, the technique has been investigated for almost four decades. Woodward [128] in 1993 collected all the research on mutation testing from those first years. After that, Jia and Harman surveyed the studies related to mutation testing, first in a technical report in 1999 [66], and second in a journal paper two years later [68]. Finally, Offutt [63] also discussed in 2011 the past, present and future of this testing technique.

In general terms, the research on mutation testing can be classified into the following branches:

- Many studies have been dedicated to **produce evidence of the usefulness of mutation testing** to evaluate and improve the quality of test suites, which will be addressed in this section. This activity includes validating the rationale behind mutation testing, its empirical evaluation in real environments and the comparison with structural coverage targets.
- As a white-box testing technique, mutation testing must be specifically designed according to the unique features of each domain. Thus, **this technique has been developed for different programming languages** as new technologies appeared, defining sets of mutation operators and implementing them in several mutation tools. We will review the papers in the literature related to this activity in Section 2.3.
- The experiments conducted applying mutation testing have shown that it can be prohibitively expensive even for small-sized programs, which has hindered its adoption by the industry. Therefore, researchers in this field have proposed multiple **methods to reduce the cost of the application of mutation testing** without lessening its effectiveness significantly, and have evaluated their performance. We will explore this topic in Section 2.4.

Mutation testing has as cornerstones two hypotheses [30]:

- *The competent programmer hypothesis*: Programmers tend to build software close to the correct version. Therefore, this hypothesis explains why most software faults have their origin in subtle defects of the code.
- *Coupling effect hypothesis*: Complex faults relate to simple faults, so a test suite that detects simple faults should also detect most complex faults.

Several works have tried to validate these underlying hypotheses, like Offutt [98] who supported experimentally the validity of the coupling effect. The study by Daran and Thévenod-Fosse [23], addressing safety-critical software, revealed a connection between mutations and real coding errors in a program from the civil nuclear field. In particular, 85% of the injected mutations were also produced by real faults. Just et al. [72] provided evidence that the simple errors introduced into the mutants were related to more complex ones, supporting the coupling effect hypothesis. Andrews et al. [7] applied four mutant types in C to explore the link between hand-seeded and real faults. Their results suggested that manually seeded mutations are different and harder to detect than real faults, whereas mutation operators are more in line with real faults. Nonetheless, the results of the experiments by Gopinath et al. [52] contradicted that hypothesis since real faults appeared to be more complex than most of the mutant types considered in that study.

There are also several studies comparing structural coverage targets and mutation testing as methods to measure test sufficiency. Smith and Williams [119] found that mutation analysis can guide on the augmentation of test suites directed by line and branch coverage tools. Andrews et al. [8] applied mutation testing to evaluate four test coverage criteria: block, decision, c-use and p-use. They showed that mutation testing can help predict the effectiveness of these criteria to detect real faults and their relative cost in terms of fault detection, test suite size and control/data flow coverage. Baker and Habli [10] carried out an empirical evaluation in the safety-critical industry. They analysed the test suites for two different projects in C and Ada, satisfying statement coverage and modified condition/decision coverage (MC/DC) respectively. The experiments revealed a subset of operators that could detect particular deficiencies in the test suites developed for both projects. Iznometsova et al. [61] studied the correlation between coverage (statement, decision and MC/DC), test suite size and effectiveness for large programs in Java. The results gave evidence that test effectiveness is not strongly correlated with coverage criteria, so coverage is not a good indicator of test quality. Yao et al. [131] defined the term stubborn mutant: non-equivalent mutants which are not killed by a test suite satisfying the branch coverage criterion. Using a branch-adequate test suite ensures that

all the mutants have been exercised, so stubborn mutants require non-trivial test cases to be detected. Their experiments revealed that some mutation operators produced many stubborn mutants, whereas other operators produce many equivalent mutants. As such, testers should prioritise those operators generating many stubborn mutants in comparison with the number of equivalent mutants.

2.3 Mutation Operators and Tools

2.3.1 Overview

Mutation operators are associated with typical categories of errors arising when using a particular programming language. Mutation operators are mainly derived from the analysis of the most frequent mistakes made by programmers, so they represent the types of faults tackled by mutation testing. Therefore, a key aspect of mutation testing is defining and implementing the set of mutation operators properly in order to generate useful mutants. The choice of mutation operators depends on the programming language. Several mutation operators are common to different languages that share part of their syntax, but each language possesses particular features making a specific study necessary. Thus, many works have been devoted to defining sets of operators for a variety of languages. Boubeta-Puig et al. [18] recently studied the equivalence of operators among different languages.

In its early years, this technique was developed for a limited number of procedural languages such as C, FORTRAN or Ada. Agrawal et al. [3] defined in 1989 a set of 77 mutation operators for C, divided into four categories: *statement*, *operator*, *variable* and *constant* mutations. Many of the sets of mutation operators for different programming languages are based on this collection of operators. King et al. [78] developed the mutation tool Mothra with 22 operators to apply mutation testing to the FORTRAN language. Offutt et al. [103] compiled a set of 65 operators for Ada. However, the appearance of new languages boosted the research in the late 1990s and shifted the focus to other kinds of domains and levels of abstraction [68]. Hence, in a short period, the technique has been applied to languages of diverse nature. Mutations has also been used at the design level, which is known as "specification mutation" [68], such as Petri Nets [47], Finite State Machines [46] or security policies [89].

Different mutation operators can be defined depending on the characteristics of the program and also depending on the testing activity that the system under test undergoes. Particularly for general purpose languages, such as Java or C++, a set of mutation operators can be defined for each of these levels [68], [91] (see Figure 2.1):

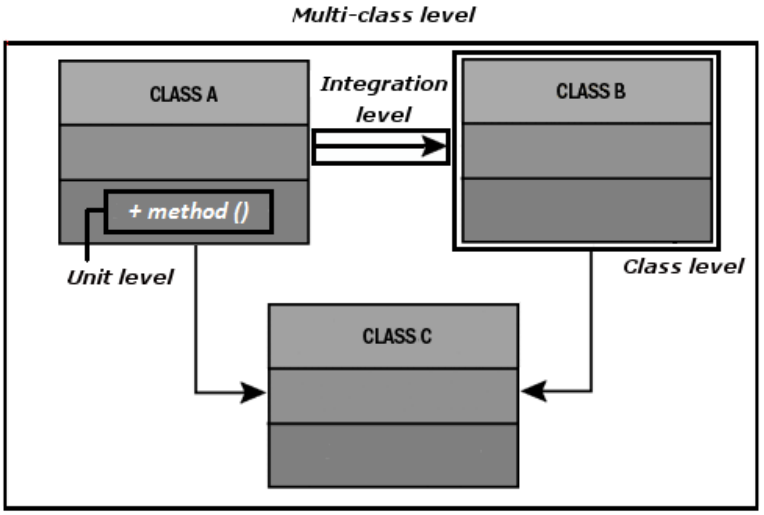


FIGURE 2.1: Mutation testing levels

- **Unit level** [3]: This level refers to conventional mutation testing applied to a function or method to test its functionality. These mutation operators are usually known as *standard or traditional operators*.
- **Class level** [87]: This level deals with the faults related to object-oriented features, such as inheritance or polymorphism. This kind of operators is known as *class mutation operators* (also class-based or class-level operators). This level will be analysed in more depth in Section 2.3.2.
- **Integration level** [26]: Mutations at this level test the connection or interaction between software units, making changes from the parameters to the values returned by the functions. These operators are called *interface mutation operators*.
- **Multi-class level** [91]: This level goes a step beyond the integration level, tackling the problem of integration testing of multiple classes or entire software products. Mutation operators at this level are also known as *system level operators*.

The number of languages that have been addressed with this technique has definitely risen in the last years, including object-oriented languages. As a result, we can find mutation tools automating the generation and execution of mutants for a wide range of them [68]. We can mention *MuJava* [87] for Java, *MILU* [65] and *Proteum/IM 2.0* [27] for C or, more recently, *MutPy* [36] for Python as successful tools applied in several research studies. Delahaye and du Busquet [25] collected in a survey their experience when using mutation tools for several programming languages, presenting the main features of the tools and positive and negative aspects.

Different techniques have been used to introduce mutations into the code, depending on the nature of the language and the complexity of the operators. Chevalley and

Thévenod-Fosse [21] were the first authors dealing with the implementation of class mutation operators. They found that traditional techniques based on syntactic analysis were not sufficient to automate object-oriented mutation testing since mutation operators developed for procedural programs languages did not modify data structure declarations. As such, different techniques have been developed in mutation tools for object-oriented languages, such as inserting faults directly into the bytecode [87], using compile-time reflection to access the internal structure of a program [21] or using a parser-based approach [71]. For instance, *CREAM* [39] uses compilation along with reflection mechanisms to analyse the original program and determine where in the code the operators can inject new mutations. After that, *ILMutator* [37] implemented a new approach for C#, manipulating both the meta-data and the intermediate code (Common Intermediate Language of .NET) originated from C# code. That study concludes that inserting mutations at that level is more efficient and faster than parsing the code as no recompilation is needed. However, the identification of mutation locations presents some limitations and requires arduous implementation work to comply with correctness conditions. Finally, other mutation tools parse the abstract syntax tree (AST), like *MAJOR* [71] for Java to generate traditional mutants.

2.3.2 Mutation testing at the class level

Class-based mutation operators deserve special attention because they are particularly devised to test structures related to object-oriented features. Traditional operators may not be sufficient to stress the test suite designed for object-oriented applications because of the new facilities and features added with this programming paradigm; features such as encapsulation, inheritance, and polymorphism provide a new scope for potential faults. Therefore, the object-oriented characteristics require their own research and class mutation operators were designed to cope with the possible flaws resulting from the misuse of those features.

As an example of a class operator related to inheritance, the operator *IHD* (Hiding Variable Deletion) deletes a member variable in a subclass which is hiding a variable in a parent class. Given this piece of code:

```
class Base{
public:
   ...
   int v;
};

class Child: public Base{
```

```
public:
    ...
    int v;
};
```

this operator removes the variable "v" from the child class, so every reference to this variable will refer to the variable "v" in the base class now:

```
class Base{
public:
    ...
    int v;
};

class Child: public Base{
public:
    ...
    /*IHD*/
};
```

Although the object-oriented 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 [75]. The class-level mutation operators for Java were refined and extended later in different works of several authors [21, 76, 86, 104]. The list of class-based operators provided by Offutt et al. [104] is shown in Table 2.1. Subsequent to the definition of Java class operators, Derezińska studied object-oriented features in C# to provide class operators for this language [34, 35]. We can find several mutation tools to test programs in these two languages since then, where *MuJava* [87] for Java and *CREAM* [39] for C# are the most popular tools.

There are several empirical studies on the effectiveness of class mutation operators [77, 82, 88, 117]. One of the first analyses was made by Kim et al. [77], evaluating with mutation testing the test suites developed following three object-oriented test strategies. They found that these strategies for the creation of test suites were not effective at dealing with some object-oriented features. Lee et al. [82] studied the orthogonality of the class mutation operators compiled by Ma et al. for Java [86], and also the distribution of the mutants stemming from large programs. Experimental results showed that class operators could reveal many faults while producing few mutants in comparison to traditional operators in procedural programs. Ma et al. [88] found that the traditional operators produced about twice as many mutants as the class operators for the same applications. Ma et al. [88] and Segura et al. [117] observed a different behaviour when applying class operators for Java with respect to the number of equivalent mutants: around 70% and

TABLE 2.1: List of class mutation operators provided by Offutt et al. [104]

Block	Operator	Description
Encapsulation	**AMC**	Access modifier change
Inheritance	**IHI**	Hiding variable insertion
	IHD	Hiding variable deletion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	*super* keyword insertion
	ISD	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	**PNC**	*new* method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-specific features	**JTI**	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* keyword deletion
	JSD	*static* keyword deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOC	Reference assignment and content assignment replacement
	EOA	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change

45% in their studies respectively. However, some operators with a high percentage of equivalence in the study by Ma et al. did not produce any mutants in the experiments conducted by Segura et al. Moreover, we should take into account that the application of these class operators is rather dependent on the characteristics of the tested program in general. There are two surveys dedicated to object-oriented mutation testing: the first by Ahmed et al. in 2010 [4] and the second by Bashir and Nadeem in 2012 [14].

2.3.3 Mutation testing and C++

C++ [58, 122] is a powerful and multiparadigm programming language widely used in strategical areas in the industry. This language can be considered to be an enhancement of C, incorporating new properties such as classes, templates or exception handling. The

TABLE 2.2: Faults identified by Derezińska [33] for the *Object* and *Member* categories

Category	Description
Object	Calls a same function member from a different object of the same class. Calls a function from an object of a different class, but both classes have the common base class. Calls a function from the derived class instead of the base class.
Member	Calls a different (complementary) function member. Calls a function inherited from the base class. Swaps calling of function members in a class. Swaps calling of functions inherited from one class. Accesses the different data in the same object.

first standard for C++ appeared in 1998, being modified in the C++03 standard and, more recently, in new standards (C++11-C++17). However, the adaptation of these new standards is taking place gradually. Because of its advanced facilities, the size of the grammar and the variety of alternatives provided, it is not considered an easy language for the ordinary programmer, calling for a necessary testing process.

As advanced in the introduction, the development of mutation testing regarding C++ was still pending. This development has been postponed in favour of other languages, mainly because of the difference in complexity. Regarding a particular set of mutation operators for this language, the research is really scarce and we cannot find a complete set of operators. Prior to this thesis, only two attempts had been carried out to define a set of operators. The first work [133] formed a collection of traditional operators, based on the operators defined for Ada [103] and FORTRAN [78]. This set categorised the operators according to four blocks: *operand replacement*, *operator insertion*, *arithmetic operator replacement* and *relational operator replacement*. The second work showed an approximation regarding plausible errors related to object-oriented features in C++[33], but no operators were formally defined. This paper proposed five categories of possible mistakes: *Inherit*, *Associate*, *Access*, *Object* and *Member*. However, only *Object* and *Member* refer to C++ coding errors (the three first blocks are applied to the Unified Modeling Language (UML) specification). The faults within these two blocks in that paper are summarised in Table 2.2. The research regarding both approaches was given up as no new progress has been published since then.

In the case of mutation systems for C++, existing commercial tools include mutation testing within a set of testing techniques (they do not focus only on mutation testing) and do not cover mutations at the class level but only some standard operations, using the technique in a selective way:

- *Insure++* (1998) [62]: This tool uses mutation testing as one more technique to enhance the software quality (especially focusing on memory problems), but it only

performs some standard mutations as mentioned in [37]. Its approach is different from classical mutation testing because it only creates *functionally equivalent mutants*, which are expected to pass the tests instead of failing. Therefore, an error in the original program is detected when a mutant is killed, revealing the ambiguities that could exist. Users can choose the number of mutants to generate (the more mutants created, the more rigorous detection), and apply mutation testing from a single function to an entire project.

- *PlexTest* (2005) [112]: As mentioned in its web page, this software implements a *highly selective* mutation testing to avoid the generation of equivalent mutants. When selecting this option, the tool only performs the mutation of deletion, removing statements and sub-expressions. This tool incorporates some other features to improve the performance, like the combination with a revision control system to determine recently-edited code and selectively test that code.

- *Certitude Functional Qualification System* (2006) [55]: This tool combines mutation testing and static analysis, qualifying a program functionally and revealing faults that might not be detected otherwise. Although this product has also been used for the analysis of software systems, it is now addressing the microelectronics industry.

As for open-source systems, *CCMutator* [80] is a mutation generator for concurrency constructs in C or C++ recently developed. This tool implements a set of operators specifically designed to mutate multi-threaded applications.

2.4 Cost Reduction Techniques

2.4.1 Motivation

As aforementioned, mutation testing is a powerful technique hampered by its computational inefficiency. Two are the main problems when applying mutation testing: the **computational cost that generating and executing all the mutants involves** and the **existence of equivalent mutants**. Several advances have been proposed in order to reduce the computational cost of mutation testing. These cost reduction techniques have been reviewed by Offutt and Untch [100] and Usaola and Mateo [125] and they will be addressed in the next sections.

The existence of mutants functionally equivalent to the artefact under test is an issue in mutation testing. Equivalent mutants are identified when alive mutants are manually inspected, so determining which of the surviving mutants are equivalent is a time-consuming and error-prone labour. Hence, a variety of works have tried to alleviate this issue, but it is not possible to fully automate the analysis of equivalent mutants since this is an undecidable problem. The first heuristics for detecting equivalence were proposed by Baldwin and Sayward [11] based on compiler optimisations. Besides, constrained-based test data (CBT) to automatically generate test data was also used by Offutt and Pan [99] to identify equivalent mutants. Hierons et al. [57] addressed the detection of equivalent mutants by means of program slicing and Adamopoulos et al. [2] used a co-evolutionary algorithm to discard equivalent mutants through a fitness function. Several interesting advances have been done regarding equivalence in the last years. Mutant classification strategies analysing the coverage impact of mutations [107, 115] have been used in studies to mitigate the effects of the equivalence. Papadakis et al. [108] also proposed a novel technique to automatically detect equivalent mutants based on compiler optimisations by comparing the executables of the original program and the mutants. They succeeded in reducing as many as 30% of all existing equivalent mutants in large real-world C programs.

Regarding object-oriented mutation testing, Lee at al. [82] proposed some hints for the implementation of three class operators in order to avoid class mutants which are known to be equivalent. Equivalence conditions were later extended by Offutt et al. [104] for sixteen operators; almost 75% of equivalent mutants on average were identified and their generation was prevented for sixteen operators. New rules for avoiding equivalent mutants generated by five class-based operators were provided and analysed by Hu et al. [59].

2.4.2 Classification

The techniques for the reduction of the computational cost in mutation testing can be classified into two distinct categories, as stated by Jia and Harman [68][1]:

1. Reduction of the mutants generated.
2. Reduction of the execution cost.

Regarding (1), several techniques for reducing the number of mutants without unacceptable information loss have been proposed:

[1] Offutt and Untch [100] summarise the approaches for the reduction of the cost into three categories: *do fewer*, *do smarter* and *do faster*.

- **Mutant Sampling** [20] selects randomly some of the available mutants (also known as random mutant selection).
- **Selective Mutation** [102] applies only a subset of the mutation operators defined for the language.
- **Mutant Clustering** [60] groups the mutants according to the set of test cases that kill them and then selects a representation of each cluster.
- **Higher Order Mutation** [67], unlike traditional first order mutation, generates mutants that contain more than a single fault.
- **Evolutionary Mutation Testing** [43] generates a selected subset of mutants through an evolutionary algorithm.

As for (2), there exist several techniques to reduce the execution cost that can be applied:

- **Weak mutation and Firm mutation**: weak mutation [49, 106] compares the internal state of the mutant and the original program once the mutated statement has been executed. Firm mutation [127] compares both the internal state of the mutant and the original program just after the execution of the mutation (as done in weak mutation) and the outputs at the end.
- **Runtime optimisations**: this group encompasses from a compiler-integrated approach to enhance the performance of compiler-based techniques [31], to the meta-mutant technique (also known as mutant schemata) representing all possible mutants in a single program [124] to speed up the execution of mutants.
- **Advanced platforms support**: advanced architectures are leveraged to distribute the computational cost among several machines, such as vector processors [93] or network computers [132].

Selective mutation and weak mutation are the most widely analysed techniques in the literature respectively for these two categories. We will focus on selective mutation in the next section.

2.4.3 Selective mutation

Selective mutation is one of the cost reduction techniques with greater acceptance because it has been satisfactorily applied following different approaches. Selective mutation works under the assumption that some mutation operators can be excluded, yet retaining a

similar fault-revealing capability. By using selective mutation, the aim is to obtain a *sufficient set of mutation operators*: a reduced set of mutation operators which is representative of the whole set of operators, that is, a subset of operators that can accurately predict the mutation score of the full set of operators. Selective mutation was first suggested by Mathur [92] to reduce the large computation expenses.

The approach of removing some of the mutation operators has been investigated since then by many researchers [13, 28, 95, 97, 101, 102, 126]. In this regard, Wong and Mathur [126] limited mutation testing to two operators in their study with FORTRAN operators; by using these two operators, they achieved similar results than using the 22 operators included in Mothra. Offutt et al. [101] explored the information loss when applying *N-selective mutation*, .i.e., when the N most commonly applied operators are removed. Among other results, by excluding the 6 operators that engendered more mutants (6-selective mutation), adequate test suites for the remaining mutants (around 40% of the complete set of mutants) maintained a high correlation with the full mutation score (99.71%). Offutt et al. [102] also explored the possibility of excluding the operators belonging to the same operator group: *replacement of operand*, *expression modification* and *statement modification*. They found that 5 of the 22 mutation operators in Mothra (those operators within the expression modification group) sufficed to implement mutation testing without a meaningful decrease of the mutation score, allowing for a reduction of 78% of mutants on average.

Removing operators of a similar syntactic category was also the approach to selective mutation followed by Mresa and Bottaci [95]. In their experiments, they found that selective mutation is a preferable option when compared to mutant sampling but only if a mutation score not very close to 100% is required. They used effective and non-redundant test cases in their empirical procedure. Barbosa et al. [13] tried to find sufficient sets of operators for C programs by defining a set of guidelines. By applying these guidelines to Agrawal et al.'s operators [3], the authors found that just with 10 operators the mutation score ranged between 95.8% and 100% in 27 cases studies. Namin et al. [97] aimed at finding a sufficient set of mutation operators by defining a statistical analysis procedure. This procedure identified 28 operators for C as sufficient for an accurate measurement of the mutation score for all the operators (128 operators implemented in Proteum/IM 2.0 [27]). The results of their approach to select a subset of operators did not support the intuition that one operator from each operator group should be selected, as in the guidelines proposed by Barbosa et al. [13]. Delamaro et al. [28] proposed a greedy algorithm for selecting a reduced subset of C mutation operators, successively adding the operators that increased the overall mutation score the most. They concluded that the high redundancy among the operators makes difficult to establish a single way to

select the best operators. Recently, the study directed by Zhang et al. [134] showed that selective mutation scales with regard to the size of the system under test.

Random mutant selection was proposed by Budd [20] and Acree [1], where they showed that just sampling 10% of the mutants was sufficient to predict the mutation score for all the mutants with high accuracy. Despite the particular attention received by selective mutation in the literature, a growing number of research studies in recent years gives evidence that selective mutation is not superior to random mutant selection [53, 135]. This conclusion was drawn by Zhang et al. [135] when comparing random mutant selection with several of the aforementioned sufficient sets of operators in the literature [13, 97, 102]. The experiments by Gopinath et al. [53] also suggested that removing operators could offer limited benefit in comparison to random mutant selection. Finally, Zhang et al. [136] applied 8 different random strategies for the selection of mutants, concluding that selective mutation and random mutant selection can be combined to further reduce the cost.

Selective strategies applied to class mutation operators have previously been studied for object-oriented languages. Derezińska and Rudnik [38] conducted their experiments with C# applying 18 class operators and 8 standard operators to three applications. The results evidenced that, even with a considerable reduction in the number of class mutants (using 74% of the mutants) still 93% of the original mutation score could be achieved. Ma et al. [88] explored the elimination of some unnecessary class operators in Java that generated very few mutants. Bluemke and Kulesza [17] performed a selective reduction of mutants generated by Java operators, including class-level operators. In their experiments, they showed that the strategy could significantly reduce the cost (between 40% and 60% of mutants) while preserving an acceptable mutation score and code coverage. Hu et al. [59] recommended the omission of the operators *PCI* and *OAC* for Java, as these operators had a low performance and were expensive.

2.4.4 Quality of mutation operators

The evaluation of the quality of mutation operators is closely related to the application of selective mutation. Assuming that not all mutation operators are equally effective at assessing a test suite, analysing desirable properties of the mutants generated by the operators has become an important research field in mutation testing. As mentioned in the previous section, the first evaluations of operator effectiveness aimed at obtaining sufficient sets of mutation operators by removing both the most prolific operators [101] and those of the same category [102] (based on the syntactic elements that they mutate). If the mutation score in the reduced set of operators is the same as in the original set

after the selective strategy, the effectiveness of the technique remains high and those operators are not actually necessary.

After that, several works have dealt with the concept of quality of a mutation operator considering other dimensions. Mresa and Bottaci [95], in addition to calculating the mutation score of FORTRAN operators, evaluated operators regarding two factors for a more accurate measurement of the trade-off of including each operator: mutation score and cost information about test data generation as well as equivalent mutant identification. Estero-Botaro et al. [44] analysed a set of WS-BPEL operators with a focus on the quantitative dimension, defining several terms to that end. A *weak mutant* is killed by every test case in the test suite, whereas a *resistant mutant* is killed by a single test case. Furthermore, the resistant mutant that is killed by a single test case only killing that mutant is known as *hard to kill*. In summary, Estero-Botaro et al. [44] considered that the quality of a mutation operator can be determined by studying the below conditions. A mutation operator should ideally:

1. Generate no invalid mutants.
2. Generate no equivalent mutants.
3. Require very specific test cases to kill its mutants. The operator will not be useful if an obvious "happy path" test case going through the most common paths is enough to kill most of its mutants.
4. Produce a high percentage of resistant and hard to kill mutants, and a low percentage of easy to kill (those which are killed by most of the test cases) and weak mutants.

In their study, they made use of the notion of quality that Derezińska [34] took to assess C# class operators, which computes how effective are test cases in killing mutants (see Equation 2.2). Derezińska also posed several questions that should be answered to judge the usefulness of an operator:

- Does an operator can be applied in real programs to simulate faults of programmers?
- Are any invalid mutants generated by an operator?
- Does an operator generate many equivalent mutants?
- Is an operator effective at assessing the quality of given test cases? If a mutant is not killed by a given test suite, is it easy to create test cases which kill it?

$$Effectiveness = \frac{Killed\,test\,runs}{(Total\,mutants - Equivalent\,mutants) \times Total\,tests} \times 100 \quad (2.2)$$

Smith and Williams [119] went a step beyond when classifying mutants. They categorised the mutants in four types: *killed by the initial test suite*, *killed by a new test case*, *killed by a new test case specifically defined to kill another mutant* or *not killed*. The mutants killed by test cases specifically designed to detect them require particular test cases which are not easy to design through other mutants. Therefore, this kind of mutants is more interesting than those mutants killed by the initial test suite. The quality of a mutation operator (named as *utility of a mutation operator* by these authors) is calculated as a linear combination of the percentage of each of the four types of mutants generated by the operator. As for object orientation, Hu et al. [59] also proposed a metric, called *mutation operator strength*, to estimate the quality of Java class operators. This metric computes the minimal number of necessary test cases to kill the set of non-equivalent mutants. An operator is regarded as strong when its mutants are detected by relatively few test cases.

Estero-Botaro et al. [45] defined a novel notion of *quality of a mutant* (Q_m), which considers the number of test cases killing a mutant and the number of mutants killed by those test cases. By way of explanation of the metric, an equivalent mutant will be punished with the minimum value (0), whilst each dead mutant will be assigned a different value in the range 0-1 depending on how difficult is to produce test cases to kill it (the higher the value, the better is that mutant).

Consequently, the quality of a mutation operator can be defined as the mean of the quality metric of the mutants generated with that mutation operator. Similarly, the quality of a set of dead mutants (D) can be defined as the mean of the quality metric of the dead mutants generated:

$$Q_D = \frac{1}{|D|} \sum_{m \in D} Q_m \quad (2.3)$$

In the study by Estero-Botaro et al. [45], they first calculated the number of equivalent, weak and resistant mutants generated by each operator. Then, they computed the quality of each mutation operator trying to establish a threshold that allowed discarding some operators with a low quality. They also analysed in depth the connection between their quality metric and the formulas of effectiveness by Derezińska [34], utility of operators by Smith and Williams [119] and mutation operator strength by Hu et al [59]. While Derezińska [34] and Smith and Williams [119] did not impose any restriction to the

test suite when computing their respective metrics, Mresa and Bottaci [95] and Hu et al. [59] assessed operators with adequate and non-redundant test suites. Estero-Botaro et al. [45] went further by establishing the condition of minimality to the test suite (see Appendix B).

2.4.5 Genetic algorithms applied to mutation testing

Search-based techniques have also been used in software testing [129] (and more specifically to mutation testing as well) in order to reduce the cost. This section focuses on those works using genetic algorithms to this purpose [51]. This thesis aims at analysing the technique known as Evolutionary Mutation Testing (EMT), which was proposed by Domínguez-Jiménez et al. [43] in 2011 to reduce the number of mutants generated for TSR by using an evolutionary algorithm. The algorithm favours the generation of *strong mutants*: *potentially equivalent* mutants, which are not detected by the initial test suite, and *difficult to kill* mutants, which are detected by one test case which only kills this mutant and no other.

Each of the mutants receives a fitness, which decreases as:

1. The number of test cases detecting the mutant increases.
2. The number of mutants killed by those test cases increases.

A genetic algorithm is implemented in *GAmera* [42], a mutation system for WS-BPEL compositions developed to apply and evaluate EMT. This system was later extended to use the genetic algorithm with Higher Order Mutants or HOMs [16].

As raised by these authors, genetic algorithms had been widely applied previously, but most of the studies had limited to test case generation [110], and only a few to mutant generation (where EMT is classified). Mutation testing has been applied in conjunction with evolutionary algorithms to generate test cases also for object-oriented software [15, 48]. Bashir and Nadeem [15] made use of the same term, Evolutionary Mutation Testing, when proposing a novel fitness function to help search effective test cases for object-oriented programs. Only Adamopoulos et al. [2] used a genetic algorithm for the co-evolution of mutant and test suite population, where difficult to kill mutants are favoured and equivalent mutants are penalised. Banzi et al. [12] also applied a genetic algorithm for the selection of mutation operators. They used a multi-objective approach: maximise the mutation score and minimise the number of mutants generated.

Nevertheless, there is an increasing body of research in the last years applying genetic programming to select a subset of mutants. Silva et al. [118] collected the studies applying

search-based techniques in the context of mutation testing, including a section dedicated to the application of search-based techniques for mutant generation.

Oliveira et al. [24] studied the evolution in parallel of the population of mutants and test cases, as done by Adamopoulos et al. [2]. However, they explored this approach describing a new representation with new genetic operators: Effective Son crossover and Muta Genes mutation (instead of the crossover and mutation operators used by Adamopoulos et al.). Schwarz et al. [116] leveraged a genetic algorithm to find mutations not detected by the test suite, which have a high impact and are also spread throughout the tested code.

Most of the studies in this context have focused on genetic programming to generate interesting HOMs [56, 64, 81, 105]. Jia and Harman [64] defined the concept of subsuming HOM as a HOM which is hard to kill when compared to the difficulty of killing the First Order Mutants (FOMs) from which it is constructed. The authors applied several search-based techniques in order to find subsuming HOMs, concluding that the genetic algorithm yielded the best results of all them thanks to its ability to generate subsuming HOMs from one generation to the next. Later, Harman et al. [56] provided a more restrictive fitness function to find strongly subsuming high order mutants (SSHOMs). Langdon et al. [81] also tried to find hard to kill HOMs as similar as possible to the original program, showing that these HOMs could simulate complex faults beyond those modelled with FOMs. Omar et al. [105] have explored in several papers the performance of search-based techniques (including genetic programming) to find subtle HOMs for Java and AspectJ programs, where guided local search obtained the best results in general.

As a final remark, Lima et al. [83] have recently compared both traditional strategies (such as random mutant selection, selective mutation and search-based mutation by using a genetic algorithm) and HOM-based strategies. This comparison was based on the number of mutants, the number of test cases and the mutation score. Randomly selecting 20% of mutants (random mutant selection) and removing the 5 most prolific operators (selective mutation) were the best strategies overall, though most of the strategies presented a similar behaviour.

Chapter 3

Definition of Mutation Operators

This chapter defines a set of mutation operators at the class level for C++ and groups them into different categories. We complete the chapter providing references about operators in other object-oriented languages and remarking differences and similarities between them and the operators defined for C++.

3.1 Defining Mutation Operators

The first step in mutation testing is the definition of mutation operators, where the set of operators has to be defined for each particular language. In our case, we seek to define a set of mutation operators for C++ at the class level (see Section 2.3).

With regard to class mutation operators, the research related to other object-oriented languages has been analysed, in particular, Java [76, 86, 104] and C# [34]. These languages are very 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 [58]). For its part, the basic syntax of C# is influenced by C/C++ as well as Java in its object model.

Thus, we followed this process for the definition of operators:

1. Check whether it is possible to adapt each of the operators defined for Java/C#, and whether the features of C++ alter their definition. In that case, perform sufficient changes to make them suitable for C++.

2. Design new mutation operators according to the language peculiarities and add them to the set of operators generated in the previous step.

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 (object reference) and the object it actually points to. This aspect often causes mistakes when referring objects, especially if using dynamic allocation. In this sense, dynamic binding is not as simple as in other languages to induce a polymorphic behaviour. When a method is declared with the *virtual* keyword, the method is dispatched based on the runtime type of the invoking pointer to object (the compiler uses a virtual method table or *v-table* to this end). The polymorphic behaviour is created through both pointers and references. Construction and destruction of objects are also sources of several known faults [58] because of the complex memory management. Construction entails memory allocation and initialization of every member variable, since references and primitive types are not automatically initialized. Besides, the existence of special methods, such as the destructor and the copy constructor, are distinguishing features of this language.

The inheritance mechanism in C++ has a particular syntax because of the possibility of using multiple inheritance. Because of this, the scope resolution is necessary to access the members of a base class, especially in presence of hiding and overriding members in the class hierarchy. The protection level of the members of a base class can also be specified in the inheritance. In addition to the above commented, there are many other characteristics that may confuse a programmer when moving from a mainstream language to C++, like method overloading, exception handling, default arguments or operator overloading.

Having said this, mutation operators have been classified into several categories according to the main characteristics of object-oriented programming. Furthermore, the aforementioned main sources of error have been taken into account to define these categories and their mutation operators. Each category is identified by an uppercase letter. At this level, mutation operators have been subdivided into seven categories, which are listed below with the letter that identifies each of them:

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

TABLE 3.1: 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.*

Mutation operators are identified with a code comprised of three uppercase letters: the first letter denotes the category, while the two other letters identify the operator within the category. Categories and their mutation operators are summarised in Table 3.1.

3.2 Mutation Operators at the Class Level for C++

In this section, we describe the purpose of each mutation operator, remarking those details related to C++ features. An example of different mutation operators from each category is also shown, allowing us to observe the nature of the mutations represented with this set of class mutation operators.

3.2.1 Access control

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

- **AMC or Access modifier change**: *AMC* checks the correct access control to members of a class. In C++, 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 another block with a different access level. In the absence of a section with a specific modifier, a new block with that access level is included to add the member. On the contrary, if the block has a single item, it is deleted after running out of members.

Example AMC:	**Mutant 1:**	**Mutant 2:**	...
`public: int a;`	`public: int a;`	`protected: int a;`	
`private: float b;`	**`string c;`**	`private: float b;`	
`string c;`	`private: float b;`	`string c;`	

- **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: *public*, *protected* or *private*. This operator changes the access modifier when inheriting to ensure the assigned access is correct.

Example AAC:

```
class A{
    ... ...
};

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

Mutant 1:

```
class B: protected A{
    ... ...
};
```

Mutant 2:

```
class B: private A{
    ... ...
};
```

3.2.2 Inheritance

Mutation operators related to inheritance relationships, mainly with respect to the presence of overridden members, are included in this group.

- **IHD or Hiding variable deletion:** When a member variable hides the variable of a parent class (both share the same name), this operator removes the hiding variable. As a result, the references to this variable access to the variable of the same name in the parent class.

- **IHI or Hiding variable insertion:** *IHI*, instead of removing the hiding variable as *IHD* does, creates a member variable in the subclass that hides the variable defined in a parent class.

Example IHI:

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

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

Mutant:

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

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

- **ISI or Base keyword insertion:** This operator ensures that 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 used to refer to the base class.

Example ISI:

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

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

Mutant:

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

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

- **ISD or Base keyword deletion:** This operator is the opposite case of *ISI*.

- **IPC or Explicit call of a parent's constructor deletion:** This operator removes the explicit call to a constructor of a parent class so that the default constructor is used. The constructor of a parent class is invoked within the initialization list of a constructor (see the operator *CID*).

- **IOD or Overriding method deletion:** The operator *IOD* deletes an overriding method (the parent's version is called instead) intending to ensure that the desired method is called.

Example IOD:

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

Mutant:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             /*IOD*/
};                                 };
```

- **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.

Example IOP:

```
class A{
    ... ...
    int a;
    void method(){
        a = 0; ... ...
    }
};

class B: public A{
    ... ...
    void method() {
        A::method();
        a = 1;
    }
};
```

Mutant:

```
class A{
    ... ...
    int a;
    void method(){
        a = 0; ... ...
    }
};

class B: public A{
    ... ...
    void method() {
        a = 1;
        A::method();
    }
};
```

- **IOR or Overridden method rename:** This operator acts when an overriding method interacts with a parent's version (see example below). This situation only arise when that method is declared *virtual*. In this way, the overriding method can

be called from a method in its parent class when the binding is dynamic. *IOR* renames the method being overridden in the parent class.

Example IOR:

```
class A{
   ... ...
   virtual void m1() {... ...}
   void m2() {... m1(); ...}
};

class B: public A{
   ... ...
   void m1() {... ...}
};
```

Mutant:

```
class A{
   ... ...
    virtual void m3() {... ...}
    void m2() {... m3(); ...}
};

class B: public A{
   ... ...
   void m1() {... ...}
};
```

- **IMR or Multiple inheritance replacement:** C++ supports multiple inheritance enabling a derived class to inherit from more than a single class. 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 by the scope resolution operator. That is the fault modelled by this operator. *IMR* can be applied when multiple inheritance is present and there is a conflict among members of the inherited classes.

Example IMR:

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

class B{
   ... ...
   int a;
};

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

Mutant:

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

class B{
   ... ...
   int a;
};

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

3.2.3 Polymorphism and dynamic binding

Mutation operators that belong to this block check that the polymorphic mechanism is properly used.

- **PCI or Type cast operator insertion:** The purpose of this operator is to cast an object reference, turning its actual type into the parent or child of the original declared type. The *dynamic_cast* conversion includes, in a safe way, the *downcasting* of pointers/references as well as the *upcasting*.

Example PCI:

```
class A{
  void m(){...}
  ... ...
}

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

Invocation:

```
B b;
A *pa = &b;
pa->m(); (*)
```

Mutant:

```
B b;
A *pa = &b;
(dynamic_cast<B*>(pa))->m(); (**)
```

(*) A::m() is invoked
(**) B::m() is invoked

- **PCD or Type cast operator deletion:** This operator represents the contrary case of *PCI*.

- **PCC or Cast type change:** *PCC* changes the cast type of an object reference to another of its class hierarchy.

Example PCC:

```
C c;
A *pa = &c;
(dynamic_cast<A*>(pa))->m();
```

Mutant:

```
C c;
A *pa = &c;
(dynamic_cast<B*>(pa))->m();
```

Where C is directly derived from B and indirectly derived from A.

- **PRV or Reference assignment with other comparable variable:** An object of a subclass can be assigned to an object reference of one of its ancestors. *PRV* changes that object, referred in a reference assignment, to an object of another subclass.

Example PRV:

```
A *a;
A1 a1;
```

Mutant:

```
A *a;
A1 a1;
```

```
    A2 a2;              A2 a2;
    a = &a1;            a = &a2;
Where A1 and A2 are subclasses of A
```

- **PNC or *new* method call with child class type:** *PNC* assigns a derived-class pointer to a base-class pointer instead of the instantiated type.

Example PNC:
```
    A *a = new A();
```
Mutant:
```
    A *a = new B();
Where B is a derived class of A
```

- **PMD or Member variable declaration with parent class type:** This operator changes the declared type of an object reference to a parent class type.

- **PPD or Parameter variable declaration with child class type:** *PPD* performs the same function as *PMD*, but it targets the parameters of a method.

- **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. To kill their mutants, there has to be at least one method overriding the behaviour of the virtual method in a derived class and a method invocation whose binding is dynamic.

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

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

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

3.2.4 Method overloading

Mutation operators in this group ensure that a method calling invokes the correct method when a class uses method overloading. C++ not only counts with method overloading, but with *operator overloading* (operators can be redefined, giving them different semantics depending on the operand types). These two concepts should not be confused,

although the mutation operators in this group can be applied to both kinds of overloading when applicable.

- **OMR or Overloading method contents replace:** The aim of this operator is to check that a method invokes the correct overloaded method, replacing the content of a method for the content of another with the same name.
- **OMD or Overloading method deletion:** The operator *OMD* removes one of the overloaded methods in each of the mutants generated. The wrong method is being called or an incorrect parameter type conversion is taking place if the mutant still runs.
- **OAN or Argument number change:** This operator focuses on the arguments in method invocations, changing the number of arguments. This operator should take into account the possibility of using *default parameters*. If a method has just another overloaded method, then only one mutant can be generated; but if the overloaded method has a default parameter, a further one can be created.

Example OAN: (using default parameters)

```
class A{
    ... ...
    void m (int a = 2) {... ...}
    void m (int a, float b) {... ...}
};
```

Invocation: a.m(0,0);

Mutant 1: a.m(0);

Mutant 2: a.m(); → a.m(2) is invoked.

- **OAO or Argument order change:** This operator is similar to *OAN*, but it changes the order instead of the number of arguments in a method calling.
- **OPO or Method parameter order change:** *OPO* changes the order of the parameters in a method declaration. This operator allows us to model the possibility that the programmer has invoked the wrong overloaded method because of implicit type conversions.

3.2.5 Exception handling

This block addresses the improper handling of exceptions[1].

[1] Although exceptions are not unique to this paradigm, they are closely related to it.

- **EHR or Exception handler removal:** *EHR* deletes one of the *catch* clauses in an exception block in each mutant generated, delaying the capture of the exception. In this sense, this operator tests the order of the handlers in an exception 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 situation is applied 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;
    };
}
```

- **EXS or Exception swallowing:** This operator adds a general *catch* clause at the end of a *try* block (provided it does not exist). When an exception is caught in this general clause, the operator *EXS* detects that the exception handling is not properly implemented.

3.2.6 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 is referenced and calls a method, *MCO* replaces that object by another one of the same class (the invoked method is not changed).

Example MCO:

```
a1.method();

Where a1 and a2 are objects of class A and they are member
variables of a class.
```

Mutant:

```
a2.method();
```

- **MCI or Member call from another inherited class:** This operator is similar to *MCO* in the sense that it is applied to an object when it is invoking a method, but the object is now replaced by another object of a different class, both objects having the same base class.

Example MCI:

```
class A{
    ... ...
    Child1 a1;
    Child2 a2;
    void m() { ... a1.method(); ... }
};
```

Mutant:

```
class A{
    ... ...
    Child1 a1;
    Child2 a2;
    void m() { ... a2.method(); ... }
};
```

Where Child1 and Child2 have a common base class

- **MNC or Method name changed:** *MNC* models the error produced when, in a method invocation, another method name is used instead of the desired one. This situation may happen if the class of the method invoked has another method which is compatible with the method calling. In this operator, standard methods and overloaded operators are dealt with separately.

Example MNC:

```
class A{
    ... ...
    void method1 (int a) {... ...}
    void method2 (int a) {... ...}
    void method3 () { ... method1(1); ... }
};
```

Mutant:

```
class A{
    ... ...
    void method1 (int a) {... ...}
    void method2 (int a) {... ...}
    void method3 () { ... method2(1); ...}
};
```

- **MBC or Member changed:** This operator accesses a different instance variable of the same object when a member variable is referred.

3.2.7 Miscellany

This block contains a blend of operators related to particular C++ characteristics that do not fit with the rest of categories.

- **CTD or *this* keyword deletion:** The operator *CTD* deletes occurrences of the keyword *this*, which is used to reference the current object, when a method parameter hides a member variable (when both have the same name).

- **CTI or *this* keyword insertion:** This operator simulates the contrary case of *CTD*.

- **CID or Member variable initialization deletion:** This operator removes the initial value given to member variables, checking that the proposed initialization is correct. Initial values are assigned using *initialization lists*, so an item initializing a member variable is removed from the list in each mutant.

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

- **CDC or Default constructor creation:** This operator removes the user-defined default constructor (when it is the only constructor) so that the compiler creates a default version. In this way, *CDC* checks initializations within this constructor.

- **CCA or Copy constructor and assignment operator overloading deletion:** In object-oriented programming, copying objects is frequently needed: a function receives or returns an object by value, or an object is initialized using another instance. This task 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). *CCA* deletes the copy constructor or the assignment operator overloading, checking that 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 to destroy the objects. *CDD* deletes the destructor method checking its correct implementation.

Example CDD:

```
class A{
    ... ...
    ~A() {... ...};
};
```

Mutant:

```
class A{
    ... ...
    // ~A(){... ...};
};
```

3.3 Comparison with other Languages

A set of 37 mutation operators has been defined and classified into seven categories. The size of this set of mutation operators is higher than the size of the set for Java [104] (29 operators) and the same as the one for C# [34] (37 operators without counting the invalid ones). In the next paragraphs, we compare the operators defined for C++ and Java/C# by categories:

- **Access control:** The operator *AMC* [21, 76, 104] is different in C++ because the access level is specified by sections and not individually, as in Java. Therefore, this operator will not change the access modifier of a member, but it will move the member to a block with a different access level.

- **Inheritance:** Most operators in the *inheritance* category have been defined for Java [21, 76, 104] and taken in C# [34]. However, *ISI, ISD, IPC* and *IOP* change with respect to Java as shown in the example for the operator *IOP*; they are related to the *super* keyword, which does not exist in C++ because of the multiple inheritance. Likewise, *ISI* (and analogously *ISD*) has the *super* keyword in its name, so it is called *base keyword insertion* instead, as shown in [34] for *ISK*. The scoping in C++ allows referencing members of classes which are deeper in the hierarchy. For instance, *ISI* can generate different mutants when a variable with the same name is declared in several classes of the hierarchy.

- **Polymorphism and dynamic binding:** All operators from [104] in the *polymorphism* group, *PNC, PMD, PRV, PCC, PCI, PCD* and *PPD,* have been considered for this language with the same meaning. Nevertheless, their implementation is completely different as polymorphism and dynamic binding are handled in a different manner in C++. As commented in Section 3.1, the use of pointers/references to dynamically bind the objects is necessary. With respect to the type

casting of objects, C++ provides specific casting operators apart from the generic form.

- **Method overloading:** Regarding *method overloading, OMR*, *OMD*, *OAO* and *OAN* have been based on [86] and *OPO* is based on *POC* from [76]. This latter one has a different name in our set, adapting it to the established naming convention. As it was mentioned earlier, operator overloading is also addressed in this category.

- **Exception handling:** Attending to the *exception handling* category, a definition of *EHR* and *EHC* can be found in [76] and *EXS* in [34] for C#. The function of the operator *EHC* is achieved in Java using a *throws* declaration instead of the *try-catch* statement. This is different from C++, where this option is not available. Besides, Java uses a singly-rooted hierarchy, so the exception will be caught in the *Object* class ultimately. In C++, the *finally* clause is not used and the exception could be captured in the *main* function instead.

- **Object and member replacement:** *MNC*, *MCO*, *MCI* and *MBC* are all named in [34] and the fault simulated by them is shown in [33] (in *Object* and *Member* blocks). For this category, only an explicit definition for *MNC* can be found in [21].

- **Miscellany:** Some operators in this block take as reference the *Java-specific* group in [104]. Those operators are not exclusively available to Java and similar operators have been created for C++, except *JSI* and *JSD*: *static members* must be declared inside the class and also initialized outside the class; inserting or deleting this keyword would suppose more than a simple change. Regarding *CID*, an initial value cannot be assigned directly to members (like in Java) but in the constructors.

  The *common programming mistakes* block included in [86], containing operators related to typical mistakes and misuse of the language, has not been considered to create analogous operators. There is no a convention in C++ with respect to the methods that they mutate, so they do not fit this language.

Chapter 4

Implementation of the C++ Mutation System

This chapter collects all the information on our approach when implementing mutation operators and the mutation system, called *MuCPP*. The operator implementation encompasses the method to inject the faults into the code, the requirements for the generation of the mutants that are expected from each mutation operator, and also the improvement rules that can be set to avoid uninteresting mutants. Regarding the implementation of the mutation system, we describe the overall system architecture and functionalities.

4.1 Mutation Operator Implementation

This section focuses on the implementation of mutation operators through the analysis of the abstract syntax tree generated by Clang. After setting several requirements for the proper operator implementation, this section ends with an example to illustrate this process.

4.1.1 Approach

4.1.1.1 LLVM and Clang

LLVM [84] is a project intending to provide the necessary infrastructure for the development of new compilers for any programming language (the target was originally the compilation of C and C++). The development of the LLVM project is completely open source, comprising a number of subprojects that have drawn a great interest, many of

them used in other commercial and open-source projects. LLVM supplies the middle layers of a complete compilation system, taking the *intermediate form* generated by a compiler for a certain programming language. One of its aims is to achieve a compile-time, link-time, run-time, and idle-time optimisation. Thus, LLVM can be configured to be used with GCC in order to obtain a compiler frontend for every programming language addressed by GCC. However, several projects are being developed to create new frontends for some languages to work specifically on top of LLVM, such as FORTRAN, Haskell or Phyton.

Clang [22] is the most outstanding frontend implemented for the LLVM project. It is devoted to the C family languages: C, C++, Objective-C and Objective-C++. This frontend was built as a native part of the LLVM system (indeed, it is part of the LLVM releases). Therefore, Clang is a compiler frontend for these programming languages, using LLVM as its backend.

This linkage of both LLVM and Clang constitutes a complete and functional toolchain. As a matter of fact, they can be used to create new tools based on them, thanks to the library-based design followed by these two projects. This fact allows us to reuse the Clang libraries to parse C/C++ code for a particular purpose. Embedding other compilers into our applications is not as easy as with LLVM/Clang (for example GCC because of its structure design). Therefore, we can conclude that these projects were created, since their inception, to design new tools at a source code level, such as refactoring, static analysis or code generation tools. The design as an API, instead of as a monolithic structure, is the main characteristic that allows us to develop a mutation framework, since in mutation testing we actually need to analyse and change the source code according to a set of mutation operators. Advantages that the Clang project claims are achieving fast compiles and low memory use, a greater compliance with the C family languages standards than other compilers and an easy integration with IDEs.

4.1.1.2 Abstract syntax tree

In order to apply mutation testing, we need to find a robust method to insert the faults into the code. Parsing the high-level code might be insufficient in the cases when it is necessary to analyse the context of the elements involved in the mutation. Thus, making use of the internal representation generated by compilers is a more appropriate option to this end.

The aforementioned intermediate form internally generated by compilers is actually the *Abstract Syntax Tree* or *AST*. We say abstract syntax because, unlike a concrete syntax tree, it does not contain every token in the source code explicitly, but the structure of

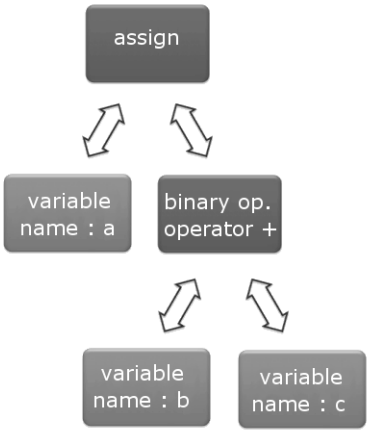


FIGURE 4.1: Abstract syntax tree for the expression "$a = b + c$"

an expression through branches of the tree. In this way, the tree gives us a simplified and clear structure of the code with only the essential aspects, facilitating to go through the tree to process the nodes. The abstract syntax tree in Figure 4.1 represents the expression "$a = a + b$", the concrete syntax in C++. However, for instance, the sum is represented as "$(+\ a\ b)$" in Lisp, but that concrete syntax is the same in structure as "$a + b$" in C++.

The search using the AST makes the task of analysing the code regarding the set of mutation operators easier, which is supported by the following features of Clang:

- **The AST generated by Clang is easily understandable** and can be even serialised (it has the form of an XML file). This shows us how the compiler works internally, providing a better understanding of the cases when a mutation operator can be used.
- **Clang maintains the same code that was passed to the compiler at all times**, i.e, it does not implicitly simplify the code. As an illustration, a compiler could delete in the AST the parentheses that were not necessary in order to simplify the code. This would not serve our interests since each mutant is expected to contain the same code as the original program except for a single change modelled by the mutation operator.
- **Clang saves information about every token in the code**. This information allows us to retrieve the token or tokens in the code to make the appropriate modification according to the mutation operator.

4.1.2 Matching nodes in the AST

Mutation operators need to be automated to create the appropriate source-to-source transformations. In certain languages, as in specification ones, the target of a mutation operator is relatively easy to identify as the options are usually limited. In contrast, a general purpose language like C++ provides a wide range of design alternatives. Hence, we should consider multiple factors before determining mutation locations.

In this sense, the representation of the code as AST allows us to omit the specific and more intricate details of each particular situation. Each element in the language is uniformly represented with a type of node in the tree. Each type of node is modelled with a class in the *Clang* API, providing members to handle the nodes and the relations among them. For instance, member methods are represented with the class `CXXMethodDecl`, so every method in our code will be bound to an object of that class (saving information of each method).

The AST generated can be traversed in two different ways:

1. **Using the *visitor* pattern**, which enables us to process each kind of node in a different manner. For instance, if we need to process only the member methods, we can use this technique to visit every `CXXMethodDecl` node. Then, we can impose as many restrictions as necessary on this kind of nodes.

2. **Using *matchers*** [90]: This compiler supplies a domain-specific language or DSL (based on the classes of the libraries) for the combination of rules, allowing us to traverse the tree and search for the desired nodes through pattern matching. These patterns, called *matchers*, use the *visitor* pattern internally to find the nodes complying with a set of conditions.

The latter option has been chosen to traverse the AST because *matchers* fit better with the sense and purpose of mutation operators; moreover, the number of bound nodes is lower using *matchers* than using the visitor pattern as different conditions can be set in the search to process a reduced number of nodes. A list of useful and frequently used *matchers* has been already included in *Clang* [90]. Besides, new simple *matchers* can be defined to invoke the members of the API classes. Our proposal implies coordinating various *matchers* to build a new one (or several) for each mutation operator (an example of the construction of a *matcher* is exposed later on in the document). Therefore, each operator is automated with the development of a pattern using this DSL.

As a high-level example, if we wanted to find every member variable marked with the *private* access modifier, the pattern would follow the next structure:

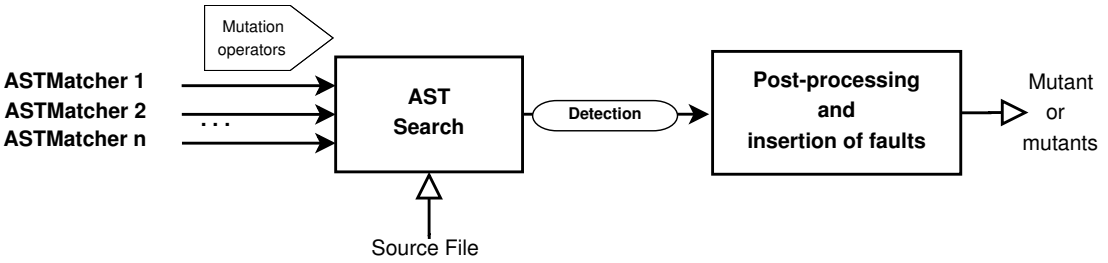


FIGURE 4.2: General diagram of the search and generation of mutants in the proposed mutation system

```
for each class x belonging to program P do
    for each variable y belonging to class x do
        if variable y is private
            retrieve variable y
        end if
    end for each
end for each
```

4.1.3 Fault injection

Before explaining how mutations are inserted into the code, we should distinguish three elements that we use to identify a mutant:

- **Mutation operator**: As it was explained earlier, each mutation operator is focused on different elements in the code and performs a different modification.
- **Mutation location**: A mutation operator can be applied several times for the same source file. The mutation locations for each mutation operator represent the positions in which that mutation operator can be applied.
- **Attribute or variant:** The attribute represents the different mutants that can be generated for each mutation location. The number of variants depends on the mutation operator. For instance, an arithmetic operator can be replaced by four other arithmetic operators (see Section 2.1 for an example). This means that four different mutants can be produced per detection of an arithmetic operator in the code.

For the injection of faults, *matchers* are first used to detect convenient nodes in the tree with regard to the mutation operator description. When a mutation location is found, the matched nodes can be mapped to their location in the source code thanks to the information saved in the AST by Clang (see advantages of Clang in Section 4.1.1.2).

Depending on the nature of the operator, we can *insert* an element into the code, *replace* an element or *delete* the involved code range. Sometimes, not only a single fault can be injected, but several variants can be applied as in the example of the replacement of arithmetic operators. Once a simple fault has been introduced, the mutated source code can be saved in a new file, generating a mutant.

The process to inject the faults into the code is depicted in Figure 4.2 and can be summarised in the following steps:

1. A pattern (or several patterns) for each mutation operator is created using *matchers*.
2. The source code is converted to the form of AST.
3. The AST is traversed searching for every mutation target according to the language elements modified by the set of mutation operators. As a result, a set of nodes is obtained, which represents potential mutation locations.
4. The retrieved nodes are post-processed to ensure that the injection of a fault is possible at that point. Moreover, we should analyse other aspects in order to produce the expected mutation (see Section 4.1.4). For instance, in the operator *CID*[1] it is essential to know the position of the initializer within the initialization list to know whether a comma (1) or the colon preceding the list (2) has to be deleted:

```
(1) A::A() : a(0), b(0)
 - Mutant deleting a(0) and the following ','
 - Mutant deleting b(0) and the preceding ','
(2) A::A() : a(0)
 - Mutant deleting a(0) and the preceding ':'
```

5. Depending on the nature of the operator, one or more variants can be introduced to each mutation location. Therefore, the different mutations that can be inserted into a location are collected in this step. Each mutation operator has to consider each of the possible variants in its implementation.
6. The corresponding mutations are inserted into the set of mutation locations detected.
7. The mutants with the form of source code are generated, containing the introduced faults.

In some cases, the attribute is known in advance (*fixed attribute*), but in other cases, this number is variable (*variable attribute*). The case of the mutation operator "arithmetic

[1]Recall that *CID* deletes the initialization of member variables in the initialization list

operator replacement" is an example of fixed attribute (attribute = 4). Also, an operator deleting an element of the code can only produce a mutant at that point (the removal of the element, attribute = 1). However, there are mutation operators whose number of mutants entirely depends on the location within the code. As an example of *variable attribute*, the application of the operator *OAN*[2] depends on the overloaded methods defined in each class and the parameters of those methods, so the mutation operator requires a study of each particular situation in order to know how many mutants can be generated.

4.1.4 Expected mutants

4.1.4.1 Generation of the expected mutants

The definition and the implementation of mutation operators are aspects that go hand in hand, but they should be addressed from different perspectives. The definition describes the purpose of each operator and when they should act, but it does not reflect low-level details in general. In contrast, the implementation should consider a more fine-grained control for the generation of appropriate mutants. A mutation system generating undesired mutants can lead to a significant increase in the computational cost and also to overlook interesting mutants if they are malformed.

Thus, four requirements have been identified in the operator implementation to generate the *expected mutants*. These requirements should be studied in each operator to match as closely as possible the implementation with the expected behaviour. The requirements are listed below and are illustrated with the operator *IOD*. The aim of this operator is to delete an overriding method (the parent's version is called instead) with the intention of ensuring that the desired method is invoked:

- **Requirement 1: Insert mutations into locations under appropriate conditions**. We should analyse the conditions posed in the operator definition to determine whether the mutation makes sense in that context.

  IOD operator: This operator should only be applied if there is an overridden method in a base class. This implies, among other aspects, comparing the methods between both classes.

- **Requirement 2: Ensure that the mutant is well formed**, that is, prevent the generation of mutants that do not compile because of an error in the operator

[2]Recall that *OAN* changes the number of arguments in method invocations when there are overloaded methods.

implementation.

IOD operator: If the method to be deleted is not defined at the time of the declaration, both the definition and the declaration of the method have to be removed in the mutant:

Example:

```
class A{
    ...
    int method();
};

int A::method(){...}
```

Mutant:

```
class A{
    ...
    /*IOD*/
};

/*IOD*/
```

We should note that inserting a further change cannot be considered to be a high order mutation (when more than a single fault is inserted into a mutant); both mutations (declaration and definition removal) are applied to serve the same purpose.

- **Requirement 3: Generate the exact number of mutants that can be expected** from an operator, producing neither a lower nor a higher number of mutants. In other words, do not miss any mutant that should be generated and, at the same time, do not generate more mutants than the expected ones. Spinellis [120] uses the terms *silence* and *noise* for the missed and extraneous matches respectively.

IOD operator: This operator could create the same mutant in a class as many times as a similar method was found in its base classes. *IOD* checks if the method to be deleted is hiding another one in a base class. When implementing this operator, the base classes are explored and the same method could be found in several of them. Hence, the same method could be deleted more than once if this fact is not contemplated, leading to different mutants that would contain the same mutation:

Example:

```
class A{
    ...
    int method(){...}
};

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

Mutant:

```
class C: public B{
    ...
    /*IOD, method found in class B*/
};
```

Duplicate mutant to avoid:

```
class C: public B{
    ...
    /*IOD, method found in class A*/
};
```

```
class C: public B{
   ...
   int method(){...}
};
```

- **Requirement 4: Avoid the generation of uninteresting mutants** for the assessment of the test suite. Given that the computational cost of the technique is a major concern, we should impose some conditions on the operators to avoid the creation of *uninteresting mutants*, i. e., mutants which do not help us assess the adequacy level of a test suite despite meeting the other three requirements. The three kinds of mutants that should be prevented as much as possible are:

  – **Invalid mutants:** The mutated code cannot be compiled to generate an executable program.

  – **Equivalent mutants:** Any input is able to detect a difference between the original program and the mutant.

  – **Trivial mutants:** The difference between the original and the mutant is detected by every input covering the mutation [103].

  IOD operator: A method is labelled as *pure virtual* when it is not defined in a class and forces inheriting classes to supply a definition for this method. If the parent's version of the method is marked as pure virtual, deleting the method in the child class will always produce an invalid mutant. This situation can be contemplated in the operator implementation:

Example:

```
class A{
   ...
   virtual int method()=0;
};

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

Mutant to avoid:

```
class B: public A{
   ... ...
   /*IOD*/
};
```

4.1.4.2 Considerations for the implementation

As aforementioned, there exist several aspects that should be considered in order to perform the expected mutations when automating mutation operators. Undesired situations can be caused by several reasons:

- The specifics of the language.
- The characteristics of the AST.
- Different peculiarities of each operator.

As a result, after studying the application of the operators to several C++ programs, diverse situations generating unexpected mutants could be detected. In this section, we address those situations that should be considered in the operator implementation at a low level to comply with the first three requirements in the previous section. The fourth requirement (avoiding the generation of uninteresting mutants) will be examined in depth later on in Section 4.2.

The following general considerations were tackled in the *basic implementation* of every operator to create the expected mutants:

1. **Duplicate mutants**: Some of the operators could create two or more identical mutants. The generation of duplicate mutants was mainly detected in various operators related to inheritance when a class inherits from several classes. This is the case of *IOD*, as explained in the third requirement in the previous section.

   In addition, it is necessary to take care of templates. A template specialization of a class is explicitly contained in the AST as a further class. When mutating a declaration, we would produce the same mutant for each one of the specializations if they are not excluded from the process.

2. **Declarations and definitions**: Several elements in C++, as functions and methods, can be declared at the moment of its declaration, but also in redeclarations outside the class definition. Both the declaration and the definition (in a redeclaration) should be modified if a mutation operator changes the signature or the value returned by a function or method. Also, sometimes a method is never defined but the programmer provides the declaration for documentation. The mutation of such declarations is pointless and should be avoided.

3. **Implicit elements**: The AST also contains implicit elements, i.e., language constructs not written in the code but automatically added by the compiler. As an example, special methods like destructors or invocations for the construction of base classes can be mentioned. Thus, implicit elements should be taken into account so that they:

   (a) Are not mutated.

   (b) Are not used as candidates for a replacement.

(c) Do not intervene in the proper mutant formation.

4. **Unchanged mutants**: This situation is produced in operators replacing elements by other similar elements in the code. In this instance, the operator seeks for candidates in the code, but the element to be mutated has to be discarded from the set of candidates.

5. **Types and namespaces**: Some mutation operators need to know the type of the elements prior to the injection of mutations. For instance, when a mutation operator replaces variables of the same type. However, evaluating the types is required in the first place: the keyword *typedef* can be used in C++ to rename a particular type. Therefore, if we do not take this feature into account, some mutants may be overlooked.

   Likewise, several declarations can be grouped together in a *namespace*. We can declare similar elements in the code provided that they are in different namespaces. This fact should be considered in the operator implementation when:

   (a) Replacing elements: we need to properly qualify the references to declarations in namespaces.

   (b) Checking some conditions for the mutation: if invocations to members are qualified (for instance, to refer to the member of a particular base class), the qualifier should be analysed.

4.1.5 Example

In this section, we show an example to clarify how the AST helps us find mutation locations. The goal of the example is to show the implementation of a mutation operator and its application to a fragment of source code.

4.1.5.1 Mutation operator

As starting point, the mutation operator *CDC* or *C++ default constructor creation* (see Section 3.2 for a definition) has been chosen to illustrate how we can combine different *matchers* satisfying the rules defined in each operator. Recall that a default constructor is provided by the compiler in C++ when a class does not contain other user-defined constructors. This operator deletes the constructor without parameters supplied by the user (so that the compiler provides the default version) in order to ensure that this constructor is correctly implemented. The following **two conditions should be considered in this operator**:

```
1  DeclarationMatcher CDC_Matcher =
2     constructorDecl(
3        parameterCountIs(0),
4        isDefinition(),
5        anyOf(
6           hasAnyConstructorInitializer(
7              isWritten()),
8           has(
9              compoundStmt(
10                has(
11                   stmt()))))),
12       ofClass(
13          recordDecl(
14             unless(
15                hasMethod(
16                   constructorDecl(
17                      allOf(
18                         hasAnyParameter(
19                            anything()),
20                         unless(
21                            isImplicit()))))))))
22    ).bind("CDC");
```

FIGURE 4.3: *Matcher* for the operator *CDC*

- **Condition 1:** In C++, a constructor with parameters can be invoked without arguments if default parameters were provided for that constructor, for instance, `A::A(int a = 0)`. In this way, that constructor is also considered an user-declared default constructor. However, we do not have to take this fact into account as deleting that constructor would remove a non-default constructor too (maintaining that constructor and deleting the default value of the parameter is not an option as the compiler would not provide the default constructor in that case).
- **Condition 2:** We are looking for a non-trivial constructor, i.e., we should not delete the constructor if it has no statements in its definition. In C++, that definition implies not only the body but the initialization list. This aspect can be classified into the fourth requirement for the generation of the expected mutants (see Section 4.1.4.1), given that this situation would lead to an equivalent mutant.

Thus, the *matcher* in Figure 4.3 has been defined to search for the user-declared default constructor with the combination of the next simple *matchers*:

- To find the default constructor:
  - **constructorDecl** (line 2): This pattern retrieves every constructor declaration.

  - **parameterCountIs(0)** (line 3): This pattern excludes constructors with parameters (this pattern also avoids the case shown in the first condition above).

- To meet the second condition (looking for a non-trivial constructor), we can add the following *matcher*:

  - **isDefinition** (line 4): To ensure that a constructor is non trivial, we need to check its definition. This pattern will retrieve only constructor declarations with a definition.
  - **anyOf** (line 5): It is the '||' or logic operator *OR*. We need to find a constructor with, at least, either one constructor initializer in the initialization list (**hasAnyConstructorInitializer** (line 6)) or one statement in the body (**has(compoundStmt(has(stmt()))))** (lines 8-11)). The pattern **isWritten** (line 7) is used to ensure that the constructor initializer is explicitly written in the code. For instance, the compiler implicitly invokes the default constructor of a base class (see "implicit elements" in Section 4.1.4.2).

- We have to ensure that this is the only constructor of the class:

  - **ofClass** (line 12): The nested code in this pattern is used to ensure that the class does not contain any other constructors. Namely, this *matcher* means that the constructor declaration belongs to a class with the features described in parentheses.
  - **recordDecl** (line 13): This pattern retrieves every *class*, *struct* and *union* declaration.
  - **unless** (line 14): Is the '!' or logic operator *NOT*. We need to look for every method that is a constructor (**hasMethod(constructorDecl())** (lines 15 and 16)) to check the condition commented in the *matcher ofClass*.
  - **allOf** (line 17): Is the '&&' or logic operator *AND*. If that constructor has at least one parameter (**hasAnyParameter(anything())** (lines 18 and 19)) and the constructor has not been implicitly added by the compiler (**unless (isImplicit())** (lines 20 and 21)), the user-declared default constructor cannot be deleted; the default constructor is not provided by the compiler if the class has another constructor. The **allOf** matcher can be omitted, as it is implicitly added if no other logical pattern is supplied (indeed, this pattern is implicit within **constructorDecl** (line 2), matching every condition given).

The string "*CDC*" inside the *matcher bind* (line 22) is a unique identifier to retrieve the nodes associated with this pattern afterwards in an *operator handler* to post-process

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A{
6    public:
7      A() {a = 1;}
8      int a;
9      int get_a() {return a;}
10 };
11
12 class B: public A{
13   public:
14     B(): A(), b(1) {}
15     float b;
16     float get_b() {return b;}
17 };
18
19 class C{
20   public:
21     C() {}
22     string c;
23     string get_c() {return c;}
24 };
25
26 class D{
27   public:
28     D() {d = false;}
29     D(bool _d){d = _d;}
30     bool d;
31     bool get_d() {return d;}
32 };
33
34 class E: public D{
35   public:
36     E();
37     int e;
38     int get_e() {return e;}
39 };
40
41 E::E() {
42   D(true);
43 }
```

FIGURE 4.4: Classes in *example.cpp*

the nodes. Then, if desired, the operator can be refined through the methods offered in the Clang API to better comply with the operator purpose. Even though new *matchers* can be defined using the members of the API, some conditions are difficult to represent with this DSL. Hence, a post-processing in the operator handler is also helpful in some situations to collect useful information for the injection of the mutations.

```
| -CXXRecordDecl 0x68f7450 <line:5:2, line:10:2> class A
| | -CXXRecordDecl 0x68f7560 <line:5:2, col:8> class A
| | -AccessSpecDecl 0x68f75f0 <line:6:3, col:9> public
| | -CXXConstructorDecl 0x68f7660 <line:7:4, col:15> A 'void (void)'
| |  `-CompoundStmt 0x68f78f8 <col:8, col:15>
| |    `-BinaryOperator 0x68f78d0 <col:9, col:13> 'int' lvalue '='
| |      | -MemberExpr 0x68f7880 <col:9> 'int' lvalue ->a 0x68f7730
| |      |  `-CXXThisExpr 0x68f7868 <col:9> 'class A *' this
| |      `-IntegerLiteral 0x68f78b0 <col:13> 'int' 1
| | -FieldDecl 0x68f7730 <line:8:4, col:8> a 'int'
| | -CXXMethodDecl 0x68f77a0 <line:9:4, col:26> get_a 'int(void)'
| |  `-CompoundStmt 0x68f7998 <col:16, col:26>
| |    `-ReturnStmt 0x68f7978 <col:17, col:24>
| |      `-ImplicitCastExpr 0x68f7960 <col:24> 'int' <LValueToRValue>
| |        `-MemberExpr 0x68f7930 <col:24> 'int' lvalue ->a 0x68f7730
| |          `-CXXThisExpr 0x68f7918 <col:24> 'class A *' this
|  `-CXXConstructorDecl 0x68f7eb0 <line:5:8> A 'void
|       (const class A &)' inline noexcept-unevaluated 0x68f7eb0
|    `-ParmVarDecl 0x68f7ff0 <col:8> 'const class A &'
```

FIGURE 4.5: AST fragment representing the class *A* in "*example.cpp*". In bold, the user-declared default constructor matched by *CDC*

4.1.5.2 Source code and mutants

In order to check how the operator *CDC* works, a source file called "example.cpp" with five simple classes has been designed (Figure 4.4). The default constructors in those classes contemplate different possible situations.

The default constructor of the classes *A* (line 7), *B* (line 14) and *E* (lines 36, 41-43) should be retrieved by *CDC* to be removed. In the Figure 4.5, the node matched in the AST for the class A is shown. On the contrary, the default constructor of class *C* (line 21) should not be bound as it is trivial (second condition); besides, class *D* does not fit the proposed mutation operator because the class defines another constructor (line 29).

In the case of the class *E*, unlike the rest of constructors in the code, its default constructor definition is located outside the class declaration (lines 41-43). The example shown in the requirement 2 for the operator *IOD* (see Section 4.1.4.1) also takes place in *CDC*; the definition of the constructor is analysed in the pattern, but the declaration has to be also deleted so that the operator has the intended effect (this can be performed in the operator handler using the methods provided by Clang to this end). The resulting mutants are depicted in Figure 4.6.

```
 5  class A{
 6    public:
 7      // CDC: A() {a = 1;}
 8      int a;
 9      int get_a() {return a;}
10  };
```

```
12  class B: public A{
13    public:
14      // CDC: B(): A(), b(1) {}
15      float b;
16      float get_b() {return b;}
17  };
```

```
34  class E: public D{
35    public:
36      // CDC: E();
37      int e;
38      int get_e() {return e;}
39  };
40
41  // CDC: E::E() {
42  //  D(true);
43  // }
```

FIGURE 4.6: Mutants generated in "*example.cpp*" (see Figure 4.4)

4.2 Mutation Operator Improvement

The study of the mutants produced in each operator can lead to the identification of different situations always producing unnecessary mutants. As it was mentioned earlier, avoiding uninteresting mutants could allow for a reduction of the computational cost of the technique, which is a major concern when using mutation testing. Specific rules for several operators to cut out equivalent mutants have been proposed for Java [59, 82, 104] (see Section 2.4). However, in general, it is not possible to avoid the creation of every invalid, equivalent or trivial mutant.

The aforementioned studies identifying mutants to be prevented locate different situations in each particular mutation operator. 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. Some unproductive mutants were detected in a particular mutation operator when reviewing its mutants.

2. This situation was then thoroughly analysed to determine whether it was a one-time situation or it always led to the generation of uninteresting mutants.

3. The detected case was studied in order to know whether it could be generalised in terms of implementation.

4. Finally, every operator was processed to establish whether the situation could be extrapolated to other operators, creating an **improvement rule** for the reduction of mutants, or it was an isolated occurrence. In the former case, the steps 2 and 3 were performed again in each of the respective operators.

In addition to some particular cases implemented in specific operators, nine general improvement rules were identified. They assisted us in developing an *improved implementation* of the corresponding operators:

1. **Check for triviality**: In the operators related to constructors and destructors, deleting these methods would be useless if the compiler provides the method by itself exactly with the same functionality. This happens when the method is trivial: it has no initializers or the default constructors are used to initialize the base classes, or the method has an empty definition.

2. **Explicit invocation of constructors**: If a non-default constructor of a base class is invoked, this call cannot be removed if that base class does not have an user-declared constructor without parameters. In this case, the class has to be always initialized explicitly as the compiler does not provide the default constructor.

   We can illustrate this rule with an example with *IPC*:

Original:

```
class A{                          class B: public A{
   ... ...                           ... ...
   A(int p1, int p2){... ...}        B(): A(1, 1) {... ...}
   A(int p1){... ...}
};                                };
```

Mutant:

```
class B: public A{
    ... ...
    B(): /*IPC*/ {... ...}
};
```

   IPC would delete the base class initialization marked in bold, but the base class does not have a default constructor and the mutant would be invalid.

3. **Member access control**: When an operator replaces 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. However, when the member belongs

to another class, the access to this candidate needs to be checked to know whether that reference is allowed within that class.

4. **Declaration scopes**: Several operators replace a mention of a class to another class, but the class selected for the replacement may not have been declared yet at that point. Therefore, it is necessary to check if the new class is available in each mutation location.

5. **Check the member invoked**: Equivalent mutants are generated in those cases when the member referenced is still the same after the mutation. For instance, if a method of a base class is referenced with the resolution operator (`Base::member`) but the member has not been overridden in the child class, *ISD* would not affect the behaviour of the program when deleting the qualifier (`Base::`).

6. **Member variables marked as *const***: Constant member variables require an explicit initialization. Thus, the operator *CDC* for instance should not remove the default constructor if the class contains a constant variable. Also, the operator *IHI* will generate invalid mutants when inserting member variables marked as *const* into a child class since those variables would need to be initialized in the constructors. This also applies to reference type variables.

7. **Default arguments**: The use of default arguments should be taken into account in some cases when a method call is changed to invoke another method. The list of parameters needs to accept the arguments provided in the invocation.

8. **Infinite recursion**: Sometimes, the mutation can make a method calls itself infinitely, as in *ISD* when deleting the base class qualifier within the overriding method. This state leads to trivial mutants that would be killed by any test case covering the mutation, as in the following example:

Original:

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

Mutant:

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

9. **Pure virtual methods**: In various operators, the mutation results in a pure virtual method being called. This leads to an invalid mutant as these methods have

no definition. This case was illustrated with *IOD* to explain the fourth requirement in Section 4.1.4.1.

Apart from these rules, some particular situations were considered in each of the operators. As an example, in the case of *IOP*, if the method invocation to move up and down is within a method with a return statement at the end, the method call should not be placed at the bottom of the method as it will be never reached: the resultant mutant would have the same effect as removing the method call.

As a final remark, taking into account that mutation testing is a white-box technique, many of these improvement rules are closely related to C++ because they have been directly derived from this language. Still, some of these rules may apply to other object-oriented programming languages.

4.3 MuCPP: Mutation System Implementation

MuCPP is the mutation system developed to apply mutation testing to C++ programs. This mutation system allows the tester to analyse C++ code with regard to the set of mutation operators implemented, and generate the mutants according to the results of that analysis. Moreover, this system is prepared to execute the test suite against the mutants. *MuCPP*, as most of the existing mutation systems, works in three distinct phases, which will be described in the next subsection. Subsequently, we will also explain the main features incorporated into the system.

4.3.1 Phases

Analysis of the source code

MuCPP traverses the AST to analyse the code and determine where mutation operators can be applied through the procedure explained in Section 4.1.3. In this step, the tester can provide one or more C++ files to the tool, as a C++ project usually comprises several source files. The respective ASTs are created and then sequentially visited for each one of the operators in the same execution. In this stage, the tester receives information on the number of mutants that can be produced per mutation operator. Note that all the 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.

Currently, the system integrates the class-level operators shown in Section 3.2 and also a selective set of traditional mutation operators (this set will be used in our experiments later on in the document in order to compare both types of mutation operators). However, we should also note that some of the operators defined for C++ were not included in the mutation 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 [87] or CREAM [39]. In the experiments conducted by Offutt et al. using FORTRAN [101], the operator *SVR* (similar to *MBC*) generated a very high number of mutants. However, the mutation scores obtained when removing this operator were still 100% in almost all cases. As a result, most other mutation tools have traditionally excluded similar operators. The construction of *MuCPP* allowed us to analyse larger programs, and then we found out that the number of mutants that *MBC* and *MNC* generated sharply increased the cost of the technique and was more in line with the number of mutants generated by similar traditional operators. On the basis of the aforementioned results reported by Offutt et al., we decided to remove these two operators from our mutation tool as well. The fact that *MBC* and *MNC* generate many mutants would produce an undesirable effect on the evaluations about class-based mutants; the contribution of the rest of operators in the set would be secondary when compared to *MNC* and *MBC* due to the large difference in the mutants generated. Likewise, *AAC* and *AMC* did not prove useful and *EXS* is difficult to analyse because killing mutants from this operator requires that exceptions not considered in the program are thrown (that is, it is unknown if some exceptions are missing).

It is worth noting that another set of operators could be added at any moment: the mutation tool can be easily extended with new operators. The system also allows the tester to enable/disable mutation operators to apply selective mutation.

Generation of mutants

Mutants are generated in this step, each one only representing a single modification (first order mutation) in one AST. Each mutant is a clone of the original program except for the file modified. As it was shown in the example of the second requirement in Section 4.1.4.1, we may need to insert a change into several parts of the code to create a single mutant. Those locations could be in different source files (for instance, the declaration could be in a header file). In that case, several files are modified in the mutant.

The files remaining unchanged are also stored in the clone. However, mutants are not created as new directories: each mutant is generated as a branch with a unique name in

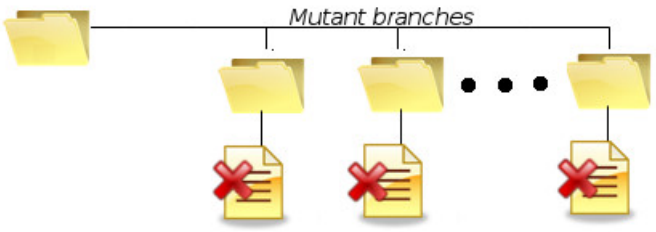


FIGURE 4.7: Generation of mutants using Git

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 in the storage resources (see Section 4.3.2 for further details). This mechanism allows testing each mutant as a stand-alone program because the mutant contains all the necessary files to build the program separately. This process is graphically shown in Figure 4.7.

MuCPP does not actually generate every available mutant in the code: the system implements the collection of improvement rules to avoid some uninteresting mutants (see Section 4.2).

Execution of tests

The mutants, once physically generated, are supplied to the execution module together with the test suite defined by the tester for the system under test. The test cases are applied to the mutants, reporting preset values when the mutant fails (1) or pass (0) a test case. Then, these results can be examined to determine whether mutants are dead or still alive after the test suite execution.

MuCPP has been implemented in such a way that the tester is not subject 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. Given that there is no a prevailing testing framework for C++ (unlike Java, for instance, where *JUnit* is widely used), this approach avoids having to translate a test suite already implemented to apply this mutation tool.

We have also developed a library to deal with this stage. This library measures the time and it also implements a timeout, which can be configured depending on the tests run; the injected mutation can lead to an unexpected behaviour, so the timeout will stop the execution of the test when exceeding a reasonable time. Moreover, a test scenario may fail at any moment because of a runtime error; this library can be configured either to stop the execution of the mutant or to continue with the rest of test cases.

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

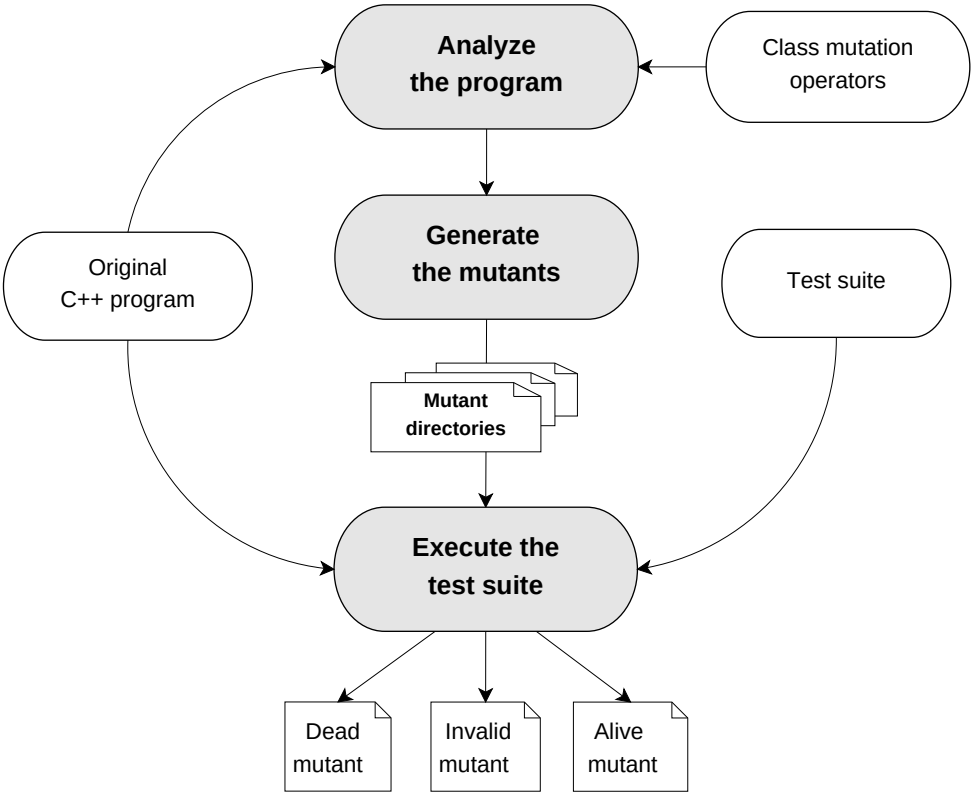


FIGURE 4.8: MuCPP work-flow

Summary of the process

The entire process from the analysis of mutants to the execution of the test suite can be seen in Figure 4.8. The application of mutation testing with *MuCPP* can be outlined as follows:

1. In the analysis of the source code, potential nodes to be mutated can be retrieved through pattern matching on the AST according to the mutation operators added to the system. The potential mutation locations are studied in depth in the operator handler to ensure that the bound nodes meet the conditions for the application of the mutation operator.

2. Mutations are injected into the mutation locations taking care of the correctness conditions in the formation of the mutant. Mutants are then produced containing the corresponding mutations, generating as many Git branches as mutants.

3. The generated mutants can be executed on the developed test suite to produce the output. Under this scheme of mutants as Git branches, each mutant can be compiled and run independently.

4. Finally, results can be analysed to determine the classification of mutants: surviving, killed or invalid.

4.3.2 Features

In this section, we describe the main features of *MuCPP*, which enable and facilitate the practical application of mutation testing:

Dependency analysis:

MuCPP, as a source code analysis system, needs to know the information handled by the build system used for the program to correctly parse its source files. For instance, the tool should be aware of the paths to header files. This information and other configuration options can be found in the commands used to compile each source file in a project. In this regard, *MuCPP* enables two options:

1. We can provide this information directly to Clang on the command line when executing *MuCPP*, using the available options in the compiler.
2. We can provide this information through a *JSON compilation database* file to Clang [69]. This file, which contains the full compilation command of each source file, can be automatically generated with *CMake*[4]. *MuCPP* will be able to find the commands to parse the different source files, their dependencies, the involved libraries... without providing any additional information to the tool.

The second option is especially convenient when we analyse several source files in the same execution. Using JSON files allows the tester to forget about these details.

Header files:

The AST contains the code of the headers included in the supplied files. *MuCPP* is able to distinguish user header files from the ones marked as system headers, only considering the former kind of headers for the insertion of faults. Thus, if the user does not want a particular header file being mutated, the tester can inform the tool of this fact using the appropriate option of the compiler (for instance, *-isystem* in Clang). This is particularly useful when working with third-party "lite libraries" provided by a single header placed within the project directory.

[4] https://cmake.org/

Git version control system:

This is the first use of Git branches in a mutation tool as far as we know. Previously, the SVN version control system had been used to reduce space when storing mutants [40]. However, Git features make this version system control more efficient for mutation testing than SVN; Git saves time when switching between branches and when committing the changes, given that Git can be informed about the modified files.

To illustrate this fact, the time was measured when using both Git[5] and SVN[6] (local repositories) with the *Kig* application[7], which occupies 91M. One hundred branches were created in an automated manner, inserting the same simple modification into one of the files for every branch. Git took 4.65 seconds to generate the branches, while SVN needed 32,40 seconds. In other words, SVN took almost seven times longer than Git for the same task. This result supports that Git is suitable for saving not only space but also much time when compared to SVN.

Although this usage of Git is unusual, generating mutants as Git branches has been especially helpful to simplify implementation and save space without impacting scalability. The system does not experience performance issues when handling a large number of mutants. As an example, when applying mutation testing to *KatePart* (see Appendix A), 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 sets of mutants without problems.

Duplicate mutants:

MuCPP has been designed to avoid the creation of duplicate mutants. As commented in the analysis stage, the system enables parsing several source files in the same execution, which are analysed sequentially. Because of header files being contained in the AST and the same headers being included in different source files, a class could be analysed more than once, leading to the creation of the same mutants. Segura et al. [117] distinguishes the terms "generated" and "executed" mutants when carrying out a mutation testing process in Java because of the existence of reusable classes. Thus, when a mutation operator finds a location to insert a mutation, *MuCPP* saves a list of the locations in the code mutated by each operator, ensuring that every mutant represents a different fault. Thanks to this fact, we do not need to post-process the mutants to remove duplicate ones.

[5] Version 1.9.1
[6] Version 1.8.8
[7] `http://edu.kde.org/kig`, version 1.0

Statistical data:

Finally, *MuCPP* has also been instrumented to yield some statistical data about the number of mutated classes and the mean of faults inserted into each class, in general and per operator. This feature has been useful to obtain experimental results in this thesis.

Chapter 5

Mutation Operator Analysis

This chapter performs a comprehensive analysis of class mutation operators. To that end, this chapter is divided into two main sections: quantitative analysis and qualitative analysis. In the former, we study the reduction in the number of mutants in those operators implementing improvement rules, the distribution of mutants, the mutation score and the improvement of the test suite. In the latter, we explore the utility of some class operators, the performance of class operators when compared to traditional operators and their ability to detect coding errors.

5.1 Quantitative Analysis

This section looks in depth at the quantitative aspect when using the set of class operators. The computational resources are very important when applying mutation testing, so we evaluate the reduction achieved thanks to the implementation of the improvement rules in the corresponding operators. We also compute different statistics related to the generation of mutants in object-oriented systems. Finally, we measure the mutation score and improve the test suite through surviving mutants.

5.1.1 Evaluation of the reduction of uninteresting mutants

For the experiments in this section, we prepared two versions of the mutation operators:

- *Basic version:* operator implementation to comply with the first three requirements in Section 4.1.4.1

- *Improved version:* operator implementation to comply also with the fourth requirement, automating the improvement rules in Section 4.2.

The mutants generated in both versions were compared to check the extent to which the improved version is able to avoid uninteresting mutants. In order to observe the impact on the number of mutants, we calculated the generation and execution times as well as the storage requirements in both cases to measure the enhancement in the efficiency of the mutation system. It is worth mentioning that:

- A tailored timeout for each program was set to stop a test scenario when it did not respond after a reasonable time (see "Execution of tests" in Section 4.3.1).
- These 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 was measured using the standard Unix utility *time*, while the execution, compilation and Git times were measured using the C++ standard library *chrono*.

The reduction achieved by the improvement rules has been computed for every application analysed in the experiments in this chapter (see Appendix A). Table 5.1 shows, in different columns, how many mutants are produced with the basic and improved versions, for those operators that produce fewer mutants after their improvement. The difference between the basic and the improved version (*Reduction*) and the percentage of reduction in the number of mutants (*Red.%*) are also presented in this table.

The total percentage of mutants excluded by the improvement rules across 16 mutation operators is 46.6%. However, we should note that several operators produce few mutants and that there are varying reductions among the operators. When considering the complete set of class operators, the reduction represents 32.1% of the total number of mutants. When studying the operators individually, we can remark the removal of all the mutants from *IMR* and *PCC*. In contrast, the number of mutants is not reduced for other operators with improvement rules implemented and generating mutants due to the characteristics of the subjects.

Table 5.2 shows a complete list of times measured when applying mutation testing to each application. Again, results have been calculated and divided according to the basic and the improved version of the mutation operators. The time for each version has been in turn divided according to the two last phases of mutation testing shown in Section 4.3.1:

TABLE 5.1: Reduction of mutants for improved class operators generating fewer mutants in the analysed programs

Operator	**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

- *Mutant generation: Total* measures the time needed to analyse the source files and produce the mutants. The time used by Git has been calculated separately, including the creation of new branches and changes in the corresponding files.
- *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.

We can observe from the results of this table that the test execution phase is the critical operation when compared with the mutant generation phase. The compilation and the execution times are almost entirely dependent on the compilation system and the duration of the tests respectively (see Table A.1 in Appendix A). Git performs the least time-consuming tasks in the execution phase. On the contrary, Git takes most of the time in the mutant generation phase. However, this result is not unexpected, taking into account that Git performs output operations which imply writing files. We should also note that the difference in the percentage of the total time spent in the generation of mutants among the applications is motivated by the number of processed source files: the more files, the more ASTs are created and analysed.

When comparing the times of the basic and the improved version, we can see from Table 5.2 that the highest reduction is achieved in the compilation time. Many of the avoided mutants are invalid, which only increase the compilation time but not the execution time as the test suite cannot be applied to them. Moreover, the rest of the

TABLE 5.2: Times for the generation of mutants and test suite execution in the analysed programs with the basic and the improved version of the set of class operators

	Basic version								**Improved version**						
		Generation		**Test suite execution**						**Generation**		**Test suite execution**			
Program	$\|M\|$	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 5.3: Storage resources taken by class mutants in the analysed programs

Program	$\|M\|$	**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.

discarded mutants help reduce compilation time further. Test suite execution times are lowered thanks to the improvement rules. We have to note that, while these rules may also 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 5.3 shows:

- *Original*: the size of the original program.
- $|M|$: size of the set of mutants generated with the improved version of the operators.
- *Git*: disk space occupied after generating the mutants (M).
- *Mean*: average of storage resources needed by each mutant (*Git* / $|M|$).

As it can be seen, that *Mean* is similar in every case study. This means that the size of the Git repository (*Git*) barely depends on the size of the program (*Original*), but mainly on the number of mutants ($|M|$). In other words, the size of the repository increases proportionally to the number of mutants and not to the size of the program. This fact supports that Git just needs to save the mutation when generating a mutant, as commented in Section 4.3.1. 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 time expenses.

5.1.2 Distribution of mutants

Several absolute and relative counts were computed to study the distribution of the generated mutants across class operators on the analysed programs to better understand the quantitative dimension when using this type of operators.

Table 5.4 depicts the number of mutants generated per operator in these programs. The total number of mutants created by this set of operators and the average number of

TABLE 5.4: Distribution of class mutants generated by program and operator, divided by the categories in Table 3.1

Operator	**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

mutants produced by class are shown at the end of the table. We should note that the mean only considers the operators producing at least one mutant.

Table 5.5 includes, for each program and operator[1]:

- C: The number of classes that are mutated.
- $C\%$: The percentage of the whole set of classes that are mutated.
- M: The average number of mutants that are generated per class.

[1]Note that the operators that do not produce any mutants in these programs are not shown in this table.

As an example of the meaning of M, the operator *PCI* produces 324 mutants in *Tinyxml2* (see Table 5.4); the number of classes in this program is 20 (see Table A.1 in Appendix A), so the value of M in Table 5.5 is 16.2 (324/20).

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 5.4 because they produce almost 69% of the total number 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 (for instance, when a tester knows after a previous inspection that the members of the analysed classes do not belong to the same semantic field and therefore 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 (it is common that a class has several 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 mainly due to a low frequency of appearance of the characteristics addressed by these operators. In other cases, the implemented improvement rules prevent several mutants from appearing. For instance, the basic version of *IMR* generates various invalid mutants in *KMyMoney* (see Table 5.1) because the rule about pure virtual methods (see improvement rule number 9 in Section 4.2) is disabled. Despite not generating any mutants for the subjects in these experiments, these operators can be valuable because the features that they address may receive less attention due to their rare use. Therefore, we do not recommend discarding them.

There is not a clear link between the percentage of large classes and the number of mutants generated within a class. As it can be observed in Table A.2 in Appendix A, *KMyMoney* is the program with the highest percentage of classes with more than 500 lines of code (17.6%), but is in the fourth position according to M (average number of mutants generated per class) in Table 5.5 ($M = 1.4$). Nevertheless, this is not surprising since many class operators are related to structural elements, and thereby do not depend on the length of the methods of the class.

A key factor in the generation of class mutants is the existence of inheritance relationships among classes. Beyond the impact on the "inheritance" category, the operators in

TABLE 5.5: Quantitative statistics by program 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

the "polymorphism and dynamic binding" group will also be affected. 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. In this program, two operators belonging to the "method overloading" category generates 82 out of 137 mutants because these classes contemplate the use of operations with different types.

The more constructors in a class, the more mutants *CID* is likely to generate. For instance, there are 40 mutants in the nine classes of *Matrix TCL Pro* with a mean of 3 constructors. The same fact happens with *IPC* when a class is inheriting; 37 mutants emerge in *KMyMonyey* with 27 inheriting classes and almost 2 constructors per class. On the contrary, *CDC* is not likely to apply many times when the average of constructors in the classes is high: the compiler only provides a default constructor when a class does not contain other user-declared constructors, as it was commented in Section 4.1.5.

5.1.3 Mutation score and test suite improvement

This section shows the calculation of the mutation score in the subjects under study when using the set of class operators. The surviving mutants are then analysed, adding new test cases to kill surviving non-equivalent mutants.

Tables 5.6–5.10 present the mutation score in each program, for each mutation operator and in general. These tables 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*).

TABLE 5.6: 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

MuCPP discards several mutants because of the improvement rules (see results of Section 5.1.1), and class operators produce fewer mutants than traditional operators in general (see results of Section 5.1.2). Therefore, the low number of mutants when compared to similar evaluations regarding traditional operators in the literature corresponds to these facts. In the case of *KMyMoney* and *KatePart*, we selected a subset of the program to enable the manual reviewing of surviving mutants.

TABLE 5.7: 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 5.8: 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

Despite the reduction of equivalent mutants thanks to the improvement rules, 27.9% of the valid class mutants (considering the programs altogether) are still found to be equivalent (357 out of 1,279). We have to note that some mutants are classified into the set of equivalent mutants 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. Also, there are some mutants which might be only killable under certain memory restrictions (we comment this fact in Section 5.2.1). All the mutants generated by *PMD* turned out to be equivalent. *PMD* is the only operator that does not produce any useful mutants among those class operators creating at least one mutant in

TABLE 5.9: 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 5.10: 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

these programs.

The mutation score is far from 100% in all the programs, especially in *XmlRpc++* (51%) and *KatePart* (32%). Therefore, we can say that these test suites are not able to detect the different faults simulated by these class operators in the code of these programs. As a consequence, the current test suites do not ensure a minimum coverage of class

TABLE 5.11: Mutation score obtained after improving the test suite for the analysed programs with respect to surviving non-equivalent class mutants

	Original		**Added**			**Augmented**		
Program	$\|S\|$	$\|A\|$	$\|M\|$	$\|S\|$	$\|A\|$	$\|S\|$	$\|A\|$	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

mutations, that is, they were designed without taking into account common mistakes when handling object-oriented features. Therefore, the mutation scores in these tables show that we can take advantage of the application of mutation testing to improve the adequacy level of test suites. As a result, we have analysed the surviving non-equivalent mutants and refined the test suite accordingly. 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 assertions from test cases or test scenarios:

- **Assertion**: It checks the current state of one or more objects at any given moment. An assertion is used to confirm that the state after performing a sequence of actions is correct.
- **Test case or test scenario**: A scenario describes a particular logic where some objects work together, testing functionalities of the program. A test case may encompass different assertions.

The tester creates several test cases to check the correct operation of a set of classes and their members, including different assertions. Still, not only the assertions determine whether a mutant is killed or not, but a different behaviour can be exhibited at any moment during the execution of the test scenario because of a runtime error or a timeout (see Section 4.3.1). Therefore, a test scenario may fail at any moment, even satisfying all the assertions.

Each test scenario is usually designed with a particular goal according to the system under test. In this way, the augmentation of the test suite to kill the surviving mutants has been performed as follows:

- Modify an existing scenario when the assertion needed to kill a mutant is closely related to the logic of that scenario.

- Add a new scenario when, in our view, there are no test scenarios checking a particular use of the program. This test case may include some assertions at the same time. In order to complete the new test case and make it as general as possible, several assertions are inserted apart from those needed to detect the mutation that induced the scenario.

Table 5.11 shows the original size of the test suite, the additions made, and the size of the final augmented test suite, where:

- $|S|$ is the number of test scenarios.
- $|A|$ is the number of assertions.
- $|M|$ is the number of modified scenarios.

This table also shows the mutation score (MS) computed with the augmented test suite. We have achieved a class-adequate test suite for the programs *Matrix TCL Pro* and *XmlRpc++*. The design of new test cases driven to kill surviving mutants by hand is a complex and laborious task, especially when testing third-party libraries, so we have added new tests within our possibilities in the rest of the programs. After inspecting the surviving mutants, we have created new tests to form better test suites, which will be used later on to compare class and traditional operators (see Section 5.2.2).

5.2 Qualitative Analysis

This section analyses class-based mutants from a qualitative perspective. In particular, it studies in which cases these class mutants can be useful, their contribution to the test assessment when compared to traditional mutants and the detection of coding errors thanks to these mutants.

5.2.1 Class mutation operator utility

To start with the qualitative analysis, we analyse in this section the kind of mutants produced with a subset of class operators. The goal is to illustrate, with particular cases, different situations where these operators can be useful to assess or improve a test suite with respect to object-oriented features in our set of case studies (see Section A.1 in Appendix A).

The operators studied are: *CDD, CCA, CID, PVI* and *IHI*. *CDD*, *CCA* and *CID* are related with the construction and destruction of objects; they refer to language elements that have some distinguishing features when compared to the rest of methods and they are always invoked whenever an object is built or destroyed respectively. Additionally, *CDD*, *CCA* and *PVI* were specifically defined for C++, so they have never been studied before. On the contrary, *IHI* was adapted from other object-oriented languages without change.

- ***CDD* operator:** *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. As a result, some mutations could be detected when the memory is not released properly. The combination of the mutation system with a tool for memory debugging like *valgrind* could help detect those situations. Nonetheless, this experiment shows that a destructor also performs other operations that should be tested with specific test cases. We detected two interesting situations:
  - 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 *Xmlrpc++*, the deletion of a destructor in a mutant also affects the execution of the program because a pointer to a boolean is not given the appropriate value.
- ***CCA* operator:** As in the case of *CDD*, *CCA* mutants are usually equivalent because copy constructors are often similar to the default copy constructor provided by the compiler. Still, this operator can suggest the inclusion of new scenarios performing a copy of objects when this constructor is somewhat different, as in the following case:
  - In *family*, a mutant from *CCA* was killed when the destructor was invoked in a specific scenario copying an object of the class `Parent`. 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.
- ***CID* operator:** This operator tends to create many mutants and some mutants are easily killed when a member pointer is not initialised in the constructor. However, after studying the mutants from this operator, we can highlight the following facts:

- In *Tinyxml2*, all the mutants generated by *CID* were executed on the test suite and three of them were killed by a single test case, which was different for each of these three mutants. This information gives evidence that some faults can be difficult to locate and shows the need for a test suite as complete as possible.
- In *garage*, we detected a case where the application of a timeout was useful. 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 by the compiler; a timeout will stop the test case when the execution takes longer than normal:

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

- ***PVI* operator:** This operator focuses on non-virtual methods in a base class which are overridden in derived classes. These methods should be declared with the *virtual* keyword when we want that they are dispatched based on the runtime type of the objects. A non-virtual method overridden in a derived class led to the generation of a mutant in the following case:
  - In *simul*, a mutant was generated marking as virtual the method `move()` in the class `cursor_controller`. Then, a new test case was created to kill the mutant by producing a runtime overriding: the method `move()` in the class `screen_controller` was dynamically invoked instead of basing this action on the static type of the object.
- ***IHI* operator:** Inheritance is the most notable and used feature of the object-oriented paradigm. Mutation operators classified into the "inheritance" category like *IHI* check whether a test suite properly addresses inheritance relationships among classes. We found several mutants which are representative of the utility of this operator:
  - In *garage*, in addition to the member variable `plate` in the class `Vehicle`, this operator inserts the same variable into the derived class `Car`. When the number plate of an object of class `Car` is printed, the `plate` in this derived class is accessed instead of the same member in the base class, with the consequent change in the output.

– In *Tinyxml2*, a test case is required to kill a concrete mutant inserting the member variable `_value` into `XMLComment`. In the original version, when an XML document is parsed, the values are saved in the member variable associated with the generic base-class `XMLNode`; in the mutant, the value was assigned to the new member in the particular derived-class `XMLComment` instead.

5.2.2 Class mutants and traditional mutants

In this section, we analyse a set of traditional operators and the set of class operators defined in this thesis to perform a quantitative comparison between both sets. To that end, we first present the set of traditional operators selected for the study. Then, we explain the experimental procedure, including the definition of a new metric that gives us an estimation of the extent to which using class operators can help refine the test suite with respect to traditional mutants. Finally, we show the results of these experiments.

5.2.2.1 Traditional operators

Our set of class mutation operators has been compared with a set of traditional operators. Table 5.12 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 [13, 95, 102]. Offutt et al. [102] 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. Recently, some authors have shown that some variants of an operator can subsume the rest [70, 73].
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 [111].

TABLE 5.12: Traditional mutation operators included in *MuCPP*

Operator	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 (−=,+=,∗=,/=,%=)

5.2.2.2 Experiments and metric

Firstly, traditional mutants were generated to compare the number of mutants created with both types of operators. Secondly, two different experiments using the execution results of traditional and class mutants were prepared:

- **First experiment:** For each case study, we generated 30 adequate test suites derived from the augmented test suite with regard to the set of class mutants (see results in Section 5.1.3). 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.

- **Second experiment:** We defined a metric (T_d) to know the extent to which the set of class mutants could help us add new test cases with respect to traditional mutants. Let T be the test suite used, M_t the results of running each traditional mutant on each test scenario in T, and M_c the analogue of M_t for class mutants. The following procedure was carried out for each program under study:

  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. Minimise the test suite with regard to M_t, M_c and $M_{t \cup c}$. This minimisation generates the minimal suites $TM(M_t)$, $TM(M_c)$ and $TM(M_{t \cup c})$ respectively (see Section B.2 in Appendix B for further information on minimal test suites).
  3. Compare the sets $TM(M_t)$ and $TM(M_{t \cup c})$. There are two possible 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 analysing the set of class mutants.

4. Compute the metric T_d, defined as:

$$T_d = \frac{|TM(M_{t\cup c})| - |TM(M_t)|}{|TM(M_{t\cup c})|} \tag{5.1}$$

This metric will allow us to know the proportion of test cases in the minimal test suite $TM(M_{t\cup c})$ that appears when considering the class mutants in addition to the traditional ones. Note the following properties of this metric:

- If $|TM(M_t)| = |TM(M_{t\cup c})|$, then $T_d = 0$;
- 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 derived from the complete test suite T.

We also compared the metric Q_D (see Equation 2.3 in Section 2.4.4) for the set of traditional and class mutants to analyse whether the value of T_d is affected by the fact that the test suite was improved only inspecting the surviving class mutants.

5.2.2.3 Results

Table 5.13 presents the number of traditional mutants generated in these programs divided by mutation operator and in total. As it can be seen from this table, the number of traditional mutants from only 12 operators is far higher than their class-level counterparts: over four times as many when considering the total number of mutants generated by class (18,865) and traditional operators (84,639). We should note however that over 55,000 of the traditional mutants are spawned by two operators (*AIU* and *AIS*). There are many more traditional mutants than class mutants for all the programs, especially in the case of *Matrix TCL Pro*, where arithmetic operations are widely used (137 class mutants and 18,734 traditional mutants). Traditional operators have also undergone a process of analysis to avoid the generation of uninteresting mutants through their implementation. However, most class mutation operators usually entail less computation expense than traditional operators.

Figure 5.1 is the result of the **first experiment** analysing the execution of the mutants, which intends to answer if class mutants or traditional mutants can subsume the other in some way. On the one hand, *Class-Adequate Traditional MS* contains, for each program, the average mutation score associated with traditional mutants when applying

TABLE 5.13: Distribution of traditional mutants generated by program and operator (see Table 5.12)

Operator	**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

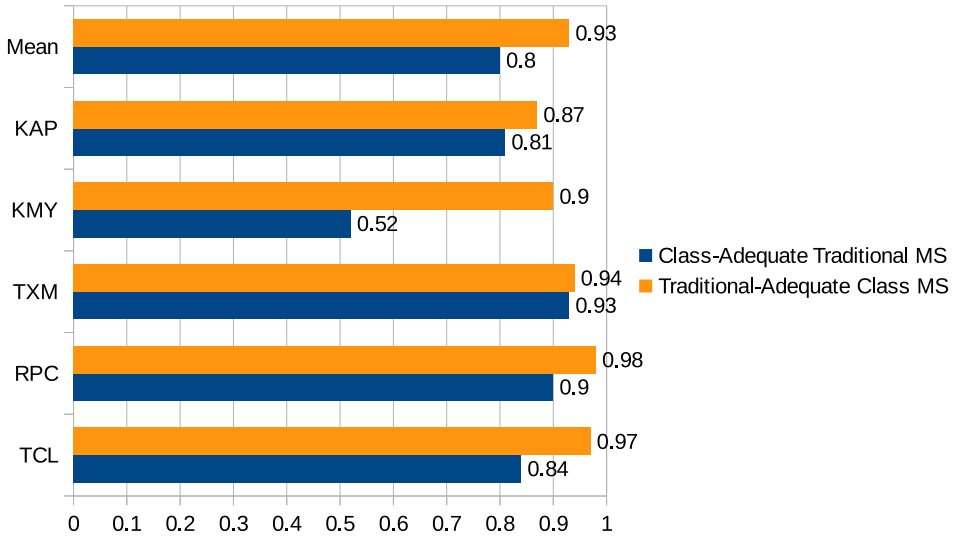


FIGURE 5.1: Average mutation scores for traditional and class mutants over 30 class-adequate and 30 test-adequate test suites

30 adequate test suites for the set of class mutants. On the other hand, *Traditional-Adequate Class MS* shows the average scores for class mutants using 30 adequate test suites with respect to traditional ones. In all cases, the class-adequate traditional mutation score is lower than the traditional-adequate class score. However, while the gap is quite significant in *KMyMoney* (0.38), the difference is small in *Tinyxml2* (0.01).

Overall, the results suggest that class mutants are easier to detect than traditional mutants and that class mutants do not cover traditional mutants (mean score of 80% in the programs). However, traditional operators are not able to completely subsume class mutants either (mean score of 93%). Moreover, even though these results suggest that only few class mutants cannot be killed through traditional mutants, we should bear

in mind that there are far fewer class mutants than traditional ones, as we have just discussed in this section.

As for the **second experiment**, the calculation of the metric T_d for each program is shown in Table 5.14, where:

- $|TM(M_t)|$ is the size of the minimal test suite to kill the set of traditional mutants.
- $|TM(M_c)|$ is the size of the minimal test suite to kill the set of class mutants.
- $|TM(M_{t\cup c})|$ is the size of the minimal test suite to kill both the set of traditional and class mutants.
- D is the result of calculating $|TM(M_{t\cup c})| - |TM(M_t)|$.
- N is the number of test cases in $TM(M_{t\cup c}) \setminus TM(M_t)$ that were modified or added to improve the test suite in Section 5.1.3.

TABLE 5.14: Calculation of Metric T_d with the improved tests for the analysed programs

Program	$\|TM(M_c)\|$	$\|TM(M_t)\|$	$\|TM(M_{t\cup c})\|$	**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

Analysing these results, it is interesting to observe in first place that $D > 0$ in all the programs; this means that the set of class-level mutants provides at least a test case to $TM(M_{t\cup c})$ which is not included in $TM(M_t)$. However, $|TM(M_t)|$ is greater than $|TM(M_c)|$ in all cases, which means that the set of traditional operators contributes with more test cases to $TM(M_{t\cup c})$ than the set of class operators. In the case of *Matrix TCL Pro*, $TM(M_{t\cup c})$ aligns with the augmented test suite (24 test scenarios, as it can be seen in Table 5.11), whereas $|TM(M_t)| = 21$ and $|TM(M_c)| = 15$ when both sets of mutants are evaluated separately. 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 in all the analysed programs except for *XmlRpc++*, ranging from 0.04 to 0.16. Recall that T_d reflects the proportion of test cases that appears in the minimal test suite exclusively when considering the mutants at the class level. That means that the rest of the test cases in $TM(M_c)$ are already included in $TM(M_{t\cup c})$ because of the execution results of the traditional mutants, so it is not unexpected that the values of T_d are low.

As commented in Section 5.1.3, when improving the test suite, we designed test scenarios which were as general as possible instead of specific scenarios to kill a particular 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 5.11), but 7 of them are also in $TM(M_t)$.

We calculated the column N because we wanted to know how many of the test cases in $TM(M_{t\cup c}) \setminus TM(M_t)$ belonged to the original test suite and to the subset of new test cases added. There are three programs in which there are some test cases from the original test suite in $TM(M_{t\cup c}) \setminus TM(M_t)$, most notably in *KMyMoney*, with 6 original test scenarios out of 13. This means that the test suite $TM(M_{t\cup c})$ is not only augmented with respect to $TM(M_t)$ because of the specificity of the new or modified test cases.

TABLE 5.15: Calculation of metric Q_D for the set of killed class and traditional mutants from the analysed programs and the improved test suite

Program	$\|K_c\|$	$\|K_t\|$	Q_{DC}	Q_{DT}	**Difference**
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 5.15 shows the calculation of:

- $|K_c|$: number of killed class mutants.
- $|K_t|$: number of killed traditional mutants.
- Q_{DC}: the metric Q_D with respect to K_c.
- Q_{DT}: the metric Q_D with respect to K_t.
- *Difference*: the result of $Q_{DC} - Q_{DT}$.

Q_D should be calculated with an adequate test suite, but when the test suite does not meet this requirement, this metric gives us an approximation to the quality of the dead mutants with that test suite.

In order to appropriately interpret these results, we have to know that:

- When $Difference > 0$, the results for K_c are of higher quality than K_t.
- When $Difference < 0$, then the results for K_t are of higher quality.

By calculating $Difference$, we have checked that there is not a significant difference in any case study after improving the test suite through surviving class mutants. In fact, Q_{DT} is even higher than Q_{DC} in *Matrix TCL Pro* and *Tinyxml2*. Again, this shows that the values obtained for the metric T_d are not due to the design of new test cases and assertions only with regard to the surviving class mutants.

5.2.3 Detected coding errors with mutation testing

Class operators have shown to be useful in suggesting key missing test scenarios in the previous sections. Nevertheless, in addition to accomplishing the main goal of mutation testing, we found some defects in the analysed code thanks to these operators. That is, we detected some coding errors while reviewing these mutants. In this section, we describe two cases in which some class mutants helped us find defects in the analysed programs:

- *Tinyxml2*: By removing the `SetAttribute(float)` method implemented in the `XMLAttribute` class, we detected that this method was not reachable by objects of the `XMLElement` class. `XMLElement` only has a `double` variant for its list of methods of the type `SetAttribute(const char*, type)`, so only the method `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.

```
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 5.2: Method "executeMulticall" in *XmlRpc++*

- *XmlRpc++*: While trying to design a test case that threw an exception in line 7 of Figure 5.2 so that it was caught in the exception handler in line 10, we detected a case of infinite recursion. The code seems to have been designed to allow the execution of multiple invocations by iterating through a data structure, but it is

not correctly implemented. The method calls itself without changing the value of `params` (marked in bold), resulting in infinite recursion and eventually a segmentation fault.

Chapter 6

Evolutionary Mutation Testing

This chapter is about the Evolutionary Mutation Testing technique. Firstly, we look in depth at how this technique operates. Secondly, we present *GiGAn*, the system implemented to apply EMT to C++ object-oriented software. Finally, we show the results of two different experiments. They are conducted to evaluate the usefulness of this technique to improve a test suite generating a reduced set of mutants.

6.1 Description

This section describes the fundamental aspects of EMT: we provide a definition of the technique, explain the *fitness function*, how *individuals* are represented and the underlying *genetic algorithm*.

As it has been mentioned throughout this document, it is desirable to reduce the number of mutants required in mutation testing as much as possible. As such, several techniques have been proposed to select a subset of mutants with almost the same ability to evaluate a test suite as the whole set of mutants (see Section 2.4.2). In the case of *Evolutionary Mutation Testing* [43], it proposes the use of an evolutionary algorithm to produce that subset of the full set of mutants. This algorithm assumes that there are some mutants with greater potential than others to guide the tester to the design of new test cases (TSR). These mutants are called *strong mutants* and the evolutionary search favours their generation because those mutants are useful to improve the quality of the test suite. Therefore, the number of mutants is reduced while preserving the power to refine the test suite.

There are two kinds of mutants considered to be strong:

- *Potentially equivalent*: mutants not detected by the test suite under evaluation. Potentially equivalent mutants either lead to the generation of new test cases or result in equivalent mutants once they are manually reviewed. Ideally, all potentially equivalent mutants help improve the current test suite with new test cases: the test suite does not cover the mutation or the test cases covering the mutation are not able to reveal it. However, some of those mutants may turn out to be equivalent as this is an undecidable problem and they cannot be automatically discarded in general.

- *Difficult to kill*: mutants killed by only one test case that kills no other mutants. These mutants represent subtle faults which require of specific test cases to be killed. Following a similar approach as the quality metric by Estero-Botaro et al [45] (see Section 2.4.4), we value that the test case detecting a difficult to kill mutant does not kill other mutants; that means that this test case might only be created by reviewing this mutant.

The rest of the mutants are regarded as *weak mutants*. In general terms, these are the steps to apply EMT:

1. Produce a subset of mutants in the first generation.
2. Execute the test suite against the subset of mutants.
3. Compute the fitness of each mutant.
4. Apply the evolutionary algorithm to produce a new generation of mutants based on the calculated mutant fitnesses.
5. Stop the algorithm if the stopping condition is reached. Otherwise, repeat the process from step 2.

As it was previously mentioned in Section 2.4.5, *GAmera* was implemented to apply EMT to WS-BPEL compositions [42]. This system is based on a genetic algorithm [51], so we will look in detail at the above process in the following sections focused on this type of algorithms. The time spent by this genetic algorithm is marginal when compared to the total time required to generate and execute the mutants, especially in non-trivial programs [43].

6.1.1 Individuals

In a genetic algorithm, each individual in a population represents a solution to the problem. The genetic algorithm tries to find the best individuals based on their fitness function.

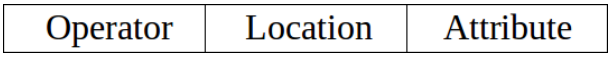


FIGURE 6.1: Encoding scheme

The individuals in EMT are the mutants and, therefore, we need to set an encoding scheme to represent them. In Section 4.1.3 we exposed that each mutant can by identified by three elements: *operator* (identifier of the mutation operator), *location* (order in the code of the mutants of an operator) and *attribute* (variant inserted into a location). These elements become important in EMT: a mutant is encoded as a combination of these three fields, as shown in Figure 6.1, given that these elements allow identifying a mutant uniquely. These fields are encoded using integer values. To help acquire a better understand of this representation, we show the example in Figure 6.2. The mutant depicted in that figure is identified as:

```
1 int f(int x, int y){
2   if(x > 0){
3     if(y > 1){
4       return y;
5     }
6   }
7   return x;
8 }
```

(a) Original

```
1 int f(int x, int y){
2   if(x > 0){
3     if(y < 1){
4       return y;
5     }
6   }
7   return x;
8 }
```

(b) Mutant

Operators	**Attributes**
1.Relational operator replacement	<=, >=, <, >, ==, !=
2.Arithmetic operator replacement	+, -, *, /, %
...	...

(c) List of operators

FIGURE 6.2: Information for mutant encoding: a) original code, b) mutant (second appearance of > replaced by <) and c) predefined positions in the list of operators and in their attributes

- **Operator = 1**: The first mutation operator in the list of operators is applied.
- **Location = 2**: The second relational operator in the code (line 3) is mutated (the first location is in line 2).
- **Attribute = 3**: The relational operator (>) is changed by the third variant (<) in the predefined set of attributes.

6.1.2 Fitness function

The fitness function measures the quality of a solution and, therefore, is devised for each specific problem. In EMT, the fitness function attaches the best value to potentially equivalent and difficult to kill mutants (strong mutants). Roughly speaking, the fitness of a mutant decreases as:

- The number of test cases detecting the mutant increases.
- At the same time, the number of mutants detected by those test cases increases.

Therefore, the generated mutants are executed on every test case to calculate their fitness function. The information is then saved in an execution matrix (see Appendix B for a definition), which helps compute the fitness of each mutant.

Equation 6.1 shows how the fitness of mutant I is computed with respect to test suite S, where M is the number of mutants, T is the number of test cases in S, and m_{ij} is 1 when mutant i is detected by test case j, and 0 otherwise.

$$\text{Fitness}(I, S) = M \times T - \sum_{j=1}^{T} \left(m_{Ij} \times \sum_{i=1}^{M} m_{ij} \right) \tag{6.1}$$

The value of the fitness function is in the range $[0, M \times T]$. In general, the greater the number of test cases killing a mutant and the number of mutants being detected by those test cases, the lower the fitness function. According to this fitness function, if mutant I is:

- Potentially equivalent, it receives the maximum value ($M \times T$) because $m_{Ij} = 0$ for all j.
- Difficult to kill, it receives a fitness of $M \times T - 1$ because $m_{Ij} = 0$ for all j except for one (test case z), which kills no other mutants ($m_{Iz} = 1$ and $\sum_{i=1}^{M} m_{iz} = 1$).
- Weak, it receives a fitness lower than $M \times T - 1$. The more test cases kill I, the lower the fitness; also, the more mutants those test cases kill, the lower the fitness.

Invalid mutants neither are assigned a fitness nor affect the fitness computation of the rest of valid mutants, as their rows are removed from the execution matrix. We should note the following regarding this fitness function:

- It penalises groups of mutants killed by the same test cases. Even if few mutants from one of those groups are produced in a generation and they are selected to breed a new generation, it is likely that several mutants from that group are created and, consequently, the fitness of the mutants in that group drops.
- Similarly, when the mutants of a group are generated by the same mutation operator, the genetic algorithm will penalise this operator focusing on other operators in successive generations.

6.1.3 Genetic algorithm

A genetic algorithm is a search-based technique that, starting from the information of a population of individuals, successively selects new individuals in order to optimise the solution. This optimisation is based on the fitness function: the algorithm maximises the sum of the fitness of the individuals in each generation to evolve toward better solutions.

As aforementioned, EMT makes use of the fitness function to find strong mutants. The genetic algorithm produces several generations formed by mutants that depend on the mutant fitnesses in the previous generations, favouring those mutants with a high value at all times. Thus, the algorithm supposes that nearby individuals are likely to be similar to those that induced their generation. The algorithm performs two main steps in each generation:

1. Generation of mutants:
   - *First generation*: a subset of mutants is generated randomly.
   - *Next generations*: a subset of mutants is generated both randomly and using *reproductive operators* with the mutants selected from the previous generation (using *selection operators*).
2. Execution of the test suite against the mutants generated:
   - *First generation*: the fitness assigned to those mutants is computed with respect to the mutants in that generation.
   - *Next generations*: unlike the first generation, the fitness of a mutant depends on all the mutants generated so far. This is achieved by storing a *second population* with the mutants created in previous generations. This helps the fitness function to produce better estimations since the fitness of a mutant can vary depending on other mutants. Therefore, this is a co-evolutive genetic algorithm because the individuals in a generation (or first population) are influenced by the individuals in the second population.

At the end of its execution, the genetic algorithm returns all the mutants stored in the second population.

6.1.4 Selection and reproductive operators

A genetic algorithm depends on two kind of operators: selection and reproductive operators.

Selection operators follow different criteria to select individuals from the population. Examples of selection methods are *tournament selection*, *rank-based selection* and *reward-based selection*. The genetic algorithm implemented in *GAmera* applies the *roulette wheel method* [51] to select the mutants. The quick convergence of this selection method is convenient in the case of EMT because we are interested in obtaining the set of strong mutants with a reduced set of mutants.

Regarding *reproductive operators*, the genetic algorithm can apply *mutation operators*[1] and *crossover operators* to individuals from the previous generation to create new ones:

- *Mutation operators:* they modify the information of one of the selected individuals to generate a new individual. As such, one of the three fields to identify an individual (*operator*, *location* or *attribute*) is mutated. The integer value of the selected field is mutated according to this equation:

$$\beta = (\alpha \pm random(1, 10(1 - p_m))) \pmod{U} \tag{6.2}$$

  Where:

  - β is the final value of the field.
  - α is the current value of the field.
  - p_m is the probability that a mutation operator is applied (configuration parameter of the algorithm).
  - α is added or subtracted a random value in the range $[1, 10(1 - p_m)]$. In this way, the upper limit of this range decreases as p_m increases, which reduces the impact of the mutation.
  - U is the maximum value that the field can be assigned.
  - The operation is carried out modulo U to avoid generating invalid mutants.

[1] Do not confuse these mutation operators with the mutation operators applied in mutation testing.

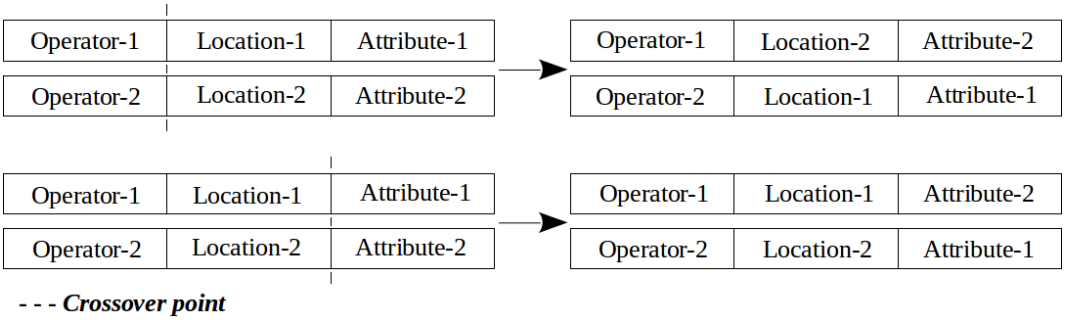


FIGURE 6.3: Mutant crossover

- *Crossover operators:* they combine the information of two individuals (parents), ($operator_1$, $location_1$, $attribute_1$) and ($operator_2$, $location_2$, $attribute_2$), to generate two new individuals (children), which inherit information from both parents. To that end, a crossover point related to the encoding scheme is selected. In this case, the genetic algorithm contemplates two crossover points (see Figure 6.3):
  - *Point 1:* generates the individuals ($operator_1$, $location_2$, $attribute_2$) and ($operator_2$, $location_1$, $attribute_1$).
  - *Point 2:* generates the individuals ($operator_1$, $location_1$, $attribute_2$) and ($operator_2$, $location_2$, $attribute_1$).

Given that each mutation operator produces a different number of mutants, we should note that a process of normalization of the fields (location and attribute) avoids that invalid representations of mutants are produced. Those fields are encoded by a value in the range 1 to the least common multiple of the number of locations and attributes respectively of all the operators that can be applied to the subject program.

6.2 GiGAn

The genetic algorithm described in the previous section is implemented in *GAmera* [42]. This system makes use of *MuBPEL* to analyse, generate and execute the mutants for WS-BPEL compositions. In order to reuse the same genetic algorithm implemented in *GAmera*, we have developed a new system called *GiGAn*[2]. Analogously to *GAmera* and *MuBPEL*, *GiGAn* connects the genetic algorithm to *MuCPP*.

Figure 6.4 displays how *GiGAn* connects *MuCPP* and *GAmera* to apply EMT to C++ object-oriented systems. As it can be seen, *GiGAn* acts as a bridge between the mutation tool and the genetic algorithm, translating the commands and mapping mutant identifiers

[2] *GiGAn*, like *GAmera*, belongs to the set of *daikaiju* and *kaiju* creatures that appears in Japanese movies of the 1960s and 1970s. Uppercase letters are used for 'GA' (Genetic Algorithm).

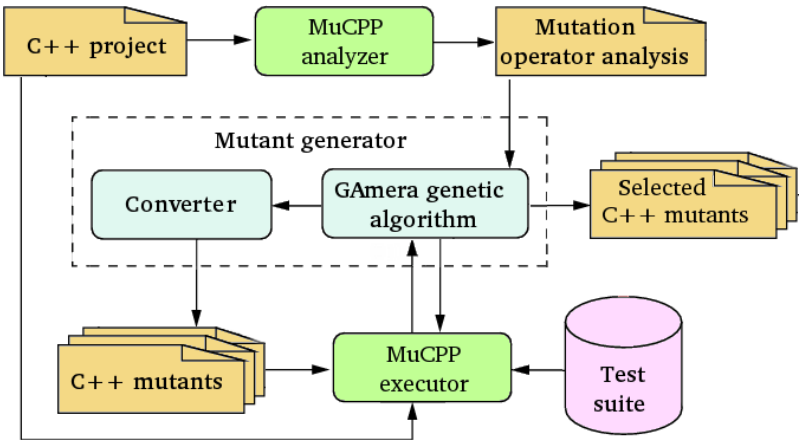


FIGURE 6.4: *GiGAn* diagram

so that *MuCPP* and *GAmera* can work together. The process orchestrated by *GiGAn* is as follows:

1. *MuCPP* analyses the C++ source code of the project. This generates a report with a list of the mutants that each mutation operator can produce in the code (mutation operator analysis).
2. The genetic algorithm implemented in *GAmera* uses the report to know the individuals that can be generated.
3. The genetic algorithm selects several mutants in a generation and a converter transforms the individuals into usable mutant identifiers for *MuCPP*.
4. *MuCPP* generates and executes the mutants on the test suite, resulting in an execution matrix that is used by the genetic algorithm to compute the fitness function.
5. Steps 2 to 5 are repeated until the stopping condition is satisfied, for instance, when reaching a percentage of the full set of mutants or a number of generations. The output is the set of selected C++ mutants in all the generations.

The experiments conducted in this thesis using *GiGAn* presents two main changes with respect to the experiments using *GAmera* [43]. These two differences are highlighted because they can impact the evaluation of EMT:

- **Attribute:** In Section 4.1.3, we explained that we distinguish between fixed attribute (when the mutations injected into a location are known in advance) and variable attribute (when the number of variants depends on the context). Only in the former case, the genetic algorithm can select the attribute of the mutant to

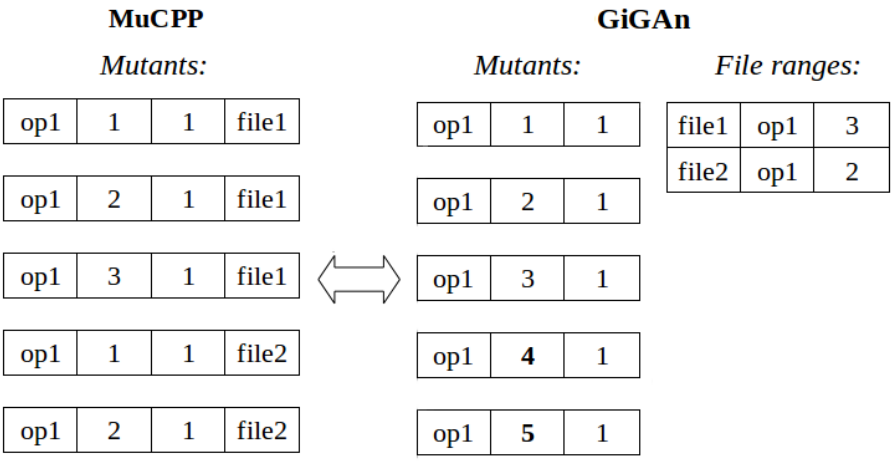


FIGURE 6.5: Example of mutant mapping between *MuCPP* and *GiGAn* when one operator (*op1*) generates mutants in several files (*file1* and *file2*)

be generated with the same probability. As a consequence, in the latter case, the attribute is marked as 1 and each mutation in the same location is simply counted as a new location instead.

In the case of mutation testing at the class level for C++, either mutation operators have $attribute = 1$ or variable attribute. In other words, none of the class operators has $attribute > 1$. Thus, we limit reproductive operators to mutation of *operator* and *location* fields and *point 1* crossover (see the previous section), as the rest of reproductive operators would result in the same mutant being created.

- **Mutants in different source files:** In Section 4.3.1, we commented that *MuCPP* allows analysing several source files of a project in the same execution. Thus, *MuCPP* uses a further field to identify a mutant: the *source file*. Provided that two mutations are in different files, they can have the same fields *operator*, *location* and *attribute*. This fact is not contemplated in the genetic algorithm implemented in *GAmera*, which only uses the aforementioned three fields to identify a mutant. To handle this disparity, we map the mutant identifiers used by *MuCPP* (with the field *file*) and the genetic algorithm (without the field *file*), as shown in Figure 6.5. We make use of the field *location* and the ranges of mutants in each file to that end. Notice that *GiGAn* knows which mutant belongs to which file because the source files are sorted beforehand. For instance, the mutant with $location = 4$ in Figure 6.5 belongs to $file2$ because $file1$ only contains 3 mutants and it is analysed before $file2$ (see "File ranges").

  This fact is relevant to this evaluation because mutants from different source files can be generated when the field *location* is modified to produce new individuals from previous ones (using reproductive operators). Even though classes in a project

often use a similar design pattern, it is possible that the behaviour of a mutation operator varies for different classes, especially when they belong to different source files.

6.3 Experiment 1: Finding Strong Mutants

In this first experiment, we want to study the ability of the genetic algorithm to find strong mutants. To do so, we calculate the number of mutants generated to reach different percentages of strong mutants, and we also compare the results of EMT and a random strategy.

6.3.1 Setup

First of all, we need to be aware that EMT has to be configured with several parameters and that the results of these experiments are subject to the selected configuration (which will be the same for all the applications). Namely, the parameters are:

- *Population size*: It is the number of individuals to produce in each generation. This parameter is a percentage of the total of mutants in each application.
- *Individuals generated randomly and by reproductive operators*: The mutants in a generation are produced either randomly or by reproductive operators (see Section 6.1.3), so these parameters set the percentage of mutants generated by each of these methods. The sum of both percentages has to be 100%.
- *Mutation and crossover probability*: These parameters reflect the probability that mutation or crossover operators are used when a mutant is generated through reproductive operators. As in the previous item, both parameters have to sum 100%.

Domínguez-Jiménez et al. [43] experimented with different values for these parameters and determined an optimal combination. These values, which can be seen in Table 6.1, have been used for the execution of this algorithm in our experiments.

All mutants were generated and run on the test suite in a previous execution, resulting in an execution matrix. This execution allows us to maintain a record of strong mutants in the subjects under study with the current test suite, which is used as a ground truth to compute our results. We can divide the experiment into two parts:

TABLE 6.1: Genetic algorithm configuration

Block	**Parameter**	**Value**
-	Population size	5%
Individual generation	Random Reproductive operators	10% 90%
Reproductive operators	Mutation probability Crossover probability	30% 70%

First part of the experiment: We established several stopping conditions for the algorithm: finding 30%, 45%, 60%, 75% and 90% of the set of strong mutants. Then EMT was run 30 times with different seeds for each of the stopping conditions. Therefore, the statistics are obtained from the results of these 30 executions.

Second part of the experiment: We executed 30 times a random strategy (*Random* from now on) where mutants were selected one by one until reaching the stopping condition, and then we compared these results with the data reported by EMT in the first part of the experiment.

6.3.2 Results

Table 6.2 collects several statistics (mean, median, minimum, maximum and standard deviation of the results of the 30 executions) about the percentage of mutants that EMT needs to generate before finding different percentages of the set of strong mutants. The results in this table are divided by program and stopping condition.

As it was expected, we can observe that the percentage of necessary mutants increases as the stopping condition becomes more demanding in all the programs. For instance, the percentage of mutants generated in *Dolphin* increases continuously from 28.35% (to find 30% of strong mutants) to 85.35% (to find 90% of strong mutants). In order to study the tendency of this increase in each program, Figure 6.6 depicts the average percentage of mutants generated in them for each of the five stopping conditions. Given that the stopping conditions have been selected in 15% increments, this graphic reflects that:

- The upward tendency is quite stable among applications.
- The relation between the number of mutants generated and the percentage of strong mutants is almost linear.
- There is often a small increment in the percentage of mutants generated as the stopping condition increases. Taking *Tinyxml2* to illustrate this fact, on average EMT needs to produce around 12.5% more mutants to find 45% of the strong

TABLE 6.2: Percentage of the total of mutants generated in the programs with EMT (a) and with Random (b) to achieve 30%, 45%, 60%, 75% and 90% of the strong mutants (SD: standard deviation)

(a) EMT results

Program	Statistic	*30%*	*45%*	*60%*	*75%*	*90%*
TCL	Mean	23.45	37.59	53.55	67.59	84.33
	Median	24.08	39.05	54.74	67.52	83.94
	Min.	13.13	25.54	40.14	51.09	70.07
	Max.	37.22	51.09	65.69	79.56	92.70
	SD	5.44	6.67	6.10	6.98	5.21
DPH	Mean	28.35	41.94	55.19	69.87	85.35
	Median	28.54	42.23	54.79	70.09	85.38
	Min.	24.65	38.35	50.68	62.55	78.99
	Max.	33.33	47.03	59.81	76.71	90.41
	SD	2.11	2.28	2.10	3.57	2.67
TXM	Mean	24.09	36.62	49.74	64.91	84.32
	Median	24.18	36.07	49.67	64.74	84.12
	Min.	20.52	31.92	44.78	60.58	77.85
	Max.	27.85	41.04	56.35	71.49	89.73
	SD	1.61	2.34	3.05	2.59	3.34
DOM	Mean	21.20	34.86	52.21	69.96	87.84
	Median	21.16	34.86	52.31	70.15	88.09
	Min.	19.02	32.28	46.59	66.05	83.33
	Max.	23.03	37.26	57.06	73.38	90.13
	SD	1.01	1.26	2.39	1.98	1.60

(b) Random results

Program	Statistic	*30%*	*45%*	*60%*	*75%*	*90%*
TCL	Mean	28.61	43.13	57.90	72.53	88.25
	Median	28.10	44.16	58.76	72.26	88.32
	Min.	16.78	31.38	44.52	60.58	79.56
	Max.	40.87	54.74	70.07	79.56	94.16
	SD	6.03	5.17	5.63	3.89	3.57
DPH	Mean	29.89	45.76	59.98	75.11	89.07
	Median	29.22	44.97	59.59	75.79	89.72
	Min.	24.65	38.35	52.96	67.57	82.64
	Max.	35.15	53.42	67.12	81.27	93.15
	SD	3.07	3.98	4.30	3.57	2.86
TXM	Mean	29.09	44.50	59.23	74.93	89.98
	Median	28.83	44.54	59.12	74.83	90.22
	Min.	22.63	38.27	54.23	69.70	85.01
	Max.	35.01	51.14	65.63	80.61	93.64
	SD	3.14	3.40	3.03	2.78	1.78
DOM	Mean	29.49	44.16	59.38	74.43	89.76
	Median	29.23	44.11	59.60	74.34	89.75
	Min.	25.65	39.09	54.88	71.64	86.21
	Max.	34.29	49.47	63.96	78.88	93.71
	SD	2.14	2.28	2.51	2.00	1.58

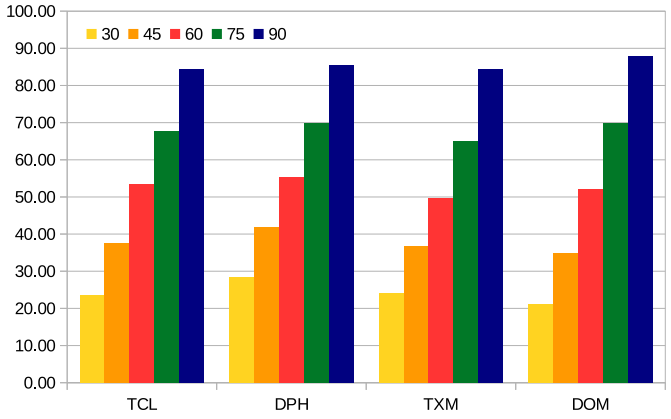


FIGURE 6.6: Average percentage of mutants generated with EMT in the programs to reach the five different stopping conditions

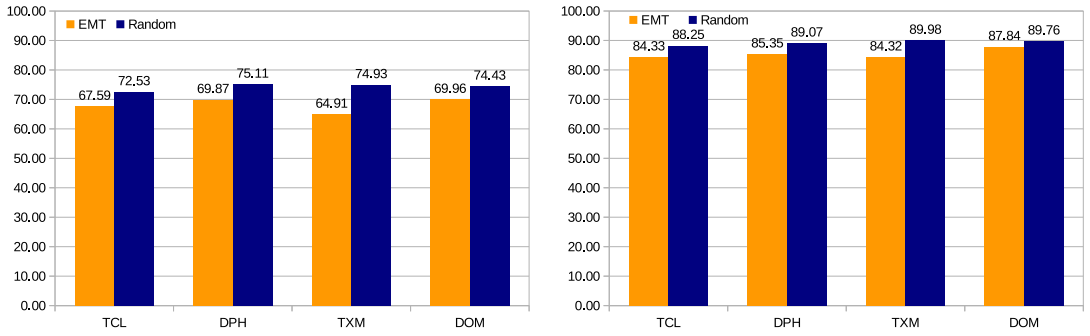
mutants than to find 30%. However, this difference increases when considering the rest of conditions: 45%-60% (13.1%) 60%-75% (15.2%) and 75%-90% (19.4%).

Despite these facts, the best results in terms of the relation between the number of mutants and the percentage of strong mutants are not necessarily obtained with the lowest stopping condition. Considering *Tinyxml2* again, the benefits of using EMT are more notable for the stopping condition 60% (49.74% of mutants are generated to find 60% of strong mutants) than for the stopping condition 30% (24.09% of mutants are generated to find 30% of strong mutants). As such, the effectiveness of this technique also depends on the moment when the algorithm stops.

The standard deviation does not follow a pattern and is quite low, except for *Matrix TCL Pro* where it might be affected by the few mutants in this program.

Table 6.2 also shows the results of Random (in the same format as the statistics of EMT are presented). In the light of the results, EMT produces a better outcome than the random selection of mutants in all cases and statistics, except for the standard deviation (where we can observe varying results). Figure 6.7 graphically shows the difference between both techniques focused on the average results of the two more demanding stopping conditions (75% and 90%).

In order to know about the significance of these results, we run a statistical test using the web application *STATService* [121], which selects an appropriate statistical test depending on the introduced data (called *smart* statistical test). The *p-value* obtained with the smart test for the stopping conditions 75% and 90% is collected by Table 6.3. These results lead us to accept that the median percentage of mutants that EMT needs to generate to find a subset of strong mutants is significantly lower than with random selection within a 99.9% confidence interval. We also computed the non-parametric *Vargha and*



(a) 75% (b) 90%

FIGURE 6.7: Average percentage of the total of mutants generated with EMT and Random to achieve 75% (a) and 90% (b) of the strong mutants

TABLE 6.3: Results of the smart and *Vargha and Delaney's* A_{12} statistical tests

	75%		***90%***	
Program	p-value	A_{12}	p-value	A_{12}
TCL	2.26×10^{-03}	0.711	1.24×10^{-03}	0.734
DPH	4.55×10^{-07}	0.848	2.72×10^{-06}	0.829
TXM	7.14×10^{-21}	0.996	1.65×10^{-06}	0.937
DOM	4.31×10^{-12}	0.962	1.71×10^{-05}	0.816

Delaney's A_{12} statistic to complement this study with the evaluation of the effect size (see Table 6.3). The difference between both algorithms can be described as large in all cases, especially in *Tinyxml2* where the best results are achieved (with a difference over 10% for the 75% stopping condition).

Still, the gap between both strategies in the experiments by Domínguez-Jiménez et al. [43] is greater than in this study in the better case (16% on average for the more complex WS-BPEL composition when trying to find all strong mutants). As the results in Chapter 7 suggest, it seems that each of the class operators addresses different object-oriented features in general. As a consequence, strong mutants may be spread across a large subset of the operators. We suspect that this fact may limit the benefits of using this genetic algorithm at the class level.

6.4 Experiment 2: Improving the Test Suite

A percentage of the strong mutants selected in the previous experiments may be later found to be equivalent mutants (once the mutants are reviewed). Being aware of this fact, finding a subset of strong mutants does not ensure that the test suite is proportionally improved with new test cases. This is the origin of the experiments in this section.

In this second experiment, we seek to simulate the process of generating a subset of mutants and then use the information of their execution to design new test cases. Instead of measuring how many of the generated mutants are strong, we estimate how much the test suite is actually improved thanks to those mutants. The goal is to know which algorithm, the genetic or the random algorithm, is able to augment the test suite in a number of test cases but generating fewer mutants.

6.4.1 Setup

The aforementioned simulation was carried out in **two different phases**: the test suite is improved in the first phase and it is used in the simulation in the second phase.

In the *first phase*, we obtain adequate test suites for the case studies as follows:

1. Execute the current non-adequate test suite (T_{NA}) against all the mutants.
2. Review the surviving mutants and identify equivalent and non-equivalent mutants.
3. Design new test cases to kill all the surviving non-equivalent mutants. At the end of this step, we achieve an adequate test suite (T_A).
4. Execute all the test cases in T_A against all the mutants, obtaining the final execution matrix associated with T_A (EM). This execution matrix contains the information about which mutants can induce the generation of which test cases in that test suite.
5. Minimise T_A using the information in EM. At the end of this step, we have a minimal and adequate test suite (T_{MA}).

For instance, if we have the execution matrix in Figure 6.8 associated with the current test suite (T_{NA}), we can use that information and augment the test suite until reaching an adequate test suite (T_A). When applied to the set of mutants, T_A produces the execution matrix (EM) in Figure 6.9, where:

- The mutants m_2, m_4 and m_7, which remained alive with T_{NA}, are now killed.
- $m2$ and m_4 are killed with two new test cases, $test_6$ and $test_7$ respectively; m_7 is killed with a modified test case ($test_4$).
- The mutant m_8 turns out to be equivalent.

	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$
m_1	1	0	1	0	0
m_2	0	0	0	0	0
m_3	0	1	1	0	0
m_4	0	0	0	0	0
m_5	2	2	2	2	2
m_6	0	0	0	0	0
m_7	0	0	0	0	0
m_8	0	0	0	0	1

FIGURE 6.8: Example of execution matrix associated with a non-adequate test suite T_{NA} and the whole set of mutants

	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$	$test_6$	$test_7$
m_1	1	0	1	0	0	0	0
m_2	0	0	0	0	0	1	0
m_3	0	1	1	0	0	0	0
m_4	0	0	0	0	0	0	1
m_5	2	2	2	2	2	2	2
m_6	0	0	0	0	0	0	0
m_7	0	0	0	1	0	0	0
m_8	0	0	0	0	1	0	0

FIGURE 6.9: Example of execution matrix associated with an adequate test suite T_A and the whole set of mutants (EM)

In the *second phase*, we make use of the information in EM to know how many mutants the genetic algorithm would need to generate to reach the stopping condition: the algorithm stops when reaching a given percentage (P) of the number of test cases in the minimal and adequate test suite ($|T_{MA}|$) with the subset of mutants generated.

Let i be an index to refer to a generation of the genetic algorithm. We run EMT and the genetic algorithm follows these steps, starting from $i = 1$:

1. Select the mutants to produce in generation i (M_i).
2. Select the rows of EM corresponding to the mutants in the set of generations $\{M_0,...,M_i\}$, producing EM_i. In this step, we create a new execution matrix associated with the generation i (EM_i), but selecting the rows in EM of the mutants generated so far by the algorithm ($\{M_0,...,M_i\}$).
3. Minimise T_A using the information in EM_i. At the end of this step, we obtain a minimal and adequate test suite for the mutants selected by the algorithm so far (T_{MA_i}).
4. Go to step 1 (the index i is increased by one) until the stopping condition is reached. Therefore, the genetic algorithm stops when $|T_{MA_i}| \geq |T_{MA}| \times P$.

	test_1	test_2	test_3	test_4	test_5	test_6	test_7
m_1	1	0	1	0	0	0	0
m_3	0	1	1	0	0	0	0
m_4	0	0	0	0	0	0	1
m_7	0	0	0	1	0	0	0

FIGURE 6.10: Example of execution matrix associated with an adequate test suite T_A and the subset of mutants generated by EMT after two generations (EM_2)

In this simulation, we count with the test suite in the present (T_{NA}) for the execution of EMT, and a test suite in the future (T_A) for the stopping condition. We also have the result of the execution of the mutants on the adequate test suite in the future (EM). The idea is to simulate a real process of test suite improvement; we stop the algorithm when it has generated enough mutants to improve the test suite in a percentage with respect to T_A. By extracting from EM the information of the execution of the mutants generated by the algorithm, we can estimate how many test cases those mutants would induce.

Using the same example as in the first phase, if EMT selected the subset of mutants m_1, m_3, m_4 and m_7 in the first two generations, we could extract from EM the information of the exectuion of those mutants, as shown in Figure 6.10. As a result of that, we could estimate how much that subset of mutants would help improve the test suite. In this case, when minimising the test suite in the second generation (T_{MA_2}) and the adequate test suite (T_{MA}), we have that $|T_{MA_2}| = 3$ ($test_3$, $test_4$ and $test_7$) while $|T_{MA}| = 5$ ($test_3$, $test_4$, $test_5$, $test_6$ and $test_7$). As such, if our stopping condition was lower than or equal to $P = 60\%$, the algorithm would stop ($|T_{MA_2}|\,(3) \geq |T_{MA}|\,(5) \times P(60\%)$); otherwise, the algorithm would produce a new generation.

The experiment has been executed with two stopping conditions (two values for P): reaching 75% and 90% of the minimal and adequate test suite (T_{MA}). Note that we do not compute the results with lower values for P (as in the previous experiment) because the current test suite already contributes to the minimal and adequate test suite. Therefore, demanding stopping conditions are required so that some of the new test cases appear in the minimal and adequate test suite T_{MA_i}.

As in the first experiment in the previous section, we executed the second phase 30 times with different seeds. The same process is followed for the Random algorithm, where only one mutant is randomly selected in step 1 of the second phase.

6.4.2 Results

Table 6.4 shows the results of the experimental procedure explained in Section 6.4.1. The same statistics shown in the first experiment have been computed. The figures shown in the table represent the percentage of mutants that have been generated to reach the stopping conditions by EMT and Random. Therefore, the lower the percentage, the better.

TABLE 6.4: Percentage of mutants generated with the evolutionary and the random strategy to reach the stopping conditions (75% and 90% of the minimal and adequate test suite) in the subjects under study

P		***75%***		***90%***	
Program	Statistic	*EMT*	*Random*	*EMT*	*Random*
TCL	Mean	37.24	32.45	49.24	47.85
	Med.	38.32	33.57	50.36	50.36
	Min.	18.97	14.59	25.54	25.54
	Max.	54.74	52.55	75.91	68.61
	SD	10.77	9.09	13.41	12.78
DPH	Mean	49.75	54.10	66.33	71.08
	Med.	48.63	52.05	65.29	69.63
	Min.	36.52	30.13	52.51	40.63
	Max.	74.42	84.01	84.93	88.58
	SD	8.51	9.95	8.61	10.45
TXM	Mean	19.26	25.75	31.93	46.79
	Med.	18.48	24.34	32.25	43.24
	Min.	10.58	11.88	20.52	24.75
	Max.	27.36	53.09	46.09	86.80
	SD	4.38	8.98	7.13	15.21
DOM	Mean	13.33	23.74	21.41	49.04
	Med.	13.00	21.90	21.16	46.68
	Min.	7.85	11.16	11.95	26.96
	Max.	23.29	49.04	35.86	81.15
	SD	3.35	7.99	5.00	12.85

Before interpreting these results, we should note that the evaluation is impacted by the test suite itself. For instance, not in all the subjects under study the current suite (T_{NA}) is at the same distance of the adequate test suite (T_A); the difference between the sizes of the current non-adequate test suite and the adequate test suite when they are minimised is not the same for all the programs. The size of the test suite (having few or many test cases) may also affect the search and the results of the genetic algorithm. Moreover, each test suite has a different killing power (whether mutants are killed by few or many of its test cases in general), which depends on the nature of the test cases (general or specific test cases). Thus, we can observe that, on average, EMT needs to generate 49.75% of mutants to reach 75% of T_{MA} in *Dolphin*, whereas it only needs to generate 13.33% in

QtDom for the same end. As such, we should not directly compare the results among applications, as done in the first experiment.

The results reported by EMT are better than the results reported by Random in *Dolphin*, *Tinyxml2* and *QtDom*, but worse in *Matrix TCL Pro*. In this experiment, we can observe that the results scale with the size of the program, given that the best result is obtained in *QtDom*, followed by *Tinyxml2* and *Dolphin* (in descending order of the number of mutants). If we focus on the 75% stopping condition, the difference between EMT and Random increases from about 5% in *Dolphin* to 10% in *QtDom*. We can also note that the outcome is better for the most demanding condition (90%). Again, the difference between EMT and Random increases from about 5% in *Dolphin* to 28% in *QtDom*. When comparing the results for both stopping conditions, we can see that the gap between EMT and Random is wide in the case of:

- *Tinyxml2*: approximately 6% for $P = 75\%$ and 15% for $P = 90\%$.
- *QtDom*: approximately 10% for $P = 75\%$ and 28% for $P = 90\%$.

The standard deviation is lower in the executions of EMT than in Random for *Dolphin*, *Tinyxml2* and *QtDom* and both stopping conditions. The standard deviation is especially low for EMT in comparison with Random in *Tinyxml2* and *QtDom*.

However, we should note that the difference between both techniques may be impacted by the number of invalid mutants, which is higher in *Tinyxml2* and *QtDom* than in *Matrix TCL Pro* and *Dolphin*. In the first experiment (see Section 6.3), selecting either a weak mutant or an invalid mutant has no effect on the stopping condition for Random; unlike the first experiment, while invalid mutants do not affect the moment when the algorithm stops, selecting a weak mutant may increase the size of the minimal and adequate test suite T_{MA_i} in this second experiment. Studying how much of the difference between both algorithms is due to the existence of invalid mutants in these programs could be addressed in future experiments. Still, this fact also means that EMT has the ability to avoid the generation of invalid mutants, especially when they are generated by a subset of the operators.

Regarding *Matrix TCL Pro*, where the result of Random was better, we suspect that these results are due to the test cases for this application. These test scenarios are quite general in the sense that several invocations are needed before testing a particular functionality and some other test scenarios cover a subset of related functionalities instead of a single functionality. This type of test cases does not usually lead to mutants killed by few test cases. On the contrary, given that these tests cover a great part of the code, these test cases tend to detect several mutants. As it was mentioned in Section 5.1.3, we created

new test scenarios as complete as possible, following a similar design pattern as the rest of scenarios in that test suite. As a result, some of the new or modified test cases (those that were manually designed) are likely to appear in this experiment without generating the mutants that led to the design of those test cases. In summary, it is easy to reach 75% and 90% of the minimal and adequate test suite with different combinations of mutants. This fact can be disadvantageous for EMT because this strategy is guided by the fitness function to find a specific subset of mutants (which includes equivalent mutants).

TABLE 6.5: Average percentage of mutants generated with the evolutionary and the random strategy to find the whole minimal and adequate test suite in the subjects under study

Program	*EMT*	*Random*
TCL	75.79	80.05
DPH	88.24	91.63
TXM	49.30	75.92
DOM	34.69	80.51

As a result of the above, we carried out the same simulation to find the complete minimal and adequate test suite ($P = 100\%$). By doing so, it is expected that EMT benefits from the fitness function to find the most specific test cases quicker than the random strategy. The average results for all the applications are shown in Table 6.5, which confirms that EMT is more effective than Random in finding the whole adequate and minimal test suite for *Matrix TCL Pro*. Again, this situation shows that the assessment is subject to the test suite. As it can be seen, the results are again especially positive for the largest programs in terms of mutants generated; the difference between EMT and Random is around 26% and 45% in *Tinyxml2* and *QtDom* respectively.

Table 6.6 also reveals that the average percentage of mutants that EMT needs to generate notably increases from $P = 90\%$ to $P = 100\%$ when compared to the difference between $P = 75\%$ and $P = 90\%$. Nonetheless, this is an expected outcome: some test cases are induced by mutants that are generated by operators either producing very few mutants or generating mutants with a low fitness function overall (that is, there are some high-quality mutants generated by low-quality operators). As such, these mutants are not easy to find by the guided search of the genetic algorithm.

TABLE 6.6: Differences in the average percentage of mutants generated between $P = 75\%$ and $P = 90\%$, and between $P = 90\%$ and $P = 100\%$ in the subjects under study

Program	*75–90%*	*90–100%*
TCL	12.00	26.55
DPH	16.58	21.91
TXM	12.67	17.37
DOM	8.08	13.28

Chapter 7

Selective Mutation Assessment

In this chapter, we delve into the analysis of mutation operators following a selective approach. In particular, we divide our analysis into two dimensions: classifying the operators into a ranking around their mutant redundancy for the evaluation of test suites and another ranking regarding their ability to improve the quality of tests. Once both rankings are obtained, we apply two selective strategies based on the best-valued mutation operators in the rankings. The results are then compared to other traditional techniques for the selection of mutants, validating the used metrics for the operator classification.

7.1 Selective Approach

In this section, we set the basis for the selective mutation study in the following sections, describing the double perspective to classify operators, the selective strategies applied in the experiments and the metrics to evaluate the effectiveness of our approach.

7.1.1 Test suite evaluation and test suite refinement

Mutation testing is mainly used for two purposes: evaluate and refine test suites. In our thesis, we conjecture that the value of each mutation operator differs depending on whether the test suite is being evaluated or refined:

- **Test Suite Evaluation** (TSE): mutation testing is used to assess how effective a test suite is at detecting faults. Several studies have observed that some

mutants can be *redundant* and therefore removed without impacting the effectiveness. Therefore, redundant mutants should be removed as much as possible to reduce the computational cost.

- **Test Suite Refinement** (TSR): mutation testing guides the tester on the improvement of the test suite by designing new test cases that kill the surviving mutants. Some mutants may be more effective than others in guiding the tester on the creation of *high-quality test cases*. We say that the quality of a test case is high when it detects non-trivial faults which are not easy to find with a straightforward test case. Therefore, those mutants that contribute to creating high-quality test cases should be favoured as much as possible.

Based on this idea, we rank C++ class mutation operators regarding their influence during TSE and TSR respectively. While the ranking for TSE arranges the operators according to their degree of redundancy, the ranking for TSR sorts them regarding their potential to contribute to the creation of high-quality test cases (we use the metric devised by Estero-Botaro et al. [45] to that end). These two rankings are the basis of the selective mutation study in this chapter, where we show the trade-off between discarding mutation operators and the loss of effectiveness.

As such, we divide our selective approach into TSE and TSR, which will be studied separately in Section 7.2 and Section 7.3 respectively.

7.1.2 Selective strategies

Mutation testing suffers from two main drawbacks. Firstly, it has a high computational cost due to the potentially large number of mutants that can be generated. Secondly, the technique is limited by the existence of equivalent mutants; even when the number of mutants is manageable, the effort required to identify equivalent mutants could make the application of the technique unbearable. This led to researchers in this field to seek other alternatives to the generation and evaluation of the whole set of mutants. *Selective mutation* is a well-known cost reduction technique to exclude some of the mutants while retaining effectiveness (see Section 2.4.3 for further information).

In this study, we will distinguish **two main selective strategies**:

- **Operator-based selective mutation**: traditional definition of selective mutation [13, 102]. It works under the assumption that not all mutation operators are equally effective and that there should be a *sufficient set of operators* which is representative of the full set of mutation operators.

- **Mutant-based selective mutation**: Unlike operator-based selective mutation, a subset of mutants (instead of a subset of operators) is discarded. Mutant sampling can be categorised as a mutant-based selective strategy [20, 126].

Both strategies will be applied later on in the study in order to know whether C++ class mutation operators exhibit any *degree of redundancy* (percentage of redundant mutants generated by each operator) or the extent to which they *contribute to creating high-quality test cases*. This could help us estimate the loss of accuracy that we concede when using them in a selective mutation process.

7.1.3 Test-Quality selective mutation

Studies in the literature about selective mutation have mainly sought to find a sufficient set of mutation operators that allows us to accurately predict the overall mutation score when applying operator-based mutant selection. In other words, we obtain a sufficient set of operators if the mutation score, when measured against the original set of operators, correlates with the mutation score associated with the reduced set of operators. This is the approach to selective mutation when it comes to evaluating the fault detection capability of the test suite (TSE).

However, during TSR we focus on the improvement of the test suite with high-quality test cases, and the best mutation operators are not necessarily those with the greatest potential to predict the mutation score of the full set of operators. As a consequence, we should not apply selective mutation and compute the mutation score (as traditionally done) to evaluate the effectiveness of the used metric in TSR.

We can illustrate with a simple example why a new approach related to test quality is required. Consider the executing matrix in Figure 7.1. If we had to select only one mutant to refine our test suite, we would select the mutant m_1 because it is a resistant hard to kill mutant (see Appendix B). Therefore, the mutation score is low because $test_1$ does not kill any other mutants, but we are retaining a test case which is not easy to design. As it can be seen, the mutation score is not an appropriate method to measure the effectiveness of a set of mutation operators in the refinement of the test suite.

As a result, our proposal in this study is applying:

- **Traditional selective mutation for TSE**, where we seek for a representative subset of mutants and the test suite effectiveness is measured using the *mutation adequacy score*.

$$\begin{array}{c} \\ \mathrm{m}_1 \\ \mathrm{m}_2 \\ \mathrm{m}_3 \\ \mathrm{m}_4 \end{array} \begin{array}{c} \begin{array}{ccc} \mathrm{test}_1 & \mathrm{test}_2 & \mathrm{test}_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

FIGURE 7.1: Execution matrix to illustrate the difference between TSE and TSR with respect to selective mutation

- **Test-quality selective mutation for TSR**, where we seek for a subset of mutants that allows us to leverage the information of surviving mutants in such a way that the test suite is enhanced with as many high-quality test cases as possible. We define the metric **test suite size** to compute its effectiveness: *percentage of test cases loss* when compared to T, the original adequate and minimal test suite (see Appendix B). Let n be a number to represent the n selected mutants (represented by $m_1...m_n$), the percentage of test cases loss is measured as follows:

$$\frac{|T| - |T_{m_1...m_n}|}{|T|} \times 100$$

  The lower the percentage, the fewer test cases we are losing because of removing the rest of mutants which are not in the subset $\{m_1...m_n\}$.

As a final remark, effectiveness in mutation testing has been traditionally associated with the capability of the test suite to kill as many mutants as possible. In our approach, whether the test suite kills a large number of mutants is unimportant as we just seek a set of operators which is effective at refining the test suite with specific test cases.

7.1.4 Rank-based selective mutation

As we mentioned earlier in Section 7.1.2, we follow a mutant-based selective strategy in addition to an operator-based selective one. However, we cannot assess the performance of the operator rankings by randomly selecting individual mutants from the operators. Instead of random mutant selection, we apply *rank-based selective mutation*, which favours the selection of mutants from the top ranked operators.

In this strategy, we follow a similar approach to the *two-round random* selection technique proposed by Zhang et al. [135]. While in the two-round random technique the number of mutants selected from each operator is probabilistically speaking about the same, in rank-based mutant selection we seek to generate more mutants from the top-ranked

operators than from the operators at the bottom of the ranking. Our rank-based mutant selection comprises two steps:

1. *Operator selection*: The probability of being selected for an operator is proportional to its position in the ranking. As an example, consider the following ranking with three operators:
   - First operator in the ranking (top ranked) → it will be selected with probability 3/6.
   - Second operator in the ranking → it will be selected with probability 2/6.
   - Third operator in the ranking → it will be selected with probability 1/6.
2. *Mutant selection*: A mutant is randomly selected from the operator previously selected.

7.1.5 Selective assessment

This section summarises the evaluation that we will perform taking into account the information presented in previous sections. The evaluation of selective mutation in this chapter is as follows:

1. We divide our assessment into **two main blocks**:
   - Test Suite Improvement.
   - Test Suite Refinement.

   For each of these blocks, we obtain a different *ranking of mutation operators* depending on how good the operators are for TSE and TSR based on two different metrics.
2. For both TSE and TSR, we apply **two selective strategies**:
   - Operator-based selective mutation.
   - Mutant-based selective mutation and, more specifically, rank-based selective mutation.

   Both strategies follow the aforementioned rankings of mutation operators to discard some of the mutants.
3. To evaluate each block, we use **two different selective approaches**:
   - *TSE*: Traditional selective mutation.

- *TSR*: Test-quality selective mutation.

At the end of this evaluation, we will compare:

- The rankings of mutation operators obtained for TSE and TSR.
- The results reported by operator-based and rank-based selective mutation with other selective approaches.
- The results between the selective strategies.

7.2 Selective Mutation for Test Suite Evaluation

In this section, we assess the value of each mutation operator for TSE. To that end, we first present the evaluation metric followed by the ranking of mutation operators and the experiments performed applying the selective strategies.

7.2.1 Evaluation metric

Not all the mutation operators offer the same effectiveness when assessing a test suite. The rationale behind selective mutation is that some mutants are redundant with regard to the whole set of mutants and they can be discarded. At the mutation operator level, an operator is *redundant* if it produces mutants that are always killed by the test cases that kill mutants from other operators. Therefore, an operator that only generates redundant mutants is said to be subsumed by the rest of mutation operators in the set [102].

We propose to measure the degree of redundancy of a mutation operator as the number of redundant mutants generated by the operator with respect to the mutants generated by the rest of operators. Roughly speaking, if a mutation operator o is eliminated and the same number of test cases are needed so that the test suite is adequate for the rest of mutants, then the mutants from o are redundant regarding the rest of mutation operators. Otherwise, some of the mutants derived from o are not redundant and they can help detect test deficiencies.

Formally, we define the metric *operator redundancy* to measure the degree of redundancy of a mutation operator o as follows:

$$R_o(T_{MO}) = \begin{cases} \frac{|D(T_{MO\setminus o})|}{|D(T_{MO})|} \times 100, & D_o \neq \emptyset \\ 100, & D_o = \emptyset \end{cases} \tag{7.1}$$

Where:

- D_o is the set of dead mutants from operator o.
- MO is the set of mutation operators.
- T_{MO} is an adequate test suite for the set of mutants in MO .
- $D(T_{MO})$ is the set of dead mutants with T_{MO}.
- $D(T_{MO\setminus o})$ is the set of dead mutants when using an adequate and minimal test suite derived from T_{MO} without considering the mutants from operator o.

Equation 7.1 measures the operator redundancy (R_o), which actually computes the percentage of dead mutants when using an adequate test suite for all the mutants except for the mutants from the operator under evaluation. The lower the value of R_o, the fewer the number of redundant mutants and therefore the more valued is that mutation operator.

The value of R_o can range from 100 to 0:

- $R_o = 100$:
  - $|D(T_{MO})| = \left|D(T_{MO\setminus o})\right|$: all the mutants from the mutation operator o are redundant, that is, the same test cases that kill the mutants generated by o are still needed to kill the mutants from other operators.
  - $D_o = \emptyset$: all the mutants are equivalent, as stated in Equation 7.1.
- $R_o = 0$: the analysed mutation operator is the only operator in the set generating non-equivalent mutants (i.e., $T_{MO\setminus o} = \emptyset$).

7.2.2 Example

To illustrate the evaluation metric in the previous section, consider the execution matrix in Figure 7.2. The set $\{test_1, test_2, test_3\}$ is an adequate and minimal test suite for the set of operators $\{o_1, o_2, o_3\}$ (T_{MO}) because all of those test cases are essential to kill the mutants from those operators. Then, we can compute the following adequate and minimal test suites when removing each of the operators in turns:

- $T_{MO\setminus o_1} = \{test_3\}$
- $T_{MO\setminus o_2} = \{test_1, test_2, test_3\}$

	test_1	test_2	test_3
$\text{op}_1 - \text{m}_1$	0	1	0
$\text{op}_1 - \text{m}_2$	1	0	0
$\text{op}_2 - \text{m}_3$	1	1	1
$\text{op}_2 - \text{m}_4$	0	1	1
$\text{op}_3 - \text{m}_5$	0	0	1
$\text{op}_3 - \text{m}_6$	1	0	1

FIGURE 7.2: Execution matrix to illustrate the metric R_o

- $T_{MO \setminus o_3} = \{test_1, test_2\}$

The subset $\{test_1, test_2\}$ is an adequate and minimal test suite for $MO \setminus o_3$ as this subset kills all the mutants without considering o_3 (m_1-m_4). Once those adequate and minimal test suites are known, we can calculate the set of dead mutants associated with those test suites:

- $D(T_{MO \setminus o_1}) = \{m_3, m_4, m_5, m_6\}$
- $D(T_{MO \setminus o_2}) = \{m_1, m_2, m_3, m_4, m_5, m_6\}$
- $D(T_{MO \setminus o_3}) = \{m_1, m_2, m_3, m_4, m_6\}$

Finally, knowing that $D_o \neq \emptyset$ in all cases, the value of the operator redundancy metric for these three operators can be calculated as follows:

- $R_{o_1} = (4/6) \times 100 = 66.6$
- $R_{o_2} = (6/6) \times 100 = 100$
- $R_{o_3} = (5/6) \times 100 = 83.3$

Interpreting these results, the operator o_1 presents the lowest redundancy: only 66.6% of the mutants (4 out of 6) would be killed with an adequate test suite for the subset of operators $\{o_2, o_3\}$, while the mutants from o_2 are redundant with regard to the mutants created by o_1 and o_3 ($R_{o_2} = 100$). The mutant 5 from o_3 would be alive after using the subset $\{test_1, test_2\}$ ($R_{o_3} = 83.3$). As a result, while the mutants from o_2 are subsumed by o_1 and o_3, the mutants from o_1 are not killed with an adequate test suite obtained from o_2 and o_3.

As a conclusion, a mutation operator with a low degree of redundancy increases the probability of losing effectiveness if mutants from that operator are discarded when following a selective mutation strategy. Therefore, the operators with the lowest R_o should be at the top of our ranking.

7.2.3 Ranking mutation operators

This section explains the general procedure followed in this study to rank mutation operators, and also shows the resulting operator classification.

7.2.3.1 Experimental procedure

We first measured the operator redundancy metric described in Section 7.2.1. R_o was calculated for each of the class mutation operators generating at least one dead mutant in the subjects under study in this experiment (see Section A.2 in Appendix A). Then, we followed this process for each of those mutation operators:

1. The mutants from the mutation operator o were removed from the execution matrix ($MO \setminus o$).
2. An adequate and minimal test suite was computed for the remaining operators ($T_{MO \setminus o}$).
3. The mutants from the mutation operator o were included again in the execution matrix.
4. The columns of the test cases which were not in the computed adequate and minimal test suite were removed from the execution matrix.
5. The operator redundancy of o was calculated with respect to a minimal test suite derived from T_{MO} ($R_o(T_{MO})$).

This procedure was carried out for each of the case studies and then a mean was calculated considering the different values of R_o for each operator. Finally, a ranking was prepared taking into account the average value of each mutation operator in descending order of R_o.

TABLE 7.1: Ranking of mutation operators based on mutant redundancy

Operator	TCL	RPC	DPH	TXM	KMY	DOM	Mean	SD
MCO	100	83.90	91.47	100	64.76	97.93	89.67	13.70
PCI		100		94.83	97.92	80.22	93.24	8.94
OMD	89.56	98.85	100	100	97.92	99.76	97.68	4.06
CID	100	97.70	96.12	98.06	96.89	100	98.13	1.60
IOD		100	96.12	98.70	99.48	98.62	98.58	1.49
OAN					98.96		98.96	-
MCI				99.03			99.03	-
IPC		100	99.22		97.92	100	99.28	0.98
OMR	98.26	100	100		98.96	100	99.44	0.80
CDC		98.85		100	100		99.62	0.66
EHC		100			99.48		99.74	0.37
CDD	*100	98.85	*100	100	100	*100	*99.81	0.47
IHI		100		100	99.48	100	99.87	0.26
IOR		*100	100	99.67		*100	*99.92	0.17
IHD					100		100.00	-
ISD		100					100.00	-
PNC						100	100.00	-
CTD						100	100.00	-
CTI			100			100	100.00	0.00
ISI		100	100		*100	100	*100.00	0.00
IOP				100		*100	*100.00	0.00
PMD				*100	*100	*100	*100.00	0.00
PPD		*100		100	100	100	*100.00	0.00
CCA	100	100	*100	*100	*100	100	*100.00	0.00

7.2.3.2 Ranking

The results of the operator redundancy metric of each operator and case study appear in Table 7.1. As aforementioned, an average is calculated per operator in order to form the operator ranking. As it can be seen, the top ranked operator is *MCO* whereas a group of 10 operators (from *IHD* to *CCA*) present the worst value ($R_o = 100$). These operators at the bottom of the table do not impact the TSE process when excluding one of them from the set of operators, that is, they are not useful to detect test deficiencies that would not be revealed by other mutants from other operators. The figures marked with '*' represent operators only producing equivalent mutants in that case study. The standard deviation (*SD*) has also been included in this table to observe the stability of the metric in each operator among case studies.

As illustrated, although ten operators have $R_o = 100$, the rest of operators show a redundancy degree between 89.67 and 99.92 on average, where 18 out of 24 mutation operators present a value over 99. These high values are explained by the fact that a test case usually reveals the mutations injected by different operators, so removing an operator does not always lead to a reduction in the number of test cases. Given these

values, the fact that the operators at the top of the classification present the highest standard deviation is not surprising.

The top 4 ranked operators are the ones spawning the highest number of mutants (see Table A.5), which suggests a correlation between the number of mutants and the metric R_o. This is not unexpected: it seems unlikely that removing a large subset of mutants does not lead to a decrease in the number of necessary test cases. However, this correlation does not hold for all the operators in the ranking: *IHI* is the fifth most prolific operator and it is placed in the 13th position. Thus, in order to know how the number of mutants influences R_o, we run the Spearman's correlation test (see Table 7.2). The results in each of the programs range from -0.56 in *XmlRpc* to -0.73 in *KMyMoney* (95% confidence level except for *Matrix TCL Pro*). Effectively, these results suggest that there is an inverse correlation between the number of mutants generated by these mutation operators and the value that the redundancy metric attaches them. Nevertheless, the correlation is not very strong, which means that the operator redundancy does not depend only on the number of mutants generated by each operator.

TABLE 7.2: Spearman's correlation test (*rho* and *p-value*) between the number of mutants generated by the operators and the value that the redundancy metric assigns them for each of the programs under test

Program	**rho**	**p-value**
TCL	-0.68	0.07018
RPC	-0.56	0.01199
DPH	-0.66	0.0139
TXM	-0.58	0.01449
KMY	-0.73	0.000493
DOM	-0.66	0.001424

The top 5 operators are from different operator groups (see Table 3.1). "Exception handling" is the only block not represented in that top 5. This fact leads us to think that the operators at the top of the ranking partially subsume the rest of operators in the same group. It also suggests that each operator block addresses different features of the language, which makes operators from different groups less likely to be redundant among them.

7.2.4 Selective mutation based on the ranking

In this section, we use the ranking of mutation operators for TSE to perform selective mutation. The goal is to observe the loss in the mutation adequacy score when some of the mutants are not taken into account for the evaluation of the test suite.

7.2.4.1 Experimental procedure

The experimental setup comprises two phases:

First phase

We gathered the operators with a similar average value of operator redundancy, grouping them into five different categories. We set the following ranges with a view to balancing the number of operators in each category (see Table 7.1). The operators in each of the categories are shown in Table 7.3.

TABLE 7.3: Categories and operators for TSE

Category	Condition	Operators
1	$98 > R_o$	MCO-PCI-OMD
2	$99 > R_o \geq 98$	CID-IOD-OAN
3	$99.5 > R_o \geq 99$	MCI-IPC-OMR
4	$100 > R_o \geq 99.5$	CDC-EHC-CDD-IHI-IOR
5	$R_o = 100$	IHD-ISD-PNC-CTD-CTI-ISI-IOP-PMD-PPD-CCA

Second phase

Once defined the categories in the first phase, we applied both selective mutation strategies: operator-based and rank-based selective mutation. These two strategies follow a different process, which will be separately explained below:

Operator-based selection We performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category[1]):

1. The operators encompassed within categories $[1...i]$ ($MO_{[1...i]}$) were selected from the execution matrix.
2. An adequate and minimal test suite was computed for the selected operators ($T_{MO_{[1...i]}}$).
3. The mutants generated by the operators that were not in $MO_{[1...i]}$ were included again in the execution matrix.
4. The mutation score associated with the test suite $T_{MO_{[1...i]}}$ and the reduction in the number of mutants were calculated.

[1]The operators classified in the category 5 are removed in the first loop as we select the categories 1–4.

Rank-based mutant selection In rank-based mutant selection, we select a subset of mutants from all operators but with different probability depending on the position of each operator in the ranking. In this case, we do not decide a particular size for that subset; instead, we select the same number of mutants as in operator-based selection in order to compare both selective strategies later on in Section 7.4.3.

We performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category):

1. As many mutants from D (set of dead mutants) as dead mutants were contained in the operators encompassed within categories $[1...i]$ ($\left|D_{MO_{[1...i]}}\right|$) were selected using rank-based mutant selection with all operators. We call M_R the set of selected mutants using rank-based selection from now on.
2. An adequate and minimal test suite was computed for the selected mutants (T_{M_R}).
3. The mutants that were not in M_R were included again in the execution matrix.
4. The mutation score associated with the test suite T_{M_R} and the reduction in the number of mutants was calculated.

We applied the above process 30 times with different seeds and computed the average.

7.2.4.2 Selective mutation results

Operator-based selection

Table 7.4 shows the mutation adequacy score when performing the experimental procedure explained in the second phase in Section 7.2.4.1 to apply operator-based selection following the five categories presented in the first phase. The mean (*Mean*) and the standard deviation (*SD*) of the results in all the case studies are computed in the two last columns.

Each value of this table is the result of excluding the operators classified into the categories under that row. As an example, only the operators *MCO*, *PCI* and *OMD* were applied to compute the mutation scores shown in the first row (category 1). We achieved a mutation score over 90% in 4 out of 6 case studies only applying these three operators. The second row presents the results of selecting the operators within category 1 (*MCO*, *PCI*, *OMD*) and 2 (*CID*, *IOD* and *OAN*), where the mutation score was greater than 90% for all the case studies.

TABLE 7.4: Operator-based selective mutation results (mutation score) based on the ranking of mutant redundancy

Category	TCL	RPC	DPH	TXM	KMY	DOM	Mean	SD
1	93.0	92.0	89.1	94.8	79.3	94.9	90.52	5.89
2	98.3	97.7	99.2	98.7	91.7	97.7	97.22	2.76
3	100	97.7	100	99.7	99.0	100	99.40	0.92
4	100	100	100	100	100	100	100	0

TABLE 7.5: Rank-based selection results based on the ranking of mutant redundancy

Program	1		2		3		4	
	M	SD	M	SD	M	SD	M	SD
TCL	92.3	2.88	98.4	1.80	100	0.00	100	0.00
RPC	95.1	3.61	98.8	1.86	99.8	0.42	99.9	0.58
DPH	93.1	2.57	98.2	1.39	99.1	0.97	99.3	0.75
TXM	99.8	0.49	100	0	100	0	100	0
KMY	95.0	1.91	97.9	1.45	99.6	0.49	100	0.13
DOM	100	0	100	0	100	0	100	0
Total	95.89	3.30	98.87	0.92	99.77	0.35	99.87	0.27

TABLE 7.6: Reduction in the number of mutants by categories when applying operator-based selective mutation based on the ranking of mutant redundancy

Category	TCL	RPC	DPH	TXM	KMY	DOM	Mean	SD
1	63.7	52.0	69.6	46.6	60.4	23.5	52.63	16.47
2	34.1	36.2	30.8	31.9	40.4	16.8	31.70	8.06
3	8.9	27.6	25.3	22.2	22.5	13.3	20.00	7.28
4	7.4	5.5	9.1	5.5	8.9	5.7	7.02	1.70

Rank-based selection

Table 7.5 contains the results of the rank-based mutant selection strategy, described in Section 7.2.4.1. The mean mutation score (M) in each of the categories and the standard deviation (SD) of the 30 executions are shown. As an example of the meaning of the figures in this table, the average in *KMyMoney* in category 3 (99.6%) is the mutation score when selecting the same number of dead mutants as dead mutants are produced by the subset of operators classified into the categories 1 (*MCO*, *PCI*, *OMD*), 2 (*CID*, *IOD* and *OAN*) and 3 (*MCI*, *IPC* and *OMR*). By selecting the same size of mutants as in the categories, the effectiveness of this strategy is comparable to the effectiveness of the operator-based strategy (99.0%) for the same case study and category (see Table 7.4). As remarkable results, we can observe that in category 2 we achieve the full mutation score in two programs and a total average score of 98.87%. The mutation score declines by 3% (95.89%) in category 1, but it is over 92% for all case studies.

Savings

Table 7.6 shows the reduction in the percentage of generated mutants because of the operators removed in each step (we also calculated the mean and the standard deviation (*SD*)). Applying the three operators in category 1, we achieve a reduction of more than half of the mutants (52.63%) with a standard deviation of 16.47. In this case, the mutation score is:

- 90.52% of the original mutation score with operator-based selection.
- 95.89% of the original mutation score with rank-based selection.

Analogously, using the six operators from the categories 1 and 2 offers a reduction in the number of mutants of 31.7% (standard deviation: 8.06), and the mutation score is:

- 97.22% with operator-based selection.
- 98.87% with rank-based selection.

The mutation score gradually decreases when removing each of the categories, except for the operators in category 5 when applying operator-based selection (the mutation score is still 100%, as it can be seen in Table 7.4 in the row of the fourth category). In this latter case, there is no loss of mutation score accuracy while lowering the number of mutants to 92.98%.

7.3 Selective Mutation for Test Suite Refinement

This section presents an evaluation of mutation operators for the refinement of the test suite analogous to the assessment carried out for TSE. In this regard, we show the evaluation metric used to form the operator ranking and the experiments conducted based on that classification.

7.3.1 Evaluation metric

After the test suite execution, those non-equivalent mutants remaining alive require additional test cases to be killed. However, a single test case could suffice to kill all of those surviving mutants if they represent faults that are not hard to detect. This usually happens when those mutations are in a part of the code not covered by the current test suite.

Thus, the mutants offering resistance to be killed should be the most valued when determining a classification of operators for TSR. Therefore, assigning a greater value to resistant and resistant hard to kill mutants (see Appendix B) over other kinds of mutants is a proper approach if we want to give preference to the generation of high-quality test cases.

This is the approach embodied in the quality metric devised by Estero-Botaro et al. [45], which was commented in Section 2.4.4. The formula of the quality metric of a mutant Q_m is presented in Equation 7.2:

$$Q_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|\,, & m \in D \end{cases} \tag{7.2}$$

Where:

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

This quality metric punishes the existence of equivalent mutants ($m \in E$) as well as takes into account a twofold aspect regarding dead mutants ($m \in D$):

- The number of test cases that kill a mutant.
- The number of mutants that those test cases kill at the same time.

Therefore, this metric considers that a mutant will assist a tester in designing high-quality test cases not only when there are few test cases killing it, but also when there are few mutants killed by those test cases. This is a desirable property because the fewer the mutants that are able to guide the tester on the creation of a test case, the more specific and hard to design is that test case. Consequently, this metric seeks that the mutants killed by few test cases that in turn kill few other mutants are included in the

subset of selected mutants: this will increase the probability that the more specific test cases are designed through the inspection of those mutants.

Every mutant receives a value in the range $[0, 1]$, which depends on the number of test cases and mutants killed by those test cases: the lower the number of test cases killing the mutant, the higher that value. In the same line, the lower the number of mutants killed by those test cases, the higher that value. As such, resistant and resistant hard to kill mutants are the kind of mutants with the highest value according to this metric.

On this basis, being M_o the set of mutants generated by the operator o, the quality of a mutation operator can be defined as:

$$Q_o = \frac{1}{|M_o|} \sum_{m \in M_o} Q_m \tag{7.3}$$

The metric Q_o can be used as a means to rate operators by their potential to help the tester to enhance the fault detection power of the test suite. The operators with the highest quality metric should be the most valued. Notice that this quality metric can be computed even when the operator only generates equivalent mutants (in that case, $Q_o = 0$).

7.3.2 Example

To illustrate the quality metric, we include an example based on the execution matrix in Figure 7.3. Let T_o be an adequate and minimal test suite for the mutation operator o. We can compute the following adequate and minimal test suites for each mutation operator:

- $T_{o_1} = \{test_1, test_2\}$
- $T_{o_2} = \{test_3\}$

T_{o_1} is formed by $\{test_1, test_2\}$ because the test cases $test_1$ and $test_2$ can be used to kill the three mutants from operator o_1 (m_1, m_2, m_3). In the same line, $\{test_3\}$ is sufficient to kill the mutants from o_3 (m_4, m_5, m_6). Then, the value of the quality metric for these two operators is:

1. Quality of operator o_1 ($|M_{o_1}| = 3$, $|E_{o_1}| = 0$, $|T_{o_1}| = 2$, $C_{test_1} = \{m_2, m_3\}$, $C_{test_2} = \{m_1\}$):
   - $Q_{m_1} = 1 - 1/((3-0) \cdot 2) = 0.83$ where $K_{m_1} = \{test_2\}$

	test_1	test_2	test_3
$\text{op}_1 - \text{m}_1$	0	1	0
$\text{op}_1 - \text{m}_2$	1	0	0
$\text{op}_1 - \text{m}_3$	1	0	1
$\text{op}_2 - \text{m}_4$	0	0	0
$\text{op}_2 - \text{m}_5$	0	0	1
$\text{op}_2 - \text{m}_6$	1	0	1

FIGURE 7.3: Execution matrix to illustrate the *quality metric*

- $Q_{m_2} = 1 - 2/((3-0)\cdot 2) = 0.67$ where $K_{m_2} = \{test_1\}$
- $Q_{m_3} = 1 - 2/((3-0)\cdot 2) = 0.67$ where $K_{m_3} = \{test_1\}$

$Q_{o_1} = (0.83 + 0.67 + 0.67)/3 = 0.72$

2. Quality of operator o_2 ($|M_{o_2}| = 3$, $|E_{o_2}| = 1$, $|T_{o_2}| = 1$, $C_{test_3} = \{m_5, m_6\}$):

   - $Q_{m_4} = 0$ (equivalent)
   - $Q_{m_5} = 1 - 2/((3-1)\cdot 1) = 0$ where $K_{m_5} = \{test_3\}$
   - $Q_{m_6} = 1 - 2/((3-1)\cdot 1) = 0$ where $K_{m_6} = \{test_3\}$

   $Q_{o_2} = (0 + 0 + 0)/3 = 0$

Interpreting these results, $test_1$ and $test_2$ may not be generated without considering o_1; $test_2$ would be generated only after analysing the first mutant generated by o_1 (m_1). On the contrary, o_2 could induce the creation of $test_3$, which would be generated inspecting either m_5 or m_6 from this operator; both mutants are killed by a single test case ($test_3$), which results in $Q_m = 0$ when just a test case suffices to kill all the non-equivalent mutants generated by an operator. In addition, o_2 produces an equivalent mutant (m_4), which is penalised by this metric. Consequently, o_1 is more valued than o_2 according to this quality metric, which is reflected in the values for these operators: $Q_{o_1} = 0.72 > Q_{o_2} = 0$.

As a conclusion, a mutation operator with a high value of Q_o increases the probability of missing some test cases when performing a selective mutation strategy without mutants generated by that operator. Therefore, the operators with the highest Q_o should be at the top of the ranking.

7.3.3 Ranking mutation operators

This section explains the application of the quality metric described in Section 7.3.1 to each mutation operator. This is useful to rank mutation operators and perform a selective study for TSR.

7.3.3.1 Experimental procedure

We computed the quality metric for each class mutation operator generating some mutants in the subject programs in this chapter (see Section A.2 in Appendix A). We should note that the authors of the original quality metric [45] established a threshold of four mutants as the minimum number of mutants that a mutation operator should generate so that the value of the metric was significant. We have seen that class operators generate fewer mutants than traditional operators (see Section 5.2.2), but we have maintained this condition in our study for consistency with the experiments performed in that paper.

We calculated the quality metric of the operators for each case study and computed a mean with the values obtained for each operator. Finally, a ranking was prepared in ascending order of Q_o.

7.3.3.2 Ranking

Table 7.7 shows the results of applying the quality metric to each operator and case study. The operators are sorted according to the mean (the standard deviation is also calculated). *IOD*, with $Q_o = 0.82$, is the most valued operator on average, while *IOP*, *PMD* and *EHC* are given the lowest value and they are placed at the bottom of the classification. Note that mutants from operators with $Q_o = 0$ in a case study are either equivalent or all the mutants are killed by the same test case in the adequate and minimal test suite for the operator, as shown in the example in Section 7.3.2.

Because of the imposed threshold of four mutants (commented in the previous section), some mutation operators could not be rated in each of the programs; these cases have been highlighted with the symbol '-'. In the same line, we should remark that five of these operators (*IHD*, *ISD*, *PNC*, *CTD* and *CTI*) did not generate more than three valid mutants in any of the case studies (see Table A.5); these operators are not shown in Table 7.7. The mutation operator *OMR*, in the third position, was the only operator with values over 0.9 in some of the subject programs. The quality metric in the rest of operators with $Q_o > 0$ range from 0.07 to 0.71. Unlike the operator classification for

TABLE 7.7: Ranking of mutation operators based on test quality

Operator	**TCL**	**RPC**	**DPH**	**TXM**	**KMY**	**DOM**	**Mean**	**SD**
IOD		-	0.80	0.78	-	0.89	0.82	0.06
MCO	-	0.74	0.79	0.72	0.89	0.73	0.77	0.07
OMR	0.88	0.92	0		0.98	0.89	0.73	0.41
OMD	0.80	0.82	-	0.52	0.68	0.71	0.71	0.12
IPC		-	0.30		0.65	0.79	0.58	0.25
CID	0.83	0.74	0.52	0.58	0.52	0.29	0.58	0.19
ISI		-	0.57		-	-	0.57	-
CDC		-		-	0.47		0.47	-
PCI		-		0.71	0	0.60	0.44	0.38
OAN					0.38		0.38	-
IHI		0		0.72	0.29	0.39	0.35	0.30
MCI				0.30			0.30	-
IOR		0	0.07	0.65		-	0.24	0.36
PPD		-		0	0.17	0.11	0.14	0.09
CCA	0.17	-	0	0	-	0.25	0.10	0.13
CDD	-	0.30	-	0	0	0	0.07	0.15
IOP				0		-	0	-
PMD				-	-	0	0	-
EHC		-			0		0	-

TSE, operators present varying standard deviations across the ranking because the differences of the quality metric among case studies are more pronounced than the differences when computing the operator redundancy. Also, unlike the ranking based on mutant redundancy, none of the operators from the "polymorphism and dynamic binding" group is in the top 5 of the ranking based on test quality, finding the first one (PCI) in the 9th position.

TABLE 7.8: Spearman's correlation test (*rho* and *p-value*) between the number of mutants generated by the operators and the value that the quality metric assigns them for each of the programs under test

Program	**rho**	**p-value**
TCL	0.20	0.4583
RPC	0.26	0.2891
DPH	0.57	0.06911
TXM	0.63	0.01387
KMY	0.59	0.02193
DOM	0.48	0.05652

We replicated the Spearman's correlation test in Section 7.2.3.2, but correlating the number of mutants and the value of the quality metric for each operator. The results divided by case study are shown in Table 7.8. In this case, we can observe a direct correlation between both factors, but considerably less strong than in the test conducted with regard to the operator redundancy metric.

7.3.4 Test-quality selective mutation based on the ranking

In this last section of the assessment for TSR, we perform test-quality selective mutation following the operator classification derived previously using the quality metric. The goal is to observe the loss in the number of test cases in an adequate and minimal test suite for the full set of mutants when applying both operator-based and rank-based mutant selection for TSR (see Section 7.1.3).

7.3.4.1 Experimental procedure

The experimental setup comprises two phases:

First phase

We gathered the operators with a similar quality metric into five categories, but also trying to balance the number of operators in each category (see Table 7.7). Table 7.10 classifies mutation operators according to the five categories. The five mutation operators that could not be evaluated because of the threshold of four mutants are included in category 5, as they are supposed not to have a significant impact on the results.

TABLE 7.9: Categories and operators for TSR

Category	Condition	Operators
1	$0.70 < Q_o$	IOD-MCO-OMR-OMD
2	$0.50 < Q_o \leq 0.70$	IPC-CID-ISI
3	$0.25 < Q_o \leq 0.50$	CDC-PCI-OAN-IHI-MCI
4	$0.00 < Q_o \leq 0.25$	IOR-PPD-CCA-CDD
5	$Q_o = 0.00$	IOP-PMD-EHC-IHD-ISD-PNC-CTD-CTI

Second phase

Following the same process as in Section 7.2.4.1, in this second phase we apply both selective mutation strategies:

Operator-based selection Once defined the categories in the first phase, we performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category):

1. The operators encompassed within categories $[1...i]$ ($MO_{[1...i]}$) were selected from the execution matrix.

2. An adequate and minimal test suite was computed for the selected operators ($T_{MO_{[1...i]}}$).

3. The loss of test cases with respect to the original adequate and minimal test suite, $|T_{MO}| - \left|T_{MO_{[1...i]}}\right|$, and the reduction in the number of mutants were calculated.

Rank-based mutant selection As in Section 7.2.4.1, we executed 30 times the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category) and computed the average:

1. As many mutants from D (set of dead mutants) as dead mutants are contained in the operators encompassed within categories [1...i] ($\left|D_{MO_{[1...i]}}\right|$) were selected using rank-based mutant selection with all operators. Recall, M_R represents the set of selected mutants.

2. An adequate and minimal test suite was computed for the selected mutants (T_{M_R}).

3. The loss of test cases with respect to the original adequate and minimal test suite was calculated: $|T_{MO}| - |T_{M_R}|$.

7.3.4.2 Test-quality selective mutation results

Operator-based selection

Table 7.10 shows the percentage of loss in the number of test cases from the original adequate and minimal test suite as a consequence of removing the operators under that category (just as in Table 7.4). Again, we obtained the mean as well as the standard deviation (SD) of the results of each case study.

We should note that the number of test cases in the minimal test suite is not very high in most case studies (from 15 to 36 test cases, as it can be seen in Table A.4), so the reduction of a single test case implies a significant percentage.

When applying operator-based selective mutation, the number of test cases in the adequate and minimal test suite decreases from 0.47% when discarding the operators belonging to category 5 to 23.68% only using the operators within category 1. Considering the 16 operators from the first four categories (with $Q_o > 0$), the same number of test cases are retained except for a loss of one test case in the adequate and minimal test suite for *KMyMoney* (2.8%).

TABLE 7.10: Percentage of test cases loss when performing operator-based selective mutation based on the ranking of test quality

Category	TCL	RPC	DPH	TXM	KMY	DOM	Mean	SD
1	0	20.0	18.2	53.3	30.6	20.0	23.68	17.57
2	0	13.3	0	33.3	13.9	20.0	13.42	12,65
3	0	6.7	0	6.7	2.8	0	2.70	3.28
4	0	0	0	0	2.8	0	0.47	1.10

TABLE 7.11: Rank-based selection results based on the ranking of test quality

Program	1		2		3		4	
	M	SD	M	SD	M	SD	M	SD
TCL	9.6	5.7	0	0	0	0	0	0
RPC	8.9	7.1	1.6	2.9	0.2	1.2	0	0
DPH	10.9	6.4	1.5	3.0	1.5	3.0	0.5	1.4
TXM	28.0	8.1	19.1	5.7	0	0	0	0
KMY	6.0	2.4	0.6	1.2	0	0	0	0
DOM	15.5	4.4	11.5	5.2	0	0	0	0
Total	13.14	7.91	5.72	7.82	0.29	0.61	0.08	0.19

Rank-based selection

Table 7.11 shows the results of the rank-based mutant selection with the same format as Table 7.5 (with the results of the rank-based selection using the operator classification based on operator redundancy). By using the same size of mutants as in the operators within categories 1 and 2, we assume a mean loss of 5.72% test cases with a standard deviation of 7.82. This percentage increases to 13.14% when we just consider the first category. Overall, we can also observe that the standard deviation successively increases from category 4 to 1: the fewer the mutants selected, the more varied are the results in the different executions.

Savings

Table 7.12 depicts the percentage of reduction in the number of mutants when we put into practice both selective strategies.

The reduction in the number of mutants is not very relevant when removing the mutants regarding the categories 4 and 5, but meaningful when considering the first two categories (39.42%). In that case, we assume a loss of:

- 13.42% test cases when applying operator-based selection.
- 5.72% test cases when applying rank-based selection.

We should note an increasing standard deviation because of disparities in the results among case studies. For instance, in the case of *Matrix TCL Pro* and *Tinyxml2* in the first category: while we reduce 38.5% mutants in the former, we save twice as much in the latter (79.8%). However, this gap is also reflected in the test suite size, assuming a loss of:

- 0% and 53.3% test cases respectively for *Matrix TCL Pro* and *Tinyxml2* when applying operator-based selection.
- 9.6% and 28% test cases respectively for *Matrix TCL Pro* and *Tinyxml2* when applying rank-based selection.

TABLE 7.12: Reduction in the number of mutants by categories when applying operator-based selective mutation based on the ranking of test quality

Category	TCL	RPC	DPH	TXM	KMY	DOM	Mean	SD
1	38.5	44.1	48.5	79.8	53.9	83.5	58.05	19.01
2	8.9	27.6	19.7	71.3	29.3	79.7	39.42	28.99
3	8.9	20.5	19.7	9.7	11.4	6.1	13.20	5.96
4	0	2.4	1	2.7	2.9	1.5	1.75	1.13

7.4 Comparison Between Evaluations

7.4.1 Comparison between rankings

In this section, we want to know whether it makes sense to distinguish between the usefulness of mutation operators for TSE and TSR by comparing the rankings arranged in Section 7.2.3 and 7.3.3.

At first sight, we can observe an appreciable similarity between these two rankings. For instance, both share some commonalities:

- *MCO*, *OMD* and *IOD* are fruitful class mutation operators because these operators occupy the first positions in both classifications
- *PMD* and *IOP* are not so useful because they are at the bottom of these two rankings.

This fact suggests that the most suitable mutation operators for TSE and TSR match. However, looking at the positions of each operator more carefully, we can notice some dissimilarities:

- *PCI* falls from 2nd in the ranking for TSE (see Table 7.1) to 9th in the ranking for TSR (see Table 7.7). Therefore, while *PCI* shows a low operator redundancy, test cases are quite effective with the mutants from this operator, so *PCI* is not such an useful operator to induce new test cases.
- *OMR* climbs six positions (from 9th in the ranking related to operator redundancy to 3rd in the ranking related to test quality) and *ISI* eight positions (from 15th to 7th).
- *EHC*, *CDD*, *IOD*, *OAN* and *MCI* also exhibit discordant positions in both rankings.

In conclusion, these disparities between rankings validate our double perspective when addressing the value of each mutation operator.

7.4.2 Validation of results

The results shown so far in this chapter do not allow to rule out the possibility that other operator rankings could report better results when performing a selective strategy based on them. Hence, this section serves as a sanity check for both selective strategies.

7.4.2.1 Operator-based selective mutation

In order to check if the same results hold with different operator classifications, we compared our operator-based selection results with other rankings of mutation operators. We carried out selective mutation determining categories derived from new rankings. To this end, we followed three classical approaches to operator-based selective mutation:

- **Random:**
  - *Ranking:* random sort of mutation operators.
  - *Category size:* For the selective strategy, we maintained the same sizes of the categories from the original experimental results (see Section 7.2.4.1 and Section 7.3.4.1). This allows for a direct comparison between this and the original ranking because the categories contain the same number of operators.
- **Number of mutants (Size):**
  - *Ranking:* sort of mutation operators by the number of mutants [101, 126], where the most prolific operators are at the bottom of the ranking.

  - *Category size:* few operators should be removed in each step if we want to maintain a significant number of mutants at all times. Thus, unlike in the previous ranking *Random*, we do not maintain the same sizes of the categories from the original experiments: in those experiments, many operators were removed at the beginning (10 and 8 operators in the evaluation for TSE and TSR respectively) and few operators belonged to the category 1 (3 for TSE; 4 for TSR).

    Therefore, the category size is proportional to the number of mutants generated in the analysed programs (see Table A.5). Thus, we divided the total number of mutants (1,868) by 5 categories, which results in 374 mutants per category. Then, we included as many operators as needed to complete 374 mutants, which depends on the mutants produced by each operator. As an example, *PCI* (the most prolific operator) is the only operator classified into the category 5 as it generates 659 mutants, which suffices to reach the number of mutants set for a category.

- **Operator type (Block):**

  - *Ranking:* sort of mutation operators related to the operator block (see Table 3.1). Class operators with similar characteristics were previously grouped in different blocks. The idea is to apply selective mutation removing operator groups [102]. We sort the blocks depending on their number of operators, where the block with more operators ("inheritance") is at the top of the ranking.
  - *Category size:* the category size is related to the number of operators within each group. In this regard, we only counted operators creating at least one mutant in our case studies. For instance, 3 out of 4 operators from the "method overloading" block were applied (*OMD*, *OMR* and *OAN*). Because of the few operators, the groups "exception handling" and "object and member replacement" are placed together in the last category.

The final arrangement of these three rankings and the division of the operators into categories is depicted in Table 7.13. While the categories are the same for TSE and TSR in the rankings *Size* and *Block*, we have to consider different categories for TSE and TSR in the ranking *Random*; as aforementioned, the category size for this ranking is related to the number of operators within each category in the original experiments. For instance, in the first category there were 3 operators for TSE (*MCO*, *PCI* and *OMD*), but 4 operators for TSR (*IOD*, *MCO*, *OMR* and *OMD*).

For each of the three rankings:

TABLE 7.13: Arrangement of the rankings *Random*, *Size* and *Block* classified into categories for TSE and TSR

Category	**Random** $_{TSE}$	**Random** $_{TSR}$	**Size** $_{TSE,TSR}$	**Block** $_{TSE,TSR}$
1	IPC,OMR,ISD	IPC,OMR, ISD,ISI	CTD,ISD,IHD,PNC, CTI,OAN,EHC,PMD, CDC,IOP,ISI,CDD, IPC,CCA,MCI,PPD, IOR	IHD,IHI,ISD, ISI,IOD,IOP, IOR,IPC
2	ISI,CCA,OMD	CCA,OMD, CTI	IOD,IHI, OMR,OMD	CTD,CTI,CID, CDC,CDD,CCA
3	CTI,PNC,MCI	PNC,MCI,IOD, MCO,IOP	CID	PCI,PMD,PPD, PNC
4	IOD,MCO,IOP, CDC,PCI	CDC,PCI, CID,PMD	MCO	OMD,OMR, OAN
5	CID,PMD,IOR, IHD,IHI,CTD, CDD,OAN, EHC,PPD	IOR,IHD,IHI, CTD,CDD, OAN,EHC, PPD	PCI	MCO,MCI,EHC

TABLE 7.14: Comparison of the mutation score when using operator-based selective mutation testing for TSE with the rankings *Random*, *Size* and *Block*

Category	***Original***		***Random***		***Size***		***Block***	
	M	SD	M	SD	M	SD	M	SD
1	90.52	5.89	48.89	29.90	56.02	21.47	53.70	31.09
2	97.22	2.76	74.66	14.10	82.76	13.52	78.15	14.77
3	99.40	0.92	78.17	13.94	85.18	13.48	84.36	15.62
4	100	0	97.54	2.12	95.50	7.75	89.43	13.75

TABLE 7.15: Comparison of the percentage of test cases loss when using operator-based selective mutation testing for TSR with the rankings *Random*, *Size* and *Block*

Category	***Original***		***Random***		***Size***		***Block***	
	M	SD	M	SD	M	SD	M	SD
1	23.68	17.57	66.71	19.38	66.97	11.76	64.46	22.17
2	13.42	12.65	48.09	23.73	26.56	11.45	37.68	12.38
3	2.68	3.28	19.31	12.98	17.69	9.81	30.79	17.92
4	0.45	1.10	4.07	4.75	5.35	7.34	13.92	10.51

- Table 7.14 shows the mutation score achieved by the operators of each category following the selective mutation procedure described in the second phase in Section 7.2.4.1.

- Table 7.15 shows the percentage of loss in the number of test cases in the adequate and minimal test suite following the test-quality selective mutation procedure described in the second phase in Section 7.3.4.1.

For each of the new rankings, the column M represents the averaged results in our case studies and SD is the standard deviation. In order to facilitate the comparison, we also show in both tables the mean and the standard deviation obtained with our original rankings (see column *Mean* and *SD* in Table 7.4 and 7.10). We label these results as *Original*.

The results of the original rankings on average are clearly better than the results of the rankings *Random*, *Size* and *Block*. Studying the results in detail, we can observe that:

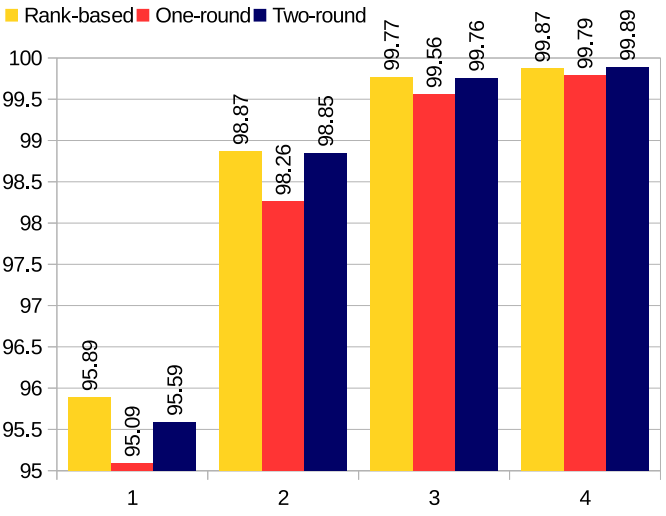
- *Random* shows the worst performance in general in terms of the mutation score and the loss in the percentage of test cases for all the categories except when removing the operators in category 5.
- *Size* gets better results than *Block* in both selective strategies in 7 out of 8 cases (four categories for TSE plus other four categories for TSR). As an exception, we have to note that the ranking *Block* is able to surpass the ranking based on test quality for *Tinyxml2* when selecting the operators from category 1 and the operators from categories 1 and 2.
- The rankings *Random* and *Size* match the outcome of our original results in a pair '(*case study*, *category*)' only in few cases, but the averaged results are still very far from the ones achieved with *Original* in both TSE and TSR.
- The high standard deviations in the three rankings suggest that they do not provide a stable performance.

7.4.2.2 Rank-based selective mutation

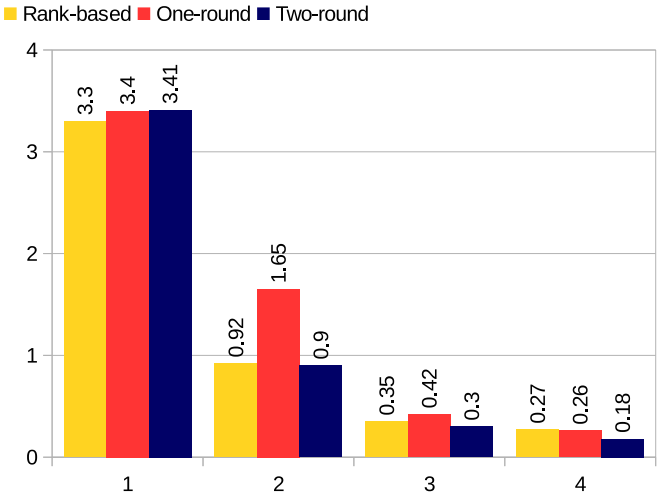
Similarly to the previous sanity check, we aim to compare our rank-based mutant selection results with other strategies for the selection of mutants. Namely, we run two random strategies proposed by Zhang et al. [135]:

- **One-round random (One-round):** random selection of mutants from all the operators. The probability of selecting each of the mutants is the same.
- **Two-round random (Two-round):** the probability of selecting a mutant from each of the operators is the same. It is performed in two different rounds:
  - *First round*: one operator is selected randomly.
  - *Second round*: one mutant is selected randomly from the operator selected in the first round.

In both strategies, we selected the same number of mutants in each category for TSE and TSR as in the rank-based mutant selection.



(a) Mean

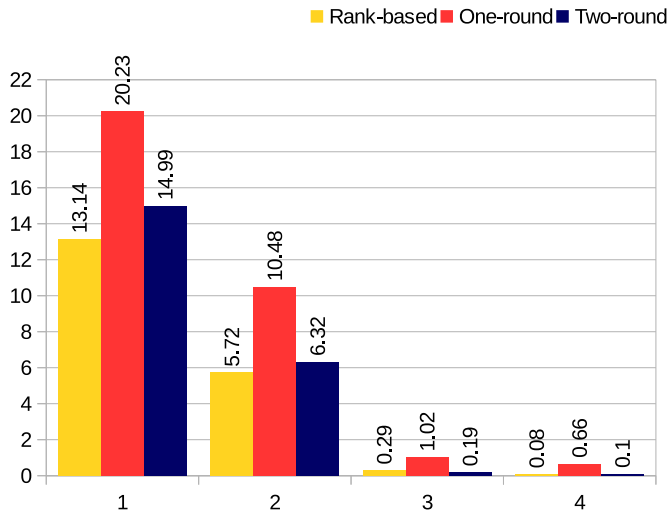


(b) Standard deviation

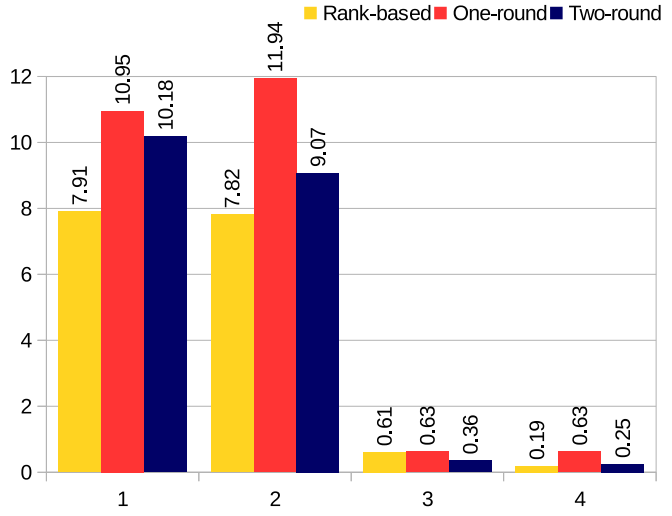
FIGURE 7.4: Comparison of the mutation score when using *Rank-based* selective mutation testing for TSE with the rankings *One-round* and *Two-round*

Figures 7.4 and 7.5 depict the comparative results (mean and standard deviation) of the two random strategies and the rank-based strategy for TSE and TSR respectively. Studying the results in detail:

- *One-round*: Rank-based mutant selection outperforms this random strategy in all cases. It also shows worse performance than *Two-round*.
  - TSE: The rank-based strategy is better than *One-round*. There is a remarkable difference of 2.6% in the pair '(*KMyMoney*, *category 2*)'. However, apart



(a) Mean



(b) Standard deviation

FIGURE 7.5: Comparison of the percentage of test cases loss when using *Rank-based* selective mutation testing for TSR with the rankings *One-round* and *Two-round*

from a few ties, the latter was able to obtain a higher score for the two first categories when analysing *Dolphin*.

– TSR: We can find a notable gap between both strategies in '(*QtDom*, *category 1*)'): 18.1% for rank-based selection and 30.6% for *One-round*.

- *Two-round*: Rank-based selection gets better results in 6 out of 8 cases on average and the gap between the two selective strategies widens as the size of the subset of mutants selected decreases. This outcome is quite interesting as that means that the operator rankings work better for large reductions of mutants (which is desirable in mutation testing). The standard deviation in *Two-round* is also higher than in the original strategy for both evaluations in category 1, which means that

the rank-based strategy is more stable in general than *Two-round* with a reduced subset of mutants.

- TSE: The rank-based strategy surpasses *Two-round* by 0.3% on average in the first category. As it was mentioned earlier, the margins are narrow due to the nature of the mutation score. If we study the results in each program individually, *Two-round* only produces better results overall in *Dolphin*.
- TSR: If we focus again in the first category, rank-based selection surpasses *Two-round* by 1.85% test cases lost. There are relevant differences in favour of rank-based mutant selection in several cases, like in the pair '(*Tinyxml2*, *category 1*)') where the difference is 6.6%.

7.4.3 Comparison between selective mutation strategies

To complete this section, it is interesting to study which one of the selective strategies, operator-based or rank-based selective mutation, provided a better result. We also made use of the results of the sanity checks in the previous section for this analysis.

We studied this aspect separately for TSE and TSR:

- **TSE:** Figure 7.6 shows graphically the average mutation score when applying the operator-based and rank-based strategy for each of the categories (see the results divided by case studies in Table 7.4 and Table 7.5).

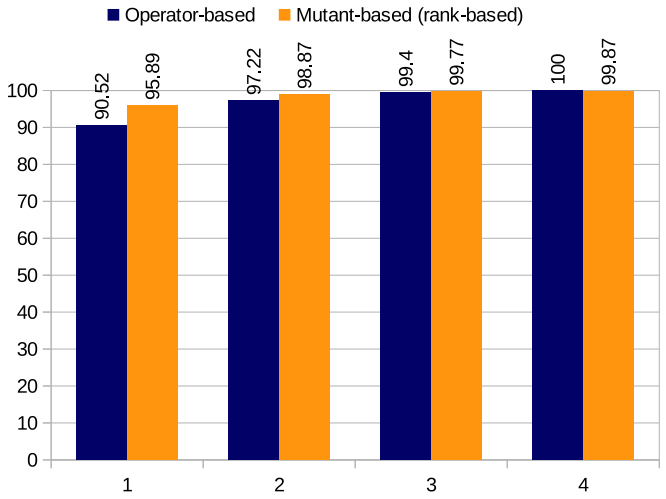


FIGURE 7.6: Comparison of the mutation score when using operator-based and rank-based selective mutation for the categories 1–4

Operator-based selection maintains the maximum mutation score when removing the operators at the bottom of the ranking. However, the rank-based mutant

selection offers better performance in the rest of categories, especially in category 1 where the gap is over 5% on average (90.52% and 95.89%).

- **TSR:** Figure 7.7 compares the average percentage of test cases loss when applying the operator-based and rank-based strategy for each of the categories (see the results divided by case studies in Table 7.10 and Table 7.11).

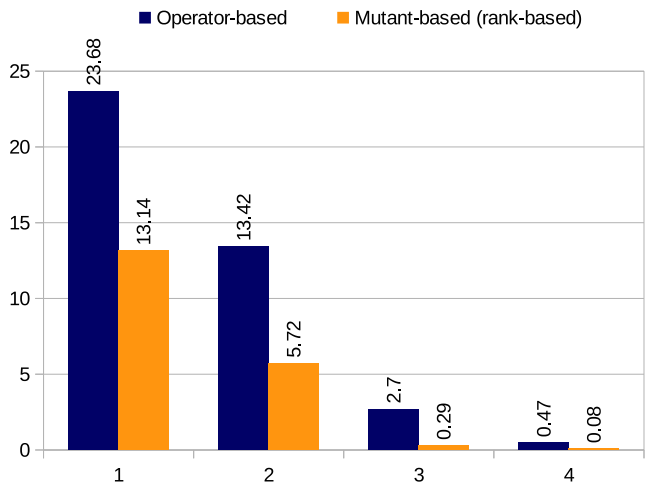


FIGURE 7.7: Comparison of the percentage of test cases loss when using operator-based and rank-based selective mutation for the categories 1–4

Rank-based mutant selection gets much better results in all of the categories. Surprisingly, the averaged result of the rank-based strategy in the first category (13.14%) not only outperforms the result of operator-based selection in the same category (23.68%) but also in the second category (13.42%).

As it can be seen, using rank-based mutant selection is not only more effective than using operator-based selective mutation, but it is also a more stable strategy in general when we analyse the standard deviations in the different categories. This fact is important since the selective process will be executed only once in practice.

It is also interesting to observe that a simple random selection of mutants (*One-round*) turned out to be better than operator-based selection except for the fourth category in both TSE and TSR evaluations (see Table 7.4 and Table 7.5). This supports the fact that each operator block addresses completely different object-oriented features. Unlike traditional operators, we suspect that several of these class operators are hardly redundant among them because they target different parts of the code and it is not common that their mutants overlap. The fact that *Two-round* random outperforms *One-round* random also supports this idea. We can draw from these results the conclusion that most of the operators can contribute to the assessment or refinement of the test

suite. Therefore, if we remove all the mutants produced by some of these operators, we might be diminishing the benefits of selective mutation.

In summary, discarding individual mutants is more convenient than discarding mutation operators when applying mutation testing at the class level.

Chapter 8

Results

This chapter summarises all the results presented throughout this thesis, highlighting the main findings. This allows for a view of the whole picture regarding the contributions and the experimental results in the different chapters. It also includes a section that collects threats to validity of the results.

8.1 Summary of Results

In the first part of this thesis, we have defined a set of 37 mutation operators for C++ at the class level. To define a set of mutation operators as complete as possible, we have surveyed existing papers in the literature addressing the definition of operators. This set of mutation operators has been classified into 7 categories, following a similar approach as previous studies in this field. Among the 37 mutation operators:

- 5 of them were particularly defined for C++ according to specific features of this language.
- 19 of them were modified with respect to their original definition or implementation for other languages.
- 13 of them were adopted without any modification.

We have compared our class mutation operators for C++ with the operators defined for other object-oriented languages, remarking the most notable differences.

After that, the approach to performing mutation testing in C++ has been presented, based on the reuse of the libraries of Clang, a parser for the C family languages. Clang

provides its libraries for the construction of new source code analysis tools. The traversal of the AST generated by this compiler is really convenient to implement a mutation tool. Moreover, the implementation technique to inject a mutation into the code has been explained step by step, with the operator *CID* as an example. This operator implementation mode is challenging as it is necessary to analyse the AST and, then, to establish a margin for each mutation operator, which sets the different situations where an operator can be applied or not. Notwithstanding, the insertion of the faults can be controlled in an accurate manner as the elements of the language are well-defined in Clang. We have illustrated the implementation of operators in *MuCPP* with the operator *CDC*.

Once a first version of the mutation operators was implemented, the set of operators was evaluated by using several case studies. The first study has been based on a qualitative analysis of the generated mutants: this was possible due to the fact that there was a manageable number of them in the analysed subjects. This study has been focused on five mutation operators (*CDD*, *CCA*, *CID*, *PVI* and *IHI*) and it has revealed interesting observations about these operators. For instance, we have pointed out that several mutants generated by *CDD* are potentially equivalent in the sense that its mutations do not necessarily propagate to the output because they only affect the memory. Some other mutants from *CCA* and *CID* have also shown that their mutation is not always reflected in the output directly: a mutant is also killed when there is a runtime error or a significant change in the execution time.

The main facilities integrated into *MuCPP* have been described: analysis of several source files, removal of duplicate mutants, use of JSON files [69], analysis of header files (avoiding system header files) and use of the Git version control system [50]. We have also stated that *MuCPP* does not suffer from scalability problems because of Git. We have also gone in depth with implementation details of the operators; by taking a closer look at the mutants generated by this type of operators, we found out some situations in which the mutations injected always led to invalid, trivial or equivalent mutants (uninteresting mutants). As a result, we have listed 9 improvement rules that can be applied to different operators in the set. These improvements have been analysed, showing that they can lead to a great reduction of the cost by avoiding the generation and execution of unproductive mutants. In particular, 46.6% of the mutants were avoided on average only taking into account those improved class operators that reduced the number of mutants in the analysed systems, and 32.1% in general. Then, we have computed the impact of that reduction in terms of time and resources. Regarding the time required, the reduction was significant in both the generation and the execution of mutants. For instance, in *KMyMoney* the test suite execution phase was reduced in almost 3 hours from the basic to the improved version of the operators. Regarding the disk space, Git

storages the mutants in a very efficient way. For that reason, the reduction of space is not as remarkable as the reduction of time when the number of mutants decreases thanks to the improvement rules.

After that, we have extended the quantitative evaluation, analysing several subjects and calculating several metrics about the performance of these operators, such as the total number of mutants, equivalent mutants or their mutation scores. These have been the main findings:

- We first calculated the number of mutants generated by a subset of traditional mutation operators based on previous studies in the literature about selective mutation. The experiment reported clear results: there were many more traditional mutants for the same applications than class mutants. On average, the subset of class mutation operators injecting mutants into a class produced 39.7 mutants per class. Class operators therefore required less time to evaluate. On the contrary, the percentage of equivalence was high. It seems that the improvement rules helped reduce the percentage of equivalent mutants, but it was still quite notable (27.9%). These two observations align with the results of similar studies in the literature.

- The quantitative analysis of the mutants revealed which are the most prolific operators: *PCI* and *MCO* . However, the operators generating more mutants did not always coincide with those mutating more classes. For instance, *MCO* injected 7 mutations on average per class, and mutated 17.8% of the classes. On the contrary, *CID* injected a mean of 2.4 mutations per class but it was applied to 47.4% of the classes. Other operators generated very few mutants (*IHD*, *ISD*, *PVI* and *PNC*) or were not applied in the tested applications (*IMR*, *PRV*, *OAO*, *EHR* and *CTD*). All these facts reflect how different mutation operators behave depending on the features used, though the existence of inheritance relationships is a factor with a strong impact on the application of this set of class operators.

- We calculated the mutation score of the mutants generated in a part of these applications. We also augmented the test suite by reviewing surviving class mutants, obtaining an adequate test suite for two of the programs. In all cases, the mutation score was far from 100% (32% in the worst case and 80% in the best case). This fact suggests that the test suites for these applications do not tackle object-oriented features properly and shows the ability of class mutants to reveal test deficiencies at the same time. While inspecting surviving mutants and designing new test cases, we found two coding errors in two of these programs.

We have to remark that the obtained experimental results can only be compared with similar studies to a certain extent. For example, C++-specific operators such as *CDD*

and *CCA* are not defined for other object-oriented languages, and some of the modified operators are completely different from the comparable mutation operators in other languages, like *CID*.

Regarding the comparison of class-based and traditional mutants, we wanted to examine the extent to which class-based mutants could contribute to the assessment of test suites for object-oriented systems when used along with traditional mutants. To do so, we have devised a new metric (T_d) which estimates the percentage of test cases in a minimal test suite that is added when evaluating class mutants in addition to traditional mutants. The result gave evidence that a proportion of those test cases only appeared when analysing class-based mutants in all the systems under test (from $T_d = 0.04$ in the worst case to $T_d = 0.16$ in the best case)[1]. Given that we only reviewed class mutants in the process of test suite improvement, it was interesting to observe that some of the test cases that had not been modified or added by ourselves appeared when considering the class mutants in 3 of the 5 subjects. Finally, we conducted an experiment to analyse the subsuming relation between both sets of operators. In this experiment, we computed the mutation score of class mutants associated with different test suites guided by traditional mutants and vice versa. On average, class-adequate test suites obtained a mutation score of 80% when they were run against traditional mutants and, conversely, the mutation score of traditional-adequate test suites when applied to class mutants was 93%. Although this aspect would require further investigation, there seems to be a higher percentage of class mutants which is trivial to detect. However, there are two aspects that should be taken into account if we want to draw a more detailed conclusion. First, a tester who wants to obtain a more effective test suite will still need to analyse the class mutants. Second, we should also consider the number of mutants generated by each set of operators, as mentioned earlier.

In this thesis, we have developed *GiGAn* to connect the genetic algorithm implemented in *GAmera* and *MuCPP*. In this way, we could analyse the performance of EMT in C++ object-oriented systems. Our experiments differed from previous experiments in two aspects that might impact the results: some of the subject programs were comprised of several source files (a mutant injected into a file can breed a new mutant belonging to a different file) and the attribute field was not taken into account (which limited the application of some reproductive operators).

We wanted to answer several research questions with our experiments. Firstly, we have studied the behaviour of EMT when searching different percentages of strong mutants, replicating previous studies with WS-BPEL compositions [43]. In this regard, we found

[1]Recall that T_d indicates the proportion of test cases that appears in the minimal test suite only when the class-level mutants are analysed.

that the relation between the number of mutants generated and the number of strong mutants did not vary much among the subject programs. We observed this relation when finding five different percentages of strong mutants (from 30% to 90% in 15% increments) in four programs. In addition, the experiments reported that the proportion of strong mutants found by EMT slightly decreased with the number of mutants generated in general.

Secondly, we wanted to know whether EMT produced better results when compared to random mutant selection. In this case, we have compared both strategies when finding the two highest percentages of strong mutants (75% and 90%). EMT yielded better results than the random strategy with high confidence (a smaller percentage of mutants was needed to achieve the same percentage of strong mutants). On average, the gap between EMT and Random was 6.17% and 3.80% to find 75% and 90% of strong mutants respectively. The most notable difference was observed in *TXM*: 10.02% when reaching 75% of the set of strong mutants. The gap between both strategies was however greater in the experiments with WS-BPEL.

Finally, in the last experiments related to EMT, instead of evaluating how useful this technique is in finding strong mutants, we wanted to know about the extent to which EMT leads to the generation of missing test cases when compared to a random strategy. Our experiment simulated a real process in which the generated mutants were reviewed and new test cases were added to detect surviving mutants. Despite the influence of test suites in this experiment, EMT outperformed Random in general. The best results were obtained for the largest programs (in terms of generated mutants), where we can highlight a difference between EMT and Random around 26% and 45% in *Tinyxml2* and *QtDom* respectively to find the whole minimal and adequate test suite. However, these results should be taken with caution because of the presence of invalid mutants.

In Chapter 7 we have arranged two operator rankings for TSE and TSR as a first step. To that end, we made use of two different metrics (degree of redundancy and quality metric) in order to rank the operators. When doing so, we found differences between both rankings, sometimes quite pronounced such us in the case of *ISI* (a difference of 8 positions between rankings) and *PCI* (7 positions).

Then we sought to determine whether those rankings were useful from a selective perspective (i.e., to what extent we can discard operators/mutants based on the rankings without significant loss in the effectiveness). We have found out that we cannot use the same metric to measure the effectiveness of a selective strategy for TSE and TSR. Instead of measuring the mutation score as in TSE, we measured the loss of test cases for TSR, which we called test-quality selective mutation. In addition to operator-based selection following our rankings, we have proposed a strategy to perform mutant-based selection

also based on these rankings: rank-based mutant selection. This strategy favours the generation of mutants from the best-valued operators. The first finding was that we could remove several operators/mutants using the rankings with minimal information loss. As an illustration:

- In the case of TSE and operator-based selection, we could discard 10 operators maintaining the same mutation score (100%).
- In the case of TSR and rank-based selection, we could discard as many mutants as the number of mutants generated by the 8 top ranked operators with a minimal loss of test cases (0.08% on average).

To complete the evaluation of the operator rankings, we have performed two comparative analyses of selective strategies:

1. Operator-based mutant selection vs mutant-based selection.
2. Rank-based mutant selection vs random mutant selection.

Regarding (1), since there does not seem to be a high redundancy among class operators from different categories, removing operators can reduce the effectiveness of mutation testing. As a consequence, mutant-based selection was found to be superior to operator-based mutant selection. For instance:

- The mutation score was 97.22% for operator-based selection and 98.87% for rank-based selection when considering the 6 top operators and the same number of mutants as generated by those 6 operators respectively.
- The percentage of test cases loss was 13.42% for operator-based selection and 5.72% for rank-based selection when considering the 7 top operators and the same number of mutants as generated by those 7 operators respectively.

Recall that the higher the mutation score, the better. Similarly, the lower the percentage of test cases loss, the better. The reduction in the number of mutants achieved in the above mentioned cases was 31.7% and 39.42% for TSE and TSR respectively.

As for the second comparative analysis (2), while mutant-based selection performed better than operator-based selection, this study showed that the results of the rank-based strategy were better than those of the random strategy. We could observe that the results for the rank-based strategy were better than those of two random strategies

overall, especially in the category 1 when the highest reduction of mutants was achieved (mutation score of 95.89% vs 95.59% for TSE and percentage of test cases loss of 13.14% vs 14.99% for TSR). We should remark that the differences between both mutant-based strategies were more notable in the case of the percentage of test cases loss than for the mutation score, but we should take into account the nature of these metrics when analysing the data (the margins are usually narrow in the case of the mutation score).

As a summary, the main finding of these experiments was that, while selecting a subset of mutants from all operators was a better approach than discarding operators, it was also true that favouring the selection of mutants generated by the best-valued operators reported better results than random mutant selection.

8.2 Threats to Validity

Several aspects pose a threat to validity of the results derived from the experiments conducted in this thesis:

Mutant equivalence. Equivalence is an inherent limitation to mutation testing because this is an undecidable problem. As such, the metrics shown may be inaccurate because they might be influenced by the manual determination of equivalent mutants. This is an error-prone task, especially when analysing third-party applications for which it is not trivial to acquire a full insight into the source code.

To counter the threat that the quality metric by Estero-Botaro et al. [45] penalised mutants incorrectly classified as equivalent, we classified as undecided [117] instead of as equivalent those mutants for which we were unsure.

Test suites. The reliance of mutation testing on the test suites supposes a threat to validity of the results. We used the test suite accompanying the analysed subjects, so we worked with test suites developed by different testers:

- Some of the test suites make a more exhaustive use of the classes and their members than others.
- We can classify some test cases as specific (testing a particular functionality) or general (testing a subset of related functionalities), which present a different killing power.

To the best of our knowledge, there are no similar test case generators to *EvoSuite* [48] (for Java) addressing object-oriented programs in C++: this would help generate new

test cases driven to kill surviving mutants. Therefore, we extended the test suites manually to achieve adequate test suites. The modified and new test cases were designed with the utmost care to develop consistent test cases. The use of a single test suite can also impact the results. The minimisation of the test suite for our calculations with an exact algorithm to exclude unproductive test cases alleviates the effect of these potential threats.

Mutation operators. One of the main limitations in the conducted studies is that only a subset of all the operators could be analysed in depth because not all the analysed operators produced mutants in the case studies. Most class mutation operators often generate no or few mutants in each class because they are less prolific than traditional operators and depend on the object-oriented features used by the programmer. Therefore, we could not measure the metrics for several of the class-level operators.

Moreover, many of the improvement rules to enhance the effectiveness of the operators produced great improvements in the operator efficiency in some cases, while having no effect on others. However, new rules could be detected applying the technique to other programs with different features and could lead to a further reduction of uninteresting mutants.

Altogether, the different behaviour of the operators in each application makes it difficult to provide conclusions and generalise the results to the whole set of operators.

Implementation. The generation and execution of mutants to obtain execution matrices, the calculation of the metrics and all the experimental procedures in this thesis rely on multiple software systems. The experiments have been automated whenever possible, but there may exist defects in the tools implemented and the systems used despite being thoroughly tested.

We cannot ensure that the improvements implemented in the mutation operators prevent valid mutants to be created due to dark corners of the language or the AST, which might not have been considered.

Generalisation. Representativeness of the programs under study is a common threat to validity of the results. It is not easy to ensure that the studied population is representative, so the results reported should be interpreted as estimations. Nonetheless, we have selected applications of varying nature so that different mutation operators were applied and those operators generated a different number of mutants. Selecting several programs of diverse complexity and sizes minimises the threat to the generalisation because it avoids the partial perspective of the individual applications.

We should note that the experimental procedures were carried out using class mutation operators in C++, so it is unknown if the results hold in other contexts.

Metrics. There are some threats related to the metrics computed in our experiments:

- *Mutation score*: The mutation score may greatly vary depending on the operators because of the few mutants injected into a class. Several trivial mutants are avoided because of the improvement rules, which may have reduced the mutation score. We only analysed a subset of the mutants generated in the applications because reviewing these mutants and designing new test scenarios to kill them is a time-consuming and laborious task.

- T_d: Being this metric dependent on the test suite, the results may change if the test suites were adequate both for class-level and traditional mutants. In the same line, the results are also dependent on the assertions and test cases added to increase the mutation score. As it was mentioned earlier, we designed the test cases as general as possible to reduce this kind of threat.

- *Quality metric*: As aforementioned, the object-oriented features of the language are used with varying frequency, so several operators did not produce a significant number of mutants. As a result, by maintaining the threshold in the number of mutants to apply the quality metric used by Estero-Botaro et al. [45], the metric could not be computed for several operators. Given that some operators could not be appropriately evaluated, it is required further research so that we can better know the usefulness of each mutation operator.

  To assess the performance of the quality metric, we measured the percentage of test cases loss instead of the mutation score. However, that percentage does not provide information on the specificity of the test cases (losing trivial test cases is not as important as losing high-quality test cases).

EMT. The performance of the genetic algorithm may vary depending on the values given to the parameters, but the best configuration is unknown in practice. As such, we have used the same configuration that Domínguez Jiménez et al. [43] found to be optimal in their experiments.

Assessing randomised algorithms requires several executions to avoid biased results. We executed the techniques 30 times, a common number of runs according to the guide by Arcuri and Briand [9].

In the second experiment to evaluate EMT, we simulated the process of extending the test suite thanks to this technique. By selecting a particular mutant, the minimal and

adequate test suite in a generation can change (T_{MA_i}). However, that does not mean that each of the mutants has the potential to help the tester design all the test cases that kill the mutant, especially when the test suite is comprised of general test cases.

Comparison between mutant-based test strategies. In Chapter 7, we compared the results of different strategies for the reduction of mutants. Namely, we prepared new operator rankings following traditional approaches to operator-based selective mutation and two random strategies for the selection of mutants. Subsequently, we compared them with operator-based and rank-based selective mutation based on our rankings respectively. Both our operator-based and rank-based strategies for TSE and TSR yielded better results overall, though this sanity check is limited to the subset of strategies analysed.

We also compared operator-based and mutant-based selective mutation under the same number of non-equivalent mutants, as done by Zhang et al. [135]. However, we should note that a mutant-based strategy will also select a subset of equivalent mutants in practice, which might impact the results. Recently, Papadakis et al. [109] found that this kind of comparative studies are vulnerable to a threat to validity if mutant subsumption is not controlled.

Chapter 9

Conclusion and Future Work

The last chapter is devoted to the conclusions drawn from this research period and the future work lines. A list of the publications derived from this thesis complements this closing chapter.

9.1 Conclusions

The set of mutation operators is a key factor in mutation testing as operators have the potential to guide us to effective test suites. Given that mutation testing is a language-dependent technique, the first step to apply mutation testing is the definition of mutation operators for the different programming languages. In this sense, the study carried out is an important contribution because there were no works focused on the definition of a set of operators for C++. Despite the dependence on the language, it is necessary that the entire development of the technique follows the same path so that the studies for a specific language are as generalisable as possible for similar languages. As such, the mutation operators for other similar languages have been analysed, mainly around Java (because this language has drawn the attention of multiple studies regarding object-oriented programming), and also C#.

As a result, a complete set of 37 operators was defined for C++ at the class level. We have seen that many of the adopted operators are impacted by different C++ characteristics and we have also created new operators according to particular C++ features. We compared these operators with those defined for Java and C#, highlighting similarities and differences. Overall, the multiple facilities provided by C++ makes more complex the operator implementation than for other languages. We evaluated certain operators in qualitative terms; this study allowed us to observe particular situations in

which operators at the class level could detect test deficiencies. Likewise, the calculation of the mutation score gave us a first evidence that test suites implemented for real programs do not deal with object-oriented features properly, justifying the incorporation of this type of operators into mutation systems for C++. This qualitative assessment was complemented with a quantitative evaluation, showing an approximation of the number and kind of mutants that these operators usually generate. The results supported several observations in previous experiments in the literature. Firstly, class operators tend to generate few mutants when compared to traditional operators. This is a positive observation since the cost has always been one of the major concerns when applying mutation testing. Secondly, the percentage of equivalence is pronounced, which is the other main problem in this technique. Thus, it is important to find ways to reduce the number of this kind of mutants. Finally, these operators show a different behaviour in each application, which mainly depends on the object-oriented features used. All the experiments are inevitably impacted by the subjects under study, but this fact makes difficult to generalise the results for any system.

The correct definition and implementation of mutation operators are fundamental to successful mutation testing so that they provide valid and useful mutants for the analysis of the technique. Thus, different situations have been considered to create the expected mutations. Moreover, the operator quality has been enhanced by establishing a specific scope for the implementation of each operator which cuts out unnecessary mutants. Previous studies have defined rules to automatically remove equivalent mutants in particular operators. In our thesis, we have set general improvement rules, which can be taken into account for different mutation operators in different languages. With this approach, the equivalence drawback can be alleviated and the mutation operator effectiveness, as well as the computational cost, can be improved in general, both when generating and when executing the mutants (mainly with regard to the compilation time), increasing the efficiency of the mutation system.

The work presented here brings down the barrier regarding the complicated task of automating the mutations in C++ by developing a feasible and comprehensive solution through the traversal of the AST generated with Clang [22]. The AST is used to determine the mutation locations through pattern matching, and to transform the code in a robust way, given that this pattern matching is not based on the concrete syntax of the language. Also, using a mature parser for this language like Clang guarantees a complete coverage of the grammar. *MuCPP* is the first mutation system for C++ implementing traditional and class-based operators. This system incorporates other interesting features that enhance the mutation testing process: *MuCPP* uses the Git version control system [50] to save storage resources and to facilitate the generation and execution of mutants. Even though this is not the first time that a version control system is used to

create mutants in a mutation tool, Git has shown to be more convenient for mutation testing than SVN.

Given that a C++ project usually comprises several source files, the analysis of more than a single file at the same time is an interesting option. To that end, *MuCPP* was designed to sequentially parse several source files in the same execution, identifying duplicate mutants when a header is included in different source files. JSON compilation database files also help us analyse different source files in the same execution. Using JSON files, each source file is independently analysed to be compiled with the proper command. This is an automatic process that allows us to forget about compilation details. The mutation system is not only a mutant generator but also handles the execution of the test suite against the mutants. While *JUnit* is broadly used to implement test suites for Java programs, there is not a prevailing framework when it comes to C++. As a result, we have found a plethora of frameworks and libraries used by testers when searching for case studies for our experiments. The fact that *MuCPP* is not subject to a specific testing framework has facilitated the experiments and will avoid testers having to translate test suites already implemented.

The conducted experiments have shown that test suites developed for object-oriented systems often fail at addressing the particularities of the object-oriented paradigm. Surviving class mutants have been helpful to improve the quality of those test suites and also to detect real coding errors. However, we wondered whether the same missing test scenarios could be found just using traditional operators. Standard operators can also be applied to test object-oriented systems, but experts in this field have hypothesised that those operators for procedural languages were not sufficient since they do not consider some types of faults related to object-oriented features. In this work, we have compared traditional and class-based operators, showing that effectively class mutants can provide information that may not be derivable from traditional mutants. The main conclusion is that class operators in conjunction with traditional operators can be applied to design more comprehensive test suites as these two sets complement each other. While it is true that traditional operators make a greater contribution to the assessment of a test suite than class operators, our experiments also confirm that there are far fewer class mutants than traditional ones. Thus, it would be interesting to explore in the future whether first analysing class mutants before traditional ones could be a good strategy to find missing test cases without inspecting many mutants.

The reduction of the expenses in mutation testing should be based on well-studied cost reduction techniques to avoid biased results. Evolutionary Mutation Testing (EMT) aims at generating a reduced set of mutants by means of an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants to guide on the

creation of new test cases. However, there was little evidence of its applicability to other contexts beyond WS-BPEL compositions. This study explored its performance when applied to C++ object-oriented programs thanks to a newly developed system, *GiGAn*. The experiments revealed that EMT has stability among the tested programs and little variation in the percentage of strong mutants found as the number of generated mutants increases. They also support previous studies about EMT when compared to random mutant selection, with better results in all case studies with high confidence.

In this thesis, we went a step further in estimating the ability of this technique to induce the generation of test cases. Instead of measuring the relation between the percentage of strong mutants and the number of mutants generated, we computed the extent to which the test suite could be actually improved thanks to the mutants selected. We can conclude from the results that the percentage of mutants generated with EMT is lower than with the random strategy to obtain a test suite of the same size. In other words, the results show that mutation testing can leverage this genetic algorithm to produce a subset of mutants that leads to a further test suite improvement when compared to the random selection of the same size of mutants. Additionally, another positive factor is that the technique scales better for complex programs. Altogether, these experiments confirm the promising results yielded by EMT in previous research studies.

Most of the studies on cost reduction techniques have analysed traditional mutants for procedural languages like C. However, it remained unclear whether the benefits yielded by those techniques could be extrapolated to other sets of operators, especially to class-level operators which have shown to be of a different nature. In the conducted experiments in this thesis applying selective mutation based on the best-valued operators for TSE and TSR, we have observed that class operators may not benefit from operator-based selection as much as traditional operators. Thus, it might be the case that, by using operator-based selective mutation with class-based operators, it is not possible to reach the great reduction and effectiveness obtained in similar studies addressing other sets of operators. As for EMT, while this technique allows reducing the number of mutants generated, the reduction was greater when mutation operators for WS-BPEL were applied. All these results suggest that the effectiveness of object-oriented mutation testing may be more related to particular mutants than to particular operators.

In our thesis, we have extended the selective approach for the evaluation of the operators by considering not only operator-based selection (selection of some of the operators) but also mutant-based selection (selection of some of the mutants). This resulted in a more comprehensive operator evaluation that allowed us to undertake a comparative study between operator-based and mutant-based selection. Moreover, we found this double perspective necessary given the significant body of research that has called the

benefits of operator-based selection into question recently. As a first interesting finding, the random selection of mutants turned out to be better than operator-based selective mutation. However, maintaining a complete set of operators and generating mutants from all of them with the same probability seems a more convenient approach, which supports the idea that discarding mutants is more effective than discarding operators in object-oriented mutation testing. Then, we proposed a new mutant-based selective strategy following our rankings of operators. This mutant-based strategy, which we call rank-based mutant selection, favours the selection of mutants from the best-valued operators (those operators at the top of our rankings). To complete the selective study, we have compared the results of using random and rank-based mutant selection. While random mutant selection was found to be superior to operator-based mutant selection, this novel study showed that the results of the rank-based strategy were better than those of the random strategy, validating the operator evaluation in this thesis, both for TSE and TSR.

As we have exposed in this thesis, there is a clear difference between finding hard-to-kill mutants (when the test cases killing those mutants kill few other mutants at the same time) and finding representative mutants of the whole set of mutants (when the test cases killing those mutants also kill other mutants), but none of the authors in this field had made this distinction before to the best of our knowledge. The differences found in the operator classifications when they were evaluated regarding TSE and TSR validate this double assessment. Therefore, this study suggests that a different sort of mutation operators should be used depending on the goal, TSE or TSR. In practice and from the operator-based selective perspective, the mutation tool should implement the full set of mutation operators and should allow us to enable/disable each of the operators (or alternatively each of the categories defined in the selective process). From the rank-based selective perspective, the tool should allow us to indicate how many mutants we want to generate, and then the tool should take into account the sort of mutation operators to favour the generation of mutants from the best-valued operators. Therefore, the tester might want to generate a different subset of mutants depending on:

1. The goal when applying mutation testing, TSE or TSR.
2. How thorough the testing process needs to be.

9.2 Future Perspectives

In this section, we suggest several investigation lines and work to undertake in the near future.

- **Mutation operators**: As for the future work in this aspect, we will divide our efforts between the refinement of our set of operators and the possibility of adding new operators according to features not covered yet. Mutation testing is in continuous development as well as the programming language itself, and we should update our list of operators in accordance with this evolution. For instance, recent studies have pointed to the usefulness of deletion operators (statement, variable, operator and constant) [29, 32]. Moreover, new standards for C++ are being approved in the last years (C++11, C++14 and C++17). Clang announced recently that it had achieved a full C++11 and C++14 compliance and the project is starting to take care of the new changes in the standard C++17. This fact could allow us to keep using Clang us the underlying technology to locate and inject mutations if the set of operators needs to be redefined in the future because of the new standards.

  The application of *MuCPP* to new case studies can also lead to the detection of new improvement rules in addition to the ones presented in this thesis, which could be incorporated into the implementation of several operators. The review of these standards can reveal novel mutation operators, but also some of the operators already defined could need to be reconsidered because of the new features.

- **Mutation system**: *MuCPP* is currently a mutation system that generates all the mutants contemplated by the mutation operators, except when the tool is applied in conjunction with the genetic algorithm thanks to *GiGAn*. This tool also executes all mutants and test cases because it does not count with runtime optimisations. As such, *MuCPP* has room for improving its efficiency and expanding the functionalities. Regarding the generation of mutants, the system could avoid those mutants not covered by the test suite as done by similar tools [40, 114]. As for the execution of mutants, the same strategy could be used to avoid the execution of those test cases that do not cover a mutant. Also, Trivial Compiler Equivalence (TCE) proposed by Papadakis et al. [108] could be used to automatically detect equivalent and duplicate mutants (see Section 2.4.1). A final goal is to study the impact of these improvements on the efficiency of mutation testing. For example, TCE has been integrated into MILU for C, but the performance of this technique is unknown for object-oriented languages. Updating the mutation system (for instance, adapting it to the new versions of Clang) and including new options to offer greater flexibility are important aspects for its maintenance in the future.

- **EMT improvements**: The findings in this thesis and related studies on mutation testing in the last years lead us to think that this technique can be further improved. In this sense, the future line will follow a multi-objective approach. Three new objectives could be integrated into the fitness function:

1. *Coverage impact*: Several papers have analysed the impact that mutations have on the code coverage [107, 115]. Roughly speaking, those mutations causing a great impact on the coverage of the test suite execution are less likely to be equivalent mutations. EMT currently assigns the highest fitness to potentially equivalent mutants, but it cannot distinguish between those mutants that turn out to be equivalent and those that help design a new test case. Analysing the coverage impact of the mutants could guide on the selection of non-equivalent mutants with a high probability.
2. *Scattering in the code*: The genetic algorithm selects mutations without taking care of the location in which it is injected. An object-oriented program is divided into different classes, which counts with different methods comprised of multiple statements. Furthermore, some methods or even statements are directly associated to specific object-oriented features (and thereby to particular class operators), such as constructors or exceptions. Therefore, spreading mutations all over the code seems an important aspect so that all code items are covered, as done by Schwarz et al. [116].
3. *Scattering in the set of operators*: This objective aligns with a finding in this thesis. Given that *Two-round* random selection yielded better results than *One-round* random selection (see Section 7.4), it is plausible to think that each class operator is useful to address a different object-oriented feature. As such, it is better to produce mutants from all operators, and this is not currently taken into account by the genetic algorithm. Consequently, the algorithm should favour the selection of mutants from operators barely applied so far.

As aforementioned, Papadakis et al. [108] devised the technique called TCE to detect some equivalent mutants automatically by comparing an optimised executable of the original program and the mutants. It could be interesting to combine both TCE and the genetic algorithm to isolate some equivalent mutants and observe the effect when applying this technique before or during the execution of the genetic algorithm.

- **Operator rankings**: Despite the good performance of the operator classifications shown in this thesis, it will be interesting to obtain more stable rankings based on the results of a greater number of case studies and covering mutation operators that were not evaluated because they did not generate any mutants in the programs analysed. It would be interesting to apply our approach for the reduction of the cost based on operator rankings to other sets of operators.

  Selective mutation based on operator rankings is studied in advance in order to incorporate this knowledge into the mutation tool, whereas the benefits of EMT

applies directly during the execution of the tool. In other words, while EMT uses the current execution information to reduce the cost, the applied metrics make use of the information once the test suite has been improved. As a result, the approach of both techniques could be merged to further reduce the cost: EMT could only generate mutants from the best-valued operators based on the ranking for TSR and mutants could be generated in a rank-based manner instead of randomly.

- **Operator-based vs mutant-based selective mutation**: The performance of mutant-based selective mutation in comparison with operator-based selection was a surprising fact. The analysis of traditional operators has usually revealed great redundancy among operators and, consequently, a subset of operators could subsume the rest. Our research suggests that this high degree of redundancy does not hold in the case of class-based operators. However, we feel that the application of these two selective techniques should be explored further to know the extent to which mutant-based selective mutation is superior to operator-based selection. For example, we would like to examine if, despite this fact, we can still remove some operators or can use both selective strategies together, as suggested by Zhang et al. [136]. We should take into account that one of the basics of EMT is the generation of strong mutants from those operators that have generated them in previous generations, so it is important to understand how much the nature of class operators can affect the performance of EMT.

  In our thesis, we have proposed the technique called rank-based selective mutation based on the rank selection method used in genetic algorithms. Despite the good results, it would be interesting to compare the results of different selection techniques, such as roulette wheel selection, tournament selection or stochastic universal sampling.

9.3 Publications

This section presents the publications with contributions derived from this thesis, divided into journals, book chapters and conferences and symposiums.

9.3.1 Journal articles

Software Testing, Verification and Reliability, 2017

Pedro Delgado-Pérez, Sergio Segura and Inmaculada Medina-Bulo

Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach

Software Testing, Verification and Reliability. Available online. doi: 10.1002/stvr.1630

Impact factor: 1.082 - Q2 (JCR 2015)

Abstract: Mutation testing is an effective but costly testing technique. Several studies have observed that some mutants can be redundant and therefore removed without affecting its effectiveness. Similarly, some mutants may be more effective than others in guiding the tester on the creation of high-quality test cases. Based on these findings, we present an assessment of C++ class mutation operators by classifying them into two rankings: the first ranking sorts the operators based on their degree of redundancy, and the second regarding the quality of the tests they help to design. Both rankings are used in a selective mutation study analysing the trade-off between the reduction achieved and the effectiveness when using a subset of mutants. Experimental results consistently show that leveraging the operators at the top of the two rankings, which are different, lead to a significant reduction in the number of mutants with a minimum loss of effectiveness.

Information and Software Technology, 2017

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

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

Information and Software Technology, 2017; volume 81, pages 169-184, ISSN 0950-5849, doi: 10.1016/j.infsof.2016.07.002

Impact factor: 1.569 - Q1 (JCR 2015)

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 on 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.

Annals of Telecommunications, 2015

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, J. J. Domínguez-Jiménez, Antonio García-Domínguez and Francisco Palomo-Lozano

Class mutation operators for C++ object-oriented systems

Annals of Telecommunications 2015; 70(3-4):137-148, ISSN 0003-4347, doi: 10.1007/s12243-014-0445-4

Impact factor: 0.722 - Q3 (JCR 2015)

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.

9.3.2 Book chapters

Encyclopedia of Information Science and Technology, 4th Edition

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

Mutation Testing Applied to Object-Oriented Languages

Khosrow-Pour, M. (Ed.) Encyclopedia of Information Science and Technology, 4th Edition, IGI Global, 2018, doi: 10.4018/978-1-5225-2255-3

Encyclopedia of Information Science and Technology, 3rd Edition

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

Mutation Testing

Khosrow-Pour, M. (Ed.) Encyclopedia of Information Science and Technology, 3rd Edition, pages 7212-7221, IGI Global, 2015, doi: 10.4018/978-1-4666-5888-2.ch710

9.3.3 Conferences and symposiums

CEC 2017

Pedro Delgado-Pérez, Inmaculada Medina-Bulo and Manuel Núñez

Using Evolutionary Mutation Testing to Improve the Quality of Test Suites

IEEE Congress on Evolutionary Computation 2017 (CEC 2017), accepted contribution.

Abstract: Mutation testing is a method used to assess and improve the fault detection capability of a test suite by creating faulty versions, called mutants, of the system under test. Evolutionary Mutation Testing (EMT), like selective mutation or mutant sampling, was proposed to reduce the computational cost, which is a major concern when applying mutation testing. This technique implements an evolutionary algorithm to produce a reduced subset of mutants but with a high proportion of mutants that can help the tester derive new test cases (strong mutants). In this paper, we go a step further in estimating the ability of this technique to induce the generation of test cases. Instead of measuring the percentage of strong mutants within the subset of generated mutants, we compute how much the test suite is actually improved thanks to those mutants. In our experiments, we have compared the extent to which EMT and the random selection of mutants help to find missing test cases in C++ object- oriented systems. We can conclude from our results that the percentage of mutants generated with EMT is lower than with the random strategy to obtain a test suite of the same size and that the technique scales better for complex programs.

SAC 2017

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Sergio Segura, Antonio García-Domínguez and Juan José Domínguez-Jiménez

GiGAn: Evolutionary Mutation Testing for C++ Object-Oriented Systems

The 32nd ACM Symposium On Applied Computing (SAC 2017), accepted contribution.

Abstract: The reduction of the expenses of mutation testing should be based on well-studied cost reduction techniques to avoid biased results. Evolutionary Mutation Testing (EMT) aims at generating a reduced set of mutants by means of an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants to help improve the test suite. However, there is little evidence of its applicability to other contexts beyond WS-BPEL compositions. This study explores its performance when applied to C++ object-oriented programs thanks to a newly developed system, GiGAn. The conducted experiments reveal that EMT shows stable behavior in all the case studies, where the best results are obtained when a low percentage of the mutants is generated. They also support previous studies of EMT when compared to random mutant selection, reinforcing its use for the goal of improving the fault detection capability of the test suite.

JISBD 2016

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Sergio Segura, Antonio García-Domínguez y Juan José Domínguez-Jiménez

Prueba de Mutación Evolutiva Aplicada a Sistemas Orientados a Objetos

XXI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2016), 13–16 septiembre 2016, Salamanca, España

Resumen: A pesar del beneficio que puede reportar la prueba de mutaciones en el proceso de prueba de software, el coste que supone su aplicación siempre ha sido visto como un obstáculo para una mayor acogida por parte de la industria. Por esta razón, se han desarrollado diversas técnicas que tratan de paliar el problema, principalmente mediante la reducción del número de mutantes que son generados. Entre ellas se encuentra la Prueba de Mutación Evolutiva (PME), que propone el empleo de algoritmos evolutivos para encontrar un subconjunto de mutantes que presenta mayor posibilidad de ayudar a refinar el conjunto de casos de prueba empleado. La técnica solo había sido probada con éxito en operadores para el lenguaje de programación WS-BPEL. En este artículo se presentan los experimentos llevados a cabo aplicando la técnica de PME con mutantes

generados por operadores de mutación para C++ relacionados con la orientación a objetos. Los resultados obtenidos, usando los parámetros considerados como más apropiados para la configuración del algoritmo, revelan que la técnica también es más efectiva que una estrategia aleatoria con operadores de clase para sistemas en C++.

SS-SSBE 2016

Pedro Delgado-Pérez

Evolutionary Mutation Testing Applied to Object-Oriented Systems.

First International Summer School on Search-Based Software Engineering (SS-SBSE 2016)

Abstract: Mutation testing is a powerful testing technique to assess and refine the fault-revealing ability of a test suite, but it involves a high cost. Evolutionary Mutation Testing proposes the generation of a subset of the mutants by means of an evolutionary algorithm in order to reduce the cost. This algorithm favours that the subset contains mutants with great potential to assist the tester in improving the test suite with new test cases. This technique had been successfully applied to WS-BPEL compositions. In this talk, we present the results when using class mutation operators in C++ object-oriented systems.

ESCIM 2015

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

Correct Application of Mutation Testing to the C++ Language

7th European Symposium on Computational Intelligence and Mathematics (ESCIM 2015), 7–10 october 2015, Cádiz, Spain.

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

JISBD 2015

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

Herramienta para la Prueba de Mutaciones en el Lenguaje C++

XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2015), 15–17 septiembre 2015, Santander, España

Resumen: La prueba de mutaciones es una técnica basada en fallos en torno a la cual se han elaborado herramientas para un amplio abanico de lenguajes de programación. Sin embargo, el desarrollo de un marco de prueba de mutaciones no comercial para C++ estaba pendiente. En este artículo se presenta una herramienta que permite analizar código C++, generar mutantes y ejecutar un conjunto de casos de prueba para obtener resultados que nos permitan determinar su efectividad en la detección de errores en el código. La herramienta está diseñada para permitir la inclusión de nuevos operadores para cubrir cualquier característica del lenguaje. En este documento, el uso de la herramienta se muestra a través de un operador de mutación al nivel de clase.

TAROT 2015

Pedro Delgado-Pérez

Advances in Mutation Testing Research for C++

11th International Summer School on Training And Research On Testing (TAROT 2015), 29 june–2 july 2015, Cádiz, Spain.

SGSOACS 2014

Pedro Delgado-Pérez

Advances in Mutation Testing Research for C++ with MuCPP

First Spanish-German Symposium on Applied Computer Science (SGSOACS 2014), 11 december 2014, Cádiz, Spain.

JISBD 2014

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

Generación de Mutantes Válidos en el Lenguaje de Programación C++

XIX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2014), 16–19 septiembre 2015, Cádiz, España

Resumen: La prueba de mutaciones es una técnica basada en fallos que se ha desarrollado alrededor de un amplio rango de lenguajes de programación. Sin embargo, la construcción de un marco de trabajo de prueba de mutaciones no comercial para C++ ha sido pospuesto en favor de otros lenguajes, principalmente por la variedad de alternativas que ofrece C++. Este artículo presenta una solución factible y completa para la implementación de los operadores de mutación en C++, la cual se basa en la búsqueda de patrones en el árbol de sintaxis abstracta (AST) que el compilador Clang genera a partir del código fuente. Estos patrones se construyen según las reglas que determinan los distintos operadores de mutación, permitiendo localizar los puntos del código en los que es posible introducir una mutación. Asimismo, en el artículo se abordan distintas situaciones que han de ser consideradas para la validez de los mutantes creados. Este proceso se ilustra a través de un operador de mutación a nivel de clase, si bien este enfoque sirve para crear operadores a cualquier nivel del lenguaje.

ICCGI 2014

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

Analysis of the Development Process of a Mutation Testing Tool for the C++ Language

The Ninth International Multi-Conference on Computing in the Global Information Technology (ICCGI 2014), 22–26 june 2014, Seville, Spain.

Abstract: Mutation testing is a fault-based software testing technique to measure the quality of a test suite depending on its ability to detect faults in the code. This technique has been applied to an assortment of languages of very diverse nature since its inception in the late 1970s. However, the researchers have postponed its development around C++ in favor of other mainstream languages. This paper aims to survey the mutation testing research regarding C++, studying the existing tools and approaches. To the same extent, we discuss the different aspects that should be taken into account in the construction of a comprehensive mutation tool for this language, from the analysis of the code to the execution of the mutants. In addition, we expound how the technique can be assessed so that it can contribute effectively in the composition of a complete test suite. The findings in this paper pose that the construction of a mutation tool for this language is complex, but still realizable.

V JORPRESI

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

Definición e Implementación de Operadores de Mutación a Nivel de Clase para el Lenguaje de Programación C++

En las V Jornadas Predoctorales de la ESI, 20–21 mayo 2014, Cádiz, España.

Resumen: La prueba de mutación es una técnica de prueba de software alrededor de la cual se han desarrollado diversas herramientas para varios lenguajes. Sin embargo, no existe ningún marco de trabajo para el lenguaje C++ en el que se traten operadores relativos a estructuras más complejas como las de la orientación a objetos. En este trabajo se presentan los avances en el trabajo de aplicación de la prueba de mutaciones a este lenguaje, desde el conjunto definido de operadores de clase hasta el sistema utilizado para la implementación de los mismos, lo cual permite la inserción de los errores que modelan en el código.

PROLE 2013

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José-Domínguez Jiménez y Antonio García-Domínguez.

Operadores de Mutación a Nivel de Clase para el Lenguaje C++

XII Jornadas sobre Programación y Lenguajes (PROLE 2013), 18–20 septiembre 2013, Madrid, España.

Resumen: La prueba de mutaciones es una técnica basada en fallos alrededor de la cual existe una gran variedad de estudios de investigación y se han elaborado herramientas para un amplio abanico de lenguajes de programación. Sin embargo, el desarrollo respecto a C++, uno de los lenguajes orientados a objetos más populares y usados, es escaso. Este trabajo aborda esta cuestión presentando un conjunto de operadores de mutación de acuerdo a las propiedades de la orientación a objetos (a nivel de clase) y a las características propias del lenguaje C++. También se ofrece una primera visión general del uso de esos operadores para analizar programas en este lenguaje, a partir de un experimento reducido con los operadores.

IV JORPRESI

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

Aplicación de la Técnica de Prueba de Mutación Evolutiva a C++

En las IV Jornadas Predoctorales de la ESI, 10–12 diciembre 2012, Cádiz, España.

Resumen: La prueba de mutación es una técnica de prueba basada en fallos de la que existen gran diversidad de estudios y aplicaciones para un amplio abanico de lenguajes. Sin embargo, el desarrollo con respecto a C++, uno de los lenguajes orientados a objetos más populares, es prácticamente inexistente. En este trabajo se presentan un conjunto de operadores de mutación asociados a este lenguaje y se propone la creación de un nuevo framework que nos permita aplicar esta técnica con el lenguaje C++.

Appendix A

Case Studies

In this appendix, we describe the programs and libraries used in the experiments throughout this thesis. We also provide several metrics about the programs which are relevant for the experiments in Chapters 5, 6 and 7.

A.1 Description

Three programs were chosen from the *LLVM 3.2 test-suite* [85], containing pieces of code written in C/C++ (most of them taken from examples in books). These programs are appropriate to observe exemplary mutations in Section 5.2.1. Namely, the programs *garage*, *family* and *simul* from *"MultiSource/Benchmarks/Prolangs-C++"* were used:

- *Garage,* with a class modelling a parking where two kinds of vehicles (represented by two subclasses) are parked and released.
- *Family,* with three classes simulating the hierarchy *grandfather-father-son*, sharing some attributes.
- *Simul*, modelling the function of a cursor on a screen.

Other seven known open-source programs and libraries were used to apply the set of operators to real applications. They are listed below, showing between parentheses the abbreviation used to refer to these programs in the tables in this thesis:

- *Matrix TCL Pro* (*Tcl*) [94]: library for performing matrix algebra calculations in C++ programs.

- *XmlRpc++* (*Rpc*) [130]: library implementing the XML-RPC protocol to incorporate client-server communication through HTTP support into other C++ programs.
- *Dolphin* (*Dph*) [41]: default navigational file manager used by desktop applications in KDE.
- *Tinyxml2* (*Txm*) [123]: lightweight and efficient XML parser that can be integrated into C++ applications.
- *KMyMoney* (*Kmy*) [79]: KDE desktop application for personal finance management.
- *QtDom* (*Dom*) [113]: Qt module that provides a C++ implementation of the DOM standard.
- *KatePart* (*Kap*) [74]: 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)[1] and *ROS* (Robot Operating System)[2]. In the case of *Tinyxml2*, we can mention *mFAST*[3], an efficient implementation of the FAST protocol for the communication of high-volume data market between financial institutions with low latency. *KatePart* is part of various popular KDE applications, such as the text editors *Kate* and *Kwrite*, the browser *Konqueror* or the IDE *KDevelop*.

Apart from the widespread use of these programs, we selected them because they were accompanied by non-trivial test suites. In these experiments, we seek to estimate the fault detection ability of these test suites with regard to object-oriented features. Furthermore, adequate test suites are required to calculate the metrics in Chapter 7, so starting from the non-trivial test suites distributed with the aforementioned programs allows us to manually extend it with new test cases based on surviving mutants.

[1] http://siremis.asipto.com
[2] http://www.ros.org
[3] http://sett.ociweb.com/sett/settOct2013.html

TABLE A.1: Metrics about the programs used in the experiments in Chapter 5

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 A.2: Number of classes in the analysed programs 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

A.2 Features

Experiments in Chapter 5

Different characteristics and measurements of the real programs used in the experiments conducted in Chapter 5 are collected in Table A.1, providing an overall picture of their complexity. We also include the time that the original programs spend executing their test suites. Table A.2 complements the information for the quantitative analysis by classifying the classes of these programs into four ranges according to the lines of code. For each program, the number of classes belonging to each range (C) and the overall percentage ($C\%$) are shown. The last column presents the total percentage of classes within each range.

Experiments in Chapter 6

Table A.3 shows different metrics related to the experiments in Chapter 6, divided by:

- *Distribution of mutants*: total, valid and strong mutants (the percentage of strong mutants with respect to the set of valid mutants is also shown).

- *Size of the test suites*: size of the original test suite, adequate test suite after adding new test cases (between parentheses, the number of test cases additionally modified) and minimal test suite.

We can remark from this table that *MuCPP* generates a different percentage of strong mutants for these applications with the test suite distributed with them.

TABLE A.3: Metrics about the programs used in the experiments in Chapter 6

		TCL	**DPH**	**TXM**	**DOM**
Mutants	Total	137	219	614	1,146
	Valid	135	208	433	681
	Strong	45	103	159	348
	% Strong mutants	33.3%	49.5%	36.7%	51.1%
Test suite	$\|Original\ T\|$	17	61	57	46
	$\|Adequate\ T\|$	24(3)	70(5)	62(3)	56(4)
	$\|Minimal\ T\|$	15	22	15	25

Experiments in Chapter 7

Table A.4 depicts several metrics about the case studies used in the experiments in Chapter 7:

- *Features*: number of classes, lines of code and mean of methods in the classes.
- *Mutants*: total of mutants, percentage of equivalent mutants and percentage of *undecided mutants* [117]. We classify as undecided those mutants for which we are unable to ascertain the condition of equivalence with high confidence. We use this term to avoid skewing of results when computing the metrics.
- *Size of the test suites*: size of the original test suite, adequate test suite after adding new test cases (between parentheses, the number of test cases additionally modified) and minimal test suite.

Table A.5 shows a breakdown of the total number of mutants and their classification into dead (D) and equivalent (E), divided by case study and mutation operator. The number of undecided mutants corresponds to the cases where the sum of dead and equivalent mutants is not equal to the number of mutants (M) in *Total*.

Notice that:

- For the sake of simplicity, we have duplicated the data from Table A.1 for some of the programs.

TABLE A.4: Features of the case studies used in the experiments in Chapter 7

		TCL	RPC	DPH	TXM	KMY	DOM
Features	Classes	9	13	13	20	17	11
	Lines of code	3,228	2,194	3,667	2,620	13,709	2,117
	Methods (mean)	21.1	11.2	16.4	15.6	35.6	23.6
Mutants	Valid	135	127	208	433	284	681
	% Equivalent	14.8	31.5	33.2	21.0	30.6	34.7
	% Undecided	0	0	4.8	7.4	1.4	1.5
Test suite	$\|Original\ T\|$	17	26	61	57	241	46
	$\|Adequate\ T\|$	24(3)	34(5)	70(5)	62(3)	248(10)	56(4)
	$\|Minimal\ T\|$	15	15	22	15	36	25

TABLE A.5: Mutants generated in each case study by operator (M: mutants; D: dead; E: equivalent)

Op.	**TCL**		**RPC**		**DPH**		**TXM**		**KMY**		**DOM**		**Total**		
	D	E	D	E	D	E	D	E	D	E	D	E	**M**	**D**	**E**
IHD	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0
IHI	0	0	2	2	0	0	41	6	8	15	21	25	120	72	48
ISD	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
ISI	0	0	2	1	9	2	0	0	0	3	1	1	19	12	7
IOD	0	0	1	2	15	3	24	1	1	0	28	2	79	69	8
IOP	0	0	0	0	0	0	8	0	0	0	0	2	10	8	2
IOR	0	0	0	15	3	27	10	1	0	0	0	1	57	13	44
IPC	0	0	1	0	2	3	0	0	12	6	8	0	32	23	9
PCI	0	0	2	1	0	0	138	20	14	1	293	155	659	447	177
PMD	0	0	0	0	0	0	0	3	0	1	0	4	8	0	8
PPD	0	0	0	1	0	0	5	2	4	14	2	12	42	11	29
PNC	0	0	0	0	0	0	0	0	0	0	2	0	2	2	0
OMD	38	8	9	1	2	1	23	14	9	4	16	6	131	97	34
OMR	33	1	10	0	5	1	0	0	32	0	16	0	98	96	2
OAN	0	0	0	0	0	0	0	0	3	4	0	0	7	3	4
MCO	3	0	38	10	68	7	18	1	76	7	36	7	285	239	32
MCI	0	0	0	0	0	0	13	26	0	0	0	0	39	13	26
EHC	0	0	1	1	0	0	0	0	1	5	0	0	8	2	6
CTD	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0
CTI	0	0	0	0	2	0	0	0	0	0	1	0	3	3	0
CID	38	2	14	3	23	18	24	10	26	22	6	9	196	131	64
CDC	0	0	2	0	0	0	3	0	4	1	0	0	10	9	1
CDD	0	2	2	3	0	2	3	3	2	2	0	4	23	7	16
CCA	3	7	2	0	0	5	0	4	0	2	4	8	37	9	26
Total	115	20	87	40	129	69	310	91	193	87	435	236	1,868	1,269	543

- The metrics for *KMyMoney* in Table A.1 and Table A.4 differ because the latter refers to a subset of the source files of this program.
- A mutant generated by *PPD* in *Tinyxml2* was classified into the set of equivalent mutants in the experiments in Section 5.1.3 (see Table 5.8), but it was found killable in a second review of the mutants. The percentage of equivalence in that table also differs from the percentage of equivalence shown in Table A.4 because some of the mutants that were classified as equivalent were later classified as undecided.

Appendix B

Useful Concepts

In this appendix, we group together several concepts that are used in different chapters of this thesis. Namely, we describe and explain several terms related to the execution of mutants and the properties of test suites.

B.1 Execution Matrix

An *execution matrix* contains the whole information about the execution of the mutants on a test suite. The rows in the execution matrix represent the mutants and the columns represent the test cases. Let M the set of mutants and T the set of test cases. The execution matrix with size $|M| \times |T|$ stores the result of running each test case against each mutant. That result depends on the behaviour of the mutant when compared with the original program. A mutant x killed by a test case y is represented with the value 1 in the intersection of the row x and the column y. On the contrary, the value 0 denotes that the mutation is not revealed by that test case.

Execution matrices are used throughout this thesis to (among others):

- Classify mutants according to the values in the matrix.
- Calculate the used metrics.
- Illustrate examples.
- Compute the fitness function of the mutants when applying EMT.

A mutant, represented by a row in the execution matrix, is said to be:

	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$
m_1	1	0	0	0	0
m_2	0	0	1	0	0
m_3	0	0	1	0	0
m_4	0	0	0	1	0
m_5	0	0	0	0	1
m_6	1	0	0	1	1
m_7	1	0	1	0	1
m_8	0	0	1	1	1
m_9	0	1	0	0	0
m_{10}	0	0	0	1	0

FIGURE B.1: Example of matrix execution with size 10×5

- **Alive** when the row is filled with the value 0.
- **Dead** when there is at least one entry with the value 1 in the row.

Invalid mutants are also represented in the execution matrix with rows filled with the value 2. Furthermore, Estero-Botaro et al. [44] defined several other terms to classify mutants (see Section 2.4.4):

- A **weak mutant**[1] is killed by every test case in the test suite. It can be identified as a row filled with the value 1.
- A **resistant mutant** is killed by a single test case, and it is identified as a row filled with the value 0 except for one entry with the value 1. In Figure B.1, the mutant 1 (m_1) is a resistant mutant.
- A **resistant hard to kill mutant** is killed by a single test case which only kills that mutant. It is identified as a row with a single entry y with the value 1, where the rest of the entries in the column y are filled with the value 0. In Figure B.1, m_1 is resistant but not resistant hard to kill because $test_1$, which kills that mutant, also kills the mutants m_6 and m_7. The mutant 9 does represent a resistant hard to kill mutant.

Note that:

- We cannot find a weak and a resistant hard to kill mutant simultaneously in the same execution matrix.

[1] Please, do not confuse weak mutants with *weak mutation* (see Section 2.4.2).

- We also use the term *weak* in Chapter 6 to refer to those mutants which are not *strong*.
- The concept *difficult to kill mutant* used in Chapter 6 is equivalent to the concept *resistant hard to kill mutant*.
- In Chapter 4, we use the term *trivial mutant* to indicate that every test case covering its mutation will kill the mutant. In contrast, a *weak mutant* is killed by all the test cases of a particular test suite, which does not necessarily imply that there might exist other test cases that do not kill the mutant.

B.2 Properties of a Test Suite

We can ascribe different properties to a test suite when analysing the execution with respect to a set of mutants. To that end, the execution matrix can be useful to ascertain these properties:

- **Non-adequate test suite**: when it does not detect the full set of non-equivalent mutants, that is, there are non-equivalent mutants that remain alive when executed on the test suite.
- **Adequate test suite**: when it detects all non-equivalent mutants. In other words, the mutation adequacy score associated with an adequate test suite is 100%.
- **Non-redundant test suite**: when none of the test cases in an adequate test suite can be removed without losing the adequacy of the test suite (there are no redundant test cases).
- **Minimal test suite**: when a non-redundant test suite is of the minimum size, that is, there are no other non-redundant test suites of smaller size.

The test suite in Figure B.1 is adequate and non-redundant, as we cannot discard any of the test cases maintaining the same mutation score. It is also a minimal test suite, as we cannot find a subset of these test cases that kills all those mutants.

We have to note that our concepts of non-redundant and minimal test suite are called minimal and minimum test suite respectively by Amman et al. [6]. Therefore, in our work we focus on minimal test suites, which are called minimum test suites by the aforementioned authors.

In the experiments in this thesis, we use the random adequate and minimal test suite generated by the exact algorithm that Estero-Botaro et al. [45] used in their study. Any

metric is dependent on the test suite. Thus, we make use of minimal test suites because that property prevents the results from being distorted by unproductive test cases, as pointed by Estero-Botaro et al. [44]. This allows us to properly assess the different metrics used in this thesis.

Bibliography

[1] A. T. Acree, Jr. *On Mutation*. PhD thesis, Atlanta, GA, USA, 1980.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO 2004: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1338–1349, 2004. doi: 10.1007/978-3-540-24855-2_155. URL http://dx.doi.org/10.1007/978-3-540-24855-2_155.

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

[4] Z. Ahmed, M. Zahoor, and I. Younas. Mutation operators for object-oriented systems: A survey. In *The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 2, pages 614–618, feb. 2010. doi: 10.1109/ICCAE.2010.5451692. URL http://dx.doi.org/10.1109/ICCAE.2010.5451692.

[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1st edition, 2008. ISBN 0521880386, 9780521880381.

[6] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-2255-0. doi: 10.1109/ICST.2014.13. URL http://dx.doi.org/10.1109/ICST.2014.13.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. URL http://dx.doi.org/10.1145/1062455.1062530.

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions*

on Software Engineering, 32(8):608–624, Aug 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.83. URL http://dx.doi.org/10.1109/TSE.2006.83.

[9] A. Arcuri and L. Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, May 2014. ISSN 0960-0833. doi: 10.1002/stvr.1486. URL http://dx.doi.org/10.1002/stvr.1486.

[10] R. Baker and I. Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, June 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.56. URL http://dx.doi.org/10.1109/TSE.2012.56.

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

[12] A. S. Banzi, T. Nobre, G. B. Pinheiro, J. C. G. Árias, A. Pozo, and S. R. Vergilio. Selecting mutation operators with a multiobjective approach. *Expert Systems with Applications*, 39(15):12131–12142, 2012. ISSN 0957-4174. doi: 10.1016/j.eswa.2012.04.041. URL http://dx.doi.org/10.1016/j.eswa.2012.04.041.

[13] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001. ISSN 1099-1689. doi: 10.1002/stvr.226. URL http://dx.doi.org/10.1002/stvr.226.

[14] M. B. Bashir and A. Nadeem. Object oriented mutation testing: A survey. In *International Conference on Emerging Technologies (ICET), 2012*, pages 1–6, Oct 2012. doi: 10.1109/ICET.2012.6375480. URL http://dx.doi.org/10.1109/ICET.2012.6375480.

[15] M. B. Bashir and A. Nadeem. A fitness function for evolutionary mutation testing of object-oriented programs. In *Emerging Technologies (ICET), 2013 IEEE 9th International Conference on*, pages 1–6, Dec 2013. doi: 10.1109/ICET.2013.6743531. URL http://dx.doi.org/10.1109/ICET.2013.6743531.

[16] E. Blanco-Muñoz, A. García-Domínguez, J. J. Domínguez-Jiménez, and I. Medina-Bulo. Towards higher-order mutant generation for WS-BPEL. In *Proceedings of the International Conference on e-Business (ICE-B), 2011*, pages 1–6. IEEE, 2011.

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

[18] J. Boubeta-Puig, A. García-Domínguez, and I. Medina-Bulo. Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 398–407, Berlin, Germany, 2011. IEEE. ISBN 978-0-7695-4345-1. doi: 10.1109/ICSTW.2011.52. URL http://dx.doi.org/10.1109/ICSTW.2011.52. Print ISBN: 978-1-4577-0019-4.

[19] P. Bourque and e. R.E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014. URL http://www.swebok.org.

[20] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[21] P. Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 267–270, Dec 2001. doi: 10.1109/APSEC.2001.991487.

[22] Clang. Clang: a C language family frontend for LLVM. URL http://clang.llvm.org. Last access: 2016.10.14.

[23] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *SIGSOFT Software Engineering Notes*, 21(3): 158–171, May 1996. ISSN 0163-5948. doi: 10.1145/226295.226313. URL http://dx.doi.org/10.1145/226295.226313.

[24] A. A. L. de Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi. A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In *IEEE Congress on Evolutionary Computation, 2013*, pages 829–836, June 2013. doi: 10.1109/CEC.2013.6557654. URL http://dx.doi.org/10.1109/CEC.2013.6557654.

[25] M. Delahaye and L. du Bousquet. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience*, 45(7):875–891, 2015. ISSN 1097-024X. doi: 10.1002/spe.2312. URL http://dx.doi.org/10.1002/spe.2312.

[26] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27

(3):228–247, Mar. 2001. ISSN 0098-5589. doi: 10.1109/32.910859. URL http://dx.doi.org/10.1109/32.910859.

[27] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. *Proteum/IM 2.0: An Integrated Mutation Testing Environment*, pages 91–101. Springer US, Boston, MA, 2001. ISBN 978-1-4757-5939-6. doi: 10.1007/978-1-4757-5939-6_17. URL http://dx.doi.org/10.1007/978-1-4757-5939-6_17.

[28] M. E. Delamaro, L. Deng, N. Li, V. H. S. Durelli, and A. J. Offutt. Growing a reduced set of mutation operators. In *Brazilian Symposium on Software Engineering (SBES), 2014*, pages 81–90, Sept 2014. doi: 10.1109/SBES.2014.14. URL http://dx.doi.org/10.1109/SBES.2014.14.

[29] M. E. Delamaro, J. Offutt, and P. Ammann. Designing deletion mutation operators. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 11–20, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-2255-0. doi: 10.1109/ICST.2014.12. URL http://dx.doi.org/10.1109/ICST.2014.12.

[30] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978. ISSN 0018-9162.

[31] R. DeMillo, E. Krauser, and A. Mathur. Compiler-integrated program mutation. In *Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International*, pages 351–356, Sep 1991. doi: 10.1109/CMPSAC.1991.170202. URL http://dx.doi.org/10.1109/CMPSAC.1991.170202.

[32] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 84–93, March 2013. doi: 10.1109/ICST.2013.20. URL http://dx.doi.org/10.1109/ICST.2013.20.

[33] A. Derezińska. Object-oriented mutation to assess the quality of tests. In *EUROMICRO Conference, 2003. Proceedings. 29th*, pages 417–420, Belek, Turkey, 2003. IEEE Computer Society. ISBN 0-7695-1996-2. doi: 10.1109/EURMIC.2003.1231626. URL http://dx.doi.org/10.1109/EURMIC.2003.1231626.

[34] A. Derezińska. Quality assessment of mutation operators dedicated for C# programs. In *Proceedings of VI International Conference on Quality Software*, pages 227–234, Beijing (China), Oct. 2006. IEEE Computer Society. ISBN 0-7695-2718-3. doi: 10.1109/QSIC.2006.51. URL http://dx.doi.org/10.1109/QSIC.2006.51. ISSN 1550-6002.

[35] A. Derezińska. *Advanced mutation operators applicable in C# programs*, pages 283–288. Springer US, Boston, MA, 2007. ISBN 978-0-387-39388-9. doi: 10.1007/978-0-387-39388-9_27. URL http://dx.doi.org/10.1007/978-0-387-39388-9_27.

[36] A. Derezińska and K. Halas. Experimental evaluation of mutation testing approaches to Python programs. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 156–164, March 2014. doi: 10.1109/ICSTW.2014.24. URL http://dx.doi.org/10.1109/ICSTW.2014.24.

[37] A. Derezińska and K. Kowalski. Object-oriented mutation applied in common intermediate language programs originated from C#. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011*, pages 342–350, 2011. doi: 10.1109/ICSTW.2011.54. URL http://dx.doi.org/10.1109/ICSTW.2011.54.

[38] A. Derezińska and M. Rudnik. Quality evaluation of object-oriented and standard mutation operators applied to C# programs. In *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_5. URL http://dx.doi.org/10.1007/978-3-642-30561-0_5.

[39] A. Derezińska and A. Szustek. CREAM - A system for object-oriented mutation of C# programs. *Annals Gdansk University of Technology Faculty of ETI*, (5): 389–406, 2007.

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

[41] Dolphin. Dolphin. https://www.kde.org/applications/system/dolphin. Last access: 2016.10.14.

[42] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. In *Proceedings of the 7th IEEE European Conference on Web Services*, pages 97–106, Eindhoven, The Netherlands, Nov. 2009. IEEE Computer Society Press. ISBN 978-0-7695-3854-9. doi: 10.1109/ECOWS.2009.18. URL http://dx.doi.org/10.1109/ECOWS.2009.18.

[43] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53 (10):1108–1123, Oct. 2011. ISSN 0950-5849. doi: 10.1016/j.infsof.2011.03.008. URL http://dx.doi.org/10.1016/j.infsof.2011.03.008.

[44] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Quantitative evaluation of mutation operators for WS-BPEL compositions. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010*, pages 142–150, 2010. doi: 10.1109/ICSTW.2010.36. URL http://dx.doi.org/10.1109/ICSTW.2010.36.

[45] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, and A. García-Domínguez. Quality metrics for mutation testing with applications to WS-BPEL compositions. *Software Testing, Verification and Reliability*, 25(5-7): 536–571, 2015. ISSN 1099-1689. doi: 10.1002/stvr.1528. URL http://dx.doi.org/10.1002/stvr.1528.

[46] S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering, 1994*, pages 220–229, Nov 1994. doi: 10.1109/ISSRE.1994.341378. URL http://dx.doi.org/10.1109/ISSRE.1994.341378.

[47] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and 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*, pages 329–337, London, UK, 1996. Chapman & Hall, Ltd. ISBN 0-412-73270-X. doi: 10.1007/978-0-387-34945-9_24. URL http://dx.doi.org/10.1007/978-0-387-34945-9_24.

[48] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL http://dx.doi.org/10.1145/2025113.2025179.

[49] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 313–319, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0620-7.

[50] Git. Git version control system. URL http://git-scm.com. Last access: 2016.10.14.

[51] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

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

[53] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. On the limits of mutation reduction strategies. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 511–522, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884787. URL http://dx.doi.org/10.1145/2884781.2884787.

[54] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977. ISSN 0098-5589.

[55] M. Hampton and S. Petithomme. Leveraging a commercial mutation analysis tool for research. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 203 –209, Sept. 2007. doi: 10.1109/TAIC.PART.2007.39. URL http://dx.doi.org/10.1109/TAIC.PART.2007.39.

[56] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 397–408, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2643008. URL http://dx.doi.org/10.1145/2642937.2643008.

[57] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999. ISSN 1099-1689. doi: 10.1002/(SICI)1099-1689(199912)9:4<233::AID-STVR191>3.0.CO;2-3. URL http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<233::AID-STVR191>3.0.CO;2-3.

[58] C. Horstmann and T. Budd. *Big C++, 2nd Edition*. Wiley, 2009. ISBN 9780470383285.

[59] J. Hu, N. Li, and J. Offutt. An analysis of OO mutation operators. In *IEEE Fourth International Conference on Software Testing, Verification and Validation*

Workshops (ICSTW), 2011, pages 334–341, March 2011. doi: 10.1109/ICSTW.2011.47. URL http://dx.doi.org/10.1109/ICSTW.2011.47.

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

[61] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL http://dx.doi.org/10.1145/2568225.2568271.

[62] Insure++. Insure++: C/C++ testing tool, detect elusive runtime memory errors - Parasoft. URL http://www.parasoft.com/insure. Last access: 2016.10.14.

[63] O. A. J. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011. ISSN 0950-5849. doi: 10.1016/j.infsof.2011.03.007. URL http://dx.doi.org/10.1016/j.infsof.2011.03.007. Special Section on Mutation Testing.

[64] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008*, pages 249–258, Sept 2008. doi: 10.1109/SCAM.2008.36. URL http://dx.doi.org/10.1109/SCAM.2008.36.

[65] Y. Jia and 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*, pages 94–98, Aug. 2008. doi: 10.1109/TAIC-PART.2008.18. URL http://dx.doi.org/10.1109/TAIC-PART.2008.18.

[66] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Tech. report TR-09-06, CREST Centre, King's College London, London, UK, September 2009.

[67] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, Oct. 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.04.016. URL http://dx.doi.org/10.1016/j.infsof.2009.04.016.

[68] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Oct. 2011. ISSN 0098-5589. URL http://dx.doi.org/10.1109/TSE.2010.62.

[69] JSON. JSON compilation database format specification. URL http://clang.llvm.org/docs/JSONCompilationDatabase.html. Last access: 2016.10.14.

[70] R. Just and F. Schweiggert. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, 25(5-7):490–507, Aug. 2015. ISSN 0960-0833. doi: 10.1002/stvr.1561. URL http://dx.doi.org/10.1002/stvr.1561.

[71] R. Just, F. Schweiggert, and 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*, pages 612–615, Nov 2011. doi: 10.1109/ASE.2011.6100138. URL http://dx.doi.org/10.1109/ASE.2011.6100138.

[72] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and 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, pages 654–665, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. URL http://dx.doi.org/10.1145/2635868.2635929.

[73] G. Kaminski, P. Ammann, and J. Offutt. Improving logic-based testing. *Journal of Systems and Software*, 86(8):2002–2012, Aug. 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2012.08.024. URL http://dx.doi.org/10.1016/j.jss.2012.08.024.

[74] KatePart. KatePart. https://kate-editor.org/about-katepart. Last access: 2016.10.14.

[75] S. Kim, J. A. Clark, and 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, 29 Nov-1 Dec 1999.

[76] S. Kim, J. Clark, and J. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net.ObjectDays*, pages 9–12, 2000.

[77] S.-W. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability*, 11(4):207–225, 2001. ISSN 1099-1689. doi: 10.1002/stvr.238. URL http://dx.doi.org/10.1002/stvr.238.

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

[79] KMyMoney. KMyMoney, version 4.6.4. https://sourceforge.net/projects/kmymoney2/. Last access: 2016.10.14.

[80] M. Kusano and 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*, pages 722–725. IEEE, 2013. URL http://dx.doi.org/10.1109/ASE.2013.6693142.

[81] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83 (12):2416 – 2430, 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2010.07.027. URL http://dx.doi.org/10.1016/j.jss.2010.07.027. TAIC PART 2009 - Testing: Academic and Industrial Conference - Practice And Research Techniques.

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

[83] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. Silva, H. L. Jakubovski Filho, and H. V. Ehrenfried. Evaluating different strategies for reduction of mutation testing costs. 2016.

[84] LLVM. The LLVM compiler infrastructure. URL http://llvm.org. Last access: 2016.10.14.

[85] LLVM test-suite. LLVM 3.2 test-suite. http://llvm.org/releases/3.2/docs/TestingGuide.html. Last access: 2016.10.14.

[86] Y.-S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of XIII International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis (Maryland), Nov. 2002. IEEE Computer Society. ISBN 0-8186-1763-3. doi: 10.1109/ISSRE.2002.1173287. URL http://dx.doi.org/10.1109/ISSRE.2002.1173287.

[87] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system: Research articles. *Software Testing, Verification and Reliability*, 15(2): 97–133, June 2005. ISSN 0960-0833. doi: 10.1002/stvr.v15:2. URL http://dx.doi.org/10.1002/stvr.v15:2.

[88] Y.-S. Ma, Y. R. Kwon, and S.-W. Kim. Statistical investigation on class mutation operators. *ETRI Journal*, 31(2):140–150, Apr. 2009. ISSN 1225-6463. doi: 10.4218/etrij.09.0108.0356. URL http://dx.doi.org/10.4218/etrij.09.0108.0356.

[89] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW

'07, pages 667–676, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242663. URL http://dx.doi.org/10.1145/1242572.1242663.

[90] Matchers. AST matcher reference. URL http://clang.llvm.org/docs/LibASTMatchersReference. Last access: 2016.10.14.

[91] P. R. Mateo, M. P. Usaola, and J. Offutt. Mutation at the multi-class and system levels. *Science of Computer Programming*, 78(4):364–387, 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2012.02.005. URL http://dx.doi.org/10.1016/j.scico.2012.02.005. Special section on Mutation Testing and Analysis (Mutation 2010) and Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing.

[92] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference, 1991. COMPSAC '91.*, pages 604–605, Sep 1991. doi: 10.1109/CMPSAC.1991.170248. URL http://dx.doi.org/10.1109/CMPSAC.1991.170248.

[93] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Tech. report SERC-TR-14-P, Purdue University, West Lafayette, Indiana, 1988.

[94] Matrix TCL Pro. Matrix TCL Pro, version 2.2. http://www.techsoftpl.com/matrix/download.php. Last access: 2016.10.14.

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

[96] S. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley-Spektrum, 2008.

[97] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, pages 351–360, May 2008. doi: 10.1145/1368088.1368136. URL http://dx.doi.org/10.1145/1368088.1368136.

[98] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, Jan. 1992. ISSN 1049-331X. doi: 10.1145/125489.125473. URL http://dx.doi.org/10.1145/125489.125473.

[99] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *Proceedings of the Eleventh Annual Conference on Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security*, pages 224 –236, jun 1996. doi: 10.1109/CMPASS.1996.507890. URL http://dx.doi.org/10.1109/CMPASS.1996.507890.

[100] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001. ISBN 978-1-4757-5939-6. doi: 10.1007/978-1-4757-5939-6_7. URL http://dx.doi.org/10.1007/978-1-4757-5939-6_7.

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

[102] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, Apr. 1996. ISSN 1049-331X. doi: 10.1145/227607.227610. URL http://dx.doi.org/10.1145/227607.227610.

[103] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, George Mason University, Fairfax, Virginia, 1996. Information and Software Systems Engineering, George Mason University.

[104] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of MuJava. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 78–84, New York, NY, USA, 2006. ACM. ISBN 1-59593-408-1. doi: 10.1145/1138929.1138945. URL http://dx.doi.org/10.1145/1138929.1138945.

[105] E. Omar, S. Ghosh, and D. Whitley. Homaj: A tool for higher order mutation testing in AspectJ and Java. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 165–170, March 2014. doi: 10.1109/ICSTW.2014.19. URL http://dx.doi.org/10.1109/ICSTW.2014.19.

[106] M. Papadakis and N. Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal*, 19(4):691–723, 2011. ISSN 1573-1367. doi: 10.1007/s11219-011-9142-y. URL http://dx.doi.org/10.1007/s11219-011-9142-y.

[107] M. Papadakis, M. Delamaro, and Y. Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*,

95(P3):298–319, Dec. 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2014.05.012. URL http://dx.doi.org/10.1016/j.scico.2014.05.012.

[108] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 936–946, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. doi: 10.1109/ICSE.2015.103. URL http://dx.doi.org/10.1109/ICSE.2015.103.

[109] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931040. URL http://dx.doi.org/10.1145/2931037.2931040.

[110] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999. ISSN 1099-1689. doi: 10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y. URL http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y.

[111] PITest. PITest mutators overview. URL http://pitest.org/quickstart/mutators/. Last access: 2016.10.14.

[112] PlexTest. PlexTest ITRegister. URL http://www.itregister.com.au/products/plextest. Last access: 2016.10.14.

[113] QtDom. QtDOM. https://github.com/qtproject/qtbase/tree/dev/src/xml/dom. Last access: 2016.10.14.

[114] D. Schuler and 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, pages 297–298, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595750. URL http://dx.doi.org/10.1145/1595696.1595750.

[115] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374, 2013. ISSN 1099-1689. doi: 10.1002/stvr.1473. URL http://dx.doi.org/10.1002/stvr.1473.

[116] B. Schwarz, D. Schuler, and A. Zeller. Breeding high-impact mutations. In *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pages 382–387, 2011. doi: 10.1109/ICSTW.2011.56. URL http://dx.doi.org/10.1109/ICSTW.2011.56.

[117] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology*, 53(10):1124–1136, 2011. ISSN 0950-5849. URL http://dx.doi.org/10.1016/j.infsof.2011.03.006. Special Section on Mutation Testing.

[118] R. A. Silva, S. do Rocio Senger de Souza, and P. S. L. de Souza. A systematic review on search based mutation testing. *Information and Software Technology*, 2016. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.01.017. URL http://dx.doi.org/10.1016/j.infsof.2016.01.017.

[119] B. H. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, June 2009. ISSN 1382-3256. doi: 10.1007/s10664-008-9083-7. URL http://dx.doi.org/10.1007/s10664-008-9083-7.

[120] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1245303. URL http://dx.doi.org/10.1109/TSE.2003.1245303.

[121] STATService. STATService. http://moses.us.es/statservice. [Last access: 2016.10.14].

[122] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201889544.

[123] Tinyxml2. Tinyxml2. https://github.com/leethomason/tinyxml2. Last access: 2016.10.14.

[124] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, pages 139–148, New York, NY, USA, 1993. ACM. ISBN 0-89791-608-5. doi: 10.1145/154183.154265. URL http://doi.acm.org/10.1145/154183.154265.

[125] M. Usaola and P. Mateo. Mutation testing cost reduction techniques: A survey. *Software, IEEE*, 27(3):80–86, 2010. ISSN 0740-7459. doi: 10.1109/MS.2010.79. URL http://dx.doi.org/10.1109/MS.2010.79.

[126] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. Tech. report, Purdue University, West Lafayette, Indiana, 1993.

[127] M. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988*, pages 152–158, 1988. doi: 10.1109/WST.1988.5370. URL http://dx.doi.org/10.1109/WST.1988.5370.

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

[129] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and Applications*, pages 625–636, 1992.

[130] XmlRpc++. XmlRpc++, version 0.7. http://xmlrpcpp.sourceforge.net/. Last access: 2016.10.14.

[131] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 919–930, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568265. URL http://dx.doi.org/10.1145/2568225.2568265.

[132] C. N. Zapf. *A Distributed Interpreter for the Mothra Mutation Testing System*. Phd thesis, Clemson University, Clemson, South Carolina, 1993.

[133] H. Zhang. Mutation operators for C++. URL http://people.cis.ksu.edu/~hzh8888/mse_project/. Last access: 2016.10.14.

[134] J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), 2014*, pages 277–287, Nov 2014. doi: 10.1109/ISSRE.2014.27. URL http://dx.doi.org/10.1109/ISSRE.2014.27.

[135] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806863. URL http://dx.doi.org/10.1145/1806799.1806863.

[136] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013*, pages 92–102, Nov 2013. doi: 10.1109/ASE.2013.6693070. URL http://dx.doi.org/10.1109/ASE.2013.6693070.