Journal of Software Engineering Research and Development, 2024, 12:3, doi: 10.5753/jserd.2024.3588

Reducing Manual Efforts in Equivalence Analysis in Mutation Testing

Samuel Amorim [**Instituto Federal de Alagoas** | ***sva2@aluno.ifal.edu.br***]
Leo Fernandes [**Instituto Federal de Alagoas** | ***leonardo.fernandes@ifal.edu.br***]
Márcio Ribeiro [**Universidade Federal de Alagoas** | ***marcio@ic.ufal.br***]
Rohit Gheyi [**Universidade Federal de Campina Grande** | ***rohit@dsc.ufcg.edu.br***]
Marcio Delamaro [**Universidade de São Paulo** | ***delamaro@icmc.usp.br***]
Marcio Guimarães [**Centro de Inovação EDGE** | ***marcio.guimaraes@ic.ufal.br***]
André Santos [**Universidade Federal de Pernambuco** | ***alms@cin.ufpe.br***]

Abstract
Mutation testing has attracted a lot of interest because of its reputation as a powerful adequacy criterion for test suites and for its ability to guide test case generation. However, the presence of equivalent mutants hinders its usage in industry. The Equivalent Mutant Problem has already been proven undecidable, but manually detecting equivalent mutants is an error-prone and time-consuming task. Thus, solutions, even partial, can help reduce this cost. To minimize this problem, we introduce an approach to suggest equivalent mutants. Our approach is based on automated behavioral testing, which consists of test cases based on the behavior of the original program. We perform static analysis to automatically generate tests for the entities impacted by the mutation. For each mutant analyzed, our approach can suggest the mutant as equivalent or non-equivalent. In the case of non-equivalent mutants, our approach provides a test case capable of killing it. For the equivalent mutants suggested, we also provide a *ranking* of mutants with a strong or weak chance of the mutant being indeed equivalent. In our previous work, we evaluated our approach against a set of 1,542 mutants manually classified in previous work as equivalents and non-equivalents. We noticed that the approach effectively suggests equivalent mutants, reaching more than 96% of accuracy in five out of eight subjects studied. Compared with manual analysis of the surviving mutants, our approach takes a third of the time to suggest equivalents and is 25 times faster to indicate non-equivalents. This extended article delves deeper into our evaluation. Our focus is on discerning the specific characteristics of mutants that our approach erroneously classified as equivalent, thereby producing false positives. Furthermore, our investigation delves into a comprehensive analysis of the mutation operators, providing essential insights for practitioners seeking to improve the accuracy of equivalent mutant detection and effectively mitigate associated costs.

Keywords: *mutation testing, equivalent mutant, automated testing*

1 Introduction

Mutation testing is a powerful technique employed to enhance the testing process (DeMillo et al., 1978; Jia and Harman, 2011). Over the years, it has garnered significant attention due to empirical studies demonstrating its ability to improve test suites (Andrews et al., 2005; Just et al., 2014; Papadakis et al., 2018). The fundamental idea behind mutation testing involves applying syntactical transformations, known as mutation operators, to introduce artificial faults into a program, resulting in what is termed a *mutant*. Subsequently, the existing test suite is executed against these mutants to ascertain whether the faults are detectable.

In the context of mutation testing, when a test suite is capable of distinguishing the output of the original program from that of the mutant, the mutant is considered *killed*. Conversely, if the mutant remains indistinguishable from the original program, it is labeled as *alive*. The underlying premise of mutation testing is that the higher the number of killed mutants, the higher the quality of the test suite.

However, the costs of using mutation testing are usually high, mostly due to the *Equivalent Mutant Problem* (Madeyski et al., 2014; Kintis et al., 2018). An equivalent mutant is syntactically different from the original program but has the same behavior as such a program regarding the observable output. This way, there is no test able to kill it. Equivalent mutants are a well-known impediment to the practical adoption of mutation testing. A previous work (Budd and Angluin, 1982) has already proven that this is an undecidable problem in its general form. Thus, no completely automated solution exists. In addition, manually detecting equivalent mutants is an error-prone and time-consuming task. 20% of the studied mutants were erroneously classified (Allen Troy Acree, 1980) and developers take, on average, 15 minutes to manually classify a mutant as equivalent or nonequivalent (Schuler and 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). In this way, research efforts to reduce these costs are still needed.

This article presents an extension of our previous work on mutation testing (Fernandes et al., 2022), wherein we propose an innovative approach to suggesting equivalent mutants using automated behavioral testing (Soares et al., 2013a) and ranking the mutants based on their behavior during test execution. Our approach leverages automated test case generation tools to create targeted test cases for the location where the mutation occurred. By executing these tests against both the original program and the mutant, we determine whether there are behavioral differences. If any test case fails against the mutant, indicating a behavioral change, the mutant is classified as *non-equivalent*. Testers can use such tests to improve their test suite, effectively reducing costs. Conversely, if no test cases kill the mutant, our approach *suggests the mutant as equivalent*. In the absence of definitive proof of equivalence, we offer a ranking system for mutants based on their behavior during test execution. This ranking hinges on two key factors observed throughout testing: the coverage impact, indicating discrepancies in test execution coverage between the mutant and the original program, and the number of test cases exercising the mutated statement. Through the synthesis of these factors, we establish a tiered ranking of mutants proposed as potential equivalents, with mutants having stronger evidence of equivalence placed at the bottom, and those with weaker evidence placed at the top. This prioritized ranking facilitates testers in determining the mutants deserving of manual review, initiating the process from the top. Our approach is implemented in a tool named Nimrod.

To the best of our knowledge, no alternative strategy has employed automated test case generation specifically targeted at the mutated point to acquire equivalence information. While several commendable strategies have been proposed to assist testers, they confront inherent scalability limitations, such as those associated with program slicing (Voas and McGraw, 1997) and the impact of dynamic invariants (Schuler et al., 2009). Recently, emerging techniques have employed machine learning to categorize equivalent mutants (Brito et al., 2020; Naeem et al., 2020; Peacock et al., 2021). Despite their promising nature, these techniques necessitate evaluation not only in terms of effectiveness but also efficiency.

In our previous work, we evaluated our approach against 1,542 mutants generated from eight methods and manually classified as equivalent or non-equivalent in prior research (Kintis et al., 2018). We submit all these mutants to Nimrod suggesting equivalent mutants, and then compute precision, recall, and F-measure to check its effectiveness. The results indicate that the approach is effective in suggesting equivalent mutants. The F-measure has reached more than 96% in five out of the eight methods we studied. To better analyze our approach, we also computed 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 and Zeller, 2013)) and 24 seconds to classify a mutant as non-equivalent. Our results include the time to generate the test cases using both Randoop and EvoSuite.

This extended article delves deeper into our evaluation, shedding light on the challenges associated with detecting equivalent mutants through mutation testing. We specifically analyze the characteristics of mutants that presented difficulties in classification by Nimrod, our tool for automated behavioral testing, thus providing valuable insights for refining the mutation testing approach and improving equivalent mutant detection in practice. Additionally, we investigate the mutation operators frequently associated with misclassifications by Nimrod, offering crucial guidance to practitioners aiming to enhance the accuracy of equivalent mutant detection and reduce costs.

The contributions of this paper encompass:

- An innovative approach for suggesting equivalent mutants based on automated behavioral testing (Section 3).
- Development and automation of the entire approach through a tool named Nimrod (available online (Fernandes, Leo, 2023)) (Section 3).
- A comprehensive evaluation of the effectiveness and efficiency of our approach, shedding light on its performance in classifying mutants as equivalent or non-equivalent (Sections 4 and 5).
- An in-depth analysis of the challenges associated with detecting equivalent mutants, with a particular focus on the characteristics of mutants and mutation operators that led to misclassifications (Section 5).
- Implications for practitioners on how to combine the methods of *Suggesting Equivalent Mutants* and *Detecting Equivalent Mutants* to optimize cost management in addressing the equivalent mutation problem (Section 6).

2 Motivating Example

Detecting equivalent mutants poses a challenging problem in mutation testing, as it is known to be undecidable (Budd and Angluin, 1982). Consequently, the task of identifying equivalent mutants often falls upon human testers. However, manual detection of equivalent mutants is error-prone, with correct judgments achieved in only about 80% of cases (Allen Troy Acree, 1980), and time-consuming, taking approximately 15 minutes per equivalent mutant (Schuler and Zeller, 2013). Thus, the need for effective heuristics to identify a subset of equivalent mutants becomes imperative in order to minimize costs.

One such heuristic, commonly employed, relies on compiler optimizations (Offutt and Craft, 1994; Kintis et al., 2018). The basic idea behind this approach is that code optimizations may result in identical compiled object codes for both the original program and an equivalent mutant. The concept of Trivial Compiler Equivalence (TCE) was already introduced and implemented for popular compiled languages such as C and Java, along with mutation testing tools like Milu and Mujava (Kintis et al., 2018). The TCE approach is sound, meaning that if two binaries are equal, the programs have the same behavior. Consequently, it does not produce false positives. However, it may not detect equivalent mutants with the same behavior but different object codes, leading to potential false negatives.

To illustrate this scenario, consider the `Triangle` class presented in Listing 1. This class contains a method named `classify` that determines the type of a triangle based on the sizes of its three sides. We have generated four different equivalent mutants using the Mujava mutation tool:

Listing 1: A code snippet extracted from the `Triangle` class.

```
 1  public static int classify( int a, int b, int c ) {
 2    int tri;
 3    if (a <= 0 || b <= 0 || c <= 0) { return INVALID; }
 4    tri = 0;
 5    if (a == b) { tri = tri + 1; }  M1 [tri + 1 ⇒ -tri + 1]
 6    if (a == c) { tri = tri + 2; }
 7    if (b == c) { tri = tri + 3; }
 8    if (tri == 0) {
 9      if (a + b < c || a + c < b || b + c < a) {
10        return INVALID;
11      } else {
12        return SCALENE;
13      }
14    }
15    if (tri > 3) {
16      return EQUILATERAL;
17    }
18    if (tri == 1 && a + b > c) {    M2 [tri == 1 ⇒ tri <= 1]
19      return ISOSCELES;
20    } else {
21      if (tri == 2 && a + c > b) {  M3 [a + c > b ⇒ a + c > b++]
22        return ISOSCELES;
23      } else {
24        if (tri == 3 && b + c > a) {  M4 [tri == 3 ⇒ tri++ == 3]
25          return ISOSCELES;
26        }
27      }
28    }
29    return INVALID;
30  }
```

- M_1 represents a mutant created by the AOIU (Arithmetic Operator Insertion - unary) operator, with the transformation: `tri + 1`⇒`-tri + 1`.
- M_2 represents a mutant generated by the ROR (Relational Operator Replacement) operator, with the transformation: `tri == 1`⇒`tri <= 1`.
- M_3 and M_4 are mutants created by the AOIS (Arithmetic Operator Insertion - short-cut) operator. Both insert a post-increment at the last access of the local variable `b`.

By running the TCE against the mutants, it detects two out of the four mutants: M_1 and M_4. When analyzing mutant M_2, one can see that the `tri` value starts with zero (line 4). However, when reaching the mutated line (line 18), the `tri` value can only be one or greater than one, which prevents the behavior change for any possible entry of the program. To identify this equivalent mutant, the compiler would need to check the conditional expression of the `if` statement at line 8. In case the condition is true, `tri` is zero and the method returns.

When applying TCE to detect equivalent mutants, it successfully identifies M_1 and M_4 as equivalent mutants. However, for M_2, detecting equivalence through code optimization becomes challenging. The value of `tri` is initialized to zero (line 4), and at the mutated line (line 18), `tri` can only be one or greater, making it difficult to detect the behavioral change for any possible program input. Similarly, for M_3, applying a post-increment to the last access of a local variable within a method will not alter the program's behavior (Kintis and Malevris, 2015; Fernandes et al., 2017). However, this mutant's equivalence is hidden in the specification of the `&&` operator (Gosling et al., 2022). The conditional-and operator `&&` only evaluates its right-hand operand if the left-hand operand is `true`. The left-hand side of the conditional expression to which M_3 belongs is mutually exclusive with the left-hand side of another conditional expression at line 24. Thus, detecting such equivalence through compilation optimization becomes intricate.

In this research, we propose an alternative approach to suggest equivalent mutants, leveraging automated behavioral testing. While our approach cannot guarantee with absolute certainty that a mutant is indeed equivalent, it can provide strong or weak suggestions based on ranking the surviving mutants. Returning to the motivating example above, our approach correctly identified all equivalent mutants. The details of our approach are elaborated in the subsequent section.

3 Suggesting Equivalent Mutants

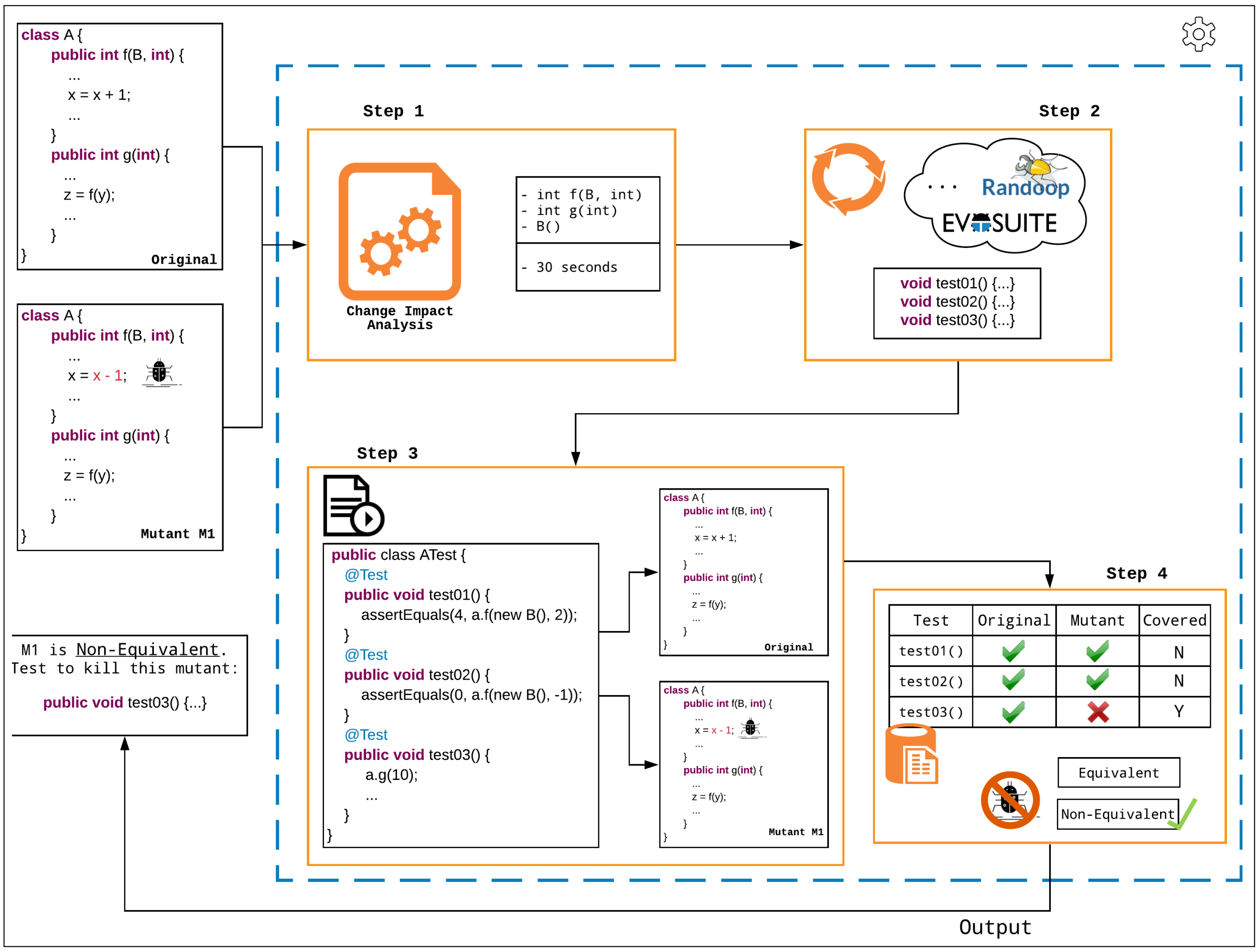
In this section, we explain the proposed approach to suggest equivalent mutants. Our approach is based on previous work in the area of refactoring (Soares et al., 2013a). While refactoring is a transformation that preserves the external behavior of a program, a mutant must transform a program so that the program's external behavior changes (Steimann and Thies, 2010). 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.

Figure 1 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.

3.1 Identifying Impacted Entities

In Step 1, our approach receives two versions of the program as input: the original and the mutant. 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. (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 and the set of public methods that need to be executed to reach these entities directly or indirectly. Besides the public methods, we also analyze the parameters of such methods to identify methods dependency.

To illustrate the process of identifying impacted entities, we consider the `FieldUtils` class from the Joda-time project as an example (Listing 2). Joda-time is a popular date and time Java library. The `FieldUtils` class has 158 lines of code, 17 methods, and no field. We introduce three different mutants: $M1$ replaces the logical AND (`&&`) with the logical OR (`||`) on line 7, $M2$ inserts a post-increment (`++`) to the `total` variable on line 17, and $M3$ inserts a pre-decrement (`--`) to the `val2` variable on line 24.



```
class A {
    public int f(B, int) {
        ...
        x = x + 1;
        ...
    }
    public int g(int) {
        ...
        z = f(y);
        ...
    }
}                                   Original
```

```
class A {
    public int f(B, int) {
        ...
        x = x - 1;
        ...
    }
    public int g(int) {
        ...
        z = f(y);
        ...
    }
}                                   Mutant M1
```

```
- int f(B, int)
- int g(int)
- B()

- 30 seconds
```

```
void test01() {...}
void test02() {...}
void test03() {...}
```

```
public class ATest {
    @Test
    public void test01() {
        assertEquals(4, a.f(new B(), 2));
    }
    @Test
    public void test02() {
        assertEquals(0, a.f(new B(), -1));
    }
    @Test
    public void test03() {
        a.g(10);
        ...
    }
}
```

Test	Original	Mutant	Covered
test01()	✓	✓	N
test02()	✓	✓	N
test03()	✓	✗	Y

```
M1 is Non-Equivalent.
Test to kill this mutant:

public void test03() {...}
```

Figure 1. Our approach to suggest equivalent mutants.

Listing 2: An excerpt extracted from the `FieldUtils` class.

```
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 &&    M_1 [ && ⇒ || ]
      value <= Integer.MAX_VALUE) {
      return (int) value;
    }
    throw new ArithmeticException(...);
  }
  public static long safeMultiply(long val1, long val2) { ...
    long total = val1 * val2; ...
    return total;     M_2 [ 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);     M_3 [ val2 ⇒ --val2 ]
    }
    return diff;
  } ...
}
```

$M1$ occurred in the `safeToInt` method. Notice that this method is called by another method, `safeMultiplyToInt`. This way, the output of Step 1 is the following:

> 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:

> m:FieldUtils.safeMultiply(long, long)
> m:FieldUtils.safeMultiplyToInt(long, long)

Regarding mutant $M3$, the mutation occurred in the `safeSubtract` method. This method is not invoked by any other method. This way, the output of Step 1 is the following:

> m:FieldUtils.safeSubtract(long, long)

Now, we pass the change impact analysis results to the test case generation step (Step 2).

3.2 Automated Generation of Test Cases

Automated generation of tests is a broad field of research (Lakhotia et al., 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 and Arcuri, 2011), Randoop (Pacheco et al., 2007), and IntelliTest (Li et al., 2016) implement such approaches. Each tool has a specific purpose, so different tools will 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), in Step 2, we use well-known automated test generation tools to generate a comprehensive set of test cases, based on the original program, for the identified impacted entities. 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. Notice that 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 the Step 2, we return to the `FieldUtils` class (Listing 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 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 $M3$.

Listing 3: Examples of test cases generated to the `FieldUtils`.

```
public class FieldUtilsTest_M1{
  @Test
  public void test001(){
    assertEquals(200, FieldUtils.safeMultiplyToInt(10L, 20L));
  }
  @Test
  public void test002(){
    try {
      int val = FieldUtils.safeToInt(2147483648L);
      fail("Failed: Should get an Arithmetic Exception");
    } catch (ArithmeticException e) { }
  } ...
}
public class FieldUtilsTest_M2{
  @Test
  public void test001(){
    assertEquals(200, FieldUtils.safeMultiplyToInt(10L, 20L));
  }
  @Test
  public void test002(){
    assertEquals(25L, FieldUtils.safeMultiply(5L, 5L));
  } ...
}
public class FieldUtilsTest_M3{
  @Test
  public void test001(){
    assertEquals(-1, FieldUtils.safeSubtract(0L, 1L));
  }
  @Test
  public void test002(){
    try {
      int val = FieldUtils.safeSubtract(Long.MIN_VALUE, 100L);
      fail("Failed: Should get an Arithmetic Exception");
    } catch (ArithmeticException e) { }
  } ...
}
```

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. This 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. Because our goal is to suggest equivalent mutants, it makes no sense to continue executing the subsequent tests since the mutant has already been identified as *non-equivalent*.

During the test execution step, we also record the test execution coverage of the original program and the mutants (Schuler and Zeller, 2013). In other words, we record the frequency in which each line has been executed by all generated tests. In addition, 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 equivalents by the approach, as we explain next.

3.4 Suggesting Equivalent Mutants

In Step 4, we analyze the test suite execution results to suggest equivalent mutants. Mutants that were killed by test cases are marked as *non-equivalent*, as they exhibit a behavioral change from the original program. For the mutants that were not killed, we suggest them as *equivalent*.

Since we cannot guarantee that the suggestion is correct, we provide a *ranking* of mutants to better support the testers. The bottom of the ranking includes mutants that we have strong confidence are equivalent, while the top includes mutants in which we have weak confidence in our suggestion.

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. Schuler and Zeller (2013) identified that coverage impact provides an effective means to separate equivalent from non-equivalent mutations. They reported that if a mutation changes the coverage, the mutant has 75% of being non-equivalent.

To better explain the last step of our approach, we rely on the code snippets presented in Listing 2 and Listing 3. As mentioned, Step 4 analyzes the test suite execution and classifies the mutant as equivalent or non-equivalent. The test `FieldUtilsTest_M1.test002` can expose the behavior change of $M1$ mutant. The approach then marks this mutant as non-equivalent and informs the tester that `FieldUtilsTest_M1.test002` is enough to kill the mutant $M1$. When considering the mutant $M2$, every generated test suite will be executed, and the mutant will not be killed, then 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`

3.5 Improvements

As explained, our approach is based on previous ideas from the refactoring field (Soares et al., 2013a; Mongiovi et al., 2014). In addition to bringing this idea to the context of mutation testing, we provide several improvements regarding related work.

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 needs to be tested after a transformation. Especially when we search for the set of indirectly impacted methods that exercise an impacted entity. The list of methods to test can get large, hindering the efficacy of the testing generation tools. 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.

Other improvements were made in the Steps 3 and 4. During the test execution step, we record coverage information to support the suggested equivalent mutants' phase. First, we follow the idea proposed by Schuler and 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 that were 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 for an important cost reduction. The cost of the tester to design and implement a new test case to kill a survived 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-adequate* test suite.

The automated implementation of our approach is provided through a tool named Nimrod[1] The tool performs the necessary steps, including change impact analysis, test case generation, and test execution, to suggest equivalent mutants in a codebase. It aims to reduce the tester's effort by suggesting potential equivalent mutants and providing relevant information to support the decision-making process.

4 Evaluation

In this section, we present the evaluation of our automatic behavioral testing approach, Nimrod, for suggesting equivalent mutants from the perspective of mutation testers in the context of mutation analysis. We detail the research questions addressed, the subjects used, the experimental setup, and the evaluation procedure.

4.1 Research Questions

To evaluate Nimrod, we address the following research questions:

RQ1. *How effective is* Nimrod *in suggesting equivalent mutants?* **RQ2.** *How long does* Nimrod *take to analyze a mutant?* **RQ3.** *What are the characteristics of the mutants that* Nimrod *failed to classify?* **RQ4.** *Which mutation operators commonly lead* Nimrod *to fail?*

The questions RQ1 and RQ2 were discussed in a previous publication (Fernandes et al., 2022). In this extended article, we introduce two new questions, RQ3 and RQ4.

We answer **RQ1** by measuring the Precision, Recall, and F-measure of Nimrod in suggesting equivalent mutants. To establish a baseline, we use a set of manually identified equivalent mutants from a previous work (Kintis et al., 2018). This enables us to calculate true positives, false positives, true negatives, and false negatives based on manual analysis. Additionally, we compare Nimrod with the TCE tool (Kintis et al., 2018), which detects equivalent mutants. It's important to note that Nimrod and TCE are complementary tools, with the former suggesting equivalent mutants and the latter detecting them. It is important to address **RQ1** as it helps us determine the potential time and effort saved by Nimrod when compared to manually analyzing each surviving mutant.

To answer **RQ2**, we calculate the average time that Nimrod takes to suggest the mutants as equivalent or non-equivalent. Once a test kills the mutant, we confirm that such a mutant is non-equivalent. This means that Nimrod has no false negatives since all mutants classified as non-equivalents are killed by at least one test case. On the other

[1]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.

hand, Nimrod can erroneously classify mutants as equivalents (false positives). These mutants may be a stubborn mutant (Yao et al., 2014), where only a very specific test case can kill it. To better understand the types of mutants that our approach classifies erroneously, we formalize the following research question:

To answer **RQ3**, we analyze the context of the program in which the mutation is inserted. 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 leads to answer **RQ4**.

We answer **RQ4** 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).

4.2 Subjects

For the evaluation, we use programs and mutants from a previous work (Kintis et al., 2018). This set of programs is accompanied by manually identified equivalent mutants, providing a ”ground truth” about the undecidability of equivalent mutants.

Table 1 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 present the lines of code and the number of generated mutants.

Table 1. Manually analyzed Java subjects.

Program	**Class**	**Method**	**LoC**	**Total Mutants**
Bisect	A program that calculates square roots	sqrt	23	135
Commons-lang	An enhancements to Java core library	capitalize	25	69
		wrap	45	198
Joda-time	A time manipulation library	add	33	257
Pamvotis	A wireless LAN simulator	addNode	53	318
		removeNode	18	55
Triangle	A classic triangle classification program	classify	44	354
XStream	A XML object serialization framework	decodeName	40	156
Total			**281**	**1,542**

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.

4.3 Experimental Setup

The evaluation is conducted on a 2.70 GHz four-core PC with 16 GB of RAM running Ubuntu 20.04.

Nimrod handles configuration files for each program to be analyzed, indicating the test generation tools and their respective input parameters. In this evaluation, we use two popular test generation solutions: EvoSuite (Fraser and Arcuri, 2011) and Randoop (Pacheco et al., 2007; Soares et al., 2013a).

EvoSuite is a search-based tool that uses a genetic algorithm to generate test suites for Java classes. We instantiate EvoSuite twice for each mutant analysis: once for EvoSuite Regression (EvoSuiteR) to generate test suites revealing differences between two versions of a Java class (the original and the mutant), and once with four coverage criteria (Statement, Line, Branch, and Weak Mutation coverage) for test generation. A time limit of 60 seconds is set for test generation.

Randoop generates unit tests for Java using feedback-directed random test generation. t randomly generates sequences of method/constructor invocations for the classes under test and creates assertions that capture the actual behavior of the program. Randoop is normally used to create regression tests. We also set a time limit of 60 seconds for test generation.

For the generated tests, we set a timeout of 80 seconds to execute the entire test suite. We also limited 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.

In order to make our analysis feasible, we have established certain limits in this paper. These include a maximum of 60 seconds to generate tests, 80 seconds to execute the test suite, and a cap of 3,000 tests. With regards to EvoSuite, we found that there was no significant increase in the test suite beyond the 60-second limit. On the other hand, Randoop generally reaches 3,000 tests in under 60 seconds. Additionally, the longest it takes for the worst test suite to execute against the original program is 20 seconds. As a result, we have set a guaranteed time of 80 seconds (4 x 20) to execute against each mutant, given our scope.

4.4 Procedure

To perform the evaluation, we use the mutants generated using the Mujava tool[2] (Kintis et al., 2018) with all available method-level mutation operators. We then execute Nimrod's equivalence analysis individually for each mutant, comparing it against the original program.

Nimrod's equivalence analysis is done individually between the original program and the mutant. This is because the change impact analysis (Section 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. For each mutant, we first execute EvoSuiteR-generated tests, followed by tests generated by EvoSuite with four coverage criteria, and finally, tests generated by Randoop. This order is chosen because EvoSuite can often reveal behavior changes with fewer tests compared to Randoop.

If a test case fails or the test suite execution reaches the

[2]`https://cs.gmu.edu/~offutt/mujava/`

timeout, Nimrod suspends the test execution, informs that the mutant is not equivalent, and write 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.

Upon analysis completion for each subject, we collect the results and create a ranking of mutants suggested as equivalents.

For a comprehensive evaluation, we also execute and gather data from the TCE tool (Kintis et al., 2018) on the same mutants, and the list of equivalent mutants manually classified by Kintis et al. is made available at our companion website (Fernandes, Leo, 2023).

In the next section, we present the experimental results and discuss the main findings of the evaluation.

5 Analysis, Results, and Discussion

In this section, we present the results and address the research questions. The basis for our analysis consists of 193 mutants[3] manually classified as equivalents in previous researches (Kintis et al., 2018). These 193 mutants represent 12.5% of the total 1,542 mutants analyzed.

Before detailing our approach, it is important to state the notion of equivalence we adopt. A mutant and an original program are equivalents 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 et al., 2013a). In an *open world assumption* (OWA), any test case can be generated to discover a behavioral change, without regarding the project or code requirements. In a *close world assumption* (CWA), the test cases must satisfy some domain constraints. For example: "*all the tested methods must be called 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. Our approach adopts an open-world equivalence notion, which means there are no constraints in the test generation.

In the Nimrod, there is a possibility that our approach may erroneously classify a mutant as equivalent when in reality, the mutant is a *stubborn* mutant. A stubborn mutant is one that remains undetected by a high-quality test suite and is, therefore, non-equivalent (Yao et al., 2014). This represents a *false positive* in our approach. Conversely, 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 incorrect; or (*ii*) the tests, although executing correctly, were written in the wrong way.

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

5.1 How effective is Nimrod in suggesting equivalent mutants?

Table 2 presents the general results of executing Nimrod on the subjects. Columns 1 and 2 show the Program and Method names, respectively. Column 3 indicates the number of mutants generated for each method. In total, the Mujava mutation tool generated 1,542 mutants. Column 4 shows the equivalent mutants according to the manual analysis, the baseline. Columns 5 and 6 show the number of equivalent mutants (N) according to TCE detection, and Nimrod suggestion, respectively. For the TCE and Nimrod results, we also present the number of False Positives (FP) and False Negatives (FN) in the table.

For clarity, we will refer to the subjects by the unique names of the methods (Column 2) and the names of the programs (Column 1) in parentheses.

The manual analysis identified 193 equivalent mutants, which we use as the baseline for comparison with our results. The TCE tool detected 109 out of the 193 equivalent mutants (56%). As TCE is a solution for detecting equivalent mutants (Madeyski et al., 2014), there are no false positives (FP) in the results. However, TCE can have false negatives (FN), i.e., it may not identify all the equivalent mutants. All 84 false negatives occurred because the optimization applied by TCE produced a bytecode that differs from the corresponding original program. It is important to note 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 paper to suggest a better solution, as our approach, together with TCE, may be complementary from the perspective of mutation analysis.

In contrast to TCE's accuracy in detecting equivalent mutants, our approach attempts to suggest whether a mutant is equivalent through automated behavioral testing. Out of the 1,542 total mutants analyzed, Nimrod classified 449 as equivalents. This is more than twice the number of 193 mutants manually identified as equivalent. In fact, we expected our solution to suggest more equivalents than the total number of mutants that are indeed equivalent. The mutants that Nimrod wrongly classified as equivalents represent the false positives (FP).

Tables 4a to 4h present the detailed 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. This information helps us assess Nimrod's performance on each subject.

Based on the F-measure, in three out of the eight subjects evaluated, namely *decodeName* (XStream), *add* (Joda-time), and *classify* (Triangle), the approach achieved an accuracy of 100%. In two subjects, *sqrt* (Bisect) and *capitalize* (Commons-lang), the accuracy was above 96%. In the other two subjects, *wrap* (Commons-lang) and *removeNode* (Pamvotis), the approach had an accuracy above 82%.

However, Nimrod exhibited a very low accuracy in the *addNode* (Pamvotis) subject, suggesting 277 out of 318 mutants as equivalents. This is eight times more mutants than

Table 2. General Results.

Program	**Method**	**Total Mutants**	**Equivalent Mutants**		
			Manual (baseline)	**TCE** N (FP-FN)	**Nimrod** N (FP-FN)
Bisect	sqrt	135	17	11 (0-6)	18 (1-0)
Commons-Lang	capitalize	69	14	2 (0-12)	15 (1-0)
	wrap	198	19	12 (0-7)	26 (7-0)
Joda-Time	add	257	35	24 (0-11)	35 (0-0)
Pamvotis	addNode	318	33	33 (0-0)	277 (244-0)
	removeNode	55	7	6 (0-1)	10 (3-0)
Triangle	classify	354	40	21 (0-19)	40 (0-0)
XStream	decodeName	156	28	0 (0-0)	28 (0-0)
Total		**1,542**	**193**	**109**	**449**

the 33 manually marked as equivalent. Upon analyzing the false positives for 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). Additionally, most mutants mistakenly marked as equivalent change fields that do not have public methods or are located in classes other than the target class where the mutation occurred (we will discuss these cases in the next section). In contrast, this was the only subject in which TCE had 100% of accuracy.

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 using the following criteria: first, the impact on coverage, and then the number of test cases that exercised the mutated point.

However, 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 will be 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 checked how many false positives remained before or after the median.

Table 3 presents the 18 mutants of *sqrt* (Bisect) suggested as equivalent by Nimrod. According to the manual analysis, 17 mutants are equivalent, which means Nimrod classification had one false positive. The false positive is AOIS_12 (in bold). No mutant has an impact on coverage. Using the number of test cases that touched the mutated point, the median value for the 18 mutants is 204 test cases. The AOIS_12 mutant was exercised by 191 test cases. Notice that this mutant is below the median value ($191 < 204$). Lower values might represent potential non-equivalent mutants. Therefore, mutants below the median could be selected for manual analysis.

Table 4 presents the false positives per subject. Only subjects that had at least one false positive are listed in the table. In the *sqrt* (Bisect) and *capitalize* (Commons-Lang) subjects, only one mutant was wrongly classified. In both cases, the false positives were below the median.

Table 3. The *sqrt* (Bisect) mutants suggested as equivalent. The AOIS_12 is the false positive (in bold) and the double line marks the division 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

In the *addNode* (Pamvotis) subject, despite having many false positives, 214 (88%) out of 244 were below the median. Upon examining the details of the result, we identified that 106 (38%) mutants were not exercised by any test case. 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). 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. This is relatively low since an average of three thousand tests were generated for these mutants.

Answer to RQ1: The accuracy of our approach reached 100% in three subjects, more than 96% in two subjects, and more than 82% in two other subjects studied. In only one subject, the performance was below 22%. We defined the median number of test cases that touched the mutated point to

Table 4. Analyzed subjects.

(a) 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%

(b) 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%

(c) Subject: *decodeName* (XStream).

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

(d) 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%

(e) 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%

(f) 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%

(g) 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%

(h) 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%

distinguish 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%.

5.2 How long does Nimrod take 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 5 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.

Table 5. Average time Nimrod took to analyze each mutant and distribution of the false positives according to the median.

Program	Method	Average Time (seconds)		False Positives		
		Equiv.	Non-Equiv.	Qty	Median ⟸	⟹
Bisect	sqrt	198.37	130.43	1	100%	0%
Commons-Lang	capitalize	358.36	22.50	1	100%	0%
	wrap	378.29	15.99	7	57%	43%
Joda-Time	add	212.61	24.04	0		
Pamvotis	addNode	404.76	38.72	244	88%	11%
	removeNode	391.89	25.79	3	66%	33%
Triangle	classify	197.10	11.46	0		
XStream	decodeName	311.01	23.72	0		

Our approach has a fast average response time for the cases where the mutant is suggested 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 allowed easy-to-kill mutants to be quickly discovered and killed.

To suggest a mutant as non-equivalent, the *sqrt* (Bisect) subject had the worst results. In this subject, some non-equivalent mutants led Nimrod to generate tests that reached the execution timeout due to infinite loops caused by the mutants. So, although this situation raises a behavioral change, Nimrod spends a lot of time until the timeout. The *classify* (Triangle) subject had the best response time to detect non-equivalent mutants. This subject has relatively simple code structures when compared to the other subjects of the study. For instance, 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. This 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 to suggest a mutant as equivalent. As explained, this subject has some features that difficult the generation of tests. One may ask whether the time is taken by Nimrod is acceptable. Notice that 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.

Manually classifying mutants as equivalent and non-equivalent takes an average of 15 minutes per mutant (Schuler and Zeller, 2013). Moreover, this task is error-prone: 20% of the studied mutants were erroneously classified (Allen Troy Acree, 1980). Nimrod can reduce this work as it takes a third of the manual time and ranks the mutants indicating which ones are likely to be equivalent. Also, for a non-equivalent mutant, the average time reduces to 36.57 seconds, 25 times less than the manual time. TCE can analyze the equivalence in less than two seconds (Kintis et al., 2018). In Section 6, we present a practical application combining TCE and Nimrod as an alternative for the equivalent mutant problem.

Answer to RQ2: Nimrod took an average of 306.55 seconds per equivalent mutant analyzed and 36.57 seconds per non-equivalent mutant. While manually analyzing a mutant to indicate whether it is equivalent or not can take 15 minutes (Schuler and Zeller, 2013), Nimrod takes a third of this time to suggest equivalent mutants and is 25 times faster to indicate non-equivalents mutants.

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

To address RQ3, we carefully examined all the false positives of each subject to identify common source code characteristics that led Nimrod to suggest mutants as equivalents when, in fact, they are not. We focused on three specific characteristics: *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. Out of the 1,542 mutants analyzed, a total of 256 (16.60%) mutants were misclassified. Remarkably, the majority of these misclassified mutants, specifically 244 (95%), were associated with the *addNode* (Pamvotis) subject. In this section, we present a qualitative evaluation of the false positives for each of these characteristics.

Access Level Modifier occurs when the test case and the source code need to be in the same package structure so that the mutant can be killed by the test. Listing 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 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 in any method of this class to set out a change in the behavior.

This led us to believe that this mutant could be equivalent, however, as can be seen in Listing 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 will have 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.

In the `Bisect` class example, if the developers of the project had defined that the unit tests and the original source code should be in 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.

To solve this problem, the Nimrod tests must follow the same package structure of the class under test and, in addition, the test assertion must use the available class fields.

Listing 4: A code snippet extracted from the *sqrt* subject.

```
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;
    }
}
```

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 the Listing 5, the mutant M_2 (line 21) was generated by the mutation operator AORB (Arithmetic Operator Replacement). 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, this mutated point was only exercised twice, even though more than 3,000 tests were generated by automatic generation tools.

Listing 5: A code snippet extracted from the *wrap* (WordUtils) subject.

```
 1 public class WorldUtils {
 2   public static String wrap( String str, int wrapLength,
 3     String newLineStr, boolean wrapLongWords ) {
 4     ...
 5     while (...) {
 6       if (...) {
 7         offset++;
 8         continue;
 9       }
10       if (...) {
11         ...
12         offset = ...
13       } else {
14         if (...) {
15           ...
16           offset = ...
17         } else {
18           spaceToWrapAt = str.indexOf(' ', wrapLength + offset);
19           if (spaceToWrapAt >= 0) {
20             ...
21             offset = spaceToWrapAt + 1;     M2 [ + ⇒ % ]
22           } else {
23             ...
24             offset = ...
25           }
26         }
27       }
28     }
29     wrappedLine.append(str.substring(offset));
30     return wrappedLine.toString();
31   }
32 }
```

Table 6. 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 the 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).

It is already known from the software testing community that automatically creating tests to achieve high branch coverage is difficult. A possible solution for Nimrod to solve this problem is to increase the time limit of the tools to generate the tests and allow a larger number of tests to be generated (we limit these settings in 60 seconds of time limit and a maximum of 3,000 tests). However, these decisions imply directly in the total time to suggest a mutant as equivalent or non-equivalent. New approaches to solving 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.

External Entities happens when the test needs to execute or assert entities that are not directly located in the target class where the mutation occurred. Listing 6 presents a code snippet extracted from the *addNode* (Pamvotis) subject. The method (lines 4-24) has no return statement and its main 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 variable `SpecParams.CW_MAX` (Line 13). This global variable is static, public, and has only one definition point in the `SpecParams` class.

Listing 6: A code snippet extracted from the *addNode* subject.

```
1  public class Simulator {
2    private java.util.Vector nodesList = new java.util.Vector();
3    ...
4    public void addNode( int id, int rate, int coverage, int
         ↪ xPosition, int yPosition, int ac ) {
5      ...
6      if (...) {...}
7      else {
8        pamvotis.core.MobileNode nd = new pamvotis.core.MobileNode();
9        ...
10       switch (ac) {
11         case 1 : {
12           nCwMin = cwMin / cwMinFact1;
13           nCwMax = SpecParams.CW_MAX / cwMaxFact1;
14           nAifsd = sifs + aifs1 * slot;   M3:
15           break;         [SpecParams.CW_MAX ⇒ SpecParams.CW_MAX--]
16         }
17         ...
18       }
19       nd.params.InitParams( id, rate, xPosition, yPosition,
             ↪ coverage, ac, nAifsd, nCwMin, nCwMax );
20       nd.contWind = nd.params.cwMin;
21       nodesList.addElement( nd );
22       nmbrOfNodes++;
23     }
24   }
25 }
```

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 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 be executed (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 was not designed to 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 have successfully identified the three main characteristics that cause Nimrod to fail in suggesting equivalent mutants. These include the *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. However, we have discovered that it is possible to significantly reduce the number of false positives by enhancing the static analysis and adjusting the automatic test generation configuration. Out of the three characteristics, the *Very Restricted Value* is the most common and challenging to address. This type of mutant is notoriously known as *stubborn* because it requires highly specific tests to be eliminated.

5.4 Which mutation operators commonly lead Nimrod to fail?

In this section, we investigate the influence of mutation operators on the performance of Nimrod and determine which ones commonly led to misclassifications. To achieve this, we counted the number of suggested equivalent mutants per operator that caused Nimrod to fail. Table 7 presents the results for each mutation operator, including the total number of mutants analyzed per operator.

Our focus is on the false positive column since it represents mutants that were incorrectly classified as equivalent by Nimrod. By analyzing the absolute numbers, we observe that the AOIS (Arithmetic Operator Insertion, short-cut) operator stands out, generating 650 (42%) mutants, the highest among all operators. Consequently, this operator was responsible for producing both equivalent and non-equivalent mutants in large quantities. Specifically, Nimrod classified 251 AOIS mutants as equivalent, out of which 125 (50%)

were misclassified. It is noteworthy that the AOIS operator is known for generating numerous equivalents, some of which could potentially be avoided before their generation (Kintis and Malevris, 2015; Fernandes et al., 2017). Additionally, this operator also generates several *stubborn* mutants, which require specific tests for mutation analysis (Yao et al., 2014).

However, when considering the relative numbers, i.e., the false positive rates for each operator, we find that the worst performance occurred with the AOIU (Arithmetic Operator Insertion, unary) and AORB (Arithmetic Operator Replacement, binary) operators, both having false positive rates close to 30%. Notably, more than 93% of the false positives from both AOIU and AORB originated from the *addNode* (Pamvotis) subject. The misclassification in these cases often resulted from the AOIU operator changing a field that either belonged to an external entity or had package-private access level.

For the remaining mutation operators (LOI and ASRS), where Nimrod also exhibited misclassifications, the hit rate in detecting non-equivalent mutants (True Negative column) was above 80%, indicating that Nimrod performed well in distinguishing non-equivalent mutants. As a result, we did not identify any specific mutation operator that was inherently leading to Nimrod's failure.

Answer to RQ4: In our investigation, we identified ten Mujava mutation operators, generating a total of 1,542 mutants. Six operators, notably AOIU and AORB, produced false positives, erroneously marked as equivalent by Nimrod. The AOIS operator, contributing 42% of mutants, stood out as the most prolific in both equivalent and non-equivalent categories. Despite AOIS's tendency to create many equivalent mutants, some could potentially be prevented before generation, as suggested by previous research (Fernandes et al., 2017). Additionally, AOIS generates *stubborn* mutants requiring very specific tests. While AOIU and AORB had the highest false positive rates at nearly 30%, the majority originated in the *addNode* (Pamvotis) subject. For other misclassified mutation operators (LOI and ASRS), Nimrod excelled in detecting non-equivalent mutants, with hit rates exceeding 80%. No specific mutation operator consistently led to Nimrod's failure; misclassifications were linked to specific code characteristics, detailed in Section 5.3.

5.5 Threats to Validity

In this section, we discuss potential threats to the validity of our study.

External Validity: The selection of projects used as subjects in this study may pose a threat to external validity, especially since we focused on a limited number of methods. To address this concern, we aimed to diversify the subjects by choosing projects from different systems and domains. Although the subjects were selected based on a previous analysis of equivalence, our approach has not been tested against methods with complex external dependencies, such as those involving objects like *ObjectC func(ObjectA , ObjectB);*. Methods with such dependencies might be challenging for test generation tools, as they require the creation of mocks (Arcuri et al., 2017) or the discovery of valid constructors, which may have further dependencies. Additionally, dependencies on external elements like graphical user interfaces or file manipulations could limit the ability of test generation tools to generate test cases that expose behavioral changes (Soares et al., 2013b).

Internal Validity: The manual analysis used to classify the mutants represents a potential threat to internal validity. However, the set of mutants was manually analyzed by two previous independent studies (Kintis et al., 2018; Kintis and Malevris, 2015) in addition to our present analysis. Furthermore, TCE was used to confirm some of the equivalent mutants. For the remaining ones, we manually verified all the equivalents and corroborated the previous manual analysis. It is important to note that this threat arises due to the undecidability of the equivalent mutant problem, and it applies to all relevant mutation testing studies on this topic.

The set of selected mutant operators also introduces threats to the internal validity of this work. However, this set includes all 15 operators available in Mujava (version 3), and we did not exclude any mutants from our investigation. We did not evaluate object-oriented related mutation operators, as previous research (Offutt et al., 2006) has shown that they yield a small number of mutants and a relatively low number of equivalent ones.

The ranking approach we used counts the number of test cases that reach the mutated point, which relies on code line coverage information. This metric may not always provide precise information. For instance, 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 if they only evaluate the left-hand side of the expression, will be computed as reaching the mutated point. To mitigate this threat, our ranking also considers another metric: the impact on coverage.

The presence of *flaky tests* (Luo et al., 2014) could represent a threat as well. For instance, Nimrod might suggest a mutant as non-equivalent because there is a test exposing a behavioral change in the mutant program, while the mutant is actually equivalent, and the difference in behavior occurred due to a flaky test. This would lead to a false negative for Nimrod. To minimize this threat, we execute the generated tests against the original program to confirm that they capture the current behavior of the original program. Additionally, both EvoSuite and Randoop have settings to avoid flaky tests. Notably, we did not identify any false negatives among the mutants analyzed manually and subsequently evaluated by TCE.

Other potential threats could arise from defects in the embedded software, such as in the static analysis or automatic test generation tools. Such defects could impact our results. However, we believe that the influence of such defects would be minimal in our study.

Construct Validity: All our results are empirical observations, and they might not necessarily hold in all cases. However, we have made all our subjects, tools, and data available on the companion website of this article, which enables independent researchers to verify, replicate, and analyze our findings. This transparency helps to mitigate construct validity threats.

Table 7. Nimrod Results by Mutation Operator.

Mutation Operator	Description	Mutants	Nimrod True Positive	Nimrod True Negative	Nimrod 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**

In conclusion, while our study presents valuable insights into the performance of Nimrod for mutation analysis, it is essential to be aware of the potential threats to validity mentioned above. These considerations provide a more comprehensive understanding of the scope and limitations of our findings.

6 Implications for Practice

We propose to extend the traditional mutation process by incorporating TCE and Nimrod to minimize the high cost of manual analysis. Figure 2 presents this extension. Step 1 presents Offutt's and Untch's proposition of the common mutation process. The solid boxes represent steps that are automated by tools such as Mujava, and the dashed boxes represent manual steps.

In Step 2, TCE receives as input a set of mutants that were not killed by the application test suite and safely discards a number of useless mutants. In Step 3, Nimrod receives as input the mutants that TCE could not confirm as equivalents. If Nimrod finds a test that exposes a behavioral change, it indicates that the mutant is not equivalent and informs which test can kill it. If after the timeout no test is capable of exposing a behavioral change, Nimrod suggests the mutant as equivalent. Step 4 shows the manual process to identify equivalent mutants. After using TCE and Nimrod, the tester now should consider a smaller number of mutants.

To better illustrate a potential cost reduction, we refer to the `FieldUtils` class (Joda-time project) presented in Section 3. Table 8 presents a summary of the effort reduction. When executing the Mujava with all the mutation operators, 1,339 mutants are generated. After executing the test suite of the Joda-time project, 543 mutants have been killed and 796 were still alive. This represents a mutation score of 40% (without any analysis of equivalence). By using the traditional process, 796 mutants should be still analyzed.

Our first step is to consider the live mutants as input for TCE. TCE took 17.33 minutes to analyze the 796 mutants. An average of 1.3 seconds per mutant. TCE detected 117 mutants, reducing the number of mutants to be investigated

Table 8. 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	Survived mutants	100.00%
-117	TCE equivalents	17.33 minutes
679	Survived mutants	⇓ 14.69%
-608	Nimrod non-equivalents	410.99 minutes
71	Survived Mutants	⇓ 91.08%

to 679. Then, we execute our approach against these 679 mutants. Nimrod was able to identify 608 mutants as non-equivalents (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 equivalents. To sum up, we achieved a total reduction of 91.08% in the number of mutants to be analyzed.

We do not intend here to generalize the cost reductions whatsoever. Instead, we intend to show that our approach might potentially reduce costs when analyzing and marking equivalent mutants.

7 Related Work

Addressing the equivalent mutant is not a recent problem (Jia and Harman, 2011; Madeyski et al., 2014; Pizzoleto et al., 2019; Papadakis et al., 2019). To tackle the mutation equivalent problem, researchers used compiler optimizations. Application of six techniques for compiler optimization in an experiment with 15 small Fortran programs along with 14 mutation operators (Offutt and Craft, 1994). Development of the Trivial Compiler Equivalence (TCE) and implemented compiler optimizations for Java and C (Kintis et al., 2018). To check the equivalence, it was used a diff program. Our approach aims to work in conjunction with detection strategies like these ones.

There are studies to avoid equivalent mutants even before they are generated. Heuristics specifications, based on equivalence conditions, that avoid equivalent mutants for class-

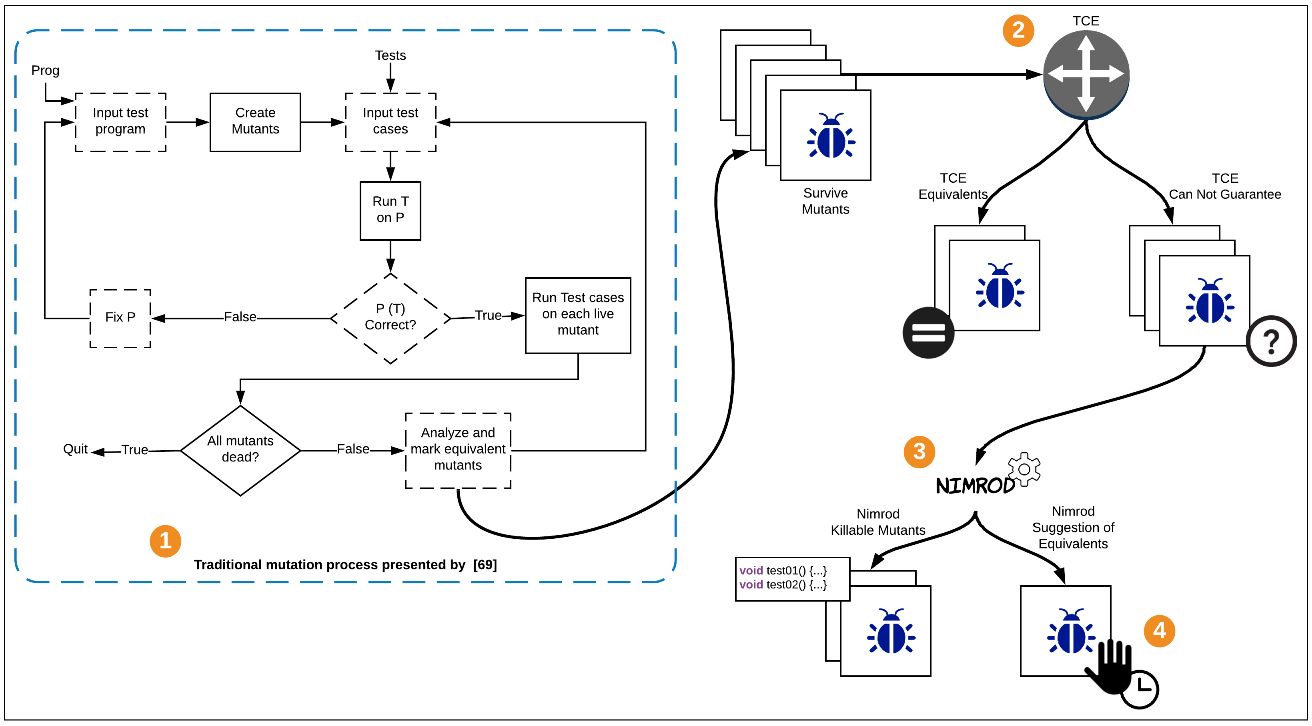


Figure 2. Combining TCE and Nimrod to minimize the manual analysis to identify equivalent mutants.

level mutation operators (Offutt et al., 2006). Application of data-flow patterns to identify equivalents (Kintis and Malevris, 2015) and introduction of a strategy to help developers with deriving rules to avoid the generation of useless (equivalents and duplicated) mutants right before their generation (Fernandes et al., 2017). Strategies to avoid equivalent mutants are alternate options to reduce cost and can be combined with complementary approaches to detect and suggest equivalent mutants after mutation analysis, such as TCE and Nimrod, may come as a complement to these solutions.

Other studies check the impact of the mutant execution. The more invariants a mutant violates, the more likely it is to be detected by actual tests (Schuler et al., 2009). Analyzes to verify whether changes in coverage can be used to detect non-equivalent mutants (Schuler and Zeller, 2013). In our work, we take advantage of the coverage impact to build our ranking (Table 3).

Recent techniques classify equivalent mutants using machine learning (ML). 80.30% accuracy while classifying equivalent mutants in investigation of seven traditional ML algorithms (Brito et al., 2020). Evaluate an Abstract Syntax Tree Neural Network Model with two mutation operators and 582 mutants, resulting in a classification accuracy of 90% (Peacock et al., 2021). Although promising, these techniques still need to assess not only effectiveness but also efficiency.

The idea of automatically generating a test suite to detect behavioral changes was proposed in the context of refactoring (Soares et al., 2010). They presented SafeRefactor, a tool for improving safety during refactoring activities, that was able to identify bugs in Eclipse and NetBeans (Soares et al., 2013a; Mongiovi et al., 2018).

We extend the initial idea to adapt to the mutation testing context.

We add more tools to automatically generate the tests. We also provide the user the test that can identify the behavioral change and calculate the confidence of the equivalence by counting the number of tests that touch the mutated point.

Finally, our approach can be used with strategies (Guimarães et al., 2020; Gheyi et al., 2021) that identify mutation subsumption relations to reduce mutation testing effort.

8 Concluding Remarks

In this paper, we introduced Nimrod, an approach based on automated behavioral testing, to mitigate the impact of equivalent mutants in mutation testing. By automatically suggesting equivalent mutants and generating tests to kill non-equivalent mutants, Nimrod reduces the manual labor required in mutation analysis.

Our results demonstrate that Nimrod achieved a high success rate in suggesting equivalent mutants, correctly identifying 100% of the equivalents in three out of eight subjects, and achieving above 96% in two subjects. Only in one subject did the performance drop below 50%. Additionally, Nimrod significantly reduced the time taken to identify non-equivalent mutants, with an average of 36.57 seconds per non-equivalent mutant. For the mutants suggested as equivalent, where test generation tools needed to execute all generated tests, Nimrod took an average of 306.55 seconds (approximately 5.1 minutes). This is substantially faster than the 15 minutes typically required for manual analysis (Schuler and Zeller, 2013). By automating this process, Nimrod saves testers valuable time and effort in thinking and implementing test cases to identify non-equivalent mutants.

The scalability of the presented approach relies significantly on the automated test generation tools employed in Step 2 (Figure 1). Given the impracticality of conducting exhaustive tests for more intricate programs, these tools utilize a time configuration as a stopping criterion. We have uniformly allocated a fixed time for all subjects, regardless of the complexity of the tested method, a practice that should not be considered standard. Extracting information from the program under test through static analysis and subsequently configuring the test generation tool according to the context can be considered an option to enhance performance.

We also investigated the characteristics of the mutants that Nimrod misclassified as equivalent (false positives). Three classes of characteristics were identified, including *Access Level Modifier*, *External Entities*, and *Very Restricted Value*. Improvements in static analysis and automatic test generation tools can help reduce false-positive instances, particularly by designing testable classes to aid Nimrod's equivalence analysis. Notably, around 69% of errors occurred due to behavioral changes in tests necessitating access to external entities or matching package structures between tests and the program under test. These issues can be mitigated through evolving automatic testing tools, employing inter-class impact analysis, and adjusting test generators to match package structures and access package-private and protected elements.

Furthermore, we assessed the impact of mutation operators on Nimrod's performance. While the AOIU and AORB operators had the highest false-positive rates, their hit rates in Nimrod surpassed the error rates, indicating no definitive set of operators leading to misclassifications. The AOIS operator generated the most equivalent mutants but maintained a high hit rate in Nimrod. Hence, we did not identify a specific set of operators responsible for misclassification.

Future work includes carrying out new experiments to statistically revalidate the findings of this paper. Both using other mutation tools such as Major and PIT, as well as different operators like class-level mutation operators (Offutt et al., 2006).

We also plan to include subjects with different levels of complexity. For instance, methods with complex external dependencies to further enhance the analysis. Novel benchmarking frameworks (van Hijfte and Oprescu, 2021) can support this plan.

As an improvement in the approach, we intend to increment the tool to reduce the number of false positives for the cases we identified. This should include improving the impact analysis. In the actual version, we only have two options in 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.

Furthermore, 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.

To summarize, our research proposes using Nimrod as a solution to address equivalent mutants in mutation testing. By automating the identification of equivalents and producing tests to reveal non-equivalents, Nimrod offers valuable assistance to software testers, greatly reducing the need for manual labor and time while ensuring the efficacy of the mutation analysis process.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and suggestions, which helped to improve the quality of this paper. This work is partially supported by INES (www.ines.org.br), CNPq grants 465614/2014-0, 312195/2021-4, 421306/2018-1 and 308636/2021-0, FACEPE grants APQ-0399-1.03/17 and APQ/0388-1.03/14, FAPEAL grant 60030.0000000462/2020, FAPESP grants 2020/09560-2, 2019/23160-0 CAPES grant 88887.136410/2017-00 and PIBIC IFAL 2023-2024.

References

Allen Troy Acree, J. (1980). *On Mutation*. PhD thesis, Georgia Institute of Technology.

Andrews, J., Briand, L., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411.

Arcuri, A., Fraser, G., and Just, R. (2017). Private api access and functional mocking in automated unit test generation. In *ICST*, pages 126–137.

Binder, R. (1994). Design for testability in object-oriented systems. *Communications of the ACM*, 37:87–101.

Braione, P., Denaro, G., Mattavelli, A., and Pezzè, M. (2017). Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 90–101.

Brito, C., Durelli, V., Durelli, R., Souza, S., Vincenzi, A., and Delamaro, M. (2020). A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants. In *ICSTW*, pages 304–313.

Budd, T. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45.

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

Fernandes, L., Ribeiro, M., Carvalho, L., Gheyi, R., Mongiovi, M., Santos, A., Cavalcanti, A., Ferrari, F., and Maldonado, J. C. (2017). Avoiding useless mutants. In *GPCE*, pages 187–198.

Fernandes, L., Ribeiro, M., Gheyi, R., Delamaro, M., Guimarães, M., and Santos, A. (2022). Put your hands in the air! reducing manual effort in mutation testing. page 198–207. Association for Computing Machinery.

Fernandes, Leo (2023). Nimrod - experimental pack repository. `https://github.com/leofernandesmo/nimrod/tree/main`. Accessed: 2024-01-18.

Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *ES-EC/FSE*, pages 416–419.

Fraser, G., Staats, M., McMinn, P., Arcuri, A., and Padberg,

F. (2015). Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, 24(4):23:1–23:49.

Gheyi, R., Ribeiro, M., Souza, B., Guimarães, M., Fernandes, L., d'Amorim, M., Alves, V., Teixeira, L., and Fonseca, B. (2021). Identifying method-level mutation subsumption relations using Z3. *IST*, 132:106496.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2022). The Java Language Specification. Accessed: 2022-07-18.

Guimarães, M., Fernandes, L., Ribeiro, M., d'Amorim, M., and Gheyi, R. (2020). Optimizing mutation testing by discovering dynamic mutant subsumption relations. In *ICST*, pages 198–208.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *ESEC/FSE*, pages 654–665.

Kintis, M. and Malevris, N. (2015). Medic: A static analysis framework for equivalent mutant identification. *IST*, 68:1 – 17.

Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Traon, Y. L., and Harman, M. (2018). Detecting trivial mutant equivalences via compiler optimisations. *TSE*, 44(4):308–333.

Lakhotia, K., McMinn, P., and Harman, M. (2009). Automated test data generation for coverage: Haven't we solved this problem yet? In *Testing: Academic and Industrial Conference-Practice and Research Techniques*, pages 95–104.

Li, S., Xiao, X., Bassett, B., Xie, T., and Tillmann, N. (2016). Measuring code behavioral similarity for programming and software engineering education. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 501–510.

Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *ESEC/FSE*, pages 643–653.

Madeyski, L., Orzeszyna, W., Torkar, R., and Jozala, M. (2014). Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *TSE*, 40(1):23–42.

Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., and Teixeira, L. (2018). Detecting Overly Strong Preconditions in Refactoring Engines. *TSE*, 44(5):429–452.

Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., and Borba, P. (2014). Making refactoring safer through impact analysis. *SCP*, 93:39–64.

Naeem, M. R., Lin, T., Naeem, H., and Liu, H. (2020). A machine learning approach for classification of equivalent mutants. *Journal of Software: Evolution and Process*, 32(5).

Offutt, J. and Craft, M. (1994). Using compiler optimization techniques to detect equivalent mutants. *STVR*, 4(3):131–154.

Offutt, J., Ma, Y.-S., and Kwon, Y.-R. (2006). The class-level mutants of mujava. In *AST*, pages 78–84.

Pacheco, C., Lahiri, S., Ernst, M., and Ball, T. (2007). Feedback-directed random test generation. In *ICSE*, pages 75–84.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., and Harman, M. (2019). Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier.

Papadakis, M., Shin, D., Yoo, S., and Bae, D.-H. (2018). Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In *ICSE*, pages 537–548.

Peacock, S., Deng, L., Dehlinger, J., and Chakraborty, S. (2021). Automatic equivalent mutants classification using abstract syntax tree neural networks. In *ICSTW*, pages 13–18.

Pizzoleto, A. V., Ferrari, F. C., Offutt, J., Fernandes, L., and Ribeiro, M. (2019). A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *JSS*, 157.

Schuler, D., Dallmeier, V., and Zeller, A. (2009). Efficient mutation testing by checking invariant violations. In *ISSTA*, pages 69–80.

Schuler, D. and Zeller, A. (2013). Covering and uncovering equivalent mutants. *STVR*, 23(5):353–374.

Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., Mcminn, P., and Arcuri, A. (2015). Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211.

Soares, G., Gheyi, R., and Massoni, T. (2013a). Automated behavioral testing of refactoring engines. *TSE*, 39(2):147–162.

Soares, G., Gheyi, R., Murphy-Hill, E., and Johnson, B. (2013b). Comparing approaches to analyze refactoring activity on software repositories. *JSS*, 86(4):1006–1022.

Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE software*, 27(4):52–57.

Steimann, F. and Thies, A. (2010). From behaviour preservation to behaviour modification: Constraint-based mutant generation. In *ICSE*, pages 425–434.

van Hijfte, L. and Oprescu, A. (2021). Mutantbench: an equivalent mutant problem comparison framework. In *ICSTW*, pages 7–12.

Voas, J. and McGraw, G. (1997). *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc.

Yao, X., Harman, M., and Jia, Y. (2014). A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, pages 919–930.