Pós-Graduação em Ciência da Computação

Leonardo Fernandes Mendonça de Oliveira

Tackling The Useless Mutants Problem

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2020

Leonardo Fernandes Mendonça de Oliveira

Tackling The Useless Mutants Problem

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Dr. André Luís de Medeiros Santos
Co-orientador: Dr. Márcio de Medeiros Ribeiro

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O48t Oliveira, Leonardo Fernandes Mendonça de
Tackling the useless mutants problem / Leonardo Fernandes Mendonça de Oliveira. – 2020.
115 f.: il., fig., tab.

Orientador: André Luís de Medeiros Santos.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de software. 2. Teste de software. I. Santos, André Luís de Medeiros (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 -86

Leonardo Fernandes Mendonça de Oliveira

"**Tackling The Useless Mutants Problem**"

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 02/03/2020.

Orientador: Dr. André Luís de Medeiros Santos

BANCA EXAMINADORA

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática/UFPE

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática/UFPE

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática/UFPE

Prof. Dr. Rohit Gheyi
Departamento de Sistemas e Computação/UFCG

Prof. Dr. Fabiano Cutigi Ferrari
Departamento de Computação/UFSCAR

ACKNOWLEDGEMENTS

Este trabalho é uma construção coletiva. Primeiramente agradeço a Deus pela minha vida e pela oportunidade de viver em uma família feliz e que me deu todo o apoio necessário para chegar ao final de mais uma etapa em minha vida. Agradeço aos meus pais, Aderson Augusto e Simone Fernandes, por sempre terem cuidado de mim e dado apoio em todos os momentos que precisei. Em especial para minha mãe, por toda a dedicação em buscar sempre o melhor para mim e para meus irmãos, por estar sempre ao meu lado em todas as decisões. Agradeço também aos meus irmãos Carlos e Pedro. Eu amo todos vocês. Agradeço a minha esposa Luiza Fernandes, a pessoa que Deus colocou na minha vida para iluminar e compartilhar comigo todos os momentos, que soube compreender minhas angustias, minhas dificuldades e também minhas conquistas. Você é uma pessoa muito especial em minha vida e meu coração sempre esteve e sempre estará com você. A todos os meus familiares: Klebel, Maria, meus amigos (que são a família que eu escolhi), tios, sogros pelo companheirismo e amizade de sempre. Meus agradecimentos. No campo acadêmico agradeço ao meu orientador, prof. André Santos, pela confiança e oportunidade. Ao meu co-orientador, prof. Márcio Ribeiro, pela dedicação quase diária e por me mostrar o caminho da pesquisa científica em engenharia de software. Ao todos que fazem Centro de Informática da UFPE, um lugar que me dá muito orgulho de ter estudado. Aos meus colegas de laboratório no EASY (UFAL) e no LabES (UFPE), saibam que eu aprendi um pouco com cada um de vocês.

ABSTRACT

Mutation testing is a fault-based testing criterion to assess and improve the quality of a test suite. Despite attracting much interest, the costs of using mutation testing are usually high, hindering its use in industry. Useless mutants (e.g., equivalent and duplicate) contribute to increase costs. The equivalent mutant problem has already been proven undecidable, and manually detecting equivalent mutants is an error-prone and time-consuming task. The duplicate mutant, although eventually killed by some test, requires an unnecessary computational cost. This way, solutions, even partial, can help reducing these costs. In this work, we tackle the useless mutants problem from two perspectives. First, we propose improving the transformation rules embedded in the mutation operators to avoid useless mutants. We present *i-rule*, a common language to avoid the generation of equivalent (*e-rule*) and duplicate (*d-rule*) mutants. We also present a strategy to help mutation tool developers find out occurrence patterns that lead to new *i-rules*. We instantiate the strategy with 100 Java programs, which led us to find out 99 *i-rules* for three common mutation testing tools (MuJava, Major, and Pit). To evaluate the effectiveness of the *i-rules* on reducing costs, we implement 32 of them in the MuJava tool and execute with classes of well-known projects. The results show we reduced the number of mutants by almost 20% on average and saved time to generate the mutants, thus demonstrating the potential of our approach for reducing mutation costs. Second, we present an approach to suggest equivalent mutants by using automated behavioral testing. We perform static analysis to automatically generate tests directed for the entities impacted by the mutation. For each analyzed mutant, our approach can suggest the mutant as equivalent or non-equivalent. In the case of non-equivalent mutants, our approach provides the test cases capable of killing them. For the equivalent mutants suggested, we also provide a *ranking* of mutants with a strong or weak chance of the mutant being indeed equivalent. We implement our approach in a tool called Nimrod. To evaluate Nimrod, we execute it against a set of 1,542 mutants from eight open-source projects. The results indicate that the Nimrod is very effective in suggesting equivalent mutants. It reached more than 96% of accuracy in five out of eight studied subjects. Compared with manual analysis of the surviving mutants, Nimrod takes a third of the time to suggest equivalent and is 25 times faster to indicate non-equivalent.

Keywords: Software Testing. Mutation Testing. Automatic Test Generation.

RESUMO

Teste de mutação é um critério, baseada em faltas, que serve para avaliar e melhorar a qualidade da suíte de testes. Apesar de atrair muito interesse, os custos para usar teste de mutação são geralmente altos, dificultando o sua adoção pela indústria. Mutantes inúteis (por exemplo, equivalentes e duplicados) contribuem para o aumento destes custos. Identificar todos os mutantes equivalentes de maneira automática não é possível, pois este problema foi provado indecidível. Contudo, deixar a detecção destes mutantes puramente manual é uma tarefa demorada e sujeita a erros. O mutante duplicado, embora eventualmente morto por algum teste, demanda um custo computacional desnecessário. Desta forma, soluções, mesmo que parciais, podem ajudar a reduzir estes custos. Neste trabalho, enfrentamos o problema dos mutantes inúteis a partir de duas perspectivas. Primeiro, propomos melhorar as regras de transformação embutidas nos operadores de mutação para evitar a geração de alguns mutantes inúteis. Para isso, apresentamos *i-rule*, uma definição para evitar a geração de mutantes equivalentes (*e-rule*) e duplicados (*d-rule*). Além disso, nós apresentamos uma estratégia para ajudar os desenvolvedores de ferramentas de mutação a descobrir padrões de ocorrência que levam a novas *i-rules*. Nós instanciamos a estratégia com 100 programas, o que nos levou a descobrir 99 *i-rules* em três ferramentas de teste de mutação (MuJava, Major, and Pit). Para avaliar a efetividade das *i-rules* na redução dos custos, nós implementamos as 32 delas na ferramenta MuJava e executamos com classes de projetos *open-source*. Nossa abordagem se mostrou promissora, pois conseguimos reduzir o número total de mutantes em quase 20%, em média, e economizamos tempo para gerar e compilar os mutantes. Na segunda abordagem, nós apresentamos uma técnica para sugerir mutantes equivalentes, utilizando testes gerados automaticamente. Nós primeiro realizamos análise estática para descobrir entidades impactadas pela mutação, e depois direcionamos a geração automática dos testes somente para estas entidades. Para cada mutante analisado, nossa abordagem pode sugeri-lo como equivalente ou não-equivalente. No caso de mutante não-equivalente, nossa abordagem fornece o caso de teste capaz de matá-lo. Para os mutantes sugeridos como equivalente, nós apresentamos um *ranking* indicando qual tem maior ou menor chance de ser realmente equivalente. Nós implementamos esta abordagem em uma ferramenta chamada Nimrod. Nós avaliamos o Nimrod com um benchmark de 1.542 mutantes de oito projetos *open-source* diferentes. Os resultados indicam que o Nimrod é muito eficaz na sugestão de mutantes equivalentes. Atingiu mais de 96% de precisão em cinco dos oito projetos estudados. Comparada à análise manual dos mutantes sobreviventes, o Nimrod leva um terço do tempo para sugerir um mutante como equivalente e é 25 vezes mais rápida para indicar um mutante não-equivalente.

Palavras-chaves: Teste de Software. Teste de Mutação. Geração Automática de Testes.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

It is estimated that 35% of the total cost and time to develop software is fully dedicated to software testing (CAPGEMINI; MICROFOCUS, 2019). As a good part of this activity can be automated through code, for some projects, the number of lines of test code exceeds the number of lines of code in the system under test.[1] This way, some important questions arise: *When should we stop generating tests?* or *How confident are we with our tests?*

Traditionally, software testing utilizes *coverage* as a test adequacy criterion. For instance, *statement coverage* requires all statements of the source code of the program under test to be covered by at least one test case, and *branch coverage* necessitates covering all branches of the program's control flow graph. Indeed, a test suite with a low coverage rate is a sign of low quality. However, a high coverage test suite does not necessarily mean high-quality tests. That is because even a test case that reaches a 100% coverage (such as statement, branch, and line) may not detect any fault if an appropriate oracle does not verify the corresponding output. A high-quality test suite must be effective in revealing faults.

Mutation testing (DEMILLO; LIPTON; SAYWARD, 1978; JIA; HARMAN, 2011) appears as an alternative to overcome this problem. It aims at guiding the design of strong (likely fault revealing) test suites. Mutation testing is a fault-based testing criterion that automatically seeds artificial faults into the code, generating modified versions of the system under test. The underlying idea of mutation testing is to force developers to design tests that reveal these planted faults. Hence, mutation testing leads developers to use the appropriate oracle (test assertion), since such an oracle is necessary to detect a fault eventually.

Mutation testing is considered the most efficient coverage criterion for determining the quality of a test suite (JIA; HARMAN, 2011; FRANKL; WEISS; HU, 1997). Furthermore, there is strong empirical evidence that mutants are a valid proxy for real faults (JUST et al., 2014; ANDREWS; BRIAND; LABICHE, 2005). However, the costs of using mutation testing are usually high (PIZZOLETO et al., 2019). Two kinds of mutants contribute to increasing such costs: equivalent and duplicate mutants. An equivalent mutant is a mutant that has the same behavior as the original program (BUDD; ANGLUIN, 1982; MADEYSKI et al., 2014); so, this mutant is useless. A duplicate mutant, on the other hand, has the same behavior as another mutant (PAPADAKIS et al., 2015; KINTIS et al., 2018); this way, one of them is useless.

Despite attracting much interest in academia, some researchers in the community believe the high cost of mutation testing hinders its use in industry. Useless mutants (e.g., equivalent and duplicate) contribute to increasing costs(MADEYSKI et al., 2014; KINTIS et al., 2018). In this dissertation, we focus on minimizing the problem of the high cost of

[1] The RxJava-3.x project, a library for asynchronous programming for the JVM, (<https://github.com/ReactiveX/RxJava>) has 95,918 lines of code and 201,533 lines of test code.

mutation test by eliminating useless mutants of the analysis. For this purpose, we present two approaches to deal with the useless mutants problem.

In what follows, we illustrate mutation testing through examples (Section 1.1), then we motivate the research (Section 1.2), and we finish summarizing the approaches (Section 1.3) and the contributions of this work (Section 1.4).

1.1 MUTATION TESTING BY EXAMPLE

Mutation testing is a criterion to better guide the testing process (DEMILLO; LIPTON; SAYWARD, 1978; JIA; HARMAN, 2011). Suppose we have an *all green* test suite running smoothly. A mutation testing tool automatically introduces faults into the code by creating many copies of the program, each containing one fault. The faults are created systematically by transformation rules embedded in *mutation operators*. These faulty programs are called *mutants* since they are "mutations" of the original program. The test cases execute these mutants intending to produce an incorrect (fail) output. During test execution, a mutant is "killed" when at least one test fails. When this happens, that mutant no longer needs to remain in the testing process because the faults represented by that mutant have been detected. If, after completing the execution of the tests, the mutant remains "alive", this means that the test suite did not detect the fault, and a new test case is necessary to kill that mutant. This way, the quality of the test suite can be assessed from the percentage of the total mutants killed over the total mutants generated (DEMILLO; LIPTON; SAYWARD, 1978).

To better understand the mutation testing process, Listing 1.1 presents the method `safeMultiply`, extracted from `FieldUtils`, a class file from project *joda-time*. *joda-time* is a popular date and time JAVA library. Also extracted from *joda-time*, Listing 1.2 presents a test code for `safeMultiply`. That is an *all green* test suite, which means all tests (or asserts) are passing successfully in the original code. To check if `testSafeMultiplyLongLong()` is a fault revealing test, a mutation testing tool takes as input the original program, the test code, and a set of mutation operators. Then, the mutation testing, through their mutation operators, creates the mutants, each one containing a specific transformation. We illustrate five of these transformations in Listing 1.1.

- Mutant M_1 is generated by the ROR (Relational Operator Replacement) mutation operator with the following transformation: `val2 == 1` ⇒ `val2 <= 1`.
- Mutant M_2 is generated by the AOIU (Arithmetic Operator Insertion - unary) mutation operator with the following transformation: `val1 == 0` ⇒ `-val1 == 0`.
- Mutant M_3 is generated by the ROR (Relational Operator Replacement) mutation operator with the following transformation: `if(...){...}` ⇒ `if(false){...}`.

```
1  public long safeMultiply(long val1, long val2) {
     if (val2 == 1) {                                          M1 [val2 == 1 ⇒ val2 <= 1]
3      return val1;
     }
5    if (val1 == 1) {
       return val2;
7    }
     if (val1 == 0 || val2 == 0) {                             M2 [val1 == 0 ⇒ -val1 == 0]
9      return 0;
     }
11   long total = val1 * val2;
     if (total / val2 != val1                                  M3 [if(...){...} ⇒ if(false){...}]
13       || val1 == Long.MIN_VALUE && val2 == -1               M4 [if(...){...} ⇒ /*if(...){...}*/]
         || val2 == Long.MIN_VALUE && val1 == -1) {            M5 [val1 == -1 ⇒ val1 == 1]
15     throw new ArithmeticException("Overflows a "
           + "long. " + val1 + " * " + val2);
17   }
     return total;
19 }
```

Listing 1.1 – A code snippet extracted from `FieldUtils`, a *joda-time* class.

- Mutant M_4 is generated by the SDL (Statement Deletion) mutation operator with the following transformation: `if(...){...}` ⇒ `/*if(...){...}*/`.
- Mutant M_5 is generated by the AODU (Arithmetic Operator Deletion - unary) mutation operator with the following transformation: `val1 == -1` ⇒ `val1 == 1`.

The mutation testing tool then executes the test code against the mutants to check which mutants are killed by the tests, and which ones stay *alive*. Of the five mutants in our example, three are killed by the `testSafeMultiplyLongLong()` test code (M_1, M_3 and M_4) and two mutants remained alive (M_2 and M_5). At this moment, the developer analyzes the surviving mutants to create a new test that can kill them. The mutant M_5 can be killed if we add a new assert code:

```
1 assertEquals(Long.MIN_VALUE, FieldUtils.safeMultiply(1L, Long.MIN_VALUE))
```

By analyzing the mutant M_2, the developer realizes that there is no test capable of killing this mutant, since any value of `val1` in the expression `val1 == 0` has exactly the same behavior as the expression `-val1 == 0`. It is because it is an *equivalent mutant*. We explain equivalent mutants later in this chapter. At the end of the process, the mutation testing tool concludes with an adequacy score to indicate the quality of the test suite. The common adequacy score is the *mutation score*. It is the ratio of the number of killed mutants over the total number of mutants minus the total of equivalent. The mutation score gives a number between ZERO and ONE, the closer to ONE the better the score.

For the introduction, this succinct description of mutation suffices. All the concepts mentioned above and additional information to comprehend this thesis are detailed further

```
@Test
public void testSafeMultiplyLongLong() {
    assertEquals(0L, FieldUtils.safeMultiply(0L, 0L));
    assertEquals(1L, FieldUtils.safeMultiply(1L, 1L));
    assertEquals(6L, FieldUtils.safeMultiply(2L, 3L));
    assertEquals(-6L, FieldUtils.safeMultiply(2L, -3L));
    assertEquals(-6L, FieldUtils.safeMultiply(-2L, 3L));
    assertEquals(6L, FieldUtils.safeMultiply(-2L, -3L));

    try {
        FieldUtils.safeMultiply(Long.MIN_VALUE, Long.MAX_VALUE);
        fail();
    } catch (ArithmeticException e) {
    }
}
```

Listing 1.2 – A code snippet extracted from `TestFieldUtils`, a *joda-time* test class.

in Chapter 2. Next, we discuss some of the problems of mutation testing and what motivated our research.

1.2 MOTIVATION

The mutation testing process has gained considerable attention because empirical studies have demonstrated its ability to improve test suites (ANDREWS; BRIAND; LABICHE, 2005; JUST et al., 2014; PAPADAKIS et al., 2018). However, despite being an effective test adequacy assessment method, mutation testing suffers from two main issues.

First, there is a high computational cost in generating a broad set of mutants, and mainly in executing the test suite against these generated mutants. To make matters worse, a considerable part of the mutants generated may be useless. For instance, they can be duplicate. As explained, two (or more) mutants are duplicate if they are equivalent to each other. In this case, either one or the other of these two mutually equivalent mutants can be discarded, saving some effort.

Second, there is a significant amount of manual effort involved in (i) distinguishing among the surviving mutants, which are killable and which are equivalent, and (ii) creating a new test to kill the non-equivalent mutants. Problem (ii) is inherent to all forms of testing. It means the developer needs to know the program's output to assert a testing criterion. Problem (i) is not new in the mutation testing community and is called: the *Equivalent Mutant Problem* (BUDD; ANGLUIN, 1982). An equivalent mutant is a mutant that is syntactically different from the original program but is semantically equal, which implies test cases can not kill this mutant.

To illustrate the problems above, we return to the example in Listing 1.1. This method has 19 lines of code. When we use MuJava (MA; OFFUTT; KWON, 2005), a standard mutation testing tool for Java, to generate the mutants, enabling all mutation operators, it creates 224 mutants in this method. In this example, it took approximately 4.5 seconds

to the developers' test suite to run in an Intel Core i7-6700 processor with 16 GB of RAM. [2] Thus, in a worst-case scenario, almost 17 minutes would be necessary to execute the test suite against all mutants. And `safeMultiply` is one out of 17 methods of the `FieldUtils` class.

However, by inspecting the MuJava's mutants at Listing 1.1, we found mutants worthless for the mutation analysis. By analyzing the mutant M_2, we can see that the value of the variable `val1` is only compared to zero. In this context, the mutant M_2 does not change the behavior of the program independent of the value of the `val1`. Therefore this mutant is equivalent and useless to the mutation analysis intent.

By analyzing M_3 and M_4, we realize that the tests killed these mutants. However, this occurred with a computational cost. Either changing the `if` conditional expression to `false`, or deleting the entire `if` statement will have the same result. That is, M_3 and M_4 have the same behavior and thus are duplicate. Therefore, one of them is useless to the mutation analysis.

Equivalent mutants are a well-known impediment to the practical adoption of mutation testing (MADEYSKI et al., 2014; KINTIS et al., 2018). It is not a new problem for the mutation testing community, and a long time ago proven an undecidable problem in its general form (BUDD; ANGLUIN, 1982). Thus, no complete automated solution exists. In addition, manually detecting equivalent mutants is an error-prone (ACREE, 1980) and time-consuming task (SCHULER; ZELLER, 2013). This problem becomes quite relevant when empirical studies report that up to 40% of all the generated mutants can be equivalent (MADEYSKI et al., 2014). Therefore, research efforts to reduce these costs are still needed. More recently, the mutation testing community has started tackling the duplicate mutant problem (PAPADAKIS et al., 2015; KINTIS et al., 2018). These mutants are less costly than equivalent mutants because they end up being killed by the tests, yet the computational cost of generating and running the test suite against them remains.

This work focuses on the problems of equivalent and duplicate mutants, which Papadakis et al. (PAPADAKIS et al., 2015) name as *useless mutants*. In what follows, we present our proposals for dealing with the useless mutants problem.

1.3 SCOPE OF THE THESIS

Madeyiski et al. (MADEYSKI et al., 2014) listed three methods to overcome the Equivalent Mutant Problem (which we can also extend to the duplicate mutant problem): *Detecting Equivalent Mutants*, *Suggesting Equivalent Mutants*, and *Avoiding Equivalent Mutants*. The best option should not even generate a useless mutant (avoiding). However, some equivalent and duplicate mutants cannot be discovered before the generation. If not avoided, and considering that a test case will eventually kill a duplicate mutant, we need to detect or

[2] We executed the complete test suite of the project.

to suggest, among the surviving mutants, the equivalent ones. Detecting techniques have the advantage of giving no false positives, as suggesting equivalent mutants do. On the other hand, detecting techniques can never be complete. In summary, these methods have different strengths and are complementary, and they do not compete with each other.

In this work, we tackle the useless mutants problem from two perspectives.

First, we propose to improve the transformation rules embedded in the mutation operators in a sense to avoid useless mutants. Mutation operators generate mutants indistinctly. That is, the transformation rules do not take into account whether the mutant will be useful or not. The transformation rules need to consider the program context information and the other mutation operators selected in the analysis to check the utility of the mutant and avoid the useless ones. We call these new improved transformation rules *i-rules*, but with distinctions in two separate classes: *d-rule* for avoiding duplicate mutants and *e-rule* for avoiding equivalent ones. In order to support the discovery of new *i-rules* by mutation tool developers, we also outline a strategy. This approach fits into the Madeyiski et al. (MADEYSKI et al., 2014) *Avoiding* method.

Second, we present an approach to suggest equivalent mutants by using automated behavioral testing. We implement our approach in a tool called NIMROD. NIMROD relies on tools to automatically generate test cases. To better guide the test generation, our approach uses a change impact analysis to identify methods impacted by the mutation. NIMROD *suggests the mutant as non-equivalent* in case it finds at least one test case that passes on the original program and fails on the mutant. If no test kills the mutant, NIMROD *suggests the mutant as equivalent*. To better support the testers when analyzing the mutants suggested as equivalent, we provide two metrics: the number of test cases that reached the point where the mutation occurred; and a boolean value indicating whether the test execution statement coverage of the mutant has changed when compared to the original program. Lastly, we propose to combine NIMROD with a method to *detect* equivalent mutants. This second approach fits into the Madeyiski et al. (MADEYSKI et al., 2014) *Suggesting* method.

More detailed information on the approaches we propose, as well as the performed empirical studies, the results obtained, and the threats to validity show up in chapters 3 and 4, respectively, for the first and second approach.

1.4 CONTRIBUTIONS OF THE THESIS

In summary, the main contributions of this work include two approaches to tackling useless mutants problems. Regarding the first approach that avoids useless mutants, we present:

- The definition of a notation (*i-rule*) to *avoid* the generation of equivalent and duplicate mutants (Section 3.3);

- A strategy to help identifying new *i-rules* (Section 3.4);
- We introduce 30 *i-rules* for equivalent and 69 *i-rules* for duplicate mutants (Section 3.5);
- We report on the results of an empirical study to evaluate some *i-rules* implemented in the MuJava mutation tool (Section 3.6);
- We provide MuJava-AUM embedded with 9 and 23 *i-rules* for equivalent and duplicate mutants, respectively (Section 3.6).

Regarding our second approach, which suggests equivalent mutants, we present:

- An approach to suggest equivalent mutants based on automated behavioral testing (Section 4.3);
- A tool that implements and automates the entire approach (namely, Nimrod) (Section 4.3);
- An evaluation to check the effectiveness and efficiency of our approach on suggesting equivalent mutants (Sections 4.4 and 4.5);
- A discussion regarding implications for practice of combining the methods *Suggesting Equivalent Mutants* and *Detecting Equivalent Mutants* in order to reduce costs when dealing with the equivalent mutant problem (Section 4.6).

1.5 ORGANIZATION OF THE THESIS

- Chapter 2 provides a more detailed view of mutation testing and its limitations.
- Chapter 3 details our approach to avoid useless mutants using *i-rules*.
- Chapter 4 details our approach to suggest the equivalent and non-equivalent mutants through automated behavioral testing.
- Chapter 5 presents the related works.
- Chapter 6 concludes this thesis by summarizing its contributions and providing possible avenues for future research.

2 MUTATION TESTING AND ITS LIMITATIONS

This chapter exposes the concepts of mutation testing and its limitations. In particular, the problem of useless mutants. In spite of being succinct, this summary suffices for the purposes of this work. A more precise introduction of mutation testing can be found elsewhere (JIA; HARMAN, 2011; OFFUTT; UNTCH, 2000; PIZZOLETO et al., 2019). The rest of the chapter is organized as follows: we introduce mutation testing concepts (Section 2.1) through the process analysis (Section 2.1), examples of mutation tools (Section 2.2), the importance of designing good mutation operators (Section 2.3), and the associated cost of the technique (Section 2.4). After that, we bring the problem of useless mutants up and how we can overcome it (Section 2.5).

2.1 THE MUTATION TESTING PROCESS

Mutation testing (also referred to *mutation analysis*) is a fault-based criterion: it injects artificial faults in the program under test by creating many copies of the original program, each one containing one (or more) simple fault(s). Then it executes the test suite against these copies to check the compliance of the output from that of the original program.

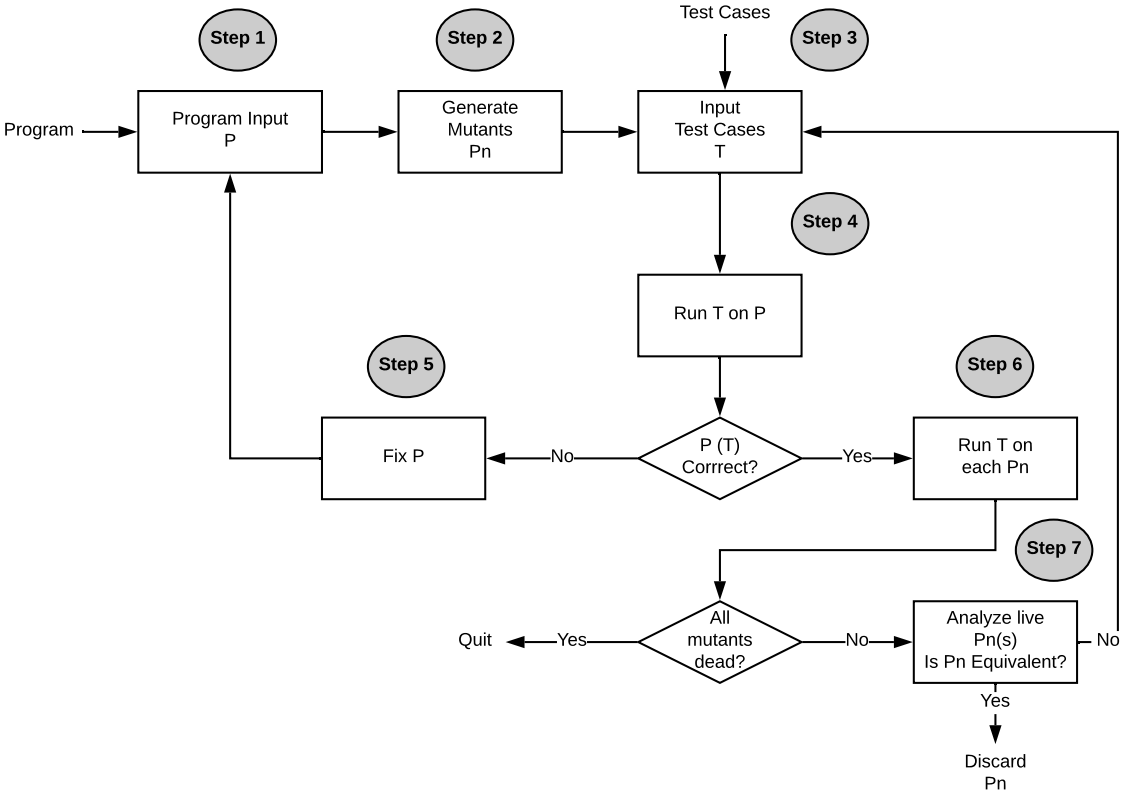
Mutation attempts to simulate real faults by inducing artificial faults to the program under test. These faults are, in general, simple syntactic changes based on two hypotheses, namely the competent programmer (ACREE et al., 1979) and the coupling effect (DEMILLO; LIPTON; SAYWARD, 1978). The competent programmer hypothesis states that programs written by competent programmers are close to being correct, then the programs require a few syntactic changes to reach the correct version. And the coupling effect hypothesis states that test data sets that detect many simple faults are sensitive enough to detect more complex faults; this way, it is sufficient to inject small defects to simulate or represent real faults.

Mutation testing has been increasingly studied since first proposed in the 1970s (LIPTON, 1971; DEMILLO; LIPTON; SAYWARD, 1978). Many research papers have appeared on several points of the mutation process seeking to turn mutation testing into a practical approach (JIA; HARMAN, 2011; PIZZOLETO et al., 2019; PAPADAKIS et al., 2019). We detail this process in what follows, indicating the automatic and manual parts.

The main aim of mutation testing is assessing the quality of a given test set or testing strategy. For this, some steps need to be taken. Offutt and Untch (OFFUTT; UNTCH, 2000) establish a generic mutation testing process. We illustrate this process in Figure 1.

Based on an original program input P provided by the developer (Step 1), the mutation testing tool generates a set of faulty programs Pn, referred to as *mutants*, by making simple syntactic changes from the original program P (Step 2). These syntactic changes

Figure 1 – Mutation Testing Process



are predefined transformation rules defined by *mutation operators*. In the next step, a test set T is supplied to the system (Step 3), and we run T on P (Step 4). In case T detects an error in P, it is necessary to fix P and start over (Step 5). But if there is no error detected in P, the mutation testing tool executes the test set T against each mutant Pn and check its correctness (Step 6). If the result of running Pn is different from the result of running P for at least one test case in T, then the mutant Pn is said to be *killed*. Consequently, an undetected mutant is referred to as *live mutant*. In case all mutants have been killed, the process ends, but if there are live mutants, the developer analyzes each live mutant to determine their equivalence or to produce a new test case to kill it (Step 7). We talk more about equivalent mutants in Section 2.5.

Mutation testing tools easily automate steps 2, 4, and 6. Steps 1, 3, 5, and 7, traditionally, need manual intervention.

The mutation testing tool concludes with an adequacy score to indicate the quality of the test set T. The common adequacy score is the *mutation score*, which is represented by the following formula:

$$MS = \frac{Mk}{Mt - Me}$$

It is the ratio of the number of killed mutants over the total number of mutants minus the

total number of equivalent mutants. Where Mt is the total number of produced mutants, Mk is the number of killed mutants, and Me is the number of equivalent mutants. The final value of the mutation score ranges between zero and one: the closer the value is to one, the better the result is.

Regarding the concept of killing a mutant, the necessary conditions for a test t to kill a mutant $m \in Mt$ can be described using the reachability, infection, and propagation (**RIP**) model (DEMILLO; OFFUTT, 1991).

1. (**R**eachability): t must execute m's mutation at least once.
2. (**I**nfection): at least one execution of m's mutation must cause m's execution state to differ from that of p.
3. (**P**ropagation): the infected execution state of m must propagate to some observable output.

This way, three common variants of mutation testing are defined: *weak*, *strong*, or *firm* mutation. In weak mutation (HOWDEN, 1982), a test kills a mutant if the test execution leads to a difference between the program state of the mutant and the program state of the original version immediately after the execution of the mutated point. In other words, a test t kills a mutant m if t satisfies the **I**nfection condition. In contrast, strong mutation (DEMILLO; LIPTON; SAYWARD, 1978), often referred to as traditional mutation testing, requires that this difference propagates to an observable output, i.e., an assertion failure or an exception. That is, a test t kills a mutant m if t satisfies the **P**ropagation condition. The weak mutation is less computationally expensive than strong mutation since we just need to execute the test until the mutated point. However, given that it is easier to kill weak mutants, then we sacrifice test effectiveness for improvements in test effort. It raises the question as to what kind of trade-off can be achieved. The idea of firm mutation (WOODWARD; HALEWOOD, 1988) is to overcome the disadvantages of both weak and strong mutations by providing a continuum of intermediate possibilities. That is, the "compare state" of firm mutation lies between the intermediate states after execution (weak mutation) and the final output (strong mutation). To the best of our knowledge, there is no publicly available firm mutation tool. Throughout this work, we use the traditional idea of killing a mutant using strong mutation.

If the test suite does not kill the mutant, we need a new test case to kill the mutant, or this mutant can be equivalent. At this point, it is worth adding an addendum to the notion of equivalence used in this work. Despite a mutant representing a valid syntactic change of the original program, the notion of equivalence applies to the semantics. As explained, in strong mutation, a mutant and an original program are equivalent if they present the same externally observable behavior for all possible inputs. However, we can divide this assumption into two different scenarios; *open world* and *closed world* (SOARES;

GHEYI; MASSONI, 2013). In an *open world assumption* (OWA), any kind of test case can be generated to find out a behavioral change, without regarding the project or code requirements. In a *closed world assumption* (CWA), the test cases must satisfy some domain constraints. For example: "*all the tested methods must be invoked through a Facade*" or "*there is a strict call sequence of methods to be followed.*" In CWA, an equivalent mutant may, for example, indicate that the test is violating a system requirement, or the system has a security flaw. In this thesis, we adopt an open-world equivalence notion, which means there are no constraints in the test generation.

Next, we take a look at some mutation testing tools responsible for the automated steps previously mentioned.

2.2 MUTATION TESTING TOOLS

To make the mutation testing possible, we need tools. Mutation testing tools traditionally have three primary responsibilities: to generate the mutants, to execute the test suite against the mutants, and to calculate the adequacy score (mostly the mutation score). In this work we have selected the following tools to execute these steps: MuJava (MA; OFFUTT; KWON, 2005; OFFUTT; MA; KWON, 2006), Major (JUST; SCHWEIGGERT; KAPFHAMMER, 2011), and Pit (PITEST, 2017).

MuJava (Mutation System for Java) (MA; OFFUTT; KWON, 2005; OFFUTT; MA; KWON, 2006) is one of the first Java mutation testing tools and has been used in many mutation testing studies. The tool manipulates the source code of the program under test and supports two types of mutation operators: class-level and method-level. The class-level mutation operators were designed for Java classes and handle object-oriented specific features such as inheritance, polymorphism, and dynamic binding (MA; KWON; OFFUTT, 2002; OFFUTT; MA; KWON, 2006). Method-level (traditional) mutants follow the selective operator set by Offutt et al. (OFFUTT; PAN, 1996) and handle primitive features of the languages, such as arithmetic expressions, predicates, and iterative structures. There is no equivalent mutants detection. In April 2015, MuJava has been released as open-source under the Apache license.[1]

Pit (PITEST, 2017) is a mutation testing framework primarily targeted at the industry but has also served in many research studies. Pit mutates the bytecode, *i.e.*, it does not compile the code; instead, it modifies the bytecode in memory. Pit only requires the location of the source code to generate a human-readable report. It employs mutation operators that affect primitive programming language features. For this work, we extended Pit to write all generated mutants in the disk. It is important to the manual analysis in Step 5. Pit has been released as open-source under the Apache license.[2]

[1] <https://github.com/jeffoffutt/MUJAVA> - accessed in October, 2019
[2] <https://github.com/hcoles/pitest> - accessed in October, 2019

MAJOR (JUST; SCHWEIGGERT; KAPFHAMMER, 2011) integrates into the JAVA compiler, in a non-invasive way, and does not require a specific mutation analysis framework. The tool manipulates the AST of the program under test. The implemented mutation operators follow a reduced set of operators defined by Offutt et al. (OFFUTT; PAN, 1996), similarly to MUJAVA. MAJOR uses mutant schemata (UNTCH; OFFUTT; HARROLD, 1993); *i.e.*, to reduce the number of generated mutant programs, it produces a *metaprogram* derived from the program under study, and each metaprogram contains multiple mutations. Each mutation is guarded by a conditional statement that can be switched on and off at runtime. To use the mutant for further equivalence analysis, the tool exports each generated mutant to an individual source-code file. MAJOR has been released as open-source (MAJOR website does not specify the license).[3]

We now discuss the primary responsibility of a mutation testing tool: to generate the mutants.

2.3 MUTANT GENERATION

Mutation testing tools have a set of mutation operators responsible for creating the mutants. The first objective of a mutation operator is to mimic typical errors that programmers can make, such as using the wrong operator or omitting a statement. And the second objective is to force tests that are usually considered valuable, such as forcing expressions to have the value zero or forcing the execution of all paths in a conditional statement.

If a mutant is created by changing one single code location, we say it is a *first order mutant*. In case the mutant is generated by changing more than one place, we say it is a *higher order mutant* (JIA; HARMAN, 2008). According to the number of changes, higher-order mutants are termed second-order mutants (if they introduce two changes), third-order mutants (in the case of three changes), and so on. Tools that implement higher order mutants are still scarce (MADEYSKI et al., 2014). In this work, we use tools that create first-order mutants.

Each tool developer designs and implements the set of mutation operators in its way. To generate the mutants, the mutation operators rely on transformation rules. In general, one mutation operator can perform one or more transformations in the original program, where each transformation generates a mutant. For instance, the mutation operator AOR (Arithmetic Operator Replacement) is present in many mutation testing tools (MA; OFFUTT; KWON, 2005; JUST; SCHWEIGGERT; KAPFHAMMER, 2011; PITEST, 2017).[4] In case this mutation operator visits the expression `x+y`, it will generate up to four mutants: `x-y`, `x*y`, `x/y`, and `x%y`. It means four different transformations.

[3] <https://bitbucket.org/rjust/major-java7> - accessed in October, 2019

[4] Some tools might use different names for the same mutation operator.

The design of mutation operators is crucial to the success of the mutation tool. The tool must generate as few mutants as possible without losing effectiveness, which means to simulate the maximum number of bugs, but, preferably without incurring in useless mutants (Section 2.5). Generating hard-to-kill useful mutants, also known as stubborn mutants (YAO; HARMAN; JIA, 2014), helps developers improve their test suite. Trivial-to-kill mutants offer no benefits. The task of generating useful mutants, however, is not trivial. That is because these useful mutants are very dependent on the context (JUST; KURTZ; AMMANN, 2017). Besides that, the operators are very dependent on the programming language of choice. For example, the set of operators for JAVA must be different from the set of operators for HASKELL, since the errors that programmers of these languages make tend to be different.

Mutation testing tools usually have many transformation rules, generating a broad set of mutants and consequently increasing the cost of the mutation analysis. Next, we discuss the main costs associated with mutation testing.

2.4 MUTATION TESTING COSTS

The barriers that prevent the practical use of mutation testing can be classified into two groups: computational cost and manual labor cost.

The major computational cost of mutation testing arises from the high number of generated mutants and the high computing time to execute each mutant. Offut and Untch (OFFUTT; UNTCH, 2000) classify three strategies to solve this problem: *do fewer*, *do faster*, and *do smarter*. The "do fewer" approaches try to decrease the number of mutants generated without losing effectiveness. The "do faster" approaches focus on ways of generating and running the mutants as quickly as possible. The "do smarter" approaches look for clever solutions to generate and run mutants, such as running only the test cases that are necessary for each mutant, or distributing the computational cost over several machines. For a more detailed list of advances in this field, please refer to Pizzoleto et al. (PIZZOLETO et al., 2019). To make matters worse, in recent work, Papadakis et al. (PAPADAKIS et al., 2015) highlight the problem of *duplicate mutants*. That is, two mutants that are not equivalent to the original program from which they are constructed, but are equivalent to each other. Thus either one or the other of these two mutually equivalent mutants can be discarded. Besides generating a high number of mutants, part of these mutants are duplicate and unnecessary to the analysis.

The other side of the cost problem comes from the manual labor cost, i.e., the amount of human effort involved to do some tasks. The first one is also known as the *Human Oracle Problem* and claims that the developer needs to know the program's output to assert a testing criterion. This problem is inherent to all forms of testing. Advances in automatic test case generation can support this laborious task. The second one is the *Equivalent Mutant Problem* (BUDD; ANGLUIN, 1982; MADEYSKI et al., 2014; KINTIS et al.,

2018). Although syntactically different from the original program, the mutant can be semantically equal, which implies that its behavior does not change, when compared to the original program, for any input data. Thus the developer needs to check whether a surviving mutant is merely hard to kill (one more test case is necessary) or equivalent (no test can kill it so that any attempt is futile).

This work focuses on the problems of equivalent and duplicate mutants, which Papadakis et al. (PAPADAKIS et al., 2015) names *useless mutants*. The next section depicts in more detail the problem of useless mutants.

2.5 USELESS MUTANTS

As explained, mutation testing is known to be very costly. Each mutant contributes to the cost since each mutant must be created, executed, and if it remains alive, be analyzed. Unfortunately, many mutants are useless. In this work, we use the term *useless* to represent *equivalent* and *duplicate* mutants.[5] These mutants do not incorporate anything to the mutation testing process (PAPADAKIS et al., 2015).

Equivalent mutants, as explained, occur when the mutant maintains the same behavior as the original program. This way, there is no test able to kill it. That is, besides the computational cost of generating and executing the complete test suite, the equivalent mutant is analyzed to check if it is indeed equivalent and, in turn, discarded from the analysis.

Budd and Angluin (BUDD; ANGLUIN, 1982) have already proven that the equivalent mutant problem is an undecidable problem in its general form. Thus, no complete automated solution exists. To worsen the situation, manually detecting equivalent mutants is an error-prone and time-consuming task. Acree (ACREE, 1980) showed that 20% of the studied mutants were erroneously classified, i.e., a killable mutant classified by mistake as equivalent or vice versa. Shuler et al. (SCHULER; ZELLER, 2013) showed that developers take, on average, 15 minutes to manually classify a mutant as equivalent or nonequivalent. This problem becomes quite relevant when empirical studies report that between 4% and 39% of all the generated mutants are equivalent (MADEYSKI et al., 2014). Therefore solutions, even partial, can significantly help to reduce this cost.

Duplicated mutants are a more recent problem (PAPADAKIS et al., 2015; KINTIS et al., 2018). That is if the mutant $P1$ is duplicate to the mutant $P2$, all the tests that kill $P1$ *are precisely the same ones* that kill $P2$. In other words, a duplicate mutant is non-equivalent to the original program, but it has the same observable behavior as other mutant(s). In this case, only one of the duplicate mutants is necessary for mutation analysis. Although it does not involve a manual effort as equivalent mutants do, duplicate mutants have a unnecessary computational cost, given that the test set needs to execute against two or

[5] In Chapter 6, we discuss other kinds of useless mutants.

more mutants that have the same behavior. Papadakis et al. (PAPADAKIS et al., 2015) was able to detect 21% of the mutants as duplicate. However, this number may be even higher since the technique proposed by Papadakis et al. does not detect all cases. Besides the cost, duplicate mutants make the mutation score imprecise, as reported by Kutz et al. (KURTZ et al., 2016). For example: Consider a set of 10 nonequivalent mutants M and a set of 10 tests T. If we execute T against M and it kills 7 mutants, the mutation score is 0.7. Now suppose we add ten more mutants to M, but all of them are duplicate when compared to a previously killed mutant in the original set. There are now 20 nonequivalent mutants, and 17 are killed. Thus the mutation score is 0.85. It shows that without any real improvement in the test suite, the value of the mutation score has improved. This example and the high number of possible duplicate mutants highlight the importance of having solutions to reduce this problem.

Madeyiski et al. (MADEYSKI et al., 2014) listed three methods to overcome the Equivalent Mutant Problem (which we can also extend to the duplicate mutant problem): *Detecting Equivalent Mutants*, *Suggesting Equivalent Mutants*, and *Avoiding Equivalent Mutants*. All three methods have strengths, so they are complementary and do not compete with each other. In this work, we present two approaches to tackle the useless mutants problem. The first one focuses on avoiding useless mutants before the generation (*do fewer*, according to Offutt and Untch (OFFUTT; UNTCH, 2000)); since we cannot eliminate all useless mutants before the generation, the second approach suggests the equivalent and non-equivalent among the live mutants. In the next two chapters, we present these two approaches. The related works on this topic are presented in Chapter 5.

3 AVOIDING USELESS MUTANTS

First things first. To tackle the problem of useless mutants, as a first step, the mutation testing tools should prevent the generation of certain useless mutants. In what follows, we depict our first approach to tackle the problem of useless mutants.

3.1 INTRODUCTION

The presence of equivalent and duplicate mutants masks the mutation testing results (KURTZ et al., 2016; JUST; KAPFHAMMER; SCHWEIGGERT, 2012), and raises human and computational costs. Previous work (MADEYSKI et al., 2014) reported that the ratio of equivalent mutants might lie between 4% and 39% of the total of mutants. Only recently, the duplicate mutants problem has been discussed and tackled. Researchers reported 21% of duplicate mutants in their empirical study (KINTIS et al., 2018). According to Madeyiski et al. (MADEYSKI et al., 2014), concerning equivalent mutants, most of the solutions try to detect these mutants after they are generated (detecting or suggesting). The few solutions (HARMAN; HIERONS; DANICIC, 2000; FERRARI; RASHID; MALDONADO, 2013; WRIGHT; KAPFHAMMER; MCMINN, 2014; KINTIS; MALEVRIS, 2015; PAPADAKIS; MALEVRIS, 2010) that try to prevent the mutant from being generated either do not provide practical effectiveness in their outcome, are specialized to domain-specific mutants, depend on valuable resources, or work with second-order mutants. Regarding the duplicate mutants, as far as we know, there is no practical solution that tries to avoid the generation of this kind of useless mutant systematically.

Mutation testing tools create mutations by modifying code using syntax-changing transformation rules. These transformation rules come embedded in *mutation operators*. The design and implementation of the mutation operators are a crucial point in the mutation testing process. Ideally, the mutation operators should generate few, non-equivalent, non-duplicate, and hard to kill useful mutants. However, these useful mutants are very dependent on the context where they are inserted. We believe that the useless mutants problem starts with the design of the mutation operators. Most of the mutation operators create useful mutants; however, on certain occasions, they should not generate some mutants. For instance, in many situations, the mutation operators responsible for the mutants M_2, M_3, and M_4 from Listing 1.1 yield useful mutants but, in that context, those mutants were equivalent or duplicate.

To avoid the generation of useless mutants, we propose to improve the transformation rules embedded in the mutation operators. The transformation rules need to consider the program context information and the other mutation operators selected to be applied. We call these new improved transformation rules *i-rules*, but we classify them in two separate

classes: *d-rule* for duplicate mutants and *e-rule* for equivalent mutants.

To help developers of mutation tools discover these *i-rules*, we have outlined a strategy. To use the strategy, we need the following input parameters: a set of programs, a mutation testing tool, and oracles to automatically identifying equivalent and duplicate mutant candidates. The strategy indicates occurrence patterns that facilitate us to derive *e-rules* and *d-rules* to avoid the generation of useless mutants in subsequent executions of the mutation tool.

In this work, we introduce and evaluate the strategy with 100 Java programs[1], mutants generated by three common mutation testing tools (namely, MuJava (MA; OFFUTT; KWON, 2005; OFFUTT; MA; KWON, 2006), Major (JUST; SCHWEIGGERT; KAPFHAMMER, 2011), and Pit (PITEST, 2017)), and we decided to use a massive set of automatically generated tests as oracle. The tools Randoop (PACHECO et al., 2007) and EvoSuite (FRASER; ARCURI, 2011) were responsible to generate the tests. From the 100 programs analyzed, 50 were automatically generated by the JDolly program generator (SOARES; GHEYI; MASSONI, 2013) and 50 were manually written Java files extracted from open-source repositories. The three mutation testing tools generate 24,826 mutants. In the end, our strategy marked 4,957 mutants as equivalent and 2,620 as duplicate. We then analyzed these mutants and identified 30 *e-rules* to avoid equivalent mutants and 69 *d-rules* to avoid duplicate mutants in the three different mutation tools.

To evaluate the effectiveness of the *i-rules* on reducing costs, we implement 32 of them (9 *e-rules* and 23 *d-rules*) in the MuJava tool. All the implemented *i-rules* use the parser available in the MuJava tool. That is, we did not implement improvements that require more diligent static analysis (e.g., def-use analysis). To better differentiate from the original version, we call this tool MuJava-AUM (AUM stands for *Avoiding Useless Mutants*). We then executed the MuJava and MuJava-AUM with classes of well-known projects such as *ant*, *bcel*, and *commons-lang*. As a result, we reduced the number of mutants by almost 20% on average and saved time to generate the mutants, thus demonstrating the potential of our approach for reducing mutation costs. These results are promising because: (i) differently from previous work (PAPADAKIS et al., 2015; KINTIS et al., 2018; BALDWIN; SAYWARD, 1979; OFFUTT; PAN, 1996; OFFUTT; PAN, 1997; VOAS; MCGRAW, 1997; HIERONS; HARMAN; DANICIC, 1999; GRÜN; SCHULER; ZELLER, 2009; SCHULER; ZELLER, 2013; SCHULER; DALLMEIER; ZELLER, 2009), we do *not* generate, compile, and analyze whether the mutants are useless or not, instead, we avoid such mutants from being generated; (ii) we can derive more *i-rules* in case we set our strategy with different Java constructs; and (iii) we have implemented only a subset of the *i-rules* identified. We also evaluate which *e-rules* and *d-rules* are most useful and those that may be excluded, since they are rarely applied. One *e-rule* (out of 9) is responsible for 90% of

[1] We will use the term 'program' to represent the input of the strategy, even if, for some cases, this program represents a single Java file.

all equivalent mutants avoided and six *d-rules* (out of 23) are responsible for approximately 86% of the duplicate mutants avoided. Regarding the execution time, despite the overhead introduced by the *e-rules* and *d-rules*, the payoff amount is 15% on average.

In summary, the main contributions of our first approach include:

- The definition of a common notation (*i-rule*) to *avoid* the generation of equivalent (*e-rule*) and duplicate (*d-rule*) mutants (Section 3.3);
- A strategy to help identifying new *i-rules* (Section 3.4);
- We introduce 30 *e-rules* and 69 *d-rules* (Section 3.5);
- We report on the results of an empirical study to evaluate some *e-rules* and *d-rules* implemented in the MuJava mutation tool. In particular, we avoid almost 20% (on average) of useless mutants in classes from well-known projects, saving time to generate and compile the mutants (Section 3.6);
- We provide MuJava-AUM embedded with 9 *e-rules* and 23 *d-rules* (Section 3.6).

3.2 MOTIVATING EXAMPLE

As mentioned, mutation effectiveness depends largely on the the design of its mutation operators. Ideally, the mutants generated must be non-equivalent, non-duplicate, and hard to kill. Previous work (MADEYSKI et al., 2014) reported that the presence of equivalent mutants could reach 40% of the total mutants generated. The only study that evaluated the presence of duplicate mutants empirically states that the number of duplicate mutants can reach 21% (KINTIS et al., 2018).

Some previous work advocate for *selective mutation* (MATHUR, 1991; OFFUTT; ROTHERMEL; ZAPF, 1993), to generate as few mutants as possible without losing effectiveness. In summary, the idea is to carefully choose a small set of mutation operators that generates fewer mutants without a significant loss of test effectiveness. Several works have carried out empirical studies to define the "ideal" set of mutation operators (ADAMOPOULOS; HARMAN; HIERONS, 2004; DEREZIŃSKA; RUDNIK, 2012; GLIGORIC et al., 2013; BLUEMKE; KULESZA, 2014; DELGADO-PÉREZ; SEGURA; MEDINA-BULO, 2017). However, recent research (GOPINATH et al., 2016) has shown that existing approaches of selective mutation do not significantly outperform random selection. The problem is that existing approaches to selective mutation do not take into account the program context (JUST; KURTZ; AMMANN, 2017). For instance, the ROR (Relational Operator Replacement) mutation operator treats a relational operator in a for loop the same way as a relational operator in an if statement. As another example, the existing AOR (Arithmetic Operator Replacement) mutation operator mutates an arithmetic operator without considering the operands of a binary expression.

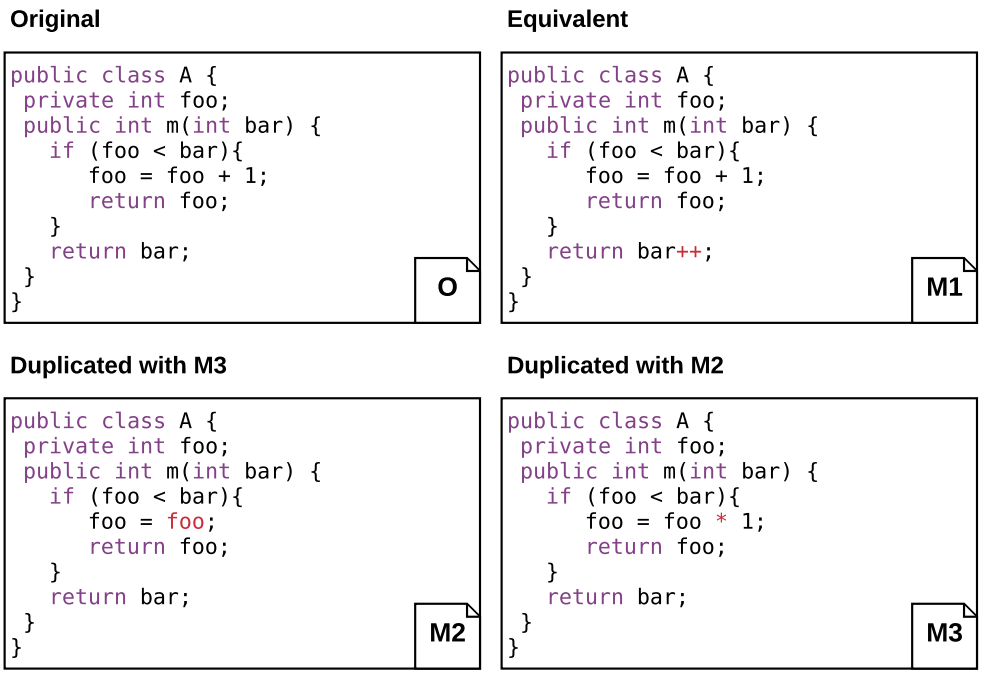


Figure 2 – Original program and three mutants.

To better illustrate this problem, Figure 2 presents one original program (O) and three mutants (*M1*, *M2*, and *M3*). Mutant *M1* was generated from a mutation operator that introduces a post increment to a local variable (`bar++`). Notice that this introduction does not change the behavior when compared to the original program since the increment would happen after the function returns and `bar` is a local variable. In this sense, *M1* is equivalent to O and, as such, useless.[2] Mutant *M2* was generated from a mutation operator that removes the right-hand side of the arithmetic expression, yielding `foo = foo`. In mutant *M3*, the mutation operator replaces the `+` operator by $*$, yielding `foo = foo * 1`. *M2* and *M3* have the same behavior and thus are duplicate. Therefore, either *M2* or *M3* is useless.

Mutant *M1* was generated from AOIS (Arithmetic Operator Insertion) mutation operator. AOIS, in the `return bar` statement, can generate up to four mutants (`return ++bar`, `return --bar`, `return bar++`, `return bar--`). Note that two mutants are useless and two mutants are useful. This AOIS operator is known by the community to generate many mutants and also generates many equivalent ones (KINTIS; MALEVRIS, 2015). However, this mutation operator also generates many hard-to-kill mutants, which in turn is suitable for mutation analysis (YAO; HARMAN; JIA, 2014).

Based on this scenario, by eliminating the mutation operator, we also eliminate useful mutants. More importantly, some of these mutants are avoidable before the generation.

King and Offutt (KING; OFFUTT, 1991) were the first to come up with "restrictions" in

[2] A previous work (PETROVIĆ et al., 2018) argues that an equivalent mutant can be productive if its analysis advances knowledge and code quality.

conjunction with the proposed transformations of the mutation operators of the *Mothra* mutation tool. The restrictions included conditions, in textual notation, to not generate mutants when they would be equivalent (in some cases they avoid duplicate mutants, but they refer to "equivalent to another mutant"). The restrictions presented were very language-dependent (*Fortran*), and the textual notation brings some difficulty to establish a common understanding. Since then, few works have presented preconditions to apply certain transformations in conjunction with the mutation operators (MADEYSKI et al., 2014; KINTIS et al., 2018).

Mresa and Bottaci (MRESA; BOTTACI, 1999) analyzed all the mutation operators used in the Mothra Tool through selective mutation. They sought to determine which operators had fewer equivalent mutants and, through heuristics, tried to identify the best subset of mutation operators. By eliminating the mutation operator, we also eliminate useful mutants for the analysis. For instance, Offutt et al. (OFFUTT; MA; KWON, 2006) observed that the AMC (Access Modifier Change) mutation operator creates a high number of useless mutants.[3]

> *If the access is strengthened (for example, public to private), the mutated program usually does not compile — the mutant tries to use a variable that is out of scope. If the access is weakened, the mutated program is often equivalent — the mutant can still use the same variables. [...] Thus, the AMC operator generates uncompilable, equivalent, or redundant mutants, and is not needed in* MuJava. (OFFUTT; MA; KWON, 2006)

However, the same authors recognize that changing the access modifier is a source of mistakes among developers.

> *In our experience in teaching OO software development and consulting with companies that rely on OO software, we have observed that access control is one of the most common sources of mistakes among OO programmers. The semantics of the various access levels are often poorly understood, and access for variables and methods is often not considered during design. It can lead to hasty decisions during implementation. It is important to note that inadequate access definitions do not always cause faults initially but can lead to faulty behavior when the class is integrated with other classes, modified, or inherited from.* (MA; KWON; OFFUTT, 2002)

Indeed, selecting the correct access modifiers in Java can become quite tricky. Nevertheless, Steimann and Thies (STEIMANN; THIES, 2010a) demonstrated that by negating the semantics-preserving rules of existing formalized refactoring tools, the AMC operator could produce useful mutants.

[3] The authors also use the therm *useless* to refer mutants that do not compile.

Harman et al. (HARMAN; HIERONS; DANICIC, 2000) proposed the use of program dependence analysis (a kind of static analysis) to detect the equivalent mutants in a probe point (weak killing). They did not present experiments nor obtained results. Afterward, Kintis and Malevris (KINTIS; MALEVRIS, 2015) introduced a static analysis tool that identifies data-flow patterns and avoids a large portion of equivalent mutants by just analyzing the original program under test. These techniques are promising but require an approach based on more in-depth static analysis, which increases the cost of the mutant generation phase.

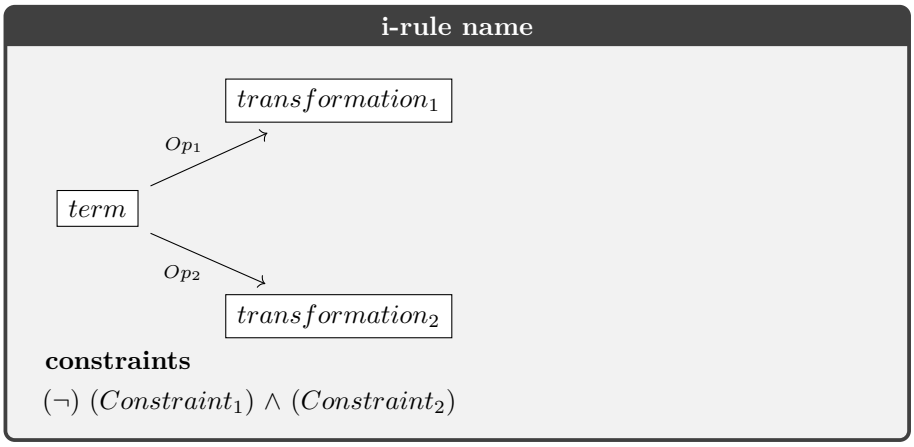
Next, we discuss a proposal to avoid equivalent and duplicate mutants by improving the transformation rules embedded in the mutation operators.

3.3 IMPROVED TRANSFORMATION RULES

The mutation operator observes a program structure looking for a particular syntactic element where it can be applied. Once found, the mutation operator performs the program transformations generating the mutants. In general, the transformation rules embedded in the mutation operators occur purely at the syntactic level. However, when applied in specific contexts, some transformations end up generating useless mutants, as presented in Figure 2. The transformation rules need to get a step further to avoid the generation of these useless mutants, which means considering the program context and also the other mutation operators selected to be applied.

In this work, we propose to improve the transformation rules embedded in the mutation operators to avoid equivalent and duplicate mutants. We formulate and check the constraints of a mutation transformation to ensure that performing it has the desired effect. We divide these improvements into two classes: *e-rule* and *d-rule*. An *e-rule* is an improvement in the transformation rule regarding avoiding equivalent mutants and states that no transformation should be applied whatsoever. On the other hand, a *d-rule* is an improvement in the transformation rule regarding avoiding duplicate mutants and states that only one of the transformations should be applied.

An *e-rule* and a *d-rule* are similar in several aspects. Because of that we use the term *i-rule* to refer to both. We define an *i-rule* to avoid useless mutants as a triple ($term$, $transformations$, $constraints$). We use the following notation to represent an *i-rule*.



where:

- *term* is any language construct;
- *transformations* is a set of mutation operators (Op_n) applied to the *term* or a part of the *term*; and
- *constraints* is a set of conditions on *term* or on the arguments of the mutation operators in $transformations$ that guarantee that the *i-rule* indeed avoids useless mutants.

We interpret an *i-rule* as follows: If the original program matches the *term* (shown on the LHS – left-hand side) and the selected mutation operators Op_n are going to perform the $transformations$ (shown on the RHS– right-hand side), then the mutation tool checks the *constraints*. In case all *constraints* are met, it means the mutant will be useless and the mutation tool must: (i) not perform the transformation (*e-rule*), or (ii) perform only one of the transformations (*d-rule*).

As the *i-rules* presented are valid for the JAVA programming language, then the *terms* and $transformations$ are expressed using a similar JAVA notation. However, when necessary, we replace language constructs with meta-variables. For instance, we use *op* to represent any JAVA binary or unary operator or v to represent any identifier of a variable for a primitive integral type (byte, short, int, long). To omit parts of code that are not relevant, we use ellipsis (...). Table 1 depicts all meta-variables referred by the *i-rules*. In the case where no constraint is necessary, we use $\varnothing$.

For all *i-rules*, we assume the following:

- Before applying an *i-rule*, all conditions required to perform the transformation by the mutation testing tool have been met, and the transformation occurs if the *i-rule* is not enabled.

Table 1 – Meta-variables referred by the *i-rules*.

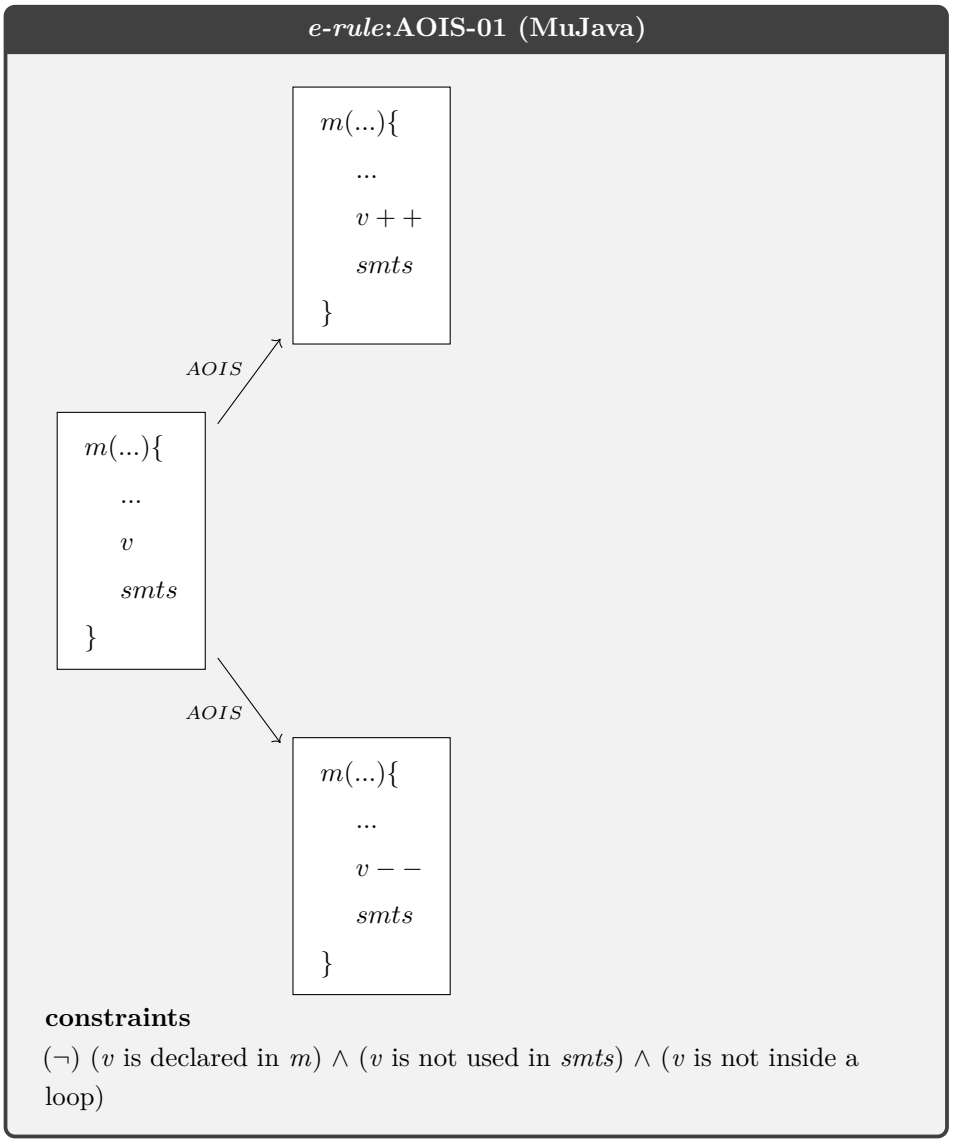
Meta-variables	Description
op	any binary or unary operator
exp	any expression
$smts$	any block of statements
s	any statement
;	an empty statement
A, B	class references
m	method references
v	any identifier of a local variable for a primitive integral type (byte, short, int, long)
sv	any identifier of a String or Array variable
f	any identifier of a field
dv	the default value of any specified type (i.e., int=0, boolean=false, char='u0000', double=0.0d, float=0.0f, long=0L, Object=null, etc.)

- All *i-rules* presented consider only the normal flow of the program execution. In other words, we are not considering anomalous or exceptional conditions that require special processing (e.g., exception handling).

Once explained the definition and all the elements involved in an *i-rule*, we illustrate an example of *e-rule* and another example of a *d-rule* extracted from the Figure 2.

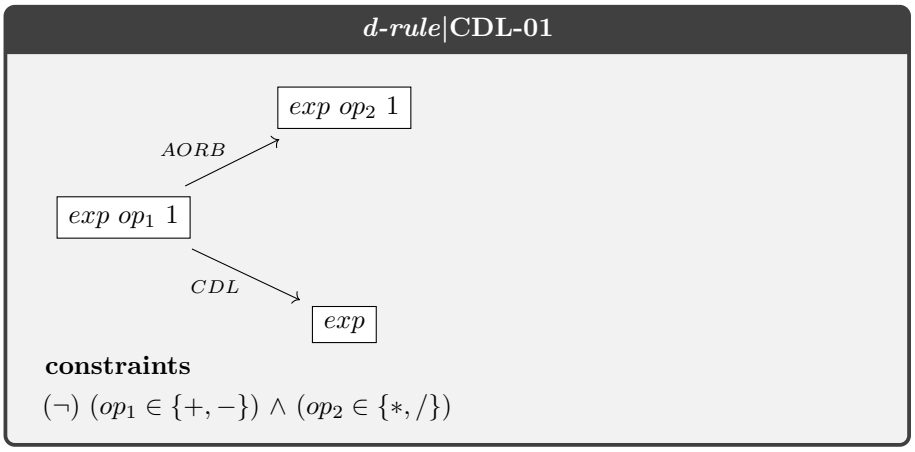
3.3.1 *e-rule* Example

In Figure 2, the *M1* mutant is equivalent to the original program. As explained, the AOIS (Arithmetic Operator Insertion - shortcut) mutation operator generates up to four mutations. Once AOIS detects a variable of a numeric type, say v, in the original program, it generates: $v++$, $v--$, $++v$, and $--v$. However, applying a post-increment or post-decrement ($v++$, $v--$) to the last use of a local variable does not change the behavior of the original program since the increment/decrement would happen after the method returns. To express this, we define the following *e-rule*:



3.3.2 *d-rule* Example

In Figure 2, the *M2* and *M3* mutants are duplicate. The reason is: the AORB (Arithmetic Operator Replacement) mutation operator replaces basic binary arithmetic operators with other binary arithmetic operators. Once AORB detects an arithmetic operator, say $+$, in the term, it replaces the operator with: $-$, $*$, $/$, and $\%$. The CDL (Constant Deletion) mutation operator removes all occurrences of constant references from every expression. Once CDL detects a constant literal, say $v + 1$, in the term, it generates: v. If AORB and CDL apply to a binary expression with an arithmetic operator (*e.g.*, PLUS (+), or MINUS (-)), and the right-hand side of the binary expression is the constant ONE (1), and the AORB transformation replaces the original operator by TIMES (*) or DIVIDE (/), and the CDL transformation removes the constant ONE, then the mutants are duplicate. To express this, we use the following *d-rule*:



3.3.3 General information: *i-rules*

We name an *e-rule* with the mutation operator, followed by a number, because the same operator can have several *e-rules*. And in brackets we inform the mutation tool, since the tools implement a different set of mutation operators and even the same mutation operator has different transformations in each mutation tool. The name of a *d-rule* differs from an *e-rule* only because it involves more than one mutation operator or more than one transformation in the same mutation operator.

In a *d-rule*, only one transformation is required, unlike in an *e-rule*, where no transformation is required. By applying a *d-rule* right before the mutant's generation, we can avoid duplicate mutants and save the computational cost of generating the mutant and executing the test set against it. By applying an *e-rule* right before the mutant's generation, we can avoid equivalent mutants, and besides saving computational cost, we also save human resource time necessary to check if the surviving mutant is indeed equivalent.

The *i-rules* can be seen as a common notation to *avoid* the generation of equivalent (*e-rule*) and duplicate (*d-rule*) mutants by the mutation operators. Besides, it is a low-cost solution that is easy to incorporate into mutation testing tools when compared to previous works. However, we realize it is a challenge to reason about the possible useless mutants before the generation. In particular, when we need to consider the various combinations of mutation operators and all possible transformations. For that reason, we present a strategy in the next section to facilitate the discovery of these transformation rule improvements.

3.4 STRATEGY

In this section, we propose a strategy that facilitates the discovery of *e-rules* and *d-rules* in mutation testing tools. In what follows, we first depict the strategy (Section 3.4.1), then we instantiate the strategy with three different mutation testing tools (Section 3.4.2).

3.4.1 Identifying Useless Mutants Candidates

To use the strategy, we need the following input parameters: a set of programs, a mutation testing tool, and oracles to indicate equivalent and duplicate mutant candidates.

Figure 3 summarizes the strategy.

- **Step 1.** We begin by providing a set of programs. Any program(s) is supported; however, we recommend these programs are preferably small because we further need to analyze them manually to extract the *i-rules*.

- **Step2.** For each program, a mutation testing tool generates mutants. We recommend enabling all mutation operators, thus allowing the generation of the full set of mutants.

- **Step 3.** Oracles must examine the original program and its mutants to identify potential equivalent and duplicate candidates. Any solution that automatically detects or suggests the equivalent and duplicate mutants may be employed.

- **Step 4.** A program gathers the results from Step 3 and clusters the candidates according to the mutation operators and transformations applied, and sort these groups according to the number of candidates. For instance, if the strategy detects many candidates of duplicate mutants with transformations of the ROR (Relational Operator Replacement) and COI (Conditional Operator Insertion) operators, the program clusters these mutants candidates and hints this group as a priority for manual analysis.

- **Step 5.** We manually analyze the clusters with more candidates to extract the *i-rules*. We reason that clusters with more candidates may indicate an occurrence pattern that leads to the identification of an *e-rule* or *d-rule*. To perform the manual analysis, we perceive all candidates in a cluster, and if identified a pattern, we perform a *manual slicing*. It means we create a simple program that isolates the components of the program involved with the mutation. We then execute tests (manually or automatically generated) against this program to bring up behavioral changes. If no behavioral changes are detected, we work to extract the *i-rule* that prevents this useless mutant from occurring again.

Computing tools easily automate steps 2, 3, and 4. Steps 1 and 5, currently, need manual support. Notice that our strategy is flexible in the sense we can collect programs, mutants, and oracles from different tools, with different configurations (e.g., restrict to a subset of mutation operators, or use compiler optimization as an oracle (KINTIS et al., 2018)). As a result, variations on any of the above steps can help discover different new *i-rules*.

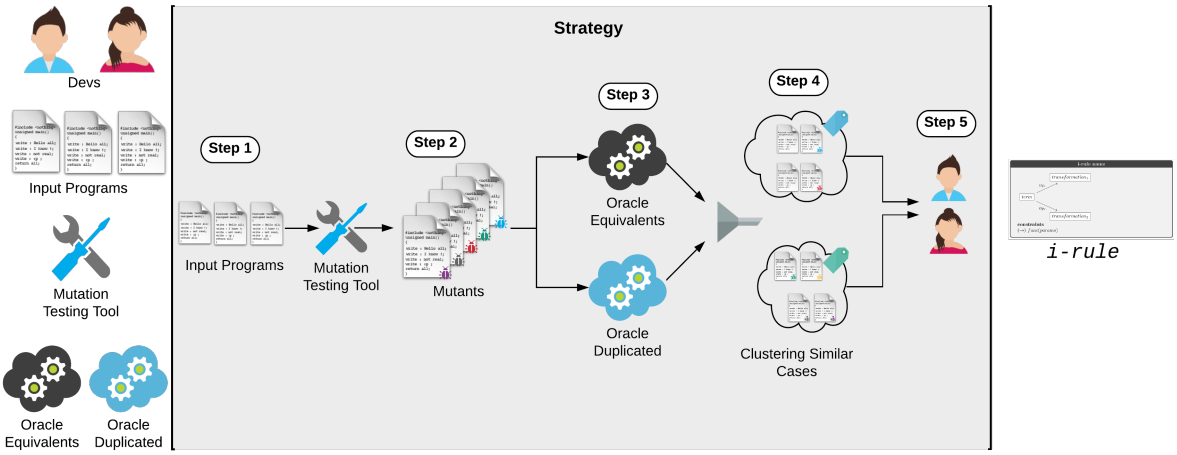


Figure 3 – Strategy to support the discovering of new *i-rules*.

Our goal with strategy is to propose a systematic way to look for occurrence patterns and consequently discover new *i-rules* that can avoid useless mutants in mutation testing tools. Despite being a costly process, once discovered an *i-rule*, the results attained pay off.

This is certainly not the only way to discover new *i-rules*. A very experienced developer may be able to extract many *i-rules* without executing our strategy. This approach requires a thorough understanding of the programming language semantics and the various aspects of a language. However, proving transformation correctness for the entire language constitutes a challenge (as well as refactoring (SCHäFER; EKMAN; MOOR, 2009)). Besides, transformations made by mutation operators usually have incomplete specifications, which makes it difficult to think about all possible mutants. Also, leaving the entire process of reasoning about transformation rules in the hands of the developer is error-prone. For instance, previous research found bugs in very sophisticated refactoring tools (SOARES; GHEYI; MASSONI, 2013; MONGIOVI et al., 2017).

Another option would be to formally verify the transformations. For instance, using the Z3 theorem prover.[4] The expressiveness of languages like JAVA, as well as the complexity of the modeled systems, makes full formalization an expensive and challenging task. A possible way in this direction can use a reduced version of the language like FEATHERWEIGHT JAVA (IGARASHI; PIERCE; WADLER, 1999) and work on *weak mutation*. Nevertheless, in addition to not focusing on the complete semantics of the language, the mutation tool developers would need a very specific knowledge to formalize the transformations.

Our strategy uses real programs as input parameters and considers the complete semantics of the language. Besides, by identifying occurrence patterns, the strategy reduces the task of reasoning about the transformation rules, which can be very difficult when it involves more than one mutation operator.

Next we instantiate our strategy with a set of JAVA programs, three mutation testing

[4] <https://github.com/Z3Prover/z3>

tools for Java (MuJava, Major, an Pit), and a proposed solution to suggesting useless mutants.

3.4.2 Instantiating the Strategy

The goal of our experiment consists of analyzing our strategy for the purpose of discovering new *i-rules* with respect to MuJava, Major, and Pit mutation testing tools from the point of view of mutation tool developers in the context of Java programs automatically generated (first round) and developer written (second round).

We address the following research question:

RQ. *How many i-rules can be discovered with the support of our strategy?*

We compute the number of *e-rules* and *d-rules* found for the three mutation testing tools. The answer to this question enables us to reason about the practical feasibility of the strategy.

In what follows, we describe the settings of the conducted experiment and then we discuss the results.

3.4.2.1 Settings

To answer the question raised above, we executed our strategy in two different rounds. The main difference between each round occurred in the input program set.

For the first round, we parameterized our strategy with a set of input programs automatically generated by JDolly (soares; gheyi; massoni, 2013); JDolly is an automated and bounded-exhaustive Java program generator based on Alloy, a formal specification language (jackson, 2012). JDolly receives as input an Alloy specification with the scope, which is the maximum number of elements (classes, methods, fields, and packages) that the generated programs may declare, and additional constraints for guiding the program generation. JDolly uses the Alloy Analyzer tool (jackson; schechter; shlyahter, 2000), which takes an Alloy specification and finds a finite set of all possible instances that satisfy the constraints within a given scope. JDolly then translates each instance found by the Alloy Analyzer to a Java program. It reuses the syntax tree available in Eclipse JDT for generating programs from those instances. We set JDolly to generate programs with at most three fields (with or without overwriting), three methods (with or without overwriting and overloading), and two classes. One class always extends the other one. We also set JDolly to use only the `int` and `long` primitive types.

For the second round, we decided to use developer-written Java files extracted from open-source repositories. First, we searched for popular Java projects in the GitHub repository. Through the Github search API, nine projects were extracted. We selected projects tagged as Java language, and we sorted the projects with more stars. We filtered out projects related to tutorials, code samples, job interview tips, and so forth. From

Table 2 – Base projects that served to extract JAVA files used in the second round.

Programs	**Description**
Leakcanary	Leakcanary is a memory leak detection library for Android.
ElasticSearch	Elasticsearch is a distributed RESTful search engine built for the cloud.
Guava	Guava is a set of core Java libraries that includes new collection types, graph library, functional types, an in-memory cache, etc.
Incubator-Dubbo	Apache Dubbo (incubating) is a high-performance, Java based open source RPC framework.
Okhttp	Okhttp is an HTTP & HTTP/2 client for Android and Java applications.
RxJava	RxJava is a library for composing asynchronous and event-based programs using observable sequences for the Java VM.
SpringBoot	Spring Boot facilitates the creation of stand-alone, production-grade applications and services with absolute minimum fuss.
Spring Framework	Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications.
Retrofit	Retrofit is a type-safe HTTP client for Android and Java.
The Benchmarks Game	The benchmarks game is a project for comparing toy benchmark programs implemented in different programming languages.

each project selected, we randomly selected five JAVA files; this resulted in a total of 45 files to analyze.[5] Five additional files for this second round were extracted from the *TheBenchMarkGame* website[6]. This site benchmarks programs in different programming languages. Table 2 presents the projects with a description of the project domain.

In Step 2 of the strategy, we needed a mutation testing tool. In both rounds we used the same set of tools to generate the mutants, namely, MUJAVA (MA; OFFUTT; KWON, 2005; OFFUTT; MA; KWON, 2006), MAJOR (JUST; SCHWEIGGERT; KAPFHAMMER, 2011), and PIT (PITEST, 2017). These tools are popular in the mutation testing community. Each one adopts a different set of mutation operators and mutant generation techniques. In the Section 2.2, we depict the main features of these mutation testing tools. Although these are very mature tools, they do not have embedded solutions to solve some of the inherent cost problems of useless mutants. Some tools have code to avoid a few equivalent mutants, such as MUJAVA that avoids the transformation that excludes the reserved word `this`, if the code does not have a local variable with the same name. However, no tool does this continuously, especially not by focusing on duplicate mutants.

In Step 3 of the strategy, we needed oracles to classify the mutant as equivalent or duplicate candidate. To perform this task, we decided to use a set of automatically generated tests. Automated generation of tests is a broad field of research (LAKHOTIA; MCMINN; HARMAN, 2009; SHAMSHIRI et al., 2015; FRASER et al., 2015). Researchers have explored different approaches to automatically generate unit tests, such as random test generation, constraint solver, symbolic execution, and genetic algorithms. Tools such as

[5] We sought for files that had little or no dependence on external classes. Classes with many external dependencies would make manual parsing difficult.

[6] <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/java.html> - accessed on October, 2019.

EvoSuite (FRASER; ARCURI, 2011), Randoop (PACHECO et al., 2007), and IntelliTest (LI et al., 2016) implement such approaches. Although not yet widely adopted by the industry, these automated unit test generation tools have become very useful in generating input data that achieve high code coverage (FRASER et al., 2015) and find real faults (SHAMSHIRI et al., 2015). Our idea was to generate a massive set of tests based on the original program and execute against the mutants, in an attempt to bring up the behavior change caused by the transformation. To classify the mutants as equivalent or duplicate candidates, we proceeded as follows: given a program P, when its test suite T executes against a mutant M (generated from P), and T does not have any failing test case, M seems not to change the behavior of the original program P. This way, the solution sets M as an equivalent mutant candidate. If two mutants generated from P (M_i and M_j) have the same non-empty set of failing test cases in T, the strategy sets M_i and M_j as duplicate mutant candidates. We are classifying mutants as useless or useful according to *strong kill*, i.e., for a given program P, a mutant M of program P is said to be killed only if mutant M gives a different observable behavior from the original program P.

In order to generate a set of massive tests for the programs, we used RANDOOP (PACHECO et al., 2007) and EVOSUITE (FRASER; ARCURI, 2011).

- RANDOOP generates unit tests for JAVA using a feedback-directed random test generation. It randomly generates sequences of method/constructor invocations for the classes under test and creates assertions that capture the actual behavior of the program within a time limit specified by the user. RANDOOP is typically used to create regression tests to warn when the program's behavior changes.

- EVOSUITE is a search-based tool that uses a genetic algorithm to automatically generate test suites for JAVA classes. It takes a hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. It also allows a combination of different coverage criteria at the same time (e.g., branch coverage and an exception coverage). EVOSUITE can be used on the command line, or through plugins for popular development tools such as IntelliJ, Eclipse, or Maven.

In both tools, we used the default configuration. However, both require a time limit for generating the tests. In the first round, with the programs generated by JDOLLY, we made small executions and observed that three seconds was enough to RANDOOP generates a set of 400-500 unit tests. For the second round, the size and complexity of the programs varied more, so we limited to 30 seconds the time to generate the tests with both RANDOOP and EVOSUITE. It allowed RANDOOP to generate around 3,000 tests for each program, and EVOSUITE, due to its search-based nature, generated more varied sets depending on the number of methods, conditionals, etc. of the original program.

Although the set of programs used in our experiment are small or made up of just a single JAVA file, these programs have a variety of language constructs and make use of

Table 3 – Useless mutants candidates identified.

Mutation Tool	First Round			Second Round		
	Mutants	Equivalents	Duplicated	Mutants	Equivalents	Duplicated
MuJava	3,170	592 (18.6%)	1,089 (34.3%)	5,846	875 (14.9%)	509 (8.7%)
Major	816	166 (20.3%)	83 (10.1%)	7,212	1,170 (16.2%)	428 (5.9%)
Pit	1,013	205 (20.2%)	160 (15.7%)	6,769	1,949 (28.7%)	351 (5.1%)
TOTAL	**4,999**	**963 (19.2%)**	**1,332 (26.6%)**	**19,827**	**3,994 (20.1%)**	**1,288 (6.5%)**

different Java language APIs. One might wonder this is a reduced scope when compared to industrial-scale systems. Nevertheless, this choice makes our entire evaluation feasible, since we wanted to explore all mutation operators from all mutation tools (47 from MuJava, 9 from Major, and 13 from Pit). Besides, simple programs facilitate manual parsing and minimize noise when reading and understanding the original and mutated programs. At this stage of the study, we were interested in verifying the capacity of the strategy to identify useless mutants candidates and support the developers to extract the *i-rules*.

3.4.2.2 Results and Discussion

As mentioned, we evaluated 100 Java programs in two different rounds. Table 3 presents the general results.

In the first round, we submitted 50 JDolly programs to the strategy. The three mutation testing tools generate 4,999 mutants. Our strategy classified 963 (19.2%) mutants as equivalent candidates and 1,332 (26.6%) mutants as duplicate candidates. In the second round, we took 50 developer-written Java files extracted from open-source repositories. The mutation testing tools yielded 19,827 mutants. Our strategy classified 3,994 (20.1%) mutants as equivalent candidates and 1,288 (6.5%) mutants as duplicate candidates.

Results varied greatly from round to round. The MuJava mutants accounted for more than 50 percent of the total generated in the first round. As well as the number of equivalent and doubled candidates. It was an expected result, because in its version four, MuJava has 47 mutation operators, against 9 from Major and 13 from Pit. Besides, MuJava is the only one that offers class-level mutation operators. On the other hand, when we observe the results of the developer-written programs (second round), MuJava failed to generate mutants in some class files because the parser used is outdated. The MuJava parser does not recognize language constructs like generics and annotations; thus, mutations didn't occur for some of the files analyzed. The Major tool generated only 816 mutants in the first round, which represents a few more than 16% of the total, while in the second round Major generated 7,212 (36.4%) out of 19,827 mutants. It is because Major had no problem parsing all files and generating the mutants.

We used automatic test generation tools to suggest useless mutant candidates. This approach can lead to false positives. In other words, the automatically generated tests were not enough to cover the code and expose a behavior change. Thus, a mutant is marked as

```java
public class ClassId_1 {
   protected long fieldId_0;
   protected int fieldId_1;
   public long methodId_1(long a) {
      if (fieldId_0 > a)
         return fieldId_0 = fieldId_1 + 1;
      return a;
   }
}
```

Figure 4 – Example of program generated by JDolly.

equivalent or duplicate, but indeed it is not. These cases occurred mostly in the second round because the complexity of the programs made it difficult to generate an excellent test suite. Nonetheless, as we order the most common cases to be manually analyzed first, these false positives became more common near the end when we analyzed clusters with a lower frequency of candidate mutants, which, in general, do not reveal any occurrence pattern.

After clustering and sorting the most frequent cases of equivalent and duplicate candidates, we started the manual analysis. In total, we manually analyzed 4,957 mutants marked as equivalent and 2,620 mutants marked as duplicate. Two researchers separately analyzed each case, trying to find the occurrence patterns that lead to extract an *i-rule*. Once identified, the researchers removed all Java constructs from input programs that are not related to the mutations. With a simple program in hand, the researchers create the mutant and execute tests against the original program and the mutant. If no behavioral change is detected, then they work to extract the *i-rule*. The manual analysis is the hardest part of the strategy. Extracting *i-rules* is not a trivial task, especially when considering developer-written programs (second round). The input programs in the first round were generated by JDolly, where we had partial control over the language constructs. This enables us to identify patterns and extract *i-rules* more easily. In this round, the researchers took an average of 10 to 20 minutes from analyzing a cluster to extracting an *i-rule*. We found 48 *i-rules*. 10 out of the 48 were *e-rules* and 38 out of the 48 were *d-rules*. The *i-rules* identified were extracted from more common language constructs and manipulating primitive data types (eg., `x = x + 1`, or `x = y`, or `return x`, or `x = m_1()`). Figure 4 presents an example of a program generated by JDolly and used in the first round.

On the other hand, only in the second round, with developer-written programs, we were able to identify patterns and extract *i-rules* related to more specific API. For instance, in the second round we found *i-rules* related to Java collections API, `StringBuffer`, and bit shift operations. In this round, the researchers took an average to 30-40 minutes to extract an *i-rule*. We found 51 *i-rules*, 20 out of 51 were *e-rules* and 31 out of 51 were *d-rules*.

Answer to RQ: At the end of the two rounds, we identified 30 *e-rules* and 69 *d-rules* in all three mutation testing tools, with the following distribution:

- MuJava: 16 *e-rules* and 36 *d-rules*
- Major: 9 *e-rules* and 14 *d-rules*
- Pit: 5 *e-rules* and 19 *d-rules*

Our results point out that there are possibilities to improve transformations to avoid useless mutants in all tools.

In the second round, the oracles based on automatic test generation tools against methods with complex external dependencies led to false positives. Dependency objects might be challenging to instantiate. Such methods require extra work from the test generation tools since they need to create *mocks* (ARCURI; FRASER; JUST, 2017) of these objects or to discover valid constructors (which in turn may have other dependencies). In these cases, a sound solution like TCE (*Trivial Compiler Equivalence*) (KINTIS et al., 2018), which is a solution based on compiler optimization, can be a better option. Another point worth noticing is the use of automatic program slicing. We reduced more sophisticated programs to simpler ones manually. Program slicing can help in this task of identifying patterns.

Next, we present some of the *i-rules* found from this experiment. Before that, we discuss possible threats to the validity of the findings of the present study.

3.4.2.3 Threats to validity

Although the study's objective was to verify the feasibility of the strategy, like any empirical study, threats can affect the validity of the results achieved.

The set of Java programs we used is a threat to external validity. To alleviate this problem, we performed two rounds with automatically generated programs and programs written by developers. This setup helped us to derive 99 new *i-rules*. Besides, the small programs allowed us to enable all available mutation operators from the three mutation tools we used in this paper.

Despite instantiating and executing the strategy only in Java programs, there is nothing particular to Java in our strategy. It means that it does not depend on the programming language. To use our strategy in a programming language p, all we need to do is to have a set p of input programs, a mutation testing tool that works with p, and oracles to detect or to suggest useless mutants.

Faults in embedded software can also be a threat. For instance, automatic test generators or mutation testing tools may have faults. In turn, such faults would push down the results.

Figure 5 – Code snippets extracted from programs used in the execution of the strategy.

Thus, it is unlikely that these faults would influence our results to a great extent.

Our solution to suggest useless mutants candidates can lead to false positives. This threat may lead to the construction of a wrong *i-rule*. To minimize this threat, we relied on manual inspection by two different researchers. Besides, all our subjects, tools, and data are available on the companion website of the present paper (FERNANDES et al., 2020). It helps to reduce all the threats mentioned above since independent researchers can check, replicate, and analyze our findings.

3.5 DISCOVERED *I-RULES*

In the previous section, we proposed a strategy that helps developers of mutation tools discover the so-called *i-rules*. We experimented the strategy with a set of 100 programs

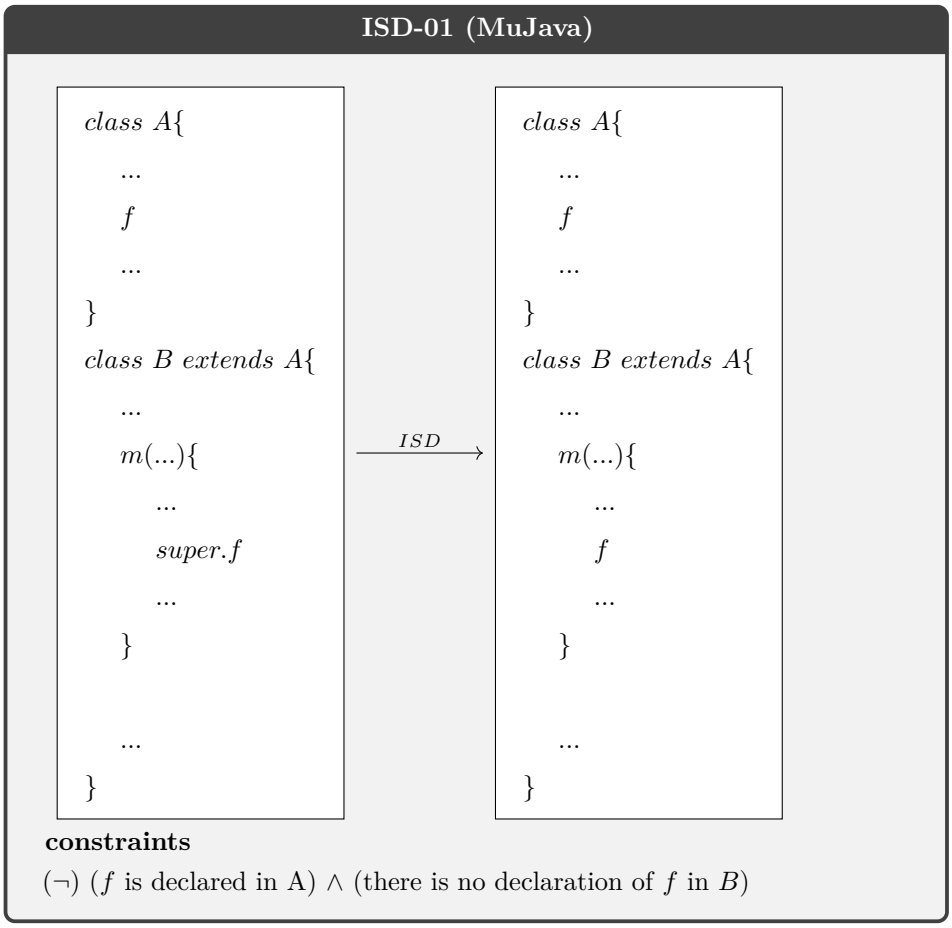
and discovered 30 *e-rules* and 69 *d-rules* in three different JAVA mutation tools; MUJAVA, MAJOR, and PIT. In this section, we depict several examples of *i-rules* derived from the results of executing the strategy. Due to space constraints, we detail just a few of the identified *i-rules*. The complete list is at the article's companion website (FERNANDES et al., 2020).

3.5.1 *e-rules*

In what follows, we present three *e-rules* found during the execution of the strategy. To better explain the *e-rules* identified, we refer to four code snippets in Figure 5 (a–d). The left box represents a code snippet from the original program. The subsequent boxes present examples of equivalent mutants. The mutation operator applied is at the top of the boxes, and the transformations are highlighted.

3.5.1.1 ISD-01 (MUJAVA)

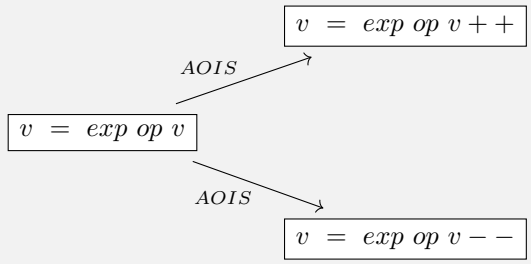
Figure 5(a) illustrates an example of an equivalent mutant derived from ISD (super keyword deletion), a MUJAVA mutation operator. The ISD deletes the `super` keyword from the $super.f_0$ term. In case f_0 exists only in the superclass, the *e-rule* would prevent the mutation testing tool from applying some transformations, avoiding equivalent mutants. To express this, we defined the following *e-rule*:



3.5.1.2 AOIS-02 (MuJava)

Next, Figure 5(b) presents a second example of an equivalent mutant from the MuJava tool. As explained in Section 3.3.1, the AOIS operator inserts post-increment, post-decrement, pre-increment, and pre-decrement operators into a numeric variable. However, applying a post-increment or post-decrement (p_0++, p_0--) generates equivalent mutants if the transformation occurs in the last use of a variable in a right-hand side (RHS) of an assignment statement, and the same variable is on the left-hand side (LHS) of the assignment. To express this, we defined the *e-rule*:

AOIS-02 (MuJava)



constraints

(¬) (v appears in the LHS and RHS of an assignment statement) ∧ (v on the RHS is the last use of v in the statement)

3.5.1.3 ROR-01 (Major)

We now present an *e-rule* extracted from the Major mutation tool (but it is also present in MuJava). Figure 5(c) shows a code example to better explain this *e-rule*. The ROR (Relational Operator Replacement) mutation operator replaces relational operators with other relational operators and replaces the entire predicate with `true` and `false`. However, in case one side of the relational expression is a `String.length` or an `Array.length` and the other side is the literal constant `0`, then some transformations are useless, as the size of a string or array cannot be negative. To express this, we defined the following *e-rule*:

ROR-01 (Major)

$if(sv.length\ op_1\ 0)\{...\} \xrightarrow{ROR} if(sv.length\ op_2\ 0)\{...\}$

constraints

(¬) ($op_1 \in \{<\} \wedge op_2 \in \{!=\}$) ∨
($op_1 \in \{!=\} \wedge op_2 \in \{<\}$) ∨
($op_1 \in \{>\} \wedge op_2 \in \{!=\}$) ∨
($op_1 \in \{!=\} \wedge op_2 \in \{>\}$) ∨
($op_1 \in \{==\} \wedge op_2 \in \{<=\}$)

3.5.1.4 AOR-01 (Major)

Figure 5(d) presents another example of an equivalent mutant derived from the Major mutation tool (but it is also present at MuJava). The AOR (Arithmetic Operator Replacement) mutation operator replaces one arithmetic operator by others. However, applying the AOR transformation inside a `StringBuilder` constructor can generate equivalent mutants. The `int` parameter inside the `StringBuilder` constructor allocate the internal buffer. If the internal buffer overflows, it is automatically extended. Then, the resulted value inside the constructor does not change the behavior of the program, as long

as *constraints* hold: the resulted value cannot be less than zero (the constructor raises `NegativeArraySizeException`).

To express this, we defined the following *e-rule*:

AOR-01 (Major)

$$\boxed{new\ StringBuilder(exp_1\ op_1\ exp_2)} \xrightarrow{AOR} \boxed{new\ StringBuilder(exp_1\ op_2\ exp_2)}$$

constraints

($\neg$) (op_2 does not turn the resulted value to negative)

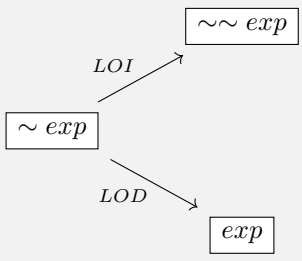
3.5.2 *d-rules*

In what follows, we present three *d-rules* found during the execution of the strategy. To better explain the *d-rules* we identified, we refer to four code snippets in Figure 5 (namely, 5(e) to 5(h)). The left-hand side represents a code snippet from the original program. The two right-hand side boxes present examples of duplicate mutants. Different from an *e-rule*, that states that no transformation should be applied, a *d-rule* states that only one of the transformations should be applied.

3.5.2.1 LOI|LOD-01 (MuJava)

Figure 5(e) presents an example of a duplicate mutant derived from MuJava. The code snippet contains a bitwise complement operator ($\sim$), which flips bits (e.g., 00000010 becomes 11111101). The LOI (Logical Operator Insertion) mutation operator inserts a bitwise complement operator before the expression, and the LOD (Logical Operator Deletion) mutation operator removes the bitwise complement operator. This way, if the original expression already contains this operator ($\sim$`f_0`), then, by applying LOI ($\sim\sim$`f_0`) and LOD (`f_0`) the mutants will be duplicates. We define a *d-rule* to identify this case as follows.

LOI|LOD-01 (MuJava)

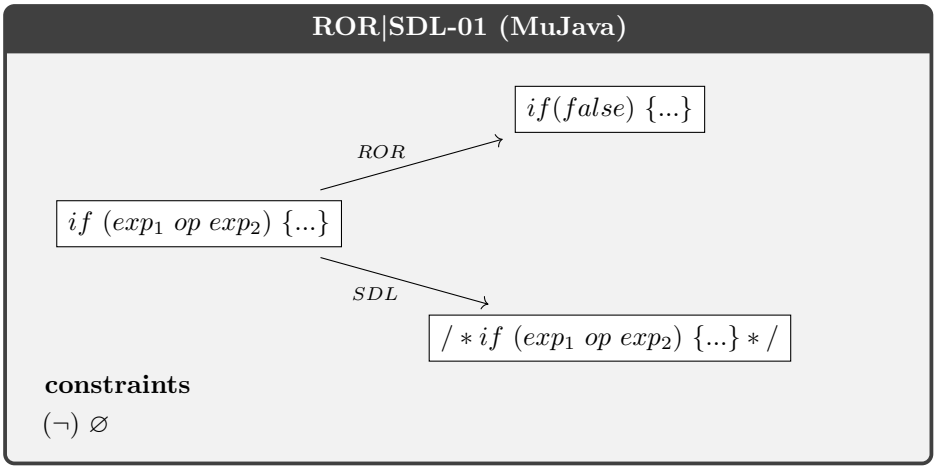


constraints

($\neg$) $\varnothing$

3.5.2.2 ROR|SDL-01 (MuJava)

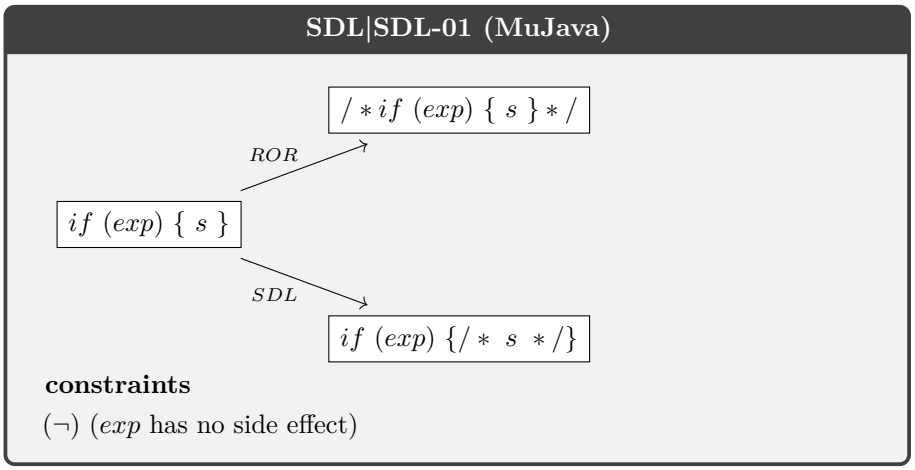
Figure 5(f) illustrates an example of a duplicate mutant generated from a transformation of ROR and SDL mutation operators. In case the original code snippet contains an `if` statement, the ROR operator (Relational Operator Replacement) sets the boolean expression to `false` (i.e., the `if` body will not be executed), whereas the SDL operator (Statement Deletion) deletes the entire `if` statement. In this context, these mutants are duplicate. We do not define constraints beyond those already required by the definition of *term* and $transformations$. To express this, we define the following *d-rule*:



3.5.2.3 SDL|SDL-01 (MuJava)

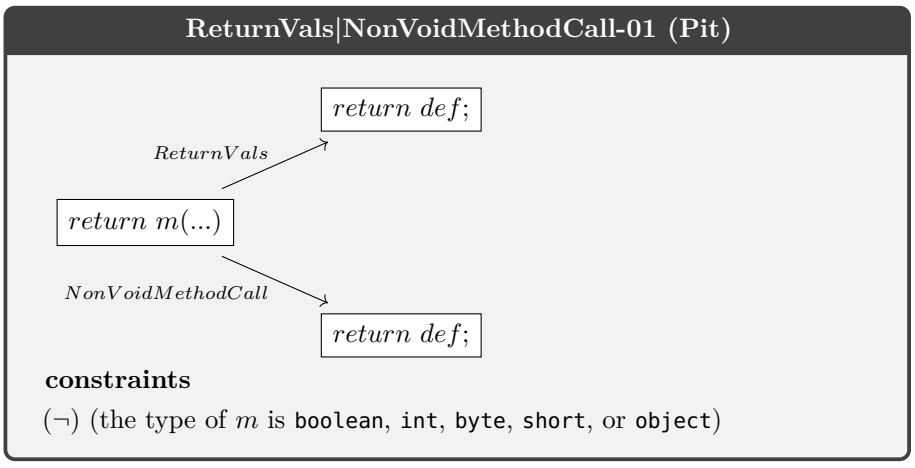
For the next *d-rule*, we avoid duplicate mutants that are generated by the same mutation operator. This is illustrated in Figure 5(g). In the first mutant, the SDL operator (Statement Deletion) deletes the entire `if` statement; in the second one, it deletes the only statement within the `if` body. Notice that both mutants are duplicate, as long as *constraints* hold: the `if` expression has no side effect[7]. In this rule, *term* matches an `if` statement containing only one statement without `else`.

[7] A side effect expression, besides calculating a value, modifies the state (i.e., the memory), such as by an assignment (simple or combined) or an increment/decrement.



3.5.2.4 ReturnVals|NonVoidMethodCall-01 (Pit)

Figure 5(h) presents an example of duplicate mutants derived from Pit tool. The ReturnVals operator (Return Values Mutator) mutates the return values of method calls. The NonVoidMethodCall operator (Non-Void Method Call Mutator) removes calls to non-void methods. In case the expression contained in the return statement is a method call, then the value is replaced by the Java default value for that specific type in both mutation operators (it works for: `boolean`, `int`, `byte`, `short`, or `object`). To express this, we define the following *d-rule*.

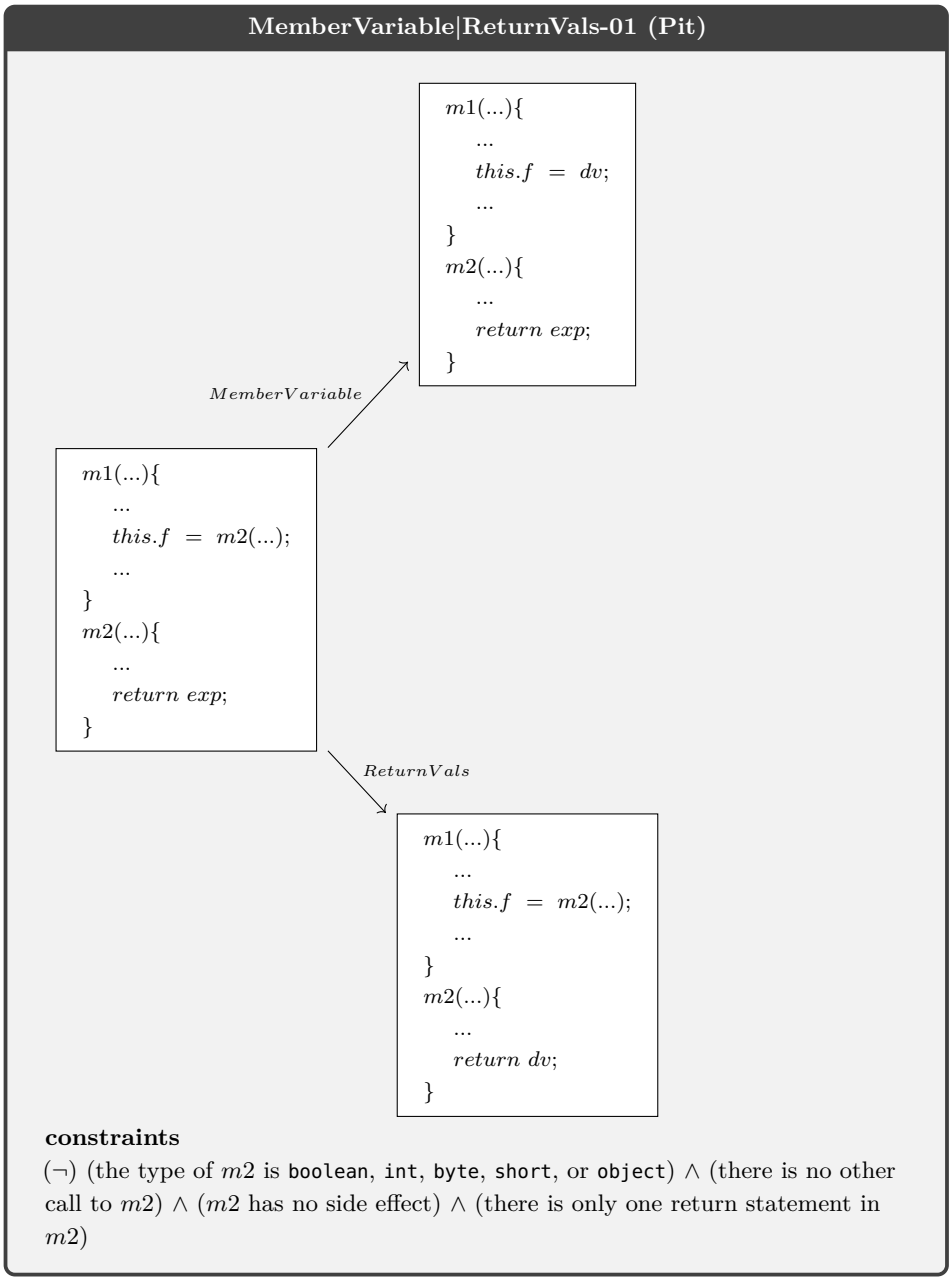


3.5.3 Implementing the *i-rules*

To be effective, the *i-rule* needs to be implemented in a mutation testing tool so that we can see a reduction in useless mutant numbers, and consequently, a reduction in computational and human resource costs. Our findings show that much of the *i-rules* we discovered can be easily implemented using the resources already available in the mutation tools. However, some *i-rules* identified need more than just navigating through the Abstract Syntax Tree (AST) to be implemented. They require advanced static analyses,

like *def-use* analysis (NIELSON; NIELSON; HANKIN, 1999).

To illustrate this scenario, consider the code snippet presented in Figure 5(i). The MemberVariable (Member Variable Mutator) mutation operator replaces member variable assignments with default values. The ReturnVals (Return Values Mutator) mutation operator, on the other hand, mutates the return value of methods. If there is an assignment to a member variable and on the RHS of the assignment there is a method call; the mutants can be duplicate as long as *constraints* hold: there is no other call to the method, the called method has no side effect, and there is only one flow to return in the called method. The constraints imposed to ensure these mutants are duplicates require advanced code analysis. However, mutation testing tools usually do not have this type of static analysis. To express the case mentioned above, we define the following *d-rule*:



We identified the majority of the *i-rules* by using the MuJava mutation testing tool. In addition, when compared to Major and Pit, MuJava achieves one of the best results on simulating real faults (KINTIS et al., 2016; KINTIS et al., 2018). This way, we selected MuJava version 4 to implement our *i-rules*[8]. We have decided to implement only the *i-rules* that are possible using the same resources available in MuJava. That is, we used the same libraries to navigate throughout the Abstract Syntax Tree (AST) of a given program. *i-rules* that need more advanced static analysis to be implemented, like *def-use* analysis, are out of the scope of this work. For this work, we have implemented 32 *i-rules* (9 *e-rules* and 23 *d-rules*). We name this tool MuJava-AUM (AUM stands for *Avoiding Useless Mutants*). Therefore, MuJava-AUM is a new version of MuJava with 32 *i-rules* implemented to prevent useless mutants. In the next section, we present an empirical study to evaluate the *e-rules* and *d-rules* implemented in MuJava-AUM.

3.6 EVALUATING THE IMPLEMENTED *I-RULES*

In this section, we present our experiment to evaluate the cost reduction of mutation testing with the implemented *i-rules*. First, we introduce the goal, the research questions, and metrics used in our study (Section 3.6.1). Thus, Sections 3.6.2 and 3.6.3 detail the experimental setup and the procedure to reproduce the study. Lastly, Sections 3.6.4 and 3.6.5 discuss the results and the threats to validity.

3.6.1 Goal and Research Questions

We have structured the experiment definition using the Goal-Question-Metric (GQM) (Basili; Rombach, 1988). The GQM template of the experiment is summarized in Figure 6.

The purpose of this study is to evaluate the cost reduction of mutation testing in industrial-scale systems by avoiding the generation of useless mutants with the implemented *i-rules* in MuJava. In particular, our experiment addresses the research questions **RQ1**, **RQ2**, and **RQ3** in Figure 6.

To answer **RQ1**, we execute MuJava and MuJava-AUM and log all the generated mutants. Then, we get all useless mutants generated by MuJava but identified and avoided by MuJava-AUM. This step is essential to validate our *i-rules* implementation and to confirm whether the avoided mutants are indeed useless or not. We then compute the reduction in the number of equivalent and duplicate mutants avoided by MuJava-AUM.

To answer **RQ2**, we verify the proportion of useless mutants avoided by each *e-rule* and *d-rule* in comparison to the total of avoided mutants. This way, we identify the most useful *i-rules* and those unnecessary, since they are rarely applied.

[8] <https://github.com/jeffoffutt/muJava>

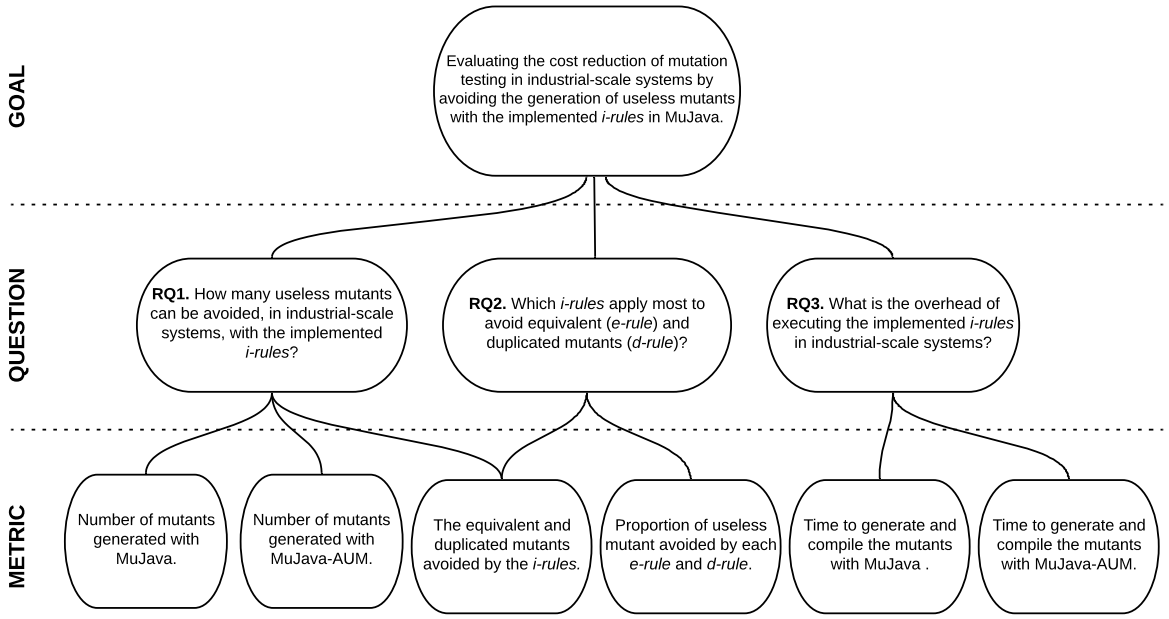


Figure 6 – Goal-Question-Metric Template

Lastly, to answer **RQ3**, we re-execute MuJava and MuJava-AUM, and then we compare the time taken by MuJava and MuJava-AUM to generate and compile the mutants.

3.6.1.1 Subjects

As we derived our *i-rules* from automatically generated or small-sized Java programs, it is important to verify if the *i-rules* implemented can reduce the mutation testing costs in industrial-scale systems. Then, we selected six open-source projects that vary in size and application domain. These projects are commonly used in mutation testing research (KINTIS et al., 2018; PAPADAKIS et al., 2015; JUST et al., 2014; KINTIS; MALEVRIS, 2015). We list the projects in Table 4. To make our analysis feasible, we follow the same procedure applied by Kintis et al. (KINTIS et al., 2018). For each project, we rank all packages according to their size. Then, we select the three largest classes that could be handled without a problem by MuJava[9] among the four largest packages. This way, we selected classes of different sizes. In this particular setup, we focus on 72 classes, 12 per project.

3.6.2 Experimental Setup

As explained, we identified the majority of the *i-rules* by using the MuJava mutation testing tool, and we developed MuJava-AUM, a new version of MuJava with 32 *i-rules*

[9] MuJava 4 uses the OJ 1.1 (<https://www.csg.ci.i.u-tokyo.ac.jp/openjava/>) parser. This parser does not support some Java constructs, such as generics and for-each loops.

Table 4 – Open-source projects used in this experiment.

System	Domain	Classes (LoC)
ant-1.8.4	Build system	DirectoryScanner (855) , AntClassLoader (783), Main (773), taskdefs.Jar (716), taskdefs.Javac (667), taskdefs.Copy (665), util.LayoutPreservingProperties (497), util.FileUtils(711), util.JavaEnvUtils(282), types.Path (427) , types.RedirectorElement (377), types.XMLCatalog (555)
bcel-5.2	Bytecode manipulation	classfile.ConstantPool(201), classfile.DescendingVisitor (277), classfile.JavaClass (537), generic.ConstantPoolGen (501), generic.InstructionFactory (563), generic.MethodGen (721), util.BCELFactory (263),~ util.InstructionFinder (225), util.CodeHTML (430),~ structurals.InstConstraintVisitor (1777), structurals.ExecutionVisitor (942), structurals.OperandStack (154)
commons-lang-2.4	Java core classes	ArrayUtils (1824), Entities (380), StringUtils (2001), builder.CompareToBuilder (450), builder.HashCodeBuilder (323), builder.ToStringBuilder (328), math.DoubleRange (189), math.Fraction (461), math.NumberUtils (715), text.StrBuilder (1363), text.StrSubstitutor (328), text.StrTokenizer (481)
commons-math-1.2	Math library	analysis.MullerSolver (190), ode.GraggBulirschStoerIntegrator (509), analysis.BrentSolver (130), distribution.BinomialDistributionImpl (83), linear.MatrixUtils (111), ode.GraggBulirschStoerStepInterpolator (193), linear.RealMatrixImpl (583), distribution.NormalDistributionImpl (102), linear.BigMatrixImpl (724), DormandPrince853StepInterpolator (186), analysis.LaguerreSolver (179), distribution.CauchyDistributionImpl (93)
h2-1.0.79	Database application	tools.Shell (557), tools.SimpleResultSet (1337), jdbc.JdbcConnection (1432), jdbc.JdbcResultSet (2433), jdbc.JdbcDatabaseMetaData (1300), expression.Aggregate (508), expression.CompareLike (448), expression.Comparison (447), command.CommandRemote (255), tools.Recover (1552), command.CommandContainer (120), command.Parser (6518)
joda-time-2.4	Date and time utility	LocalDate (749), LocalDateTime (829), Period (560) chrono.BasicMonthOfYearDateTimeField (205), chrono.BasicWeekyearDateTimeField (129), ZoneInfoProvider (170), chrono.GJLocaleSymbols (204), format.DateTimeFormat (513), format.DateTimeFormatter (406), format.FormatUtils (291), tz.CachedDateTimeZone (157), tz.FixedDateTimeZone (67)

implemented to prevent useless mutants. This tool is available online.[10] The *i-rules* can be easily enabled or disabled. If the rules are enabled, MuJava-AUM does not generate the useless mutants identified by the *i-rules*. Otherwise, MuJava-AUM generates all mutants, the same way as the original MuJava version does. However, it always labels the useless mutants that would be avoided. For this study, we call these labeled mutants as *useless mutants candidates*. This decision is necessary to validate if the *i-rules* were implemented correctly. To support us in the process of validating the *i-rules*, we submit the original classes and the useless mutants candidates to a sound tool, TCE (Trivial Compiler Equivalence) (KINTIS et al., 2018). Given two Java files (original *versus* mutant or mutant *versus* mutant), TCE applies compiler optimizations and check their bytecode. In case they are the same, TCE guarantees that the files have the same behavior (which

[10] <https://github.com/easy-software-ufal/muJava-AUM>

means they are equivalent or duplicate).

All necessary information to collect the metrics are logged into the file system during MuJava and MuJava-AUM execution. We performed the evaluation on a 2.70 GHz four-core PC with 16 GB of RAM equipped with a Ubuntu 17.10 operating system. In what follows, we detail the procedure to reproduce the experiment.

3.6.3 Procedure

To carry out our evaluation, we execute the GUI version of MuJava and MuJava-AUM. In particular, we execute MuJava-AUM with the *i-rules* disabled. In this configuration, the mutants are generated the same way as the original MuJava, but all equivalent and duplicate mutants are labeled. We proceed in this way to access the mutants that would be avoided, and these mutants are later submitted to the TCE tool and, subsequently, to manual analysis to confirm that they are indeed useless. For each subject, we select the 12 classes to be evaluated (in total 72 classes), then check all method-level and class-level mutation operators available and trigger the mutation tool to generate the mutants. In total, 47 mutation operators are available in version 4 of MuJava.

At the end of each subject's mutant generation, we gather the results. To confirm that the *i-rules* are defined and implemented correctly, we validate through two steps. In the first step, we submit the useless mutants candidates to TCE. In the case of mutants labeled as equivalent, TCE compares the bytecode of the mutant with the original program. Otherwise, in the case of duplicate, TCE compares the bytecode of the mutant with the other mutants. If TCE guarantees that the files are equivalent, we consider that the *i-rule* is correct. For cases TCE cannot confirm that mutants are equivalent or duplicate, we proceed to the second step. In the second step, we manually analyze a subset of those mutants. For each class (out of 72), we randomly select 10% of the useless mutants identified by each *i-rule*. To better explain this selection, suppose we have 100 useless mutants identified by four of our *i-rules* and not confirmed by TCE (50 by *e-rule* A, 30 by *d-rule* B, 10 by *e-rule* C, and 10 by *d-rule* D). In this situation, we randomly select 10% of mutants identified per *i-rule*, i.e., 5 from *e-rule* A, 3 from *d-rule* B, 1 from *e-rule* C, and 1 from *d-rule* D. In this way, our manual analysis comprised all *i-rules* we implemented in MuJava-AUM. This procedure is enough to collect the metrics to answer the **RQ1** and **RQ2**.

To verify the overhead introduced by the *i-rules* and answer **RQ3**, we follow a slightly different approach. We generate and compile the mutants with MuJava and MuJava-AUM, selecting the same six open-source projects to compute the time. But, at this time, we enable the *i-rules*, such that useless mutants do not generate. And we also execute the original MuJava version. In both cases, we execute each subject three times and consider the average time.

Table 5 – Results of executing MuJava-AUM.

Project	Mutants		#(%)	Confirmed by TCE
ant	12,894	Eq.	241(1.87%)	89(36.93%)
		Dup.	2,856(22.15%)	2,344(82.07%)
bcel	17,558	Eq.	352(2.00%)	239(67.90%)
		Dup.	3,053(17.39%)	2,462(80.64%)
commons-lang	34,194	Eq.	1,374(4.02%)	973(70.82%)
		Dup.	6,774(19.81%)	5,783(85.37%)
commons-math	21,970	Eq.	336(1.53%)	258(76.79%)
		Dup.	3,028(13.78%)	2,530(83.55%)
h2	35,698	Eq.	973(2.73%)	812(83.45%)
		Dup.	5,160(14.45%)	4,291(83.16%)
joda-time	12,703	Eq.	754(5.94%)	608(80.64%)
		Dup.	1,822(14.34%)	1,449(79.53%)
TOTAL	**135,017**	**Eq.**	**4,030(2.98%)**	**2,979(73.92%)**
		Dup.	**22,693(16.81%)**	**18,859(83.10%)**

The list of tools, programs, and result data are available on our companion website (FERNANDES et al., 2020).

3.6.4 Results and Discussion

This section presents the results and answers for the research questions.

3.6.4.1 How many useless mutants can be avoided, in industrial-scale systems, with the implemented *i-rules*?

Table 5 presents the general results of executing MuJava and MuJava-AUM. For each subject, we present the name of the project ("Project" column), the number of mutants generated by MuJava ("Mutants" column), the number of equivalent and duplicate mutants that our *i-rules* avoided ("#(%)" column), and the number of useless mutants confirmed by TCE ("Confirmed by TCE" column). These results support us in discussing **RQ1**. In total, the 12 subjects were responsible for 135,017 mutants. MuJava-AUM avoided the generation of 4,030 equivalent and 22,693 duplicate mutants. It represents, respectively, 2.98% and 16.81% of the total generated mutants. TCE confirmed 2,979 (out of 4,030) equivalent mutants and 18,859 (out of 22,693) duplicate mutants. This means that, from 4,030 mutants avoided by 9 implemented *e-rules*, TCE confirmed 73.92%. And from 22,692 mutants avoided by 23 implemented *d-rules*, TCE confirmed 83.10%. This way, we had 4,885 mutants avoided by our strategy but not confirmed by the TCE. We grouped the unconfirmed mutants into sets according to the *i-rule* that was applied. So we randomly selected and analyzed 10% of each set. Two distinct researchers manually analyzed each

mutant. In this manual task, we faced a few bugs in our *i-rules* implementation (e.g., one constraint missing). After fixing these bugs, we re-executed the entire analysis until we confirmed all useless mutants found by the *i-rules*. In the last execution, we did not identify false positives in the random subset we manually analyzed.

Regarding the equivalent mutants, as demonstrated by Table 5, our strategy was able to recognize and avoid 4,030 mutants, which represents 2.98% of the total mutants generated by MuJava. By analyzing each subject separately, we identified characteristics in the original programs that led to the prevalence of some *i-rules* over others. For instance, the subject *joda-time* had the best result in avoiding 5.94% of the equivalent mutants. This subject has an average of 6.66 lines of code per method. It is the lowest among the analyzed subjects. It resulted in an average of 26.40 mutants per method. Besides, *joda-time* date and time manipulation methods use a lot of local variables (parameter variables or method-declared variables) of primitive numeric types. Thus, *e-rules* such as AOIS-01 (MuJava) were applied to a wide variety of program locations. On the other hand, the *commons-math* subject had the worst result; for it, only 1.5% of mutants were avoided after being identified as equivalent. This subject has an average of 13.31 lines of code per method, which caused MuJava to generate an average of 110.95 mutants per method. Thus, even though this subject contains extensive use of numeric primitive type variables, such as *joda-time*, the total number of generated mutants was very high, which decreased the proportion of equivalent mutants avoided by the *e-rules*.

Regarding the duplicate mutants, Table 5 shows that our *d-rules* avoided 22,693 mutants, which represents 16.81% of the total mutants generated. The *ant* subject had the best result in avoiding 22.15% of the duplicate mutants. This subject has an average of 10.87 lines of code per method, which caused MuJava to generate an average of 65.12 mutants per method. By analyzing the *ant* code, we observed many binary expressions with a variable or a constant on the right- or left-hand side. In these cases, respectively, the VDL|ODL-01 (MuJava) and CDL|ODL-01 (MuJava) *d-rules* are usually applied. On the other hand, the *commons-math* subject had the worst result in avoiding 13.78% of the duplicate mutants. By analyzing the *commons-math* code, we identified many locations where the *d-rule* VDL|ODL-01 (MuJava) was applied. However, as explained, this subject has a high number of mutants per method, and in many code locations, we do not have applicable *d-rules*. For this reason, results for this subject were inferior when compared to results for the other subjects.

In summary, despite being derived from artificial or small Java programs, the *i-rules* implemented in MuJava-AUM avoided useless mutants in industrial-scale projects. The results are quite encouraging. In the work of Kintis et al. (KINTIS et al., 2018), researchers summarized several pieces of related work on useless mutants. Only a few provided techniques to *avoid* their generation (ADAMOPOULOS; HARMAN; HIERONS, 2004; MADEYSKI et al., 2014). However, they focused only on equivalent mutants. In this work,

Table 6 – Applied *e-rules*.

e-rule	Avoided Mutants	Confirmed by TCE
AOIS-01 (MuJava)	3,632	2,974(81.88%)
AOIU-01 (MuJava)	205	0(0.00%)
JID-01 (MuJava)	120	0(0.00%)
ROR-01 (MuJava)	64	5(7.69%)
ISD-01 (MuJava)	8	0(0.00%)
TOTAL	4,030	2,979(73.92%)

besides avoiding equivalent mutants, we also provide *d-rules* to avoid duplicate mutants. Finally, we answer **RQ1**.

Answer to RQ1: Considering a whole set of mutants generated by the original MuJava implementation, our *i-rules* avoided the generation of 19.79% of them, all identified as useless mutants. Out of these, 16.81% are duplicate mutants, and 2.98% are equivalent to the original program. This way, we avoid the cost of generation, compilation, and execution of the tests on these mutants. In addition, by eliminating equivalent ones, we also avoid the cost of manually analyzing these mutants.

3.6.4.2 Which *i-rules* are most applied to avoid equivalent and duplicate mutants?

To answer **RQ2**, we analyzed the frequency of each *e-rule* and *d-rule* successfully applied across all projects. Consequently, we can identify which *i-rules* are most prevalent and which could be discarded by the developers of mutation testing tools. In total, we implemented 32 *i-rules* in MuJava, from which 9 are *e-rules* and 23 are *d-rules*. Some *i-rules* are known to be applied in different domains and contexts, such as the AOIS-01 (MuJava) and AOIU-01 (MuJava) *e-rules*. Other *i-rules* depend on a more specific context, such as the AORB-01 (MuJava) *d-rule*, where the analyzed software needs to make use of more specific APIs (`StringBuilder`, in this case).

Tables 6 and 7 presents the results for *e-rules* and *d-rules* subsequently. For each *i-rule* we present its name (first column), the number of mutants avoided (second column), and the number of avoided mutants confirmed by TCE (third column).

Table 6 outlines the *e-rules* that were applied to the subjects studied, sorted in descending order by the number of mutants avoided. By examining the table, we see that only five out of 9 *e-rules* were triggered at least once. Note that the AOIS-01 (MuJava) *e-rule* (explained in Section 3.3.1) avoided most of the equivalent mutants. More precisely, this *e-rule* prevented the generation of 3,632 equivalent mutants, which represents, alone, 90.12% of all equivalent mutants avoided. The AOIS mutation operator is known by

Table 7 – Applied *d-rules*.

d-rule	Avoided Mutants	Confirmed by TCE
ODL|VDL-01 (MuJava)	6,796	6,796(100.00%)
CDL|ODL-01 (MuJava)	5,926	5,926(100.00%)
ROR|SDL-01 (MuJava)	2,487	1,952(78.49%)
COI|ROR-01 (MuJava)	2,214	2,023(91.37%)
SDL|SDL-01 (MuJava)	1,782	178(9.99%)
ODL|AODS-01 (MuJava)	933	933(100.00%)
COD|ODL-01 (MuJava)	484	484(100.00%)
ODL|AODU-01 (MuJava)	388	388(100.00%)
AORB|ODL-01 (MuJava)	381	2(0.52%)
AORB|AORB-01 (MuJava)	372	2(0.54%)
LOI|LOI-01 (MuJava)	273	1(0.37%)
AOIU|AOIU-01 (MuJava)	164	1(0.61%)
AOIU|ASRS-01 (MuJava)	125	0(0.00%)
SDL|VDL-01 (MuJava)	114	114(100.00%)
ROR|ROR-01 (MuJava)	112	50(44.64%)
LOI|ROR-01 (MuJava)	79	0(0.00%)
SOR|SOR-01 (MuJava)	56	2(3.57%)
LOD|ODL-01 (MuJava)	5	5(100.00%)
COD|ROR-01 (MuJava)	2	2(100.00%)
TOTAL	22,693	18,859(83.10%)

the community to generate many mutants, as well as many equivalent mutants (KINTIS; MALEVRIS, 2015). As an example, from the 21,970 mutants generated for the *commons-math* project, the AOIS mutation operator was responsible for 6,880 of them; that is, one mutation operator generates almost one-third of the mutants. Although the AOIS operator produces many equivalent mutants, it also generates many *stubborn* mutants. Such mutants tend to require very specific tests, which in turn is suitable for mutation analysis (YAO; HARMAN; JIA, 2014). Therefore, we believe this mutation operator adds value to the mutation operator tool, and should not be ruled out.

Observing the *d-rules* at Table 7, we see that 19 of them (out of 23) were responsible for avoiding at least one mutant. By closely examining the table, we observed that six *d-rules* accounted for 88.74% of the avoided duplicate mutants (namely, ODL|VDL-01 (MuJava), CDL|ODL-01 (MuJava), ROR|SDL-01 (MuJava), COI|ROR-01 (MuJava), SDL|SDL-01 (MuJava), ODL|AODS-01 (MuJava)), all particularly related to the following mutation operators: ODL (Operator Deletion), SDL (Statement Deletion), and ROR (Relational Operator Replacement).

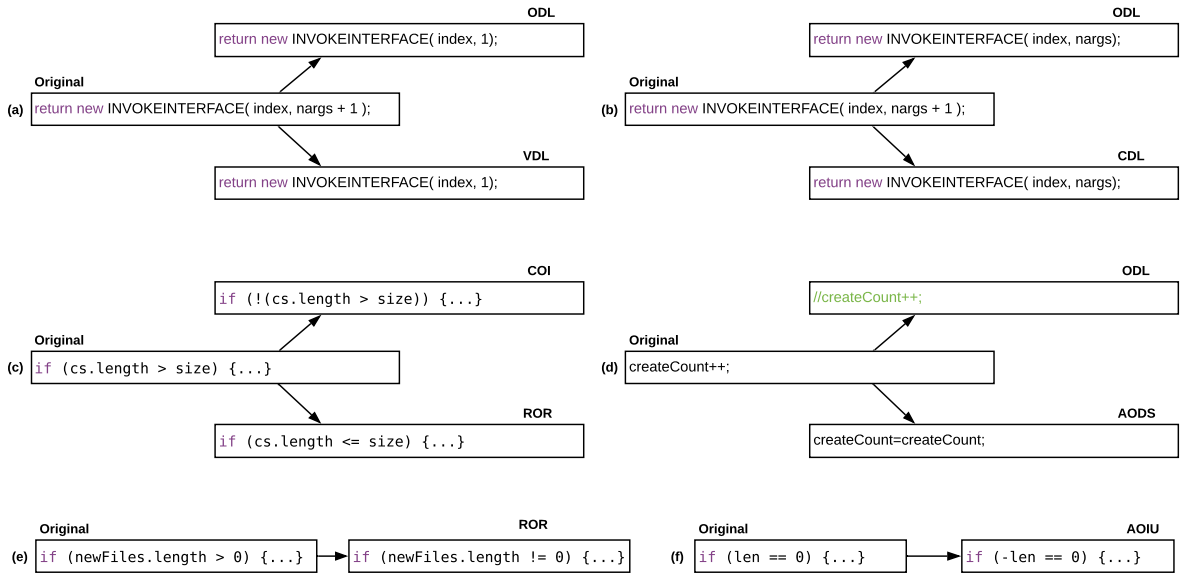


Figure 7 – Code snippets extracted from subjects used in the study.

The ODL|VDL-01 (MuJava) avoided 6,796 mutants. In regard to it, note that the VDL and ODL mutation operators occur, frequently, in binary Java expressions involving mathematical or logical operators. The second most applied *d-rule* was CDL|ODL-01 (MuJava). It is similar to VDL|ODL-01 (MuJava), with the difference that CDL is applied to code constants, while VDL applies to code variables. For instance, Figure 7(a) and (b) show examples of mutants avoided by the aforementioned *d-rules*. The original code and mutants were extracted from the `InstructionFactory` class of the *bcel* project. Next, we have the ROR|SDL-01 *d-rule*. It occurs in `if` statements that have relational operators in their conditional expression (explained in Section 3.5.2.2). The COI|ROR-01 (MuJava) *d-rule* was the fourth most applied. This *d-rule* occurs in conditional expressions involving `==` or `!=`. Figure 7(c) shows an example of the original code and two mutants avoided by COI|ROR-01 (MuJava). This example was extracted from the `ConstantPoolGen` class of the *bcel* project. The SDL|SDL-01 (MuJava) *d-rule* is also present in the list of most applied *d-rules* (explained in Section 3.5.1). The sixth most applied *d-rule* was ODL|AODS-01 (MuJava). This *i-rule* happens mainly when we use pre-increment/decrement or post-increment/decrement Java operators. Figure 7(d) shows an original code and two duplicate mutants avoided by ODL|AODS-01 (MuJava). This example was extracted from the the`Copy` class of the *ant* project.

It is essential to note the number of mutants confirmed by TCE. By examining Table 6, we see that TCE confirmed a high rate of mutants for AOIS-01 (MuJava), but indeed failed to detect more than 18% of the mutants avoided by this *e-rule*. In other cases, TCE numbers were deficient, not confirming any mutants in three *e-rules*. For example, equivalent mutants like the ones shown in Figure 7(e) and (f) have not been confirmed by the TCE.

In Table 7, we can observe that many *d-rules* were confirmed by TCE in 100% of avoided mutants. It is because the duplicate mutants generated have the same source codes. However, TCE failed to detect others. In eight *d-rules*, TCE had a confirmation rate below 10%. This is the case of the SDL|SDL-01 (MuJava) *d-rule* (explained in Section 3.5.2.3) which avoided 1,782 mutants, but only 9.99% were confirmed by TCE.

Designing a new mutation operator for a mutation testing tool needs to be done considering the program context and all the other mutation operators already available in the tool. We observed that the set of mutants generated by a mutation operator could be found spread among the mutations of other mutation operators. For instance, the ODL (Operator Deletion) mutation operator deletes each arithmetic, relational, logical, bitwise, and shift operator from expressions and assignment operators. In case of a binary expressions, *e.g.*, $X = X + 1$, ODL yields $X = X$ and $X = 1$. These transformations are valid and useful in many contexts, but these same mutations spread on other mutation operators like VDL, CDL, and SDL. In some cases, a mutation operator acts as a *superset* of another operator; for instance, ODL is a *superset* of VDL and CDL. In this context, the *subset* operators should be disabled if the *superset* operator is enabled. On the other hand, some mutation operators, such as ODL with SDL, have only one mutation in common. Since it is up to the tester—that is, the mutation testing tool user—to perform a finer (many operators enabled) or grosser (only specific operators enabled) analysis of the system under test, then the developer of the mutation tool generally makes all operators available for analysis. As a result, we believe the *i-rules* represent a viable way to avoid useless mutants. Next, we present the answer to **RQ2**.

Answer to RQ2: Regarding the *e-rules* related to equivalent mutants, we managed to avoid 4,030 mutants. The AOIS-01 (MuJava) *e-rule*, itself, was responsible for the vast majority of avoided equivalent mutants. From the 9 implemented *e-rules*, five were triggered successfully at least once, what means that 4 *e-rules* have not been applied. Regarding the applied *d-rules*, they avoided 22,693 mutants. Six *d-rules* were responsible for approximately 89% of the avoided duplicate mutants. From the 23 *d-rules* implemented, 19 were triggered successfully at least once, whereas and 4 *d-rules* have not been applied. We also found that some *i-rules* are challenging to be detected by TCE, which shows that we can use both solutions in conjunction to reduce even more the number of useless mutants. Note that these results link to the set of programs evaluated, as well as to the implementation of the mutation operators in the MuJava-AUM mutation tool.

3.6.4.3 What is the overhead of executing our *i-rules* in industrial-scale systems?

Now we discuss the research question related to the overhead of executing our *i-rules* concerning time. Table 8 presents the execution time (in seconds) of the original MuJava

version (without the *i-rules*) and MuJava-AUM version (embedded with some *i-rules*). For all systems, our version saved time. The best result came from the *ant* and *bcel* projects. In the *ant* project, on average, it took 36 minutes (2,143 seconds) for the original version of MuJava to generate the mutants, while it took 29 minutes (1,775 seconds, *i.e.,* a reduction of 17.29%) for the MuJava-AUM. In the *bcel* project, it took 38 minutes (2,287 seconds) for the MuJava and 31 minutes (1,891 seconds, (*i.e*, a reduction of 17.32%) for the MuJava-AUM. The *h2* project had the slightest difference between executions. It took about 75 minutes (4,544 seconds) for the MuJava to generate the mutants and it took 66 minutes (3,973 seconds *i.e.,* a reduction of 12.57%) for the MuJava-AUM.

Note that we are computing the time to generate and compile the mutants. In a real mutation analysis scenario, there is also time to execute the test suite against the mutants, and the time to analyze the surviving mutants (equivalent and non-equivalent). Obviously, with fewer mutants to execute on the test suite and with fewer equivalent mutants to analyze, MuJava-AUM performs much better than MuJava. However, our goal was to check if the additional overhead brought with *i-rules* could cover up the time gained by not generating the useless mutants. With this, we respond to **RQ3** below.

Answer to RQ3: Although our *i-rules* introduce an additional overhead, the payoff amount is 15% on average. Because we generate and compile fewer mutants, we have fewer I/O operations. These operations are more expensive than executing our *i-rules*.

3.6.5 Threats to Validity

The projects we used represent a threat to the external validity. To alleviate this threat, we selected projects of different sizes and domains. Also, these projects have been used by the mutation testing community (KINTIS et al., 2018; MADEYSKI et al., 2014; KINTIS; MALEVRIS, 2015).

Table 8 – Time to generate and compile the mutants with MuJava (original version) and MuJava-AUM. The presented numbers represent an average of three executions per project.

Project	MuJava	MuJava-AUM	Difference
ant	2,143s	1,775s	17.29%
bcel	2,287s	1,891s	17.32%
commons-lang	4,463s	3,781s	15.28%
commons-math	2,789s	2,354s	15,60%
h2	4,544s	3,973s	12.57%
joda-time	1,541s	1,314s	14.73%
Average Reduction			**15.08%**

Our *i-rules* implementation is a threat to the internal validity. We minimize this threat because the majority of the useless mutants have been confirmed as useless by the TCE tool. For the remaining ones, we manually analyzed a sample of 10% of the useless mutants identified by each *i-rule*. We found a few bugs in our implementation, fixed them, and re-executed the entire analysis. In this context, the manual analysis also represents a threat. We alleviate such a threat by double-checking the controversial cases with a second researcher. Additionally, the sample represents a threat as well. Nevertheless, we tried to minimize this threat by sampling useless mutants identified by all *i-rules* we have implemented.

Our *i-rules* can avoid the generation of useless mutants. In this way, we can reduce costs not only regarding the generation itself; in fact, we also reduce costs regarding the following two tasks: executing the test suite, and analyzing the surviving mutants. As we considerably reduced the number of generated mutants (see Table 5) and we observed that the additional overhead brought on by the *i-rules* does not lead to a performance loss during mutant generation (see Table 8), so as a result, this would lead to a reduction of the time to execute the test suite and also the time to analyze the surviving mutants. However, calculating mutant generation time also poses an internal threat since the number of I/O tasks is usually high for this type of system, and other system processes may block execution. So we tried to minimize this threat by running each project three times in each version of MuJava and getting the average time.

It is easy to reason about the execution time results, given that we implemented our *i-rules* by using a set of conditionals. So, the cost of executing an *i-rule* should not be higher than generating a mutant and compiling it. However, it is essential to remember that we have not implemented some costly *i-rules*, *e.g.*, the ones that need advanced static analyses. This is a threat to the conclusion validity, given that the differences presented in Table 8 can change if we consider such *i-rules*.

3.7 SUMMARY

In this chapter, we proposed an approach to avoid useless mutants by improving the transformation rules embedded in the mutation operators. We call these improvements *i-rules*. We divide the *i-rules* in two classes; *e-rule* for avoiding equivalent mutants and *d-rule* for avoiding duplicate mutants. To help mutation tool developers discover new *i-rules* to its tools, we also presented a strategy. We instantiated the strategy with 100 Java programs as input, three mutation testing tools to generate the mutants, (MuJava, Major, and Pit), and automatic test generation tools (Randoop and EvoSuite) to work as an oracle to automatically classify the mutant as equivalent or duplicate candidate. As a result, we identified 30 *e-rules* and 69 *d-rules* in all three mutation tools. We chose MuJava to implement a subset of the *i-rules* discovered, and we name this new version MuJava-AUM. By executing MuJava-AUM with well-known open-source projects, we

could avoid the generation of almost 20% of useless mutants, on average. We also identified that one *e-rule*, alone, was responsible for approximately 90% of all equivalent mutants avoided and six *d-rules* were responsible for approximately 89% of the duplicate mutants avoided. Last but not least, MuJava-AUM saved time to generate and compile the mutants in all systems evaluated.

For more information of how to reproduce the experiment, the detailed results, and download MuJava-AUM, visit our companion website (FERNANDES et al., 2020)

4 SUGGESTING EQUIVALENT MUTANTS THROUGH AUTOMATED BEHAVIORAL TESTING

Previously, we presented an approach to avoid useless mutants. Unfortunately, we cannot discover all equivalent and duplicate mutants before the generation. By following the mutation testing process (Figure 1), after running the tests on the mutants and finding out there are surviving mutants, we need to analyze each mutant to mark it as equivalent, which means no test case can kill it, or to identify as non-equivalent and create a new test case capable of killing it. To support developers in this scenario, in what follows, we present our second approach to deal with useless mutants.

4.1 INTRODUCTION

Equivalent mutants are a well-known impediment to the practical adoption of mutation testing. A previous work (BUDD; ANGLUIN, 1982) has already proven that this is an undecidable problem in its general form. Thus, no complete automated solution exists. In addition, manually detecting equivalent mutants is an error-prone (ACREE, 1980) and time-consuming task (SCHULER; ZELLER, 2013). This problem becomes quite relevant when empirical studies report that up to 40% of all the generated mutants can be equivalent (MADEYSKI et al., 2014).

Remembering Madeyiski et al. (MADEYSKI et al., 2014) work, there are three methods to deal with equivalent mutants: *Avoiding*, *Detecting*, and *Suggesting*. Previous works proposed using compiler optimization to detect equivalent mutants (BALDWIN; SAYWARD, 1979; OFFUTT; CRAFT, 1994; PAPADAKIS et al., 2015; KINTIS et al., 2018). The intuition is that code optimization can transform the original program and the mutant in a way in which their compiled object codes get identical. The most recent work (KINTIS et al., 2018) presented the TCE (*Trivial Compiler Equivalence*). Using a set of knowingly equivalent mutants, TCE identified 56% of the equivalent mutants in the benchmark. And it took approximately three seconds to analyze each mutant. Despite an efficient solution, some equivalent mutants would require costly compiler optimization techniques to be detected, which could increase the general cost of the mutation analysis. Other work demonstrated that changes in coverage would suggest non-equivalent mutants (SCHULER; ZELLER, 2013). According to the authors, if a mutation impacts coverage, it has a 75% chance to be non-equivalent. However, this information alone is not enough. Mutations that impact coverage have higher detection rates, which means naive tests could easily detect them. Besides, there are equivalent mutants that impact coverage and non-equivalent mutants that keep the coverage information equal to the original program.

In this work, we propose an approach to suggest equivalent mutants by using automated behavioral testing. We perform static analysis to automatically generate a massive set

of tests directed for the entities impacted by the mutation. For each analyzed mutant, our approach can suggest the mutant as equivalent or non-equivalent. In this sense, our approach relies on tools to generate the test cases automatically. These test cases capture the current behavior of the program under test. To avoid the generation of tests that focus on methods that have not been impacted by the mutation, we adapt a change impact analysis proposed in previous work (MONGIOVI et al., 2014). After generating the tests, we execute them against the original program and the mutant. If at least one test case fails, there is a behavioral change, which means that the mutant is *non-equivalent*. Notice that testers might use such a test to improve their test suite, minimizing costs. If no test kills the mutant, the approach *suggests the mutant as equivalent*. To better support the testers when analyzing the mutants suggested as equivalent, we provide two metrics: the number of test cases that reached the point where the mutation occurred; and a boolean value indicating whether the test execution coverage of the mutant has changed when compared to the original program. We implement our approach in a tool called NIMROD. The tool is available online (EASY, 2019).

To evaluate our approach, we execute NIMROD against a benchmark with 1,542 mutants generated from eight methods of six different open-source systems, such as *Apache commons-lang* and the *joda-time* library. A previous work manually classified these mutants as equivalent and non-equivalent (KINTIS et al., 2018). This classification represents the baseline (*ground truth*) of our evaluation. To generate the tests, we instantiate NIMROD to use well known tools by the software testing community: RANDOOP (PACHECO et al., 2007) and EVOSUITE (FRASER; ARCURI, 2011; FRASER; ARCURI, 2013). They follow different strategies. The former generates unit tests for JAVA using a feedback-directed random test generation. The latter is a search-based tool that uses a genetic algorithm to generate test suites for JAVA classes automatically. Testing generation tools, in general, need a stopping criterion. In our evaluation, we set the time limit to 60 seconds for both tools in each of the eight methods we studied.

After submitting all these mutants to NIMROD, we compute precision, recall, and F-measure (BONNIN, 2017) to check the effectiveness of our approach. Precision identifies the fraction of all elements that are actually correct. Recall is the fraction of the selected elements that are successfully classified. F-Measure provides a single score that balances both the concerns of precision and recall in one number.

The results indicate that the approach is effective in suggesting equivalent mutants. The F-measure has reached more than 92% in five out of the eight methods. On the other hand, the F-measure was very low in one of the methods, i.e., 21%. We noticed that the results might strongly depend on the code design characteristics. Such characteristics may prevent the test generation tools from generating effective test cases. This happens, for instance, when there are tight coupling classes that a statement in class `A` changes a package-private field from class `B`. To better analyze our approach, we also compute

the time taken by NIMROD to suggest equivalent mutants. On average, NIMROD took approximately five minutes to classify a mutant as potentially equivalent (three times faster when compared to the manual estimations (SCHULER; ZELLER, 2013)) and 24 seconds to classify a mutant as non-equivalent.

In summary, the main contributions of our second approach include:

- An approach to suggest equivalent mutants based on automated behavioral testing (Section 4.3);
- A tool that implements and automates the entire approach (Section 4.3);
- An evaluation to check the effectiveness and efficiency of our approach on suggesting equivalent mutants (Sections 4.4 and 4.5);
- A discussion regarding implications for practice to combine the methods *Suggesting Equivalent Mutants* and *Detecting Equivalent Mutants* in order to reduce costs when dealing with the equivalent mutation problem (Section 4.6).

4.2 MOTIVATING EXAMPLE

After generating the mutants and executing the test suite against them, a tester should analyze the surviving mutants to discard the equivalent ones and create mutation-guided tests to kill the non-equivalent ones (see Figure 1). Detecting equivalent mutants is a well-known undecidable problem, which means that the detection of equivalent mutants alternatively may have to be carried out by humans. However, this manual task is error-prone (people judged equivalence correctly in about 80% of the cases (ACREE, 1980)) and time-consuming (approximately 15 minutes per equivalent mutant (SCHULER; ZELLER, 2013)). This way, heuristics to identify at least a subset of the equivalent mutants are essential to minimize the costs.

In previous works, first Papadakis et al. (PAPADAKIS et al., 2015) and then Kintis et al. (KINTIS et al., 2018) presented and evaluated the Trivial Compiler Equivalence (TCE). An effective heuristic based on compiler optimizations for popular compiled languages (C and JAVA) and mutation tools (MILU and MUJAVA). Notice that this approach is sound in the sense that two equal binaries mean that the programs have the same behavior. This way, TCE does not raise false positives. However, despite having the same behavior, the original program and the equivalent mutant may have different object codes after applying code optimizations. In this sense, TCE is not able to detect such an equivalent mutant. Therefore, TCE may raise false negatives.

To better illustrate this scenario, we present a code snippet of a `triangle` class (Listing 4.1). This class contains a method (`classify`) to determine the type of a triangle given sizes of the three sides. We present four different equivalent mutants. We generate these mutants using the MUJAVA mutation tool.

```
1  public static int classify( int a, int b, int c ) {
       int trian;
3      if (a <= 0 || b <= 0 || c <= 0) { return INVALID; }
       trian = 0;
5      if (a == b) { trian = trian + 1; }          M1 [trian + 1 ⇒ -trian + 1]
       if (a == c) { trian = trian + 2; }
7      if (b == c) { trian = trian + 3; }
       if (trian == 0) {
9          if (a + b < c || a + c < b || b + c < a) {
               return INVALID;
11         } else {
               return SCALENE;
13         }
       }
15     if (trian > 3) {
           return EQUILATERAL;
17     }
       if (trian == 1 && a + b > c) {          M2 [trian == 1 ⇒ trian <= 1]
19         return ISOSCELES;
       } else {
21         if (trian == 2 && a + c > b) {          M3 [a + c > b ⇒ a + c > b++]
               return ISOSCELES;
23         } else {
               if (trian == 3 && b + c > a) {      M4 [trian == 3 ⇒ trian++ == 3]
25                 return ISOSCELES;
               }
27         }
       }
29     return INVALID;
   }
```

Listing 4.1 – A code snippet extracted from the `Triangle` class.

- M_1 represents the mutant generated by the AOIU (Arithmetic Operator Insertion - unary) with the following transformation: `trian + 1` ⇒ `-trian + 1`. However, at that point in program, `trian` can only be zero;
- M_2 represents the mutant generated by the ROR (Relational Operator Replacement) with the following transformation: `trian == 1` ⇒ `trian <= 1`. However, at that point in program, `trian` can only have the value of one or more;
- M_3 represents the mutant generated by the AOIS (Arithmetic Operator Insertion - short-cut) with the following transformation: `a + c > b` ⇒ `a + c > b++`. However, this is the last access to the local variable `b`.
- M_4 is like the previous case. This mutant is generated by the AOIS operator and inserts a post-increment to the last access of the local variable `trian`.

We apply TCE to verify its ability to detect the listed mutants. According to Kintis et al. (KINTIS et al., 2018), TCE uses two types of optimization for the JAVA version, one

with the standard JAVA compiler[1] and another one with the SOOT[2] analysis framework. By running the TCE against the mutants, it detects two out of the four mutants: M_1 and M_4.

By analyzing mutant M_2, one can see that the `trian` value starts with zero (line 4). However, when reaching the mutated line (line 18), the `trian` value can only be one or greater than one, which prevents the behavior change for any possible entry of the program. The compiler would need to check the conditional expression at line 8 to identify this equivalent mutant. In case the condition is `true`, `trian` is zero, and the method returns.

By considering the mutant M_3, applying a post-increment to the last access of a local variable within a method does not change the behavior of the program, as reported before (KINTIS; MALEVRIS, 2015; FERNANDES et al., 2017). However, we can see that this is not the last access to the variable `b` since, at line 24, this variable can be reaccessed. The secret, in this case, is in the specification of the `&&` operator (GOSLING et al., 2015). The conditional-and operator `&&` is like `&`, but it evaluates its right-hand operand *only if* the value of its left-hand operand is `true`. It turns out that the left-hand side of the conditional expression to which mutant M_3 belongs is mutually exclusive with the left-hand side of the conditional expression at line 24. This scenario would also be quite complicated (perhaps impossible) of detection with compiler optimization.

Next, we present an alternative approach to suggest equivalent mutants by using automated behavioral testing. By suggesting, we mean we cannot guarantee that the mutant is indeed equivalent, but we can increase the developer's confidence by ranking the surviving mutants who have strong or weak chances of being equivalent. In the motivating example scenario presented above, our approach was able to suggest all equivalent mutants correctly. We detail it in the next section.

4.3 SUGGESTING EQUIVALENT MUTANTS

Our approach is based on previous work in the area of refactoring (SOARES; GHEYI; MASSONI, 2013). While refactoring is a transformation that preserves the external behavior of a program, a mutant must transform a program so that the program's external behavior changes (STEIMANN; THIES, 2010b). We adapt the refactoring solution to the mutation testing context and add improvements in the impact analysis phase, the automated test generation, and post-testing execution to improve the accuracy of mutant classification.

Before detailing our approach, it is important to remember the notion of equivalence we adopt (see Section 2.1). As explained, a mutant and an original program are equivalent if they present the same externally observable behavior for all possible inputs. However, there are two different scenarios to consider; *open world* and *closed world* (SOARES; GHEYI; MASSONI, 2013). In an *open world assumption* (OWA), any kind of test case

[1] <http://www.oracle.com/technetwork/java/index.html>
[2] <http://sable.github.io/soot/>

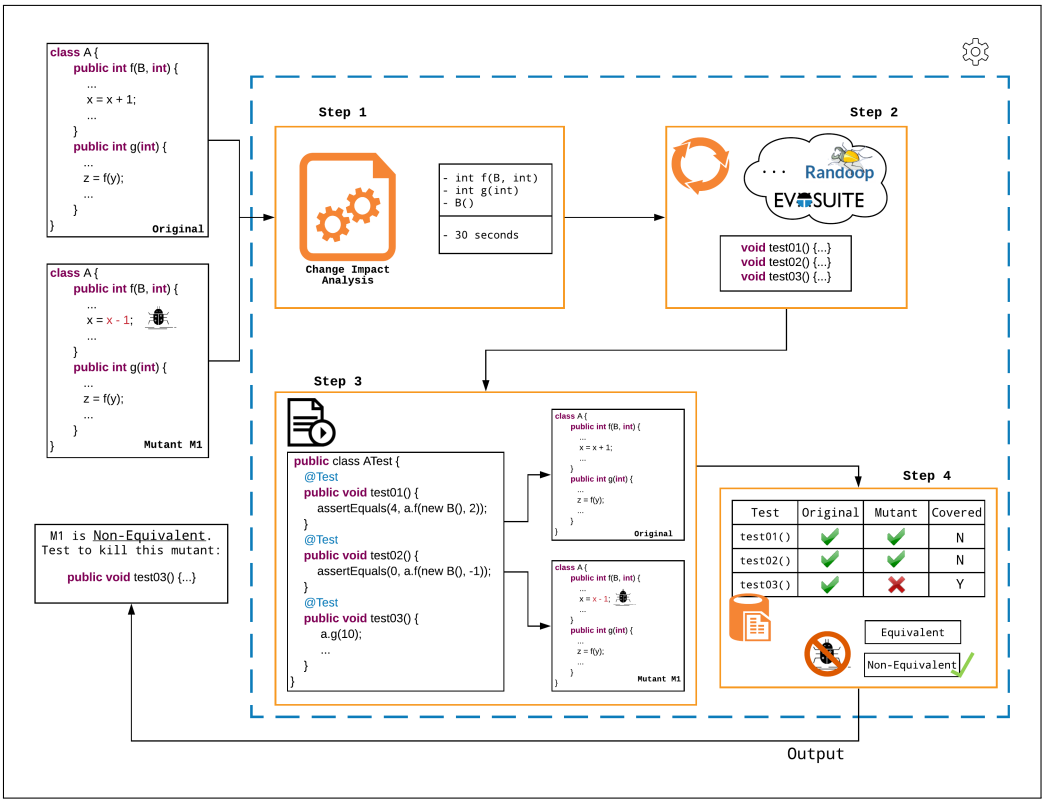


Figure 8 – Our approach to suggest equivalent mutants.

can be generated to find out a behavioral change, without regarding the project or code requirements. In a *closed world assumption* (CWA), the test cases must satisfy some domain constraints. Our approach adopts an open-world equivalence notion, which means there are no constraints in the test generation.

Figure 8 depicts an overall view of the approach. It consists of four major steps. First, it carries out a change impact analysis of the mutation. Second, it uses the change impact analysis output to guide the generation of automated behavioral tests. In the third step, it executes each generated test case against the original program and the mutant. In the final step, our approach suggests whether the mutant is equivalent or not and supports the tester in case a test to kill the mutant is found. We now detail each of the steps.

4.3.1 Identifying Impacted Entities

In Step 1 (Figure 8), our approach receives two versions of the program as input: the original and the mutant source codes. We *diff* the two programs to find out where the mutation occurred. We handle this information to carry out a *change impact analysis* and generate tests only for the entities impacted by the transformation. Our approach is based on the change impact analysis proposed by Mongiovi et al. (MONGIOVI et al., 2014). It checks both the original and mutant programs, beginning by decomposing a coarse-grained transformation into smaller transformations. For each small-grained transformation, we

identify the set of impacted entities. Finally, we sort a set of public methods in common that exercises, directly or indirectly, the impacted entities. A method in common must have the same signature in the original and mutant programs. Besides the public methods, we also analyze the parameters of such methods to identify methods dependency.

To better explain the idea of identifying the entities, we present an example using the `FieldUtils` class, a file from the joda-time project. Joda-time is a popular date and time JAVA library. The `FieldUtils` class has 158 lines of code, 17 methods, and no fields. For demonstration purposes, we extract a code snippet from the `FieldUtils` file and present in Listing 4.2. We introduce three different mutants. Mutant $M1$ replaces the logical AND (`&&`) signal to the logical OR (`||`) (line 9). Mutant $M2$ inserts a post-increment (`++`) to the `total` variable (line 20). Mutant $M3$ inserts a pre-decrement () to the `val2` variable (line 28)[3].

$M1$ occurred in the `safeToInt` method and `safeMultiplyToInt` invokes `safeToInt`. This way, the output of Step 1 is the following.

```
#List of Target Methods and Constructors
m:FieldUtils.safeToInt(long)
m:FieldUtils.safeMultiplyToInt(long, long)
```

Regarding mutant $M2$, the mutation occurred in the `safeMultiply` method. This method is invoked by the `safeMultiplyToInt` method. In this case, the output of the change impact analysis is:

```
#List of Target Methods and Constructors
m:FieldUtils.safeMultiply(long, long)
m:FieldUtils.safeMultiplyToInt(long, long)
```

Regarding mutant $M3$, the mutation occurred in the `safeSubtract` method. No other method invokes `safeSubtract`. This way, the output of Step 1 is the following:

```
#List of Target Methods and Constructors
m:FieldUtils.safeSubtract(long, long)
```

Since all impacted methods have no dependency on external objects, we do not search for valid constructors. Now, we pass the change impact analysis results to the test case generation step (Step 2).

[3] Some members of the community (eg., (PETROVIĆ; IVANKOVIĆ, 2018)) argue that generating mutants that change exception messages is useless. However, many mutation tools continue to generate mutants in these statements.This work does not discuss this.

```
public class FieldUtils {

    public static int safeMultiplyToInt(long val1, long val2) {
        long val = FieldUtils.safeMultiply(val1, val2);
        return FieldUtils.safeToInt(val);
    }

    public static int safeToInt(long value) {
        if (Integer.MIN_VALUE <= value &&                    M1 [ && ⇒ || ]
        value <= Integer.MAX_VALUE) {
            return (int) value;
        }
        throw new ArithmeticException(...);
    }

    public static long safeMultiply(long val1, long val2) {
        ...
        long total = val1 * val2;
        ...
        return total;                                        M2 [ total ⇒ total++ ]
    }

    public static long safeSubtract(long val1, long val2) {
        long diff = val1 - val2;
        if ((val1 ^ diff) < 0 && (val1 ^ val2) < 0) {
            throw new ArithmeticException
            ("The calculation caused an overflow: " +
                val1 + " - " + val2);                        M3 [ val2 ⇒ --val2 ]
        }
        return diff;
    }
    ...
}
```

Listing 4.2 – An excerpt extracted from the `FieldUtils` class.

4.3.2 Automated Generation of Test Cases

Automated generation of tests is a broad field of research (LAKHOTIA; MCMINN; HARMAN, 2009; SHAMSHIRI et al., 2015; FRASER et al., 2015). Researchers have explored different approaches to automatically generate unit tests, such as random test generation, constraint solver, symbolic execution, and genetic algorithms. Tools such as EvoSuite (FRASER; ARCURI, 2011), Randoop (PACHECO et al., 2007), and IntelliTest (LI et al., 2016) implement such approaches. Each tool has a specific purpose. Thus different tools generate a different sequence of method calls and assertions. These sequences and assertions of the generated test capture the current behavior of the original program under test. Although not yet widely adopted by the industry, these automated unit test generation tools have become very effective in generating input data that achieves high code coverage (FRASER et al., 2015) and finds real faults (SHAMSHIRI et al., 2015).

After collecting the information of the change impact analysis (Step 1), we use well-known tools to generate tests automatically. The idea is to generate a massive set of tests to only exercise the entities affected by the mutation, in an attempt to bring up the behavior change caused by the transformation. We can instantiate our approach using different test generation tools or even instantiate the same tool more than once using different input parameters.

To exemplify Step 2, we return to the `FieldUtils` class (Listing 4.2). In the previous step, only two methods have been impacted by the mutant $M1$ (`safeToInt` and `safeMultiplyToInt`) and the mutant $M2$ (`safeMultiply` and `safeMultiplyToInt`). For mutant $M3$, only one method was impacted (`safeSubtract`). Listing 4.3 shows test cases generated for $M1$, $M2$, and $M3$ mutants. Tests from the `FieldUtilsTest_M1` class will execute against the original program and the mutant $M1$. The same happens to the tests of class `FieldUtilsTest_M2` with the mutant $M2$ and of class `FieldUtilsTest_M3` with the mutant $M3$.

```
1  public class FieldUtilsTest_M1{
       @Test
3      public void test001(){
           int val = FieldUtils.safeMultiplyToInt(10L, 20L);
5          assertEquals(200, val);
       }
7      @Test
       public void test002(){
9          int val = FieldUtils.safeToInt(10L);
           assertEquals(10, val);
11     }
       @Test
13     public void test003(){
           try {
15             int val = FieldUtils.safeToInt(2147483648L);
               fail("Failed: Should get an Arithmetic Exception");
17         }
           catch (ArithmeticException e) {
19         }
       }
21     ...
   }
23
   public class FieldUtilsTest_M2{
25     @Test
       public void test001(){
27         int val = FieldUtils.safeMultiplyToInt(10L, 20L);
           assertEquals(200, val);
29     }
       @Test
31     public void test002(){
           long val = FieldUtils.safeMultiply(5L, 5L);
33         assertEquals(25L, val);
       }
35     @Test
       public void test003(){
```

```
37          long val = FieldUtils.safeMultiply(100L, 2L);
            assertEquals(200L, val);
39      }
        ...
41  }

43  public class FieldUtilsTest_M3{
        @Test
45      public void test001(){
            long val = FieldUtils.safeSubtract(0L, 1L);
47          assertEquals(-1, val);
        }
49      @Test
        public void test002(){
51          try {
                int val = FieldUtils.safeSubtract(Long.MIN_VALUE, 100L);
53              fail("Failed: Should get an Arithmetic Exception");
            }
55          catch (ArithmeticException e) {
            }
57      }
        @Test
59      public void test003(){
            long val = FieldUtils.safeSubtract(-10L, -20L);
61          assertEquals(10, val);
        }
63      ...
    }
```

Listing 4.3 – Examples of test cases generated to the `FieldUtils` class.

4.3.3 Test Execution

After generating tests (Step 2), we execute them against the original program and one mutant at a time (Step 3). In case a test fails in the original program, we discard it. It is not a common situation for unit test generation tools since they capture the current behavior of the original program. But, the presence of non-deterministic outcomes (like *flaky tests* (LUO et al., 2014)) could hinder the execution. This way, we end up with a green test suite for the original program. Once we identify a test able to expose a behavioral change in the mutant program, we do not execute the subsequent tests. It is because our goal is to suggest equivalent mutants and once identified a non-equivalent, we do not need to execute the subsequent tests.

During the test execution step, we also record the test execution coverage of the original program and the mutants. In other words, we record the frequency in which all generated tests have executed each line. Besides, we also track the number of test cases that cover the statement where the mutation occurred. We use these data to create a ranking of mutants suggested as equivalent by the approach, as we explain next.

4.3.4 Suggesting Equivalent Mutants

In the last step (Step 4), we analyze the test suite execution results to suggest equivalent mutants. As explained, if at least one test case fails, our approach reports a behavioral change, which means the mutant is *non-equivalent*. Besides, we provide the test case capable of killing the mutant, so the tester might use such a test to improve the test suite.

In case no test kills the mutant, our approach suggests the mutant as *equivalent*. Since we cannot guarantee that the suggestion is correct, we provide a *ranking* of mutants to better support the testers. At the bottom of the ranking, we place mutants that we have strong confidence they are indeed equivalent. At the top, we place mutants in which we have weak confidence that our suggestion is correct. Our ranking of mutants relies on two information recorded during the test execution: a boolean value indicating whether the test execution coverage of the mutant has changed when compared to the original program (also called *coverage impact*); and the number of test cases that reached the mutated point. In our ranking, we prioritize the coverage impact over the number of tests that exercised the mutation. Shuler and Zeller (SCHULER; ZELLER, 2013) identified that coverage impact provides effective means to separate equivalent from non-equivalent mutations. They reported that if a mutation changes the coverage, the mutant has 75% of chances to be non-equivalent.

For example, if our approach suggests a mutant as equivalent and the test execution reveals no coverage difference between the mutant and the original program and, among the suggested equivalent mutants, this one had the highest number of test cases executing the mutated statement, we place this mutant at the bottom of the ranking. On the other hand, if we identify differences in the coverage, we place the mutant at the top of the ranking, indicating that our confidence regarding the equivalence suggestion is weak.

To summarize, we place at the top the mutants that our strategy has "weak" confidence of being equivalent, and at the bottom, the mutants which our strategy has "strong" confidence of being equivalent. However, due to the features of each program and the undecidability of the equivalent mutant problem, we cannot define a general threshold that guarantees the accuracy of the ranking. This way, it is up to the tester to decide which mutants will be manually reviewed.

To better explain the last step of our approach, we rely on the code snippets presented in Listings 4.2 and 4.3. As mentioned, Step 4 analyzes the test suite execution and classifies the mutant as equivalent or non-equivalent. To produce a different behavior in the $M1$ mutant it is required, for instance, an input parameter that denies the `if` expression (lines 9-10). The test `FieldUtilsTest_M1.test003` can expose this behavior change. The approach then marks this mutant as non-equivalent and informs the tester that `FieldUtilsTest_M1.test003` is enough to kill the mutant $M1$. By considering $M2$ mutant, we soon realize that it is an equivalent mutant since it applies a post-increment at the last access of a local variable (FERNANDES et al., 2017; KINTIS; MALEVRIS, 2015). In

this case, every generated test suite executes, and the mutant is suggested as equivalent.

In the case of mutant $M3$, although this mutant is not equivalent, no generated test identified the behavior change. This way, the approach suggests $M3$ as equivalent. At this point, we have two mutants suggested as equivalent. Now we check the information collected at the test execution step to create the ranking. Both suggested mutants did not yield any impact on the coverage. Regarding the exercised statement, the mutant $M2$ had three test cases exercising the mutated statement, while the mutant $M3$ had only one test case exercising the mutated statement. This way, the mutant $M3$ goes to the top of the rank and the mutant $M2$ stays at the bottom of the ranking. The output of our approach would be:

> #Testing Execution Results
> $M3$ | Possibly Equivalent | Coverage Impact: NO | Num. Test Cases Exercise: 1
> $M2$ | Possibly Equivalent | Coverage Impact: NO | Num. Test Cases Exercise: 3
>
> ---
>
> $M1$ | Non-Equivalent | Killed by: `FieldUtilsTest_M1.test003`

4.3.5 Improvements

As explained, our approach is based on previous ideas from the refactoring field (SOARES; GHEYI; MASSONI, 2013; SOARES et al., 2010; MONGIOVI et al., 2014). In addition to bringing this idea to the context of mutation testing, we provide several improvements regarding prior work.

Mongiovi et al. (MONGIOVI et al., 2014) presented a change impact analysis to help with identifying the entities impacted by a code transformation. They provide only the *interclass* analysis option. However, for large projects with complex dependencies, it is difficult to identify what we need to test after a transformation. Mainly when we search the set of indirectly impacted methods that exercise an impacted entity, the list of methods to test can get large, hindering the testing generation tools' efficacy. Then, we add the option of the analysis to be *intraclass*, that is, the impact analysis identifies only the public methods in the class where the mutation occurred. We also add analysis to the parameters of the methods. If any of the parameters are not from a primitive type, wrapper class, or String type, we search in the *classpath* for constructors needed to initialize the objects and to perform the method call. The output of the change impact analysis is a set of public methods and, if necessary, a set of constructors to build up object dependencies.

We made other improvements in Steps 3 and 4. During the test execution step, we record coverage information to support the phase of suggesting equivalent mutants and we apply it in two ways. First, we follow the idea proposed by Shuler and Zeller (SCHULER; ZELLER, 2013) to calculate the impact on the coverage when we execute the tests in the

original program and the mutant. Second, we use the coverage information to count the number of test cases able to exercise the statement where the mutation occurred. These pieces of information can help assess the behavior of the mutation during computation. As we cannot guarantee that the suggestion of equivalence is correct, these improvements were fundamental in supporting the results of the approach.

The nature of our approach allows a critical cost reduction. The cost of the tester to design and implement a new test case to kill a surviving mutant identified as non-equivalent. The output of our approach indicates which test case can kill the mutant. This way, the *mutation tester* might use the automated generated test to improve their *mutation-based* test suite.

4.4 EVALUATION

To better understand the advantages and disadvantages of our approach, we evaluate its effectiveness and efficiency in suggesting equivalent mutants against real programs. To perform the evaluation, we automate all the steps of our approach and pack in a tool named NIMROD[4]. The link to download the tool is available at the companion website of the thesis (EASY, 2019). To facilitate the explanation of our study, we refer to the approach using the name of the tool. In what follows, we detail the research questions, the subjects used, the experimental setup, and the procedure to reproduce the study.

4.4.1 Research Questions

The purpose of this study is to evaluate automated behavioral testing to suggest equivalent mutants from the mutation tester point of view in the context of mutation analysis.

Automated solutions can help to reduce the manual effort of analyzing the surviving mutants. These solutions need to be effective, which naturally brings out our first research question.

- **RQ1.** How effective is NIMROD in suggesting equivalent mutants?

We answer **RQ1** by reporting the precision, recall, and f-measure of NIMROD in suggesting equivalent mutants. Our baseline is a set of manually identified equivalent mutants provided in a previous work (KINTIS et al., 2018). This way, our numbers regarding true positives, false positives, true negatives, and false negatives are based on such a manual analysis. To better understand the strengths and weaknesses of NIMROD in our evaluation, we also present the result of executing TCE [5] against the same set of mutants.

[4] Nimrod is a fictional character appearing in *Uncanny X-Men* (March 1985). Nimrod is a powerful, virtually indestructible descendant of the robotic mutant-hunting Sentinels.

[5] <https://bitbucket.org/marinosk/ted>

Notice, however, that NIMROD and TCE are complementary tools. The former suggests equivalent mutants, and the latter detects. Observe that answering RQ1 is crucial because it allows us to estimate the amount of effort that can be saved by NIMROD in comparison to a manual analysis for each surviving mutant.

As important as effectiveness, a solution must be efficient in the sense that it can scale to many mutants. This way, we formulate the following research question:

- **RQ2.** How long does it take for NIMROD to analyze a mutant?

To answer **RQ2**, we calculate the average time that NIMROD takes to suggest the mutants as equivalent or non-equivalent. Because we rely on OWA (see Section 4.3), once a test kills the mutant, we confirm that such a mutant is non-equivalent. It means that NIMROD has no false negatives since all mutants classified as non-equivalent are killed by at least one test case. On the other hand, NIMROD can erroneously classify mutants as equivalent (false positives). These mutants may be a stubborn mutant (YAO; HARMAN; JIA, 2014), where only a particular test case can kill it. To better understand the types of mutants that our approach classifies erroneously, we formalize the following research question:

- **RQ3.** What are the characteristics of the mutants that NIMROD incorrectly identifies as equivalent (false positives)?

We answer this question by manually analyzing the false positives. In particular, we analyze the context of the program where the mutation occurs. Mutation testers usually employ a subset of the mutation operators to perform the analysis. Therefore, having information about the relationship between the mutation operators and the equivalent mutants is useful This way, we ask the following research question.

- **RQ4.** Which mutation operators commonly lead NIMROD to fail?

We answer this question by computing the contribution of each operator to the proportion of equivalent mutants, as the ratio of each operator to NIMROD false positives. We enable all 15 method-level mutation operators available in MUJAVA (Version 3).

4.4.2 Subjects

To perform our evaluation, we rely on programs and mutants of a previous work (KINTIS et al., 2018). We select this set because manually identified equivalent mutants accompany it. The availability of known equivalent mutants provides a "ground truth" about the problem of the undecidability of equivalent mutants. Besides, this manual analysis also followed the idea of OWA.

Table 9 – Manually analyzed JAVA subjects.

Program	Class	Method	LoC	Total Mutants
bisect	Bisect	sqrt	23	135
commons-lang	WordUtils	capitalize	25	69
		wrap	45	198
joda-time	BasicMonthOfYearDateTimeField	add	33	257
pamvotis	Simulator	addNode	53	318
		removeNode	18	55
triangle	Triangle	classify	44	354
xstream	XmlFriendlyNameCoder	decodeName	40	156
Total			**281**	**1,542**

Table 9 details the JAVA programs used in the evaluation. The first three columns of the table present the examined programs, the selected classes, and the considered methods. The last two columns of the table present the lines of code and the number of generated mutants.

The list of evaluated subjects covers: *bisect* - a simple program that calculates square roots, *commons-lang* - an enhancements to JAVA core library, *joda-time* - a time manipulation library, *pamvotis* - a wireless LAN simulator, *triangle* - a classic triangle classification program, and *xstream* - a XML object serialization framework.

4.4.3 Experimental Setup

We performed the evaluation on a 2.70 GHz four-core PC with 16 GB of RAM equipped with the Ubuntu 17.10 operating system.

We implement NIMROD in a way that it is flexible in accepting different test-generation tools. For this experiment, we decide to use two of the most popular solutions in the research community: EVOSUITE (FRASER; ARCURI, 2011; FRASER; ARCURI, 2013) and RANDOOP (PACHECO et al., 2007; SOARES; GHEYI; MASSONI, 2013).

In Section 3.4.2.1, we depicted the main features of these tools. Below we complement the information needed for this study;

EVOSUITE is a search-based tool that uses a genetic algorithm to generate test suites for JAVA classes automatically. EVOSUITE has a wide variety of configuration parameters. For this experiment, we decided to instantiate EVOSUITE twice for each mutant analyzed. For the first instance, we use EVOSUITE's Regression test suite generation (EVOSUITER), where it tries to generate a test suite revealing differences between two versions of a JAVA class. For the second instance, we set EVOSUITE to comply with four coverage criteria: Statement, Line, Branch, and Weak Mutation coverage. We also specified the impacted

methods that returned from the impact analysis. In both configurations, we set up 60 seconds as the time limit to generate tests.

RANDOOP generates unit tests for JAVA using a feedback-directed random test generation. RANDOOP is typically used to create regression tests to warn when the program's behavior changes. We also set up 60 seconds as the time limit to generate tests.

For the generated tests, we set a timeout of 80 seconds to execute the entire test suite. Then, in case the suite reaches the timeout execution for a mutant, and the same does not occur in the original program, we terminate the execution and classify the mutant as non-equivalent. We also limit the maximum number of test cases to 3,000. This was necessary mainly due to the features of RANDOOP, which tries to generate the widest variety of tests in the established time.

The limits we define in this work (60 seconds to generate tests, 80 seconds to execute the test suite, and a maximum of 3,000 tests) aim at making our analysis feasible. Concerning EVOSUITE, we did not observe a real increment in the test suite after 60 seconds of the time limit. Concerning RANDOOP, we observed it usually reaches 3,000 tests in less than 60 seconds. Besides, for our scope, the worst test suite takes 20 seconds to execute against the original program. Thus, we set a guaranteed time of 80 seconds (4 x 20) to execute against each mutant.

4.4.4 Procedure

To carry out our evaluation, we first generated the mutants. Kintis et al. (KINTIS et al., 2018) did not provide the mutants, but they explain the experimental procedure to generate them and make available the list of equivalent. We manually check that some generated mutants match the mutants in the equivalent list. The mutants were created by the MUJAVA tool[6] with all method-level mutation operators selected. Table 23 lists all mutation operators available in the version 3 of MUJAVA. MUJAVA allows the generation of mutants for a given class in the project but does not let the user generate mutants for a single method of the class. This way, we generate the mutants for a class and filter out the mutants that do not belong to the target methods we focus on this work (see Table 9).

NIMROD's equivalence analysis is done individually between the original program and the mutant. It is because the change impact analysis (Section 4.3) reports the entities that need to be exercised by the tests and the undecidable nature of the automatic test generation tools used. Thus, the set of tests generated to analyze a given mutant is not necessarily equal to the set of tests generated to analyze another mutant. This way, all generated test cases are also available for future analysis.

In the execution step, we first execute the tests generated by EVOSUITE Regression (EVOSUITER). Then, we execute the tests generated by the EVOSUITE with the four coverage criteria, and finally, the tests generated by RANDOOP. We perform the execution

[6] <https://cs.gmu.edu/~offutt/mujava/>

in this order because EvoSuite can, with fewer tests, expose behavior changes faster than Randoop.

In case all the test suite is passing in the original program, we execute the test suite against the mutant. If a test case failing or the test suite execution reaches the timeout, Nimrod suspends the test execution, informs that the mutant is not equivalent, and writes out the test case that exposes the behavior change. If all the test suite executes against the mutant without any test case fails, nor does it reach the timeout, Nimrod terminates the analysis of the mutant, informs the mutant is *possibly equivalent*, and writes out the coverage impact and the number of test cases that reached the mutated statement. After executing the last mutant of each subject, we create the ranking of mutants suggested as equivalent.

To better understand the strengths and weaknesses of Nimrod, we also execute and gather data from the TCE tool on the same mutants. To use TCE, we followed the same steps outlined by Kintis et al. (kintis et al., 2018) and gathered the equivalence information provided by the solution based on compiler optimization.

The procedure to reproduce the steps of our study, the list of equivalent mutants manually classified, as well as the mutants labeled as equivalent by TCE and Nimrod are available at our companion website (easy, 2019).

In the next section, we provide experimental results and discuss the main findings.

4.5 ANALYSIS AND DISCUSSION OF THE RESULTS

This section presents the results and answers the research questions. Our basis for analysis is 193 mutants[7] classified manually as equivalent in the work of Kintis et al. (kintis et al., 2018). These 193 mutants represent 12.5% of the total of 1,542 mutants analyzed.

Notice that Nimrod's equivalence notion is based on the behavior exposed by the mutant program through the execution of automatically generated tests. Therefore, our approach can erroneously classify a mutant as equivalent, but the mutant is indeed a *stubborn*, i.e., a mutant that remains undetected by a high-quality test suite and thus is non-equivalent (yao; harman; jia, 2014). So, this is a *false positive*. On the other hand, if Nimrod finds a test that kills the mutant, but the manual analysis reports that this mutant is equivalent, two situations may occur: (*i*) the manual analysis is wrong; or (*ii*) the tests, although executing correctly, were written in the wrong way. For instance, it violates a project code standard, which represents a kind of CWA. As we are adopting the idea of OWA, and we did not identify any mistake in the manual analysis, Nimrod did not face *false negatives* in this experiment.

We now answer and discuss each of the research questions.

[7] Initially, the paper reported 196 equivalent mutants, but after reanalysis, the authors updated the companion website, and this number dropped to 193.

Table 10 – General Results.

Program	Class	Method	Total Mutants	Equivalent Mutants		
				Manual	TCE	Nimrod
bisect	Bisect	sqrt	135	17	11	18
commons-Lang	WordUtils	capitalize	69	14	2	15
		wrap	198	19	12	26
joda-time	BasicMonthOfYearDateTimeField	add	257	35	24	35
pamvotis	Simulator	addNode	318	33	33	277
		removeNode	55	7	6	10
triangle	Triangle	classify	354	40	21	40
xstream	XmlFriendlyNameCoder	decodeName	156	28	0	28
Total			**1,542**	**193**	**109**	**449**

4.5.1 How effective is Nimrod in suggesting equivalent mutants?

Table 10 presents the general results of executing NIMROD on the subjects. Columns 1 to 3 show the Program, Class, and Method's name respectively. Column 4 indicates the number of mutants generated for each method. In total, the MUJAVA mutation tool generated 1,542 mutants. Columns 5 to 7 show the equivalent mutants according to the manual analysis, TCE detection, and NIMROD suggestion. For a better understanding, we will refer to the subjects by the unique name of the methods (Column 3) and the name of the program (Column 1) in parenthesis. In other words, the subjects studied were: *sqrt* (bisect), *capitalize* (commons-lang), *wrap* (commons-lang), *add* (joda-time), *addNode* (pamvotis), *removeNode* (pamvotis), *classify* (triangle), and *decodeName* (xstream).

The manual analysis identified 193 equivalent mutants. We use these mutants as the baseline for the comparison to our results. The TCE tool detected 109 out of 193 equivalent mutants. It represents 56% of the total of equivalent. Because it is a solution for detecting equivalent mutants (MADEYSKI et al., 2014), there are no false positives in the results. However, the TCE can have false negatives, i.e., it may not identify all the equivalent mutants. All the 84 false negatives occurred because the optimization applied by TCE produced a *bytecode* that is different from the corresponding original program. Notice that this situation may happen even when the mutant exhibits the same behavior as the original program. We use the TCE data as a reference to better understand the NIMROD results. It is not the purpose of this work to suggest a better solution, since our solution, together with TCE, may be complementary from the mutation analysis point of view.

Different from the TCE accuracy in detecting equivalent mutants, our approach attempts to suggest whether a mutant is equivalent through automated behavioral testing. From the 1,542 total mutants analyzed, NIMROD classified 449 as equivalent. It is more than twice the 193 manually identified. We expected our solution to suggest more equivalent mutants than the total number of mutants that are indeed equivalent. The mutants that NIMROD wrongly classified as equivalent mutants are the false positives.

Tables 11 - 18 present in more detail the results of NIMROD and TCE execution on the analyzed subjects. Each table represents a subject and shows the number of equivalent mutants: manually identified, suggested by NIMROD, and detected by TCE. For each table, we also calculate the Precision, Recall, and F-Measure for TCE and NIMROD. With this information, we can have a more accurate base on which subjects NIMROD performed better or worse.

Based on the F-measure, in three out of eight subjects evaluated, *decodeName* (xstream), *add* (joda-time) and *classify* (triangle), the approach had an accuracy of 100%. In two subjects, *sqrt* (bisect) and *capitalize* (commons-lang), the accuracy was above to 96%. In other two subjects, *wrap* (commons-lang) and *removeNode* (pamvotis), the approach had an accuracy above to 82%.

On the other hand, NIMROD had a very low accuracy in the *addNode* (pamvotis). NIMROD suggested 277 out of 318 mutants as equivalent. It is eight times more mutants than the 33 manually marked as equivalent. By analyzing the false positives of this subject, we found some characteristics in the target program and in the mutants that might explain this result. The `addNode` method has the following signature: `void addNode (int, int, int, int, int, int)`. It does not return a value, which requires the test to use an assert that checks the state of the program by using another method or a field (or an exception for exceptional cases). Besides, most mutants mistakenly marked as equivalent change fields that do not have public methods or stay in classes other than the target class where the mutation occurred (we discuss these cases in the next section). In contrast, this was the only subject in which TCE had 100% of accuracy.

Table 11 – Subject: *sqrt* (bisect)

	MANUAL	NIMROD	TCE
EQUIVALENTS	17	18	11
PRECISION		94.44%	100.00%
RECALL		100.00%	64.71%
F-MEASURE		97.14%	78.57%

Table 12 – Subject: *classify* (triangle)

	MANUAL	NIMROD	TCE
EQUIVALENTS	40	40	21
PRECISION		100.00%	100.00%
RECALL		100.00%	52.50%
F-MEASURE		100.00%	68.85%

Table 13 – Subject: *decodeName* (xstream)

	MANUAL	NIMROD	TCE
EQUIVALENTS	28	28	0
PRECISION		100.00%	0.00%
RECALL		100.00%	0.00%
F-MEASURE		100.00%	-

Table 14 – Subject: *add* (joda-time)

	MANUAL	NIMROD	TCE
EQUIVALENTS	35	35	24
PRECISION		100.00%	100.00%
RECALL		100.00%	64.86%
F-MEASURE		100.00%	78.69%

Table 15 – Subject: *capitalize* (commons-lang)

	MANUAL	NIMROD	TCE
EQUIVALENTS	14	15	2
PRECISION		93.33%	100.00%
RECALL		100.00%	14.29%
F-MEASURE		96.55%	25.00%

Table 16 – Subject: *wrap* (commons-lang)

	MANUAL	NIMROD	TCE
EQUIVALENTS	19	26	12
PRECISION		73.08%	100.00%
RECALL		100.00%	63.16%
F-MEASURE		84.44%	77.42%

Table 17 – Subject: *addNode* (pamvotis)

	MANUAL	NIMROD	TCE
EQUIVALENTS	33	277	33
PRECISION		11.91%	100.00%
RECALL		100.00%	100.00%
F-MEASURE		21.29%	100.00%

Table 18 – Subject: *removeNode* (pamvotis)

	MANUAL	NIMROD	TCE
EQUIVALENTS	7	10	6
PRECISION		70.00%	100.00%
RECALL		100.00%	85.71%
F-MEASURE		82.35%	92.31%

The main problem with suggesting equivalent mutants regards the uncertainty of the suggestions. That is, the tester is not sure whether the mutant is indeed equivalent or not. This way, to minimize this problem and provide the mutants that deserve more attention from the tester, we provide a ranking of mutants. As explained, Nimrod computes two metrics to create the ranking and thus support the tester: the number of test cases that reached the mutated point and a boolean value indicating whether the test execution had a coverage impact. We rank the mutants as follows: The first sort criterion was the impact on coverage, and then the number of test cases that exercised the mutated point. That is, if the mutation resulted in a coverage impact, then the mutant has a higher chance of being a false positive, and the tool places it at the bottom of the ranking. If there was no change in coverage, then we use the number of test cases that touched the mutated point. The lower the number of test cases, the lower the Nimrod's confidence in the suggestion, so these mutants get closer to the top of the ranking. In this way, the tester can analyze the mutants of the ranking from top to bottom.

Nevertheless, despite the use of the metrics, we cannot define a general threshold number that determines how many mutants must be manually analyzed in all projects. It is up to the tester to decide which mutants are manually reviewed. In this study, we defined the median number of test cases that touched the mutated point as the threshold to verify the accuracy of the ranking. After that, we then check how many false positives keep before or after the median.

To better explain our evaluation, Table 19 presents the 18 mutants of *sqrt* (bisect) suggested as equivalent by Nimrod. According to manual analysis, 17 mutants are equivalent, which means Nimrod classification had one false positive. The AOIS_12 (in bold) is the mutant wrongly classified. No mutant had an impact on coverage. Using the number of test cases that touched the mutated point as data set, the median value for the 18 mutants was 204 test cases. As the mutation point of the AOIS_12 mutant was exercised by 191 test cases, this mutant was before our threshold. By our evaluation, the mutants before the threshold had a weak chance of being equivalent, and these mutants would be manually analyzed.

Table 20 presents the false positives per subject. Only subjects that had at least one false positive are listed. In the *sqrt* (bisect) and *capitalize* (commons-Lang) subjects only one mutant was wrongly classified. In both, the false positives were before the median. In the *addNode* (pamvotis) subject, despite many false positives, 214 (88%) out of 244 were before the threshold. We checked the details of the result and identified that no test case exercised 106 (38%) mutants. Most of these mutants were inside a switch-case structure, which was nested with a conditional if. The worst-case happened with the *wrap* (commons-lang) method. This subject has a structure with three conditional nested ifs. All false positives were at some point in this structure, and the number of test cases that touched these mutants ranged from two to seven. It is relatively low since the tool generated three thousand tests, on average, for these mutants.

Answer to RQ1: By using the F-measure values, our approach accuracy has reached 100% in three subjects, more than 96% in two subjects and more than 82% in the other two subjects studied. In only one subject, the performance was below 22%. The ranking also presented an interesting result. For this experiment, we defined the median of test cases that touched the mutated point as the threshold to define the mutants with a strong or weak chance of being equivalent. In two cases, the results reached 100% accuracy, and in the worst case, it reached an accuracy of 57%. We believe that Nimrod is effective in what it proposes to be, that is, an approach to suggest equivalent among the surviving mutants of the mutation analysis.

Table 19 – The *sqrt* (bisect) mutants suggested as equivalent by NIMROD. The AOIS_12 is the only false positive (in bold) and the double line marks the division (threshold) based on the median.

Mutant	Coverage Impact	Num. Test Cases Exercise
AOIS_43	NO	61
AOIS_48	NO	137
AOIS_60	NO	142
AOIS_31	NO	143
ROR_13	NO	146
AOIS_45	NO	156
AOIU_12	**NO**	**191**
AOIS_74	NO	199
ROR_12	NO	200
AOIS_59	NO	208
AOIU_4	NO	213
ROR_8	NO	221
AOIS_44	NO	235
AOIS_47	NO	245
AOIS_79	NO	311
AOIU_3	NO	329
AOIS_73	NO	386
AOIS_80	NO	465
MEDIAN		204

Table 20 – Distribution of the false positive according to the median.

Program	**Method**	**False Positives**	**Median** $\Longleftarrow$ | $\Longrightarrow$	
bisect	sqrt	1	100%	0%
commons-Lang	capitalize	1	100%	0%
	wrap	7	57%	43%
pamvotis	addNode	244	88%	11%
	removeNode	3	66%	33%

4.5.2 How long does it take for Nimrod to analyze a mutant?

To evaluate the efficiency of the approach and answer RQ2, we calculate the average time that NIMROD took to suggest each mutant as equivalent or non-equivalent. Table 21

presents the average time in seconds for each subject. For example, the *classify* (triangle) subject took an average of 197.10 seconds to suggest a mutant as equivalent and 11.46 seconds to detect one non-equivalent mutant.

Our approach has a fast average response time for the cases where it suggests the mutant as non-equivalent. That happens because once a test kills the mutant, we finish the analysis. We chose to perform this phase sequentially and defined EvoSuite Regression Testing (EvoSuiteR) as the first option. This decision allowed Nimrod to quickly discover *easy-to-kill* mutants.

The *sqrt* (bisect) subject had the worst results when we calculated the time to detect a non-equivalent mutant. In this subject, some non-equivalent mutants led Nimrod to generate tests that reached the execution timeout. That is, the mutant transforms the original program causing a loop with no terminating condition. This way, although this situation raises a behavioral change, Nimrod spends much time until the timeout.

The *classify* (triangle) subject had the best response time to detect a non-equivalent mutant. This subject has relatively simple code structures when compared to the other subjects of the study. For instance, there are no dependencies with external classes and no complex conditional expressions. This condition leads to the generation of many easy-to-kill mutants.

To suggest a mutant as equivalent, Nimrod needs to generate and execute all tests from all test-generation tools. The subjects *classify* (triangle) and *sqrt* (bisect) had the best results with an average of 197.10 seconds and 198.37 seconds to analyze a single equivalent mutant. Both classes of the two subjects do not have dependencies with external classes, which allows the test generation tools, especially EvoSuite, not to take so long to generate the tests.

On the other hand, the subject *addNode* (pamvotis) took an average of 404.47 seconds

Table 21 – Average time, in seconds, the Nimrod took to analyze each mutant.

Program	**Method**	**Average Time (seconds)**	
		Equivalent	**Non-equivalent**
bisect	sqrt	198.37	130.43
commons-Lang	capitalize	358.36	22.50
	wrap	378.29	15.99
joda-Time	add	212.61	24.04
pamvotis	addNode	404.76	38.72
	removeNode	391.89	25.79
triangle	classify	197.10	11.46
xstream	decodeName	311.01	23.72
Average		**306.55**	**36.57**

to suggest a mutant as equivalent. As explained, this subject has some features that make difficult the generation of tests. For instance, the return of the method is void, and there are many dependencies with entities of external classes that do not have public access. One may question whether the time taken by NIMROD is acceptable. The time results we report are dependent on the settings we use in the test generation tools. For instance, we set up 60 seconds of time limit for each instance (RANDOOP once, EVOSUITE twice) to generate the tests. These settings led NIMROD to take approximately five minutes to suggest a mutant as equivalent. It is worth mentioning that identifying equivalent mutants is a manual task in the last case.

This way, NIMROD can help reduce this work as it takes a third of the manual time. Besides, NIMROD ranks the mutants indicating which ones are more or less likely to be equivalent. Besides, for a non-equivalent mutant, the average time reduces to 36.57 seconds, which is 25 times less than the manual time.

As reported by Kintis el al. (KINTIS et al., 2018), TCE can analyze the equivalence of a mutant in less than two seconds. In Section 4.6, we present a practical application combining TCE and NIMROD as an alternative for the equivalent mutant problem.

Answer to RQ2: Our approach took an average time of 306.55 seconds per equivalent analyzed mutant and 36.57 seconds per non-equivalent mutant. Once NIMROD kills the mutant, we have the test case. This way, we save time the developer would spend to create a test case to kill the mutant. If compared to manual classification reported in previous work, NIMROD performs better, while manually analyzing a mutant to indicate whether it is equivalent or not can take 15 minutes, NIMROD takes a third of this time to suggest equivalent and is 25 times faster to indicate non-equivalent.

4.5.3 What are the characteristics of the mutants that Nimrod failed to classify?

In this section, we answer RQ3 by analyzing all the false positives of each subject manually. Table 22 presents common source code characteristics that led NIMROD to raise false positives, i.e., it suggested the mutants as equivalent but, in fact, they are not. We focus on three characteristics: *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. In total, 256 (16.60%) mutants out of 1,542 were misclassified. The *addNode* (pamvotis) subjects was responsible for 244 (95%) misclassified mutants. Here we intend to evaluate the false positives qualitatively. In what follows, we discuss each characteristic.

Access Level Modifier occurs when the test case and the source code need to be in the same package structure so that the test can kill the mutant. Listing 4.4 presents a code snippet of the *sqrt* (bisect) subject. In the M_1 mutant, the operator AOIU (Arithmetic Operator Insertion - Unary) inserts a minus operator at the right-hand side of an assignment

Table 22 – Common characteristics (false positives) in the NIMROD's results.

Problem	Description	Subjects
Access Level Modifier	To kill the mutant, the test needs to be in same package structure as the program source code.	sqrt (bisect), addNode (pamvotis), removeNode (pamvotis)
Very Restricted Value	To kill the mutant, the test needs to generate a very specific value.	capitalize (commons-lang), wrap (commons-lang), addNode (pamvotis), removeNode (pamvotis)
External Entities	To kill the mutant, the test needs to execute and assert entities in classes different from the mutated location.	addNode (pamvotis), removeNode (pamvotis)

to a field variable. This transformation changes the value assigned to the field `mResult`. This field has no other use or definition in the `sqrt` method. Likewise, it has no other access to any method of this class to set out a change in the behavior.

This context led us to believe that this mutant could be equivalent, however, as it can be seen in Listing 4.4, this field was declared as *package-private* (no explicit modifier). So, to kill this mutant, it is necessary to use a JAVA language artifice to bypass the field visibility constraint. The test should be created in the same package as the original class under test, and the assertion should observe the state of the field. It has direct access to the field without the need to go through an access method (e.g., `getMResult()`) for this purpose. We were able to configure EVOSUITE to follow the same package structure as the original program. However, we did not get the tests to perform assertions in *package-private* fields.

```
public class Bisect {
    double mEpsilon, mResult;
    ...
    public double sqrt( double N ){
        ...
        while (Math.abs( diff ) > mEpsilon) {
            ...
        }
        r = x;
        mResult = r;          M1 [mResult = r; ⇒ mResult = -r;]
        return r;
    }
}
```

Listing 4.4 – A code snippet extracted from the *sqrt* subject.

In the `Bisect` class example, if the developers of the project had defined that the unit tests and the original source code should be at different packages, the AOIU mutant would be equivalent (CWA). As we are considering all projects based on the notion of OWA, the mutant AOIU is considered non-equivalent. So, here NIMROD failed.

The NIMROD tests must follow the same package structure of the class under test to solve this problem and, also, the test assertion must use the available class fields.

Very Restricted Value occurs when the automatic testing tool does not generate a test with an input that exercises the behavior change made by the mutation. To better explain

this characteristic, we use an example extracted from the *wrap* (commons-lang) subject.

In Listing 4.5, the M_2 mutant (line 20) was generated by the mutation AORB (Arithmetic Operator Replacement) operator. Here, it replaces the arithmetic operator `+` by `%`. To change the behavior of this mutant, the test must reach the mutated line, which is inside several nested `if` statements. Also, the `offset` variable cannot be redefined in the subsequent repetitions of the while. During our study, the tests exercised this mutated point only twice, even though automatic generation tools generated more than 3,000 tests.

The software testing community knows it is challenging to create tests to achieve high branch coverage automatically. A possible solution for NIMROD to solve this problem is to increase the time limit of the tools to generate the tests and allows the generation of a larger number of tests (we limit these settings in 60 seconds of the time limit and a maximum of 3,000 tests). However, these decisions impact directly in the total time to suggest a mutant as equivalent or non-equivalent. New approaches to solve this problem have been presented in recent years (BRAIONE et al., 2017). So, the next versions of the automatic test generation tools are likely to show improvements in this regard.

```
1  public class WorldUtils {
       public static String wrap( String str, int wrapLength, String newLineStr, boolean wrapLongWords ){
3          ...
           while (...) {
5              if (...) {
                   offset++;
7                  continue;
               }
9              if (...) {
                   ...
11                 offset = ...
               } else {
13                 if (...) {
                       ...
15                     offset = ...
                   } else {
17                     spaceToWrapAt = str.indexOf(' ', wrapLength + offset);
                       if (spaceToWrapAt >= 0) {
19                         ...
                           offset = spaceToWrapAt + 1;              M2 [ + ⇒ % ]
21                     } else {
                           ...
23                         offset = ...
                       }
25                 }
               }
27         }
           wrappedLine.append(str.substring(offset));
29         return wrappedLine.toString();
       }
31 }
```

Listing 4.5 – A code snippet extracted from the *wrap* (WordUtils) subject.

External Entities happens when the test needs to execute or assert entities not directly located in the target class where the mutation occurred. Listing 4.6 presents a code snippet extracted from the *addNode* (pamvotis) subject. The method (lines 4-24) has no return statement, and its primary goal is to construct a `MobileNode` object and put this object into a Vector (line 21). In the M_3 mutant, the AOIS operator inserts a post-decrement in the `SpecParams.CW_MAX` variable (Line 13). This global variable is static, public, and has only one definition point in the `SpecParams` class.

```
1  public class Simulator {
       private java.util.Vector nodesList = new java.util.Vector();
3      ...
       public void addNode( int id, int rate, int coverage, int xPosition, int yPosition, int ac )  {
5          ...
           if (...) {...}
7          else {
               pamvotis.core.MobileNode nd = new pamvotis.core.MobileNode();
9              ...
               switch (ac) {
11                 case 1 : {
                       nCwMin = cwMin / cwMinFact1;
13                     nCwMax = SpecParams.CW_MAX / cwMaxFact1;  M3 [SpecParams.CW_MAX ⇒ SpecParams.CW_MAX
                           ↪ --]
                       nAifsd = sifs + aifs1 * slot;
15                     break;
                   }
17                 ...
               }
19             nd.params.InitParams( id, rate, xPosition, yPosition, coverage, ac, nAifsd, nCwMin, nCwMax );
               nd.contWind = nd.params.cwMin;
21             nodesList.addElement( nd );
               nmbrOfNodes++;
23         }
       }
25 }
```

Listing 4.6 – A code snippet extracted from the *addNode* subject.

To identify the behavior change of this mutant, the test needs to assert the state of the `SpecParams.CW_MAX` variable after executing the method `addNode`. However, our impact analysis (presented in Section 4.3) is intraclass and does not consider impacted entities in external classes/files.

As explained, the NIMROD notion of equivalence is based on the behavior exposed by the program through the execution of automatically generated tests. Therefore, to have a satisfactory result, the class under test must be designed so that unit tests can perform properly (BINDER, 1994). However, this is not always the case, so writing a good test in this sense is difficult. When this occurs, NIMROD fails to generate a test that could change the mutant's behavior in comparison to the original program. This situation can lead NIMROD to produce *false positives*. We noticed that the high false-positive number of subject *addNode* (pamvotis) occurred because the system design does not facilitate the

use of unit tests (the project does not have developer written unit tests). In the future, we intend to do refactoring actions to make the code more testable and then repeat the experiment.

Answer to RQ3: We found three classes of characteristics that lead NIMROD to fail in suggesting equivalent mutants: *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. We have seen that there are solutions to decrease the number of false positives by improving the static analysis and, consequently, the configuration to generate automatic tests. The most common characteristic and the most difficult to solve was the *Very Restricted Value*. These mutants are called *stubborn* because they need precise tests to kill them.

4.5.4 Which mutation operators commonly lead Nimrod to fail?

We also analyzed the mutation operators to determine the influence of them on the effectiveness of NIMROD. We counted the number of suggested equivalent mutants per operator that most induced NIMROD to fail. Table 23 reports the total number of mutants analyzed per operator.

For this research question, we are mainly interested in the false positive column. By analyzing the numbers, the AOIS operator stands out in absolute numbers. It generated 650 (42%) mutants. Therefore, this operator was the one that generated the most equivalent and non-equivalent mutants. NIMROD classified 251 AOIS mutants as equivalent. Of these, 125 (50%) were misclassified. This mutation operator is known by the community to generate many mutants and generate many equivalent. Most of these equivalent mutants can be avoided even before their generation (KINTIS; MALEVRIS, 2015; FERNANDES et al., 2017; FERNANDES; RIBEIRO; SANTOS, 2018). Although the AOIS operator produces many equivalent, it also generates many *stubborn* mutants. Such mutants tend to require precise tests, which in turn is good for mutation analysis (YAO; HARMAN; JIA, 2014).

If we analyze from the relative numbers' perspective, the worst performance of NIMROD occurred with the AOIU and AORB operators, where the false-positive rates were close to 30%. However, in both AOIU and AORB, more than 93% of the false positives came from the *addNode* (pamvotis) subject and the failure occurred because this operator changed a field that either belonged to an external entity, or the access level of the field was package-private.

In the other two cases (LOI and ASRS) where NIMROD misclassification occurred, the hit rate on detecting non-equivalent mutants (True Negative column) was above 80%. This way, we did not identify a specific mutation operator that was leading NIMROD to fail.

Table 23 – NIMROD Results by Mutation Operator.

Mutation Operator	Description	Mutants	Nimrod		
			True Positive	True Negative	False Positive
AORB	Arithmetic Operator Replacement (binary)	232	14 (6%)	151(65%)	67 (29%)
AORS	Arithmetic Operator Replacement (short-cut)	8	0	8 (100%)	0
AOIU	Arithmetic Operator Insertion (unary)	108	9 (8%)	67 (62%)	32 (30%)
AOIS	Arithmetic Operator Insertion~(short-cut)	650	126 (20%)	399 (61%)	125 (19%)
AODU	Arithmetic Operator Deletion~(unary)	2	1 (50%)	1 (50%)	0
AODS	Arithmetic Operator Deletion~(short-cut)	0	0	0	0
ROR	Relational Operator Replacement	254	34 (13%)	220 (87%)	0
COR	Conditional Operator Replacement	24	3	21	0
COD	Conditional Operator Deletion	0	0	0	0
COI	Conditional Operator Insertion	67	0	67 (100%)	0
SOR	Shift Operator Replacement	0	0	0	0
LOR	Logical Operator Replacement	0	0	0	0
LOI	Logical Operator Insertion	181	6 (3%)	146 (81%)	29 (16%)
LOD	Logical Operator Deletion	0	0	0	0
ASRS	Assignment Operator Replacement~(short-cut)	16	0	13 (81%)	3 (19%)
Total		**1,542**	**193**	**1,093**	**256**

Answer to RQ4: We identify that 10 MUJAVA mutation operators were responsible for generating 1,542 mutants. Six out of the 10 operators had at least one false-positive mutant. That is, NIMROD classified as equivalent, but this was not. The two mutation operators with the highest false-positive rate were AOIU and AORB. However, the NIMROD hit rate for these operators was higher than the error rate. This way, we did not identify a set of operators that indeed lead to misclassifications. The problem happened due to the three program characteristics we presented.

4.5.5 Threats to Validity

The projects we used represents a threat to external validity, mainly because we focused only on a few methods. To ameliorate this issue, the subjects selected come from different systems and different domains. We selected these subjects due to the previous analysis of equivalence. Nevertheless, we did not test our solution against methods with complex external dependencies, such as: *ObjectC func(ObjectA , ObjectB);*. Dependency objects might be challenging to instantiate. Such methods require more work from the test generation tools since they need to create mocks (ARCURI; FRASER; JUST, 2017) of these objects or to discover valid constructors (which in turn may have other dependencies). Besides, dependencies to some external elements like the graphical user interface or manipulating files could limit test generation tools to generate a test case exposing behavioral change (SOARES et al., 2013).

The previous manual analysis we used to classify the mutants represents a threat to internal validity. However, two previous studies manually analyzed the set of mutants

independently of the present one (KINTIS et al., 2018; KINTIS; MALEVRIS, 2015). Also, TCE confirmed part of the equivalent mutants. For the remaining ones, we also manually analyzed all the equivalent and confirmed the previous manual analysis. It is essential to say that this threat is a consequence of the undecidability of the equivalent mutant problem and covers all the relevant mutation testing studies about this theme.

The selected set of mutation operators also introduces threats to the internal validity of this work. However, this set of mutants is composed of all 15 operators available in MuJava (version 3). We did not pre-exclude any mutant from our investigation. We did not evaluate object-oriented related mutation operators. Previous research (OFFUTT; MA; KWON, 2006) has shown that object-oriented mutation operators yield a small number of mutants and a rather low number of equivalent ones.

In the ranking, we count the number of test cases that reach the mutated point. The code line coverage information computes this metric. This information alone may be narrow. For example, given the expression `if (a > 0 && b < 10)`, if the mutation occurs at the right-hand side of the `&&` operator (e.g., b >= 10), all tests that reach this line, even analyzing only the left-hand side of the expression, are computed as reaching the mutated point. To alleviate this threat, our ranking relies on another metric, i.e., the impact on coverage.

The presence of *flaky tests* (LUO et al., 2014) could also represent a threat. For instance, Nimrod suggests a mutant as non-equivalent because there is a test exposing a behavioral change in the mutant program. However, the mutant is equivalent, and the difference in behavior occurred due to a flaky test. This scenario would represent a false negative for Nimrod. To minimize this threat, we execute the generated tests against the original program to confirm that the tests capture the current behavior of the original program. Besides, both EvoSuite and Randoop have settings to avoid flaky tests. Last but not least, we did not identify any false negative presence among the mutants analyzed manually and subsequently evaluated by TCE.

Other threats are due to the defects in the embedded software. For instance, the static analysis, or the automatic test generators may have defects. In turn, such defects would push down the results. Thus, it is unlikely that the remaining defects would influence our results to a great extent.

It is important to note that all our results form empirical observations that might not hold in the general case. Finally, all our subjects, tools, and data are available on the companion website of the present work (EASY, 2019). It can help to reduce all the threats mentioned above since independent researchers can check, replicate, and analyze our findings.

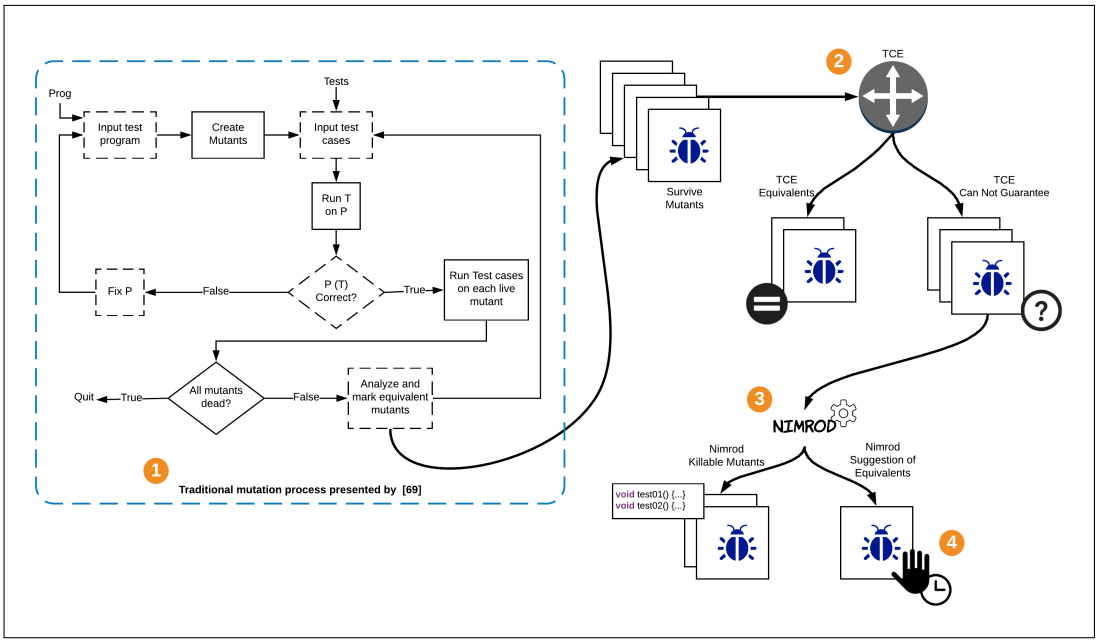


Figure 9 – Combining TCE and NIMROD to minimize the manual analysis to identify equivalent mutants.

4.6 IMPLICATIONS FOR PRACTICE: MINIMIZING THE MANUAL ANALYSIS

Offutt and Untch (OFFUTT; UNTCH, 2000) presented a traditional mutation process. As an implication for the practice of our approach, we extend the mutation process by using TCE and NIMROD to minimize the high cost of manual analysis in the detection of equivalent.

To present the extension, we rely on Figure 9. At the left-hand side (Step 1), there is a reproduction of the Offutt's and Untch's proposition of the typical mutation process. The solid boxes represent steps that are automated by traditional tools such as MUJAVA, and the dashed boxes represent steps done manually. A costly step in the mutation analysis is the manual process on the surviving mutants with the aim of *analyzing and marking the equivalent mutants*. Besides that, the tester needs to identify the non-equivalent to create a test case to kill them. It is precisely in this manual process of analyzing surviving mutants that NIMROD, in combination with TCE, can reduce the cost of mutation testing.

In Step 2, TCE receives as input a set of mutants that were not killed by the application test suite. As demonstrated before (KINTIS et al., 2018), TCE can analyze a broad set of mutants in a few minutes (less than two seconds per mutant). TCE then uses its ability to detect equivalent mutants to automatically and safely discard a significant number of useless mutants in only a small fraction of time.

Because TCE cannot guarantee that the undetected mutants are indeed non-equivalent, we use NIMROD to help with this task. In Step 3, NIMROD receives as input the mutants that TCE could not confirm as equivalent. The time spent by NIMROD to examine each

mutant can vary greatly (refer to Section 4.5.2). In case NIMROD finds a test that exposes a behavioral change, it indicates to the tester that the mutant is not equivalent and informs which test can kill it. If, after exhaustively testing the program and after the timeout, no test is capable of exposing a behavioral change, NIMROD suggests that the mutant may be equivalent, and indicates how many test cases have been able to exercise the mutated point and if occurred any impact on coverage.

Step 4 represents the manual process to identify the equivalent mutants. Instead of analyzing all the surviving mutants from the traditional mutation process, or the mutants that TCE could not confirm, the tester now should consider a smaller number of mutants. Notice that NIMROD also identifies some killable mutants, providing the tests to kill then.

To better illustrate a potential cost reduction, we refer to the `FieldUtils` class (joda-time project) presented in Section 4.3. By executing the MUJAVA, selecting all the available mutation operators, the tool generates 1,339 mutants. After executing the joda-time test suite, 543 mutants got killed, and 796 stayed alive. This result represents a mutation score of 40% (without any analysis of equivalence). By using the traditional process, 796 mutants should be analyzed to identify the killable mutants and mark the equivalent to be ignored in the mutation score.

Our first step in reducing this effort is to provide the live mutants as input for TCE. TCE took 17.33 minutes to analyze 796 mutants. An average of 1.3 seconds per mutant. TCE detected 117 mutants with the *bytecode* equivalent to the original class. This outcome reduces to 679 the number of mutants for analysis. Our next step is to execute our approach against these 679 mutants. NIMROD was able to identify 608 mutants as non-equivalent. This outcome represents 76% of all the mutants analyzed by NIMROD. The total time to analyze these 679 mutants was 24,659 seconds (6.8 hours).

At the end, NIMROD suggested 71 mutants as potential equivalent. We ranked the suggested mutants according to the metrics used by our approach. If we consider the same limit used by our study (Section 4.5), that is, the median is the threshold point, 35 mutants have strong confidence of being indeed equivalent. These mutants had a coverage impact, or a higher number of test cases exercised the mutated point. On the other hand, the 36 out of 71 mutants suggested as equivalent had no impact on coverage, and fewer tests exercised them. Then, NIMROD had weak confidence that these 36 mutants are indeed equivalent.

Table 24 presents a summary of the effort reduction when applying TCE and NIMROD to the traditional mutation process (OFFUTT; UNTCH, 2000).

The question raised here is whether the time required by TCE and NIMROD is acceptable in practice. Previous research estimated the time of the manual identification of a single equivalent mutant to be approximately 15 minutes (SCHULER; ZELLER, 2013). According to Table 24, mutation testers would need to analyze 796 mutants manually. If we apply the suggested average time, a developer would take 199 hours (15 minutes x

Table 24 – Summary of the effort's reduction when combining TCE and NIMROD for the class `FieldUtils` of project joda-Time.

Mutants	Description	
1,339	Total FieldUtils mutants	
-543	Killed by joda-time test suite	
796	Surviving mutants	100.00%
-117	TCE equivalent	17.33 minutes
679	Surviving mutants	⇓ 14.69%
-608	NIMROD non-equivalent	410.99 minutes
71	Surviving Mutants	⇓ 91.08%

796 mutants) to investigate all the first set of surviving mutants. Notice, however, we are not considering the time to create new test cases for the non-equivalent mutants. For the `FieldUtils` class, combining TCE and NIMROD took about 7 hours.

Regarding the 71 mutants suggested as equivalent, it is up to the tester to define how many mutants are manually analyzed. In our study, we defined the threshold based on the median number of test cases that touched the mutated point. In this way, 36 mutants should be manually analyzed.

Notice that we do not intend to generalize the cost reductions we presented in this section. Instead, we intend to show that our approach might reduce costs when analyzing and marking equivalent mutants and consequently might also improve the traditional mutation process (OFFUTT; UNTCH, 2000). The reductions depend on several factors, such as the source code itself. For instance, the `FieldUtils` class public methods are invoked in more than 180 places in program source code and more than 120 places in test code. This way, we understand that this class is designed for testing, which makes the task of automatically generating tests more straightforward. Therefore, the numbers presented in Table 24 might bias in our favor.

4.7 SUMMARY

In this chapter, we defined an approach, implemented in a tool named NIMROD, and empirically investigated its use as a way to reduce the cost of equivalent mutants in mutation testing. The NIMROD approach reduces the manual labor cost on two fronts. First, it suggests equivalent mutants by reducing the number of mutants needed to be analyzed manually. Second, it outputs a test to kill the non-equivalent mutants. Our first findings show that NIMROD can suggest 100% of the mutants correctly in 3 out of 8 studied subjects and achieved above 90% in 2 subjects. In only one case, the performance was below 50%. We also showed that the average time to identify a non-equivalent mutant was 36.57 seconds, and 306.55 seconds (5.1 minutes) to suggest as equivalent, which is much

less than executing this task manually. Also, we found three classes of characteristics that take NIMROD to erroneously classify mutants as equivalent (false positives) and indicated possible improvements. Finally, we identified AOIU and AORB as the mutation operators with the highest false-positive rate and AOIS as the mutation operator with the most significant absolute number of equivalent.

The idea of automatically generating a test suite that is suited for detecting behavioral changes appeared first in the context of refactoring by Soares et al. (SOARES et al., 2010). In this work, they presented *SafeRefacor*, a tool for improving safety during refactoring activities. It checks the observable behavior of the program before and after the transformation to report whether or not the refactoring should be applied. *SafeRefactor* was able to identify bugs in common refactoring engines like Eclipse JDT, NetBeans, and the JastAdd Refactoring Tool (SOARES; GHEYI; MASSONI, 2013). NIMROD is based on *SafeRefactor*. However, we extend the initial idea to adapt to the mutation testing context. While in refactoring, there is a transformation that must preserve the behavior, in the mutation, we want to carry out a transformation that changes the observable behavior of the program. In this way, we have added more tools to generate the tests automatically, we also give back to the user the test that can identify the behavior change (kills the mutant) and calculate the confidence of the equivalence by counting the number of tests that touch the mutated point.

For more information on how to reproduce the experiment, the detailed results, and to download NIMROD, visit our companion website (EASY, 2019). Madeyiski et al. (MADEYSKI et al., 2014) listed three methods to overcome the Equivalent Mutant Problem (which we can also extend to the duplicate mutant problem): *Detecting Equivalent Mutants*, *Suggesting Equivalent Mutants*, and *Avoiding Equivalent Mutants*.

5 RELATED WORK

Addressing the equivalent mutant is not a recent issue. Previous works have been addressing this problem and surveys covering this topic have been published (JIA; HARMAN, 2011; MADEYSKI et al., 2014; PIZZOLETO et al., 2019; PAPADAKIS et al., 2019). The duplicate mutant problem is more recent but has also been faced (PAPADAKIS et al., 2015; KINTIS et al., 2018).

In this chapter, we describe some of the previous studies that proposed solutions to deal with useless mutants and we compare these solutions with our approaches. To better guide reading, we divide this chapter by different topics: strategies to detect useless mutants (Section 5.1), to avoid useless mutants (Section 5.2), and to suggest useless mutants (Section 5.3). We also discuss techniques to deal with useless mutants for specific domains (Section 5.4), and techniques based on *selective mutation* (Section 5.5) Lastly, we discuss other kinds of useless mutants(Section 5.6).

5.1 STRATEGIES TO DETECT

Offutt and Pan (OFFUTT; PAN, 1996; OFFUTT; PAN, 1997) developed a technique to detect equivalent mutants based on mathematical constraints that introduce a set of strategies to formulate the killing conditions of the mutants. Those conditions are generated from a control-flow analysis (some manually) and heuristics to recognize infeasible constraints are applied. If these conditions are not feasible, the mutant is equivalent. In their experiment, they make use of 11 small Fortran programs that generated 7,636 mutants, where 695 were equivalent. The constraint-based technique was able to detect roughly half of the equivalent mutants on average. Voas and McGraw (VOAS; MCGRAW, 1997) first, and Hierons et al. (HIERONS; HARMAN; DANICIC, 1999) afterward, suggested to use program slicing to create simple mutants for C programs and to help with equivalence identification. These approaches suffer from inherent limitations in the scalability of constraint handling and slicing technology. We could have the same scalability problems in case we implement the *i-rules* that need *def-use* analyses. Also, they target detecting equivalent mutants, thus these approaches are orthogonal to our approach.

A long time ago, Baldwin and Sayward (BALDWIN; SAYWARD, 1979) investigated some heuristics for determining the equivalence of mutants using compiler optimizations. The intuition is that code optimization can transform the original program and the mutant in a way in which their compiled object codes are going to be identical. After that, Offutt and Craft (OFFUTT; CRAFT, 1994) performed data-flow analysis in mutants of FORTRAN programs and employed six techniques for compiler optimization. In their experiment, they make use of 15 small FORTRAN programs along with 14 mutation operators. The results

show that, on average of 19,80% of equivalent mutants are detected automatically. More recently, Kintis et al. (KINTIS et al., 2018) developed the Trivial Compiler Equivalence (TCE) and implemented compiler optimizations for JAVA (with `Javac` and `Soot`) and C (with `GCC`). To check the equivalence, TCE used a simple diff program. They reduced 7.4% and 5.7% of all C and JAVA mutants classified as equivalent. Kintis and Malevris (KINTIS; MALEVRIS, 2013) introduced the concept of *mirrored mutants*, which are mutants that exhibit similar behavior, such that identifying one mirrored mutant as being equivalent could help recognize other equivalent mutants. Experiments with 6 small JAVA programs reached the identification 56% equivalent mutants. These strategies have proved to be quite efficient as a solution to *detect* equivalent mutants. We even present a combined solution with TCE + NIMROD (Section 4.6). However, we demonstrated that NIMROD identified equivalent mutants that TCE did not detect. Besides, our *i-rules* can avoid useless mutants saving the mutation generation time.

5.2 STRATEGIES TO AVOID

Previous studies also tackled the problem of equivalent mutants from the *avoiding* perspective. Some of them were discussed in Section 3.2 (KING; OFFUTT, 1991; MRESA; BOTTACI, 1999; HARMAN; HIERONS; DANICIC, 2000). In addition, the work of Offutt et al. (OFFUTT; MA; KWON, 2006) specified heuristics that avoid equivalent mutants for class-level mutation operators. The approach is based on equivalence conditions, which in turn is based on the condition a mutant is killed. They implemented heuristics based on a data-flow analysis for 16 mutation operators and evaluated in 866 classes from six applications. The technique discovered that 74.60% of the class-level mutants could be equivalent and so were avoided. Kintis and Malevris (KINTIS; MALEVRIS, 2015) developed a tool to automate the identification of equivalent and partially equivalent mutants (mutants that are equivalent to the original program for a specific subset of paths) through problematic data-flow patterns. They introduced data-flow patterns and showed that a large portion of equivalent mutants could be avoided by just analyzing the original program under test. They evaluated 165 equivalent mutants from real-world JAVA programs. The tool detected 56% of the equivalent mutants in just 125 seconds. Most of them from the AOIS operator. These ideas require a data-flow or control-flow analysis execution. On the other hand, many of our *i-rules* depend only on AST traversal. Our *i-rules* also work to avoid duplicate mutants. Besides, we also propose a strategy to help with the task of deriving new *i-rules* for mutation tool developers.

5.3 STRATEGIES TO SUGGEST

Other studies check the impact of mutant execution and suggest non-equivalent mutants. Shuler et al. (SCHULER; DALLMEIER; ZELLER, 2009) assessed the impact of mutations using

dynamic invariants, and they demonstrated that mutations that violate invariants are much less likely to be equivalent. The authors evaluated the approach in seven open-source projects and, on average, the top 5% of invariant-violating mutants yield less than 3% of equivalent mutants, which in turn increases the number of killable mutants automatically detected. Shuler and Zeller (SCHULER; ZELLER, 2013) examined whether changes in coverage can be used to detect non-equivalent mutants. On a sample of 140 mutations on seven JAVA programs, the live mutant that alters the control flow of execution, or the data passed between methods has a 75% chance to be likely killable. To deal with fewer equivalent mutants, developers must focus on the mutations with the most impact. Other approaches leverage software clones to identify killable mutants (KINTIS; MALEVRIS, 2013). Since software clones behave similarly, their (non-)equivalent mutants tend to be the same. Therefore, likely killable mutants can be identified by projecting the mutants of one clone to the other. In (KINTIS; PAPADAKIS; MALEVRIS, 2015), Kintis et al. introduced Isolating Equivalent Mutants (I-EQM), a new classification technique to isolate first-order equivalent mutants based on a dynamic execution scheme to detect equivalent mutants. They observed that killable mutants are likely to compose a higher-order mutant that behaves differently than the first-order ones that it is composed of. The results indicate that I-EQM achieves the correct classification in 81% of the killable mutants with a precision of 71%. Comparing with NIMROD, these works trust only on developer written test suites. In our work, besides indicating killable mutants, we also give the test to kill it.

5.4 STRATEGIES FOR VERY SPECIFIC DOMAINS

Other techniques try to solve the equivalent mutant problem for particular domains. Devroey et al. (DEVROEY et al., 2017) used language equivalence of non-deterministic finite automata to detect equivalent in finite behavioral models. Ferrari et al. (FERRARI; RASHID; MALDONADO, 2013) proposed the analysis of join point static shadows to avoid AspectJ equivalent mutants. Wright et al. (WRIGHT; KAPFHAMMER; MCMINN, 2014) used static control-flow and data-flow analysis through a DBMS-independent representation of the relational schema to identify and remove equivalent mutants in database schemas.

All these approaches cannot be extended to general purpose source code mutants.

5.5 ELIMINATING MUTATION OPERATORS

Previous works claimed to reduce the number of useless mutants selecting a small set of mutation operators, known as *Selective Mutation* (OFFUTT; ROTHERMEL; ZAPF, 1993; ADAMOPOULOS; HARMAN; HIERONS, 2004; DEREZIŃSKA; RUDNIK, 2012; GLIGORIC et al., 2013; BLUEMKE; KULESZA, 2014; DELGADO-PÉREZ; SEGURA; MEDINA-BULO, 2017; CARVALHO et al., 2018). Just et al. (JUST; KURTZ; AMMANN, 2017) criticize the strategies based on reducing the number of applied mutation operators because they might work in a

set of programs but might not in another set. This way, depending on the program, these strategies continue to generate a high number of useless mutants. The authors conclude that to discard some of the mutation operators, we need to first understand the program constructs to which the operators will be applied. In our work, we follow this rationale. We do not remove the mutation operators entirely to reduce costs. Rather, we can use all operators, but we avoid specific transformations defined in each *i-rule*.

5.6 OTHER TYPES OF USELESS MUTANTS

In this thesis we focus on two types of useless mutants; *equivalent* and *duplicate*. However, the mutation testing community has been arguing that other types of useless mutants also increase the cost of the analysis.

Petrovic et al. (PETROVIĆ et al., 2018) identified mutants which they call *Unproductive Killable Mutants*. These mutants lead developers to write tests they rarely, if ever, write. For instance, a mutant that change the string message associated with an exception or logging output, even though mutating string messages provide potentially killable mutant. Yet, adding a test that detects such a mutant arguably does not improve the test suite effectiveness. Instead, it bloats the test suite with a meaningless and hard-to-maintain test. Notice that the notion of unproductive mutants is inherently qualitative: different developers may sometimes reach different conclusions. Petrovic et al. conducted a large-scale experiment with 30,000 software developers in six different programming languages, and through a feedback system they classified various unproductive mutants. Unfortunately, results regarding cost reduction were not explicitly shown in the study.

Several other works have been focusing on a different type of mutant called *Redundant Mutant* (KINTIS; PAPADAKIS; MALEVRIS, 2010; PAPADAKIS; MALEVRIS, 2010; PAPADAKIS et al., 2016; AMMANN; DELAMARO; OFFUTT, 2014). Given the mutant set M for program P and the mutants m_1 and $m_2 \in M$, we say that m_1 subsumes m_2 if every test t that kills m_1 also kills m_2, but the reciprocal is not true. This way, redundant mutants are always subsumed by other mutants. The generation of these mutants increases the total cost and does not help to improve the test suite. Ammann et al. (AMMANN; DELAMARO; OFFUTT, 2014) empirically identified that almost 99% of the generated mutants are redundant. The other 1% useful mutants are called the *minimal set*. Also, Papadakis et al.(PAPADAKIS; MALEVRIS, 2010) identified that such redundant mutants inflate the mutation score and that 68% of recent research papers are vulnerable to threats to validity due to the effect of these mutants. The problem is that computing minimal mutant sets for all possible test sets is clearly undecidable (AMMANN; DELAMARO; OFFUTT, 2014), thus we can just propose approximations.

Kaminski et al. (KAMINSKI; AMMANN; OFFUTT, 2013) first and Just et al. (JUST; KAPFHAMMER; SCHWEIGGERT, 2012) afterwards analyzed and evolved redundancy among mutants for relational mutation operators which eliminate redundant mutants. The analysis

proved that only 3 out of 7 mutants of ROR operator should be generated. They used truth table to infer logical relationships across the operations. However, this only works for logical and relational operators. Although the idea is promising, we cannot apply it for non-logical operators. For instance, a binary expression with two numeric variables `a + b` has a very large set of input possibilities, which turn the manual and logical approach much more difficult.

Guimaraes et al. (GUIMARAES et al., 2020) mapped all possible targets that MuJava's mutation operators can apply transformations. Then, they grouped these targets and empirically defined a set of non-redundant mutants (also called *dominants*) for each target. Guimaraes et al. achieved a 64.43% reduction in the number of mutants. Eliminating redundant mutants can greatly assist mutation analysis, however, the work to identify equivalent continues. Our *i-rules* inspired Guimaraes' work to avoid redundant mutants.

6 CONCLUSIONS AND FUTURE WORKS

As we previously described in Chapter 1, this thesis tackles the useless mutants problem. Useless mutants contribute to increasing the general costs of mutation testing. In particular, we focus on two kinds of mutants: equivalent and duplicate mutants. Although techniques to eliminate useless mutants have been applied, their practical use is an open question. This thesis explores and investigates the useless mutants from two perspectives. First, we proposed to improve the transformation rules embedded in the mutation operators. Second, we proposed an approach to suggest equivalent mutants by using automated behavioral testing. Next, we conclude this thesis by discussing the proposals, results, and possibilities for future paths.

6.1 AVOIDING USELESS MUTANTS

The automatic identification of equivalent mutants is an undecidable problem (BUDD; ANGLUIN, 1982). That is, no approach can be defined to detect all equivalent mutants automatically. The same can be said to detect all duplicate mutants. Fortunately, heuristics for detecting some cases exist (JIA; HARMAN, 2011; MADEYSKI et al., 2014; PAPADAKIS et al., 2019; PIZZOLETO et al., 2019).

In Chapter 3, we proposed to improve the transformation rules embedded in the mutation operators to avoid useless mutants. We called these improvements *i-rules*. We divided the *i-rules* in two classes; *e-rule* for avoiding equivalent mutants and *d-rule* for avoiding duplicate mutants.

We realized that it is a challenge to reason about the several possible useless mutants, in particular, if we evaluate the various combinations of mutation operators and possible transformations. Thus, we also presented a strategy to help with identifying new *i-rules*. We evaluated the strategy with 100 JAVA programs as input. To generate the mutants, we used three of the most popular JAVA mutation testing tools; MUJAVA, MAJOR, and PIT. To automatically classify the mutant as equivalent or duplicate candidate, we decided to use automatic test generation tools (RANDOOP and EVOSUITE). As a result, we identified 30 *e-rules* and 69 *d-rules* in all three mutation tools.

We decided to implement a subset of our *i-rules* in the MUJAVA mutation tool to check whether the *e-rules* and *d-rules* can identify useless mutants in industrial-scale projects. We named this tool MUJAVA-AUM. By using well-known open-source projects, we avoided the generation of almost 20% of all considered useless mutants, on average. We also identified that one *e-rule*, alone, was responsible for 90% of all equivalent mutants avoided on average, and six *d-rules* were responsible for approximately 86% of the duplicate mutants avoided. Last but not least, MUJAVA-AUM saved time to generate and compile

the mutants in all systems evaluated.

The high cost of mutation testing is well known. Our *i-rules* can contribute to the development of better tools and decrease costs in mutation testing analysis. In the end, the final users can benefit from mutation tools that generate less useless mutants. Besides, as the programming languages evolve, new constructs are added (e.g., mutation operators for code annotation (PINHEIRO et al., 2018; PINHEIRO et al., 2020)). In this way, developers of mutation testing tools tend (i) to create new mutation operators to cover these new language constructs and (ii) to evolve the existing operators. Therefore, they could use our strategy to derive and implement *i-rules* to deal with both aforementioned situations. Thus, before releasing the tool, developers can improve confidence that the new version can avoid useless mutants when considering the new operators and the evolved ones. However, differently from our evaluation, developers would focus on specific operators (not all available in the tool), reducing costs on carrying out our strategy.

6.2 SUGGESTING EQUIVALENT MUTANTS

In Chapter 4, we defined an approach and empirically investigated the use of automated behavioral testing as a way to reduce the cost regarding equivalent mutants in mutation testing. The NIMROD approach reduces the manual labor cost on two fronts. First, it suggests equivalent mutants by reducing the number of mutants needed to be manually analyzed. Second, it outputs a test to kill the non-equivalent mutants.

Our first findings indicate that NIMROD can be used as a complement to solutions that detect equivalent mutants, such as TCE. We have been able to suggest 100% of the mutants correctly in 3 out of 8 studied subjects and achieved above 90% in 2 subjects. In only one case, the performance was below 50%.

Besides being effective, any practical solution needs to be efficient. We showed that the average time to identify a non-equivalent mutant was 36.57 seconds for all subjects considered. For the mutants suggested as equivalent, where it is necessary to execute all generated tests, the NIMROD took an average of 306.55 seconds (5.1 minutes). This is much less than the 15 minutes previously reported to do this task manually (SCHULER; ZELLER, 2013). In addition, the approach saves the testers time from thinking and implementing the test case to kill the non-equivalent mutant.

We also evaluated the characteristics of the mutants that NIMROD incorrectly identifies as equivalent (false-positive). We found three classes of characteristics, which we call: *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. We observed that improvements in the static analysis and automatic test generation tools could help decrease the number of false-positive. Besides, we pointed out that classes designed to be testable (BINDER, 1994) make NIMROD equivalence analysis easier.

Finally, we investigated the mutation operators. We identified that the two mutation operators with the highest false-positive rate were AOIU and AORB. However, the NIMROD

hit rate for these operators was higher than the error rate. Besides, we also identified that the operator responsible for the most significant absolute number of equivalent mutants was AOIS, and even so, the NIMROD hit rate at this operator was well above the error rate. Thus, we did not identify a set of operators that indeed lead to misclassifications.

6.3 FUTURE WORK

Regarding avoiding useless mutants, we intend to set our strategy to use different JAVA programs and a different solution to detect useless mutants (e.g., TCE). This setup could help to discover new *i-rules*. Besides, we intend to implement the *i-rules* in different mutation testing tools (e.g., PIT). Also, we intend to implement *i-rules* that depend on deeper static analysis.

Regarding suggesting equivalent mutants, we intend to execute NIMROD with different mutation tools such as MAJOR and PIT, as well as different operators like class-level mutation operators (OFFUTT; MA; KWON, 2006). Besides, it is important to add subjects that have complex external dependencies. As an improvement in the approach, we intend to increment NIMROD to reduce the number of false positives for the cases we identified. This decision should include improving the impact analysis. In the current version, we only have two options of impact analysis: *intraclass* and *interclass*. In the case of intraclass, we direct the tests only to the entities impacted in the mutated class itself, which may be insufficient. In the case of interclass, we direct the tests to the entities impacted throughout the project, which can be very large. Perhaps, a middle ground can bring more benefits to our context. Also, we want to increase the confidence in the suggestion of equivalence by assessing the impact of mutant execution with different metrics (SCHULER; ZELLER, 2013). Lastly, we can also evolve in the generation of automatic tests. By understanding the different cases of stubborn mutants, we can better guide automatic testing, or even change these tools for something like mutant-based test generation.

REFERENCES

ACREE, A. T.; BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. *Mutation Analysis*. Atlanta, GA, USA, 1979.

ACREE, J. A. T. *On Mutation*. Tese (Doutorado) — Georgia Institute of Technology, 1980.

ADAMOPOULOS, K.; HARMAN, M.; HIERONS, R. M. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In: *Proceedings of the 3rd Genetic and Evolutionary Computation Conference*. Seattle, USA: Springer, 2004. p. 1338–1349.

AMMANN, P.; DELAMARO, M. E.; OFFUTT, A. J. Establishing Theoretical Minimal Sets of Mutants. In: *Proceedings of the 7th Conference on Software Testing, Verification and Validation*. Cleveland, USA: IEEE, 2014. p. 21–30.

ANDREWS, J.; BRIAND, L.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *Proceedings. 27th International Conference on Software Engineering*. St. Louis, USA: ACM, 2005. p. 402–411.

ARCURI, A.; FRASER, G.; JUST, R. Private api access and functional mocking in automated unit test generation. In: *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*. [S.l.]: IEEE, 2017. p. 126–137.

BALDWIN, D.; SAYWARD, F. *Heuristics for Determining Equivalence of Program Mutations*. [S.l.], 1979.

Basili, V. R.; Rombach, H. D. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, v. 14, n. 6, p. 758–773, June 1988.

BINDER, R. Design for testability in object-oriented systems. *Communications of the ACM*, v. 37, p. 87–101, 1994.

BLUEMKE, I.; KULESZA, K. Reduction in Mutation Testing of Java Classes. In: *Proceedings of the 9th International Conference on Software Engineering and Applications*. Vienna, Austria: IEEE, 2014. p. 297–304.

BONNIN, R. *Machine Learning for Developers*. [S.l.]: Packt Publishing, 2017.

BRAIONE, P.; DENARO, G.; MATTAVELLI, A.; PEZZè, M. Combining symbolic execution and search-based testing for programs with complex heap inputs. In: *Proceedings of the 26th International Symposium on Software Testing and Analysis*. Santa Barbara, USA: ACM, 2017. p. 90–101.

BUDD, T.; ANGLUIN, D. Two notions of correctness and their relation to testing. *Acta Informatica*, v. 18, n. 1, p. 31–45, 1982.

CAPGEMINI; MICROFOCUS. *World Quality Report 2019-20 Top software testing trends for CIOs*. [S.l.], 2019. Disponível em: <www.worldqualityreport.com>.

CARVALHO, L.; GUIMARAES, M.; RIBEIRO, M.; FERNANDES, L.; AL-HAJJAJI, M.; GHEYI, R.; THUM, T. Equivalent mutants in configurable systems: An empirical study. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. Madrid, Spain: ACM, 2018. p. 11–18.

DELGADO-PÉREZ, P.; SEGURA, S.; MEDINA-BULO, I. Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach. *Software Testing, Verification and Reliability*, John Wiley & Sons, v. 27, n. 4-5, p. 1630–1649, 2017.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. *Computer*, v. 11, n. 4, p. 34–41, 1978.

DEMILLO, R. A.; OFFUTT, A. J. Constraint-based Automatic Test Data Generation. *Transactions on Software Engineering*, IEEE, v. 17, n. 9, p. 900–910, 1991.

DEREZIŃSKA, A.; RUDNIK, M. Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs. In: *Proceedings of the 50th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Prague, Czech Republic: Springer, 2012. p. 42–57.

DEVROEY, X.; PERROUIN, G.; PAPADAKIS, M.; LEGAY, A.; SCHOBBENS, P.-Y.; HEYMANS, P. Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation. In: *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*. Tokyo, Japan: IEEE, 2017. p. 424–429.

EASY. *Nimrod - A tool to suggest equivalent mutants*. 2019. <https://github.com/easy-software-ufal/nimrod-hunor/wiki/Nimrod> - Accessed: 2019-08-30.

FERNANDES, L.; RIBEIRO, M.; CARVALHO, L.; GHEYI, R.; MONGIOVI, M.; SANTOS, A.; CAVALCANTI, A.; FERRARI, F. C.; MALDONADO, J. C. Avoiding Useless Mutants. In: *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experiences*. Vancouver, Canada: ACM, 2017. p. 187–198.

FERNANDES, L.; RIBEIRO, M.; PINHEIRO, P.; GHEY, R.; SANTOS, A. *Improving Transformation Rules to Avoid Useless Mutants – Companion Website*. 2020. Online – last accessed on January 2020. <https://easy-software-ufal.github.io/i-rules/>.

FERNANDES, L.; RIBEIRO, M.; SANTOS, A. Rules to avoid useless mutants. In: *8th Workshop on Theses and Dissertations of CBSoft*. São Carlos, Brazil: ACM, 2018.

FERRARI, F. C.; RASHID, A.; MALDONADO, J. C. Towards the Practical Mutation Testing of AspectJ Programs. *Science of Computer Programming*, Elsevier, v. 78, n. 9, p. 1639–1662, 2013.

FRANKL, P. G.; WEISS, S. N.; HU, C. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, v. 38, n. 3, p. 235 – 253, 1997. ISSN 0164-1212.

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*. Szeged, Hungary: ACM, 2011. p. 416–419.

FRASER, G.; ARCURI, A. Whole test suite generation. *IEEE Transactions on Software Engineering*, v. 39, n. 2, p. 276 –291, 2013.

FRASER, G.; STAATS, M.; MCMINN, P.; ARCURI, A.; PADBERG, F. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, ACM, v. 24, n. 4, p. 23, 2015.

GLIGORIC, M.; ZHANG, L.; PEREIRA, C.; POKAM, G. Selective Mutation Testing for Concurrent Code. In: *Proceedings of the 22nd International Symposium on Software Testing and Analysis*. Lugano, Switzerland: ACM, 2013. p. 224–234.

GOPINATH, R.; ALIPOUR, M. A.; AHMED, I.; JENSEN, C.; GROCE, A. On the Limits of Mutation Reduction Strategies. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, 2016. p. 511–522. ISBN 978-1-4503-3900-1.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A. *The Java® Language Specification - Java SE 8 Edition*. 2015. <https://docs.oracle.com/javase/specs/> - Accessed: 2019-01-30.

GRÜN, B. J.; SCHULER, D.; ZELLER, A. The impact of equivalent mutants. In: *The 5th International Workshop on Mutation analysis*. Paris, France: IEEE, 2009. p. 192–199.

GUIMARAES, M. A.; FERNANDES, L.; RIBEIRO, M.; D'AMORIM, M.; GHEYI, R. Optimizing mutation testing by discovering dynamic mutant subsumption relations. In: *International Conference on Software Testing, Verification and Validation*. Porto, Portugal: IEEE, 2020. To Appear.

HARMAN, M.; HIERONS, R. M.; DANICIC, S. The Relationship Between Program Dependence and Mutation Analysis. In: *Proceedings of the Mutation 2000 Symposium*. San Jose, USA: Kluwer Academic Publishers, 2000. p. 5–13.

HIERONS, R.; HARMAN, M.; DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, v. 9, n. 4, p. 233–262, 1999.

HOWDEN, W. Weak mutation testing and completeness of test sets. *Transactions on Software Engineering*, IEEE, n. 4, p. 371–379, 1982.

IGARASHI, A.; PIERCE, B.; WADLER, P. Featherweight java: A minimal core calculus for java and gj. *ACM SIGPLAN Notices*, ACM, v. 34, n. 10, p. 132–146, 1999.

JACKSON, D. *Software Abstractions: logic, language, and analysis*. Boston, USA: MIT press, 2012.

JACKSON, D.; SCHECHTER, I.; SHLYAHTER, H. Alcoa: the alloy constraint analyzer. In: *Proceedings of the 22nd International Conference on Software Engineering*. Limerick, Ireland: ACM, 2000. p. 730–733.

JIA, Y.; HARMAN, M. Constructing subtle faults using higher order mutation testing. In: *International Working Conference on Source Code Analysis and Manipulation*. Beijing, China: IEEE, 2008. p. 249–258.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, v. 37, n. 5, p. 649–678, 2011.

JUST, R.; JALALI, D.; INOZEMTSEVA, L.; ERNST, M. D.; HOLMES, R.; FRASER, G. Are Mutants a Valid Substitute for Real Faults in Software Testing? In: *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014. p. 654–665.

JUST, R.; KAPFHAMMER, G. M.; SCHWEIGGERT, F. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis? In: *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*. Montreal, Canada: IEEE, 2012. p. 720–725.

JUST, R.; KURTZ, B.; AMMANN, P. Inferring mutant utility from program context. In: *Proceedings of the International Symposium on Software Testing and Analysis*. Santa Barbara, USA: ACM, 2017. p. 284–294.

JUST, R.; SCHWEIGGERT, F.; KAPFHAMMER, G. Major: An efficient and extensible tool for mutation analysis in a Java compiler. In: *Proceedings of the 26th International Conference on Automated Software Engineering*. Lawrence, USA: IEEE/ACM, 2011. p. 612–615.

KAMINSKI, G.; AMMANN, P.; OFFUTT, A. J. Improving logic-based testing. *Journal of Systems and Software*, Elsevier, v. 86, n. 8, p. 2002 – 2012, 2013.

KING, K. N.; OFFUTT, A. J. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, v. 21, n. 7, p. 685–718, 1991.

KINTIS, M.; MALEVRIS, N. Identifying More Equivalent Mutants via Code Similarity. In: *Proceedings of the 20th Asia-Pacific Software Engineering Conference*. Bangkok, Thailand: IEEE, 2013. p. 180–188.

KINTIS, M.; MALEVRIS, N. Medic: A static analysis framework for equivalent mutant identification. *Information and Software Technology*, Elsevier, v. 68, p. 1 – 17, 2015.

KINTIS, M.; PAPADAKIS, M.; JIA, Y.; MALEVRIS, N.; TRAON, Y. L.; HARMAN, M. Detecting trivial mutant equivalences via compiler optimisations. *Transactions on Software Engineering*, IEEE, v. 44, n. 4, p. 308–333, 2018.

KINTIS, M.; PAPADAKIS, M.; MALEVRIS, N. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In: *Proceedings of the 17th Asia-Pacific Software Engineering Conference*. Sydney, Australia: ieee, 2010. p. 300–309.

KINTIS, M.; PAPADAKIS, M.; MALEVRIS, N. Employing second-order mutation for isolating first-order equivalent mutants. *Software Testing, Verification and Reliability*, wiley, Malden, MA, USA, v. 25, n. 5-7, p. 508–535, 2015.

KINTIS, M.; PAPADAKIS, M.; PAPADOPOULOS, A.; VALVIS, E.; MALEVRIS, N. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In: *16th International Working Conference on Source Code Analysis and Manipulation*. Raleigh, USA: IEEE, 2016. p. 147–156.

KINTIS, M.; PAPADAKIS, M.; PAPADOPOULOS, A.; VALVIS, E.; MALEVRIS, N.; TRAON, Y. L. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, p. 2426–2463, 2018.

KURTZ, B.; AMMANN, P.; OFFUTT, J.; KURTZ, M. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In: *Proceedings of the 11th International Workshop on Mutation Analysis*. Chicago, USA: IEEE, 2016. p. 142–151.

LAKHOTIA, K.; MCMINN, P.; HARMAN, M. Automated test data generation for coverage: Haven't we solved this problem yet? *Journal of Systems and Software*, ACM, p. 95–104, 2009.

LI, S.; XIAO, X.; BASSETT, B.; XIE, T.; TILLMANN, N. Measuring code behavioral similarity for programming and software engineering education. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin, USA: ACM, 2016. p. 501–510.

LIPTON, R. Fault diagnosis of computer programs. *Student Report, Carnegie Mellon University*, 1971.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An empirical analysis of flaky tests. In: *Proceedings of the 22Nd International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014. p. 643–653.

MA, Y.-S.; KWON, Y.-R.; OFFUTT, J. Inter-class mutation operators for java. In: *Proceedings of the International Symposium on Software Reliability Engineering*. Annapolis, USA: IEEE, 2002. p. 352–.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Mujava: An automated class mutation system. *Software Testing, Verification and Reliability*, v. 15, n. 2, p. 97–133, 2005.

MADEYSKI, L.; ORZESZYNA, W.; TORKAR, R.; JOZALA, M. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *Transactions on Software Engineering*, IEEE, v. 40, n. 1, p. 23–42, 2014.

MATHUR, A. P. Performance, Effectiveness, and Reliability Issues in Software Testing. In: *Proceedings of the 15th Annual Computer Software and Applications Conference*. Tokyo, Japan: IEEE, 1991. p. 604–605.

MONGIOVI, M.; GHEYI, R.; SOARES, G.; TEIXEIRA, L.; BORBA, P. Making refactoring safer through impact analysis. *Science of Computer Programming*, Elsevier, v. 93, p. 39–64, 2014.

MONGIOVI, M.; GHEYI, R.; SOARES, G.; RIBEIRO, M.; BORBA, P.; TEIXEIRA, L. Detecting overly strong preconditions in refactoring engines. *Transactions on Software Engineering*, IEEE, PP, n. 99, p. 1–1, 2017.

MRESA, E. S.; BOTTACI, L. Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study. *Software Testing Verification and Reliability*, wiley, Malden, USA, v. 9, n. 4, p. 205–232, 1999.

NIELSON, F.; NIELSON, H.; HANKIN, C. *Principles of Program Analysis*. New York, USA: Springer, 1999.

OFFUTT, A. J.; MA, Y.-S.; KWON, Y.-R. The Class-Level Mutants of MuJava. In: *Proceedings of the International Workshop on Automation of Software Test*. Shanghai, China: ACM, 2006. p. 78–84.

OFFUTT, A. J.; PAN, J. Detecting Equivalent Mutants and the Feasible Path Problem. In: *Proceedings of the 11th Annual Conference on Computer Assurance*. Gaithersburg, USA: ACM, 1996. p. 224–236.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An Experimental Evaluation of Selective Mutation. In: *Proceedings of the 15th International Conference on Software Engineering*. Baltimore, USA: ACM/IEEE, 1993. p. 100–107.

OFFUTT, A. J.; UNTCH, R. H. Mutation 2000: Uniting the Orthogonal. In: *Proceedings of the Mutation 2000 Symposium*. San Jose, CA, USA: Kluwer Academic Publishers, 2000. p. 34–44. ISSN 1386-2944.

OFFUTT, J.; CRAFT, M. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, Wiley, Malden, USA, v. 4, n. 3, p. 131–154, 1994.

OFFUTT, J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, v. 7, n. 3, p. 165–192, 1997.

PACHECO, C.; LAHIRI, S.; ERNST, M.; BALL, T. Feedback-directed random test generation. In: *Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, USA: ACM/IEEE, 2007. p. 75–84.

PAPADAKIS, M.; HENARD, C.; HARMAN, M.; JIA, Y.; TRAON, Y. L. Threats to the validity of mutation-based test assessment. In: ACM. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrucken, Germany, 2016. p. 354–365.

PAPADAKIS, M.; JIA, Y.; HARMAN, M.; Le Traon, Y. 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*. Florence, Italy: ACM, 2015. p. 936–946.

PAPADAKIS, M.; KINTIS, M.; ZHANG, J.; JIA, Y.; TRAON, Y. L.; HARMAN, M. Mutation testing advances: an analysis and survey. *Advances in Computers*, Elsevier, v. 112, p. 275–378, 2019.

PAPADAKIS, M.; MALEVRIS, N. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In: *Proceedings of the 5th International Workshop on Mutation Analysis*. Paris, France: IEEE, 2010. p. 90–99.

PAPADAKIS, M.; SHIN, D.; YOO, S.; BAE, D.-H. Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg, Sweden: ACM/IEEE, 2018. p. 537–548.

PETROVIĆ, G.; IVANKOVIĆ, M. State of Mutation Testing at Google. In: *Proceedings of the 40th International Conference on Software Engineering - Software Engineering in Practice Track*. Gothenburg, Sweden: IEEE, 2018. p. 163–171. ISBN 978-1-4503-5659-6.

PETROVIĆ, G.; IVANKOVIĆ, M.; KURTZ, B.; AMMANN, P.; JUST, R. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In: *Proceedings of the 13th International Workshop on Mutation Analysis*. Vasteras, Sweden: IEEE, 2018. p. 47–53.

PINHEIRO, P.; VIANA, J.; FERNANDES, L.; RIBEIRO, M.; FERRARI, F.; FONSECA, B.; GHEYI, R. Mutation operators for code annotations. In: . São Carlos, Brazil: ACM, 2018. p. 77–86.

PINHEIRO, P.; VIANA, J. C.; RIBEIRO, M.; FERNANDES, L.; FERRARI, R. G. F.; FONSECA, B. Mutating code annotations: An empirical evaluation on java and c# programs. *Science of Computer Programming*, February 2020. Accepted for publication.

PITEST. *PITest - Mutation Testing Tool for Java*. 2017. <http://pitest.org/> - Accessed: 2017-05-20.

PIZZOLETO, A. V.; FERRARI, F. C.; OFFUTT, A. J.; FERNANDES, L.; RIBEIRO, M. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software*, v. 157, 2019. (in press).

SCHäFER, M.; EKMAN, T.; MOOR, O. de. Challenge proposal: Verification of refactorings. In: *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*. Savannah, GA, USA: ACM, 2009. p. 67–72.

SCHULER, D.; DALLMEIER, V.; ZELLER, A. Efficient Mutation Testing by Checking Invariant Violations. In: *Proceedings of the 18th International Symposium on Software Testing and Analysis*. Chicago, USA: ACM, 2009. p. 69–80.

SCHULER, D.; ZELLER, A. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, v. 23, n. 5, p. 353–374, 2013.

SHAMSHIRI, S.; JUST, R.; ROJAS, J. M.; FRASER, G.; MCMINN, P.; ARCURI, A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: *30th International Conference on Automated Software Engineering*. Lincoln, USA: ACM/IEEE, 2015. p. 201–211.

SOARES, G.; GHEYI, R.; MASSONI, T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, v. 39, n. 2, p. 147–162, 2013.

SOARES, G.; GHEYI, R.; MURPHY-HILL, E.; JOHNSON, B. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, v. 86, n. 4, p. 1006–1022, 2013.

SOARES, G.; GHEYI, R.; SEREY, D.; MASSONI, T. Making program refactoring safer. *IEEE software*, v. 27, n. 4, p. 52–57, 2010.

STEIMANN, F.; THIES, A. From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation. In: *Proceedings of the 32th International Conference on Software Engineering*. Cape Town, South Africa: ACM, 2010. p. 425–434.

STEIMANN, F.; THIES, A. From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation. In: *Proceedings of the 32th International Conference on Software Engineering*. Cape Town, South Africa: ACM, 2010. p. 425–434.

UNTCH, R. H.; OFFUTT, A. J.; HARROLD, M. J. Mutation Analysis Using Mutant Schemata. In: *Proceedings of the International Symposium on Software Testing and Analysis*. Cambridge, MA, USA: ACM, 1993. p. 139–148.

VOAS, J.; MCGRAW, G. *Software fault injection: inoculating programs against errors*. New York, USA: John Wiley & Sons, 1997.

WOODWARD, M.; HALEWOOD, K. 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*. Banff, Canada: IEEE, 1988. p. 152–158.

WRIGHT, C.; KAPFHAMMER, G.; MCMINN, P. The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis. In: *Proceedings of the 14th International Conference on Quality Software*. Dallas, TX, USA: IEEE, 2014. p. 57–66.

YAO, X.; HARMAN, M.; JIA, Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India: ACM, 2014. p. 919–930.