INSTITUTO FEDERAL DE ALAGOAS

***CAMPUS* MACEIÓ**

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

SAMUEL VELOSO DE AMORIM

REDUZINDO ESFORÇOS MANUAIS NA ANÁLISE DE EQUIVALÊNCIA DOS TESTES DE MUTAÇÃO

MACEIÓ, AL

2024

SAMUEL VELOSO DE AMORIM

REDUZINDO ESFORÇOS MANUAIS NA ANÁLISE DE EQUIVALÊNCIA DOS TESTES DE MUTAÇÃO

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Sistemas de Informação do Instituto Federal de Alagoas - *Campus* Maceió, como requisito parcial para obtenção grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Leonardo Fernandes Mendonça de Oliveira

MACEIÓ, AL

2024

Dados Internacionais de Catalogação na Publicação
Instituto Federal de Alagoas
***Campus* Maceió**
Biblioteca Benevides Monte

005.1
A524r Amorim, Samuel Veloso de.

Reduzindo esforços manuais na análise de equivalência dos testes de mutação [recurso eletrônico] / Samuel Veloso de Amorim. – Dados eletrônicos (1 pdf : 1,24 MB). – 2024.

Trabalho com 49 f.

Inclui figuras, tabelas e referências.

Orientação: Prof. Dr. Leonardo Fernandes Mendonça de Oliveira.

Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) – Instituto Federal de Alagoas, *Campus* Maceió, Maceió, 2024.

1. Sistemas de Informação. 2. Testes de mutação. 3. Mutantes equivalentes. 4. Automatização de testes I. Título.

Franciane Monick Gomes de França
Bibliotecária – CRB 4/1831

FOLHA DE APROVAÇÃO

SAMUEL VELOSO DE AMORIM

REDUZINDO ESFORÇOS MANUAIS NA ANÁLISE DE EQUIVALÊNCIA DOS TESTES DE MUTAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informaçnao, do Instituto Federal de Alagoas, *Campus* Maceió, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Aprovado em: 30/10/2024.

BANCA EXAMINADORA

Documento assinado digitalmente
gov.br
LEONARDO FERNANDES MENDONCA DE OLIVEII
Data: 05/12/2024 06:51:33-0300
Verifique em https://validar.iti.gov.br

_____

Prof. Dr. Leonardo Fernandes Mendonça De Oliveira (Orientador)
Instituto Federal de Alagoas

Documento assinado digitalmente
gov.br
AUGUSTO CESAR MELO DE OLIVEIRA
Data: 09/12/2024 13:39:18-0300
Verifique em https://validar.iti.gov.br

_____

Prof. Me. Augusto César Melo de Oliveira
Instituto Federal de Alagoas

Documento assinado digitalmente
gov.br
FERNANDO KENJI KAMEI
Data: 06/12/2024 01:36:28-0300
Verifique em https://validar.iti.gov.br

_____

Prof. Dr. Fernando Kenji Kamei
Instituto Federal de Alagoas

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão ao meu orientador, Prof. Dr. Leonardo Fernandes, pela inestimável orientação, apoio e incentivo durante todo o processo de elaboração deste trabalho. Sua expertise e dedicação foram fundamentais para que eu superasse os desafios e alcançasse este resultado.

Ao Instituto Federal de Alagoas - Campus Maceió, expresso minha gratidão pela formação acadêmica recebida, pela estrutura disponibilizada e pelo ambiente de aprendizado, que contribuíram significativamente para meu crescimento pessoal e profissional.

Minha gratidão à minha namorada, que me incentivou a seguir em frente, ofereceu suporte emocional constante e compreendeu minha ausência enquanto eu me dedicava à realização deste trabalho.

Aos meus amigos, que estiveram ao meu lado ao longo desta jornada, oferecendo apoio e momentos de descontração, meu muito obrigado.

Por fim, às pessoas com quem convivi ao longo desses anos de curso, que me incentivaram e certamente tiveram impacto na minha formação acadêmica.

RESUMO

Os testes de mutação têm atraído muito interesse devido à sua reputação como um poderoso critério de adequação para suítes de teste e por sua capacidade de guiar a geração de casos de teste. No entanto, a presença de mutantes equivalentes dificulta seu uso na indústria. O Problema do Mutante Equivalente já foi provado indecidível, mas detectar manualmente mutantes equivalentes é uma tarefa sujeita a erros e que consome muito tempo. Assim, soluções, mesmo que parciais, podem ajudar a reduzir esse custo. Para minimizar esse problema, introduzimos uma abordagem para sugerir mutantes equivalentes que consistem em casos de teste baseados no comportamento do programa original. Realizamos uma análise estática para gerar automaticamente testes para as entidades impactadas pela mutação. Para cada mutante analisado, nossa abordagem pode classificar o mutante como equivalente ou não equivalente. No caso de mutantes não equivalentes, nossa abordagem fornece um caso de teste capaz de matá-lo. Para os mutantes equivalentes sugeridos, também fornecemos um *ranking* de mutantes com maior ou menor chance de serem de fato equivalentes. Testamos nossa abordagem em um conjunto de 1.542 mutantes classificados manualmente em trabalhos anteriores como equivalentes ou não equivalentes. Notamos que a abordagem sugere efetivamente mutantes equivalentes, atingindo mais de 93% de precisão em cinco dos oito projetos estudados. Comparado à análise manual dos mutantes sobreviventes, nossa abordagem leva um terço do tempo para sugerir equivalentes e é 25 vezes mais rápida para indicar os não equivalentes. Fizemos também um estudo para discernir as características específicas dos mutantes que nossa abordagem classificou erroneamente como equivalentes, gerando falsos positivos. Além disso, nossa investigação se aprofunda em uma análise abrangente dos operadores de mutação, fornecendo informações essenciais para profissionais que buscam melhorar a precisão da detecção de mutantes equivalentes e mitigar efetivamente os custos associados.

Palavras-chave: Mutantes equivalentes. Testes de mutação. Automatização de testes.

ABSTRACT

Mutation testing has attracted significant interest due to its reputation as a powerful criterion for test suite adequacy and its ability to guide test case generation. However, the presence of equivalent mutants hinders its use in industry. The Equivalent Mutant Problem has been proven undecidable, but manually detecting equivalent mutants is an error-prone and time-consuming task. Therefore, even partial solutions can help reduce this cost. To minimize this problem, we introduce an approach to suggest equivalent mutants. Our approach is based on automated behavioral tests, which consist of test cases based on the original program's behavior. We perform static analysis to automatically generate tests for the entities impacted by the mutation. For each analyzed mutant, our approach can suggest whether the mutant is equivalent or non-equivalent. In the case of non-equivalent mutants, our approach provides a test case capable of killing the mutant. For the suggested equivalent mutants, we also provide a ranking of mutants with a higher or lower likelihood of being truly equivalent. We tested our approach on a set of 1,542 mutants manually classified in previous works as either equivalent or non-equivalent. We found that the approach effectively suggests equivalent mutants, achieving over 93% accuracy in five of the eight studied subjects. Compared to the manual analysis of surviving mutants, our approach takes one-third of the time to suggest equivalent mutants and is 25 times faster in identifying non-equivalent mutants. We also conducted a study to discern the specific characteristics of mutants that our approach mistakenly classified as equivalent, generating false positives. Furthermore, our investigation delves into a comprehensive analysis of mutation operators, providing essential insights for professionals seeking to improve equivalent mutant detection accuracy and effectively mitigate the associated costs.

Keywords: Mutation testing. Equivalent mutant. Automated testing.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O Teste de Mutação é uma técnica poderosa empregada para aprimorar o processo de testes (DEMILLO; LIPTON; SAYWARD, 1978; JIA; HARMAN, 2011). Ao longo dos anos, têm atraído atenção significativa devido a estudos empíricos que demonstram sua capacidade de melhorar suítes de teste (ANDREWS; BRIAND; LABICHE, 2005; JUST *et al.*, 2014; PAPADAKIS *et al.*, 2018). A ideia fundamental por trás do teste de mutação envolve a aplicação de transformações sintáticas, conhecidas como operadores de mutação, para introduzir falhas artificiais em um programa, resultando no que é chamado de *mutante*. Posteriormente, a suíte de teste existente é executada contra esses mutantes para verificar se as falhas são detectáveis.

No contexto do teste de mutação, quando uma suíte de teste é capaz de distinguir a saída do programa original da saída do mutante, o mutante é considerado *morto*. Por outro lado, se o mutante permanecer indistinguível do programa original, ele é rotulado como *vivo*. O princípio subjacente do teste de mutação é que, quanto maior o número de mutantes mortos, maior a qualidade da suíte de teste.

No entanto, os custos de usar testes de mutação geralmente são altos, principalmente devido ao *Problema do Mutante Equivalente* (MADEYSKI *et al.*, 2014; KINTIS *et al.*, 2018; PIZZOLETO *et al.*, 2019; PAPADAKIS *et al.*, 2019). Um mutante equivalente é sintaticamente diferente do programa original, mas possui o mesmo comportamento em relação à saída observável. Dessa forma, não há nenhum teste capaz de matá-lo. Mutantes equivalentes são um impedimento bem conhecido para a adoção prática do teste de mutação. Um trabalho anterior (BUDD; ANGLUIN, 1982) já provou que este é um problema indecidível em sua forma geral. Assim, não existe uma solução completamente automatizada. Além disso, detectar manualmente mutantes equivalentes é uma tarefa propensa a erros e que consome muito tempo. Dos mutantes estudados, 20% foram classificados erroneamente (ACREE, 1980), e os desenvolvedores levam, em média, 15 minutos para classificar manualmente um mutante como equivalente ou não equivalente (SCHULER; ZELLER, 2013). Esse problema torna-se bastante relevante quando estudos empíricos relatam que até 40% de todos os mutantes gerados podem ser equivalentes (MADEYSKI *et al.*, 2014). Dessa forma, são necessários esforços de pesquisa para reduzir esses custos.

Neste trabalho propomos uma abordagem inovadora para sugerir mutantes equivalentes usando testes comportamentais automatizados e classificando os mutantes com base em seu comportamento durante a execução dos testes. Nossa abordagem aproveita ferramentas automatizadas de geração de casos de teste para criar testes direcionados para o local onde a mutação ocorreu. Ao executar esses testes tanto no programa original quanto no mutante, determinamos se há diferenças comportamentais. Se algum caso de teste falhar contra o mutante, indicando uma mudança de comportamento, o mutante é classificado como *não-equivalente*. Os testadores

podem usar tais testes para melhorar sua suíte de testes, reduzindo efetivamente os custos. Por outro lado, se nenhum caso de teste matar o mutante, nossa abordagem *sugere o mutante como equivalente*. Na ausência de uma prova definitiva de equivalência, oferecemos um sistema de classificação de mutantes com base em seu comportamento durante a execução dos testes. Essa classificação baseia-se em dois fatores principais observados ao longo dos testes: o impacto na cobertura, indicando discrepâncias na cobertura de execução entre o mutante e o programa original, e o número de casos de teste que exercem a instrução mutada. Através da síntese desses fatores, estabelecemos um ranking hierárquico de mutantes propostos como potenciais equivalentes, com aqueles que possuem evidências mais fortes de equivalência colocados no final, e aqueles com evidências mais fracas colocados no topo. Esse ranking priorizado facilita aos testadores a determinação dos mutantes que merecem revisão manual, iniciando o processo pelo topo. Nossa abordagem é implementada em uma ferramenta chamada NIMROD.

Até onde sabemos, nenhuma estratégia alternativa empregou geração automatizada de casos de teste especificamente direcionados ao ponto mutado para adquirir informações de equivalência. Embora várias estratégias louváveis tenham sido propostas para auxiliar os testadores, elas enfrentam limitações inerentes de escalabilidade, como aquelas associadas ao program slicing (VOAS; MCGRAW, 1997) e ao impacto de invariantes dinâmicos (SCHULER; DALLMEIER; ZELLER, 2009). Recentemente, técnicas emergentes têm empregado aprendizado de máquina para categorizar mutantes equivalentes (BRITO *et al.*, 2020; NAEEM *et al.*, 2020; PEACOCK *et al.*, 2021).

Apesar de sua natureza promissora, essas técnicas precisam ser avaliadas sob dois aspectos importantes: sua *eficácia*, que se refere à capacidade de atingir os resultados desejados ou cumprir os objetivos propostos, e sua *eficiência*, que diz respeito à execução dessas ações de forma otimizada, utilizando o menor tempo e esforço humano possível. Em outras palavras, é fundamental verificar se as técnicas não apenas alcançam os resultados esperados, mas também o fazem de maneira ágil e com um uso racional de recursos.

Avaliamos nossa abordagem contra 1.542 mutantes gerados a partir de oito métodos classificados manualmente como equivalentes ou não equivalentes em pesquisas anteriores (KINTIS *et al.*, 2018) aos quais 193 haviam sido avaliados como equivalentes. Submetemos todos os 1.542 mutantes ao NIMROD, sugerindo mutantes equivalentes, e depois calculamos precisão, recall e F-measure comparando o total de mutantes equivalentes sugeridos pelo NIMROD e o total de mutantes equivalentes encontrados na analise manual para verificar sua eficácia. A precisão mede a proporção de mutantes sugeridos como equivalentes que são realmente equivalentes, enquanto o recall avalia a proporção de todos os mutantes equivalentes corretamente identificados pelo NIMROD. Já a F-measure combina precisão e recall em uma única métrica, balanceando ambos os aspectos para oferecer uma visão mais ampla do desempenho do sistema. Os resultados indicam que a abordagem é eficaz em sugerir mutantes equivalentes. A F-measure atingiu mais de 93% em cinco dos oito métodos estudados. Para melhor analisar nossa abordagem, também calculamos o tempo gasto pelo NIMROD para sugerir mutantes equivalentes. Em média, o NIM-

ROD levou aproximadamente cinco minutos para classificar um mutante como potencialmente equivalente (três vezes mais rápido quando comparado às estimativas manuais (SCHULER; ZELLER, 2013)) e 24 segundos para classificar um mutante como não equivalente. Desenvolvido em Java, o NIMROD utiliza as ferramentas Randoop e EvoSuite para a geração automática de casos de teste, nossos resultados incluem o tempo para gerar os casos de teste usando ambas ferramentas.

Aprofundamos nossa avaliação, lançando luz sobre os desafios associados à detecção de mutantes equivalentes através do teste de mutação. Analisamos as características dos mutantes que apresentaram dificuldades de classificação pelo NIMROD, nossa ferramenta de testes comportamentais automatizados, fornecendo insights valiosos para refinar a abordagem dos testes de mutação e aprimorar a detecção de mutantes equivalentes na prática. Além disso, investigamos os operadores de mutação frequentemente associados a classificações incorretas pelo NIMROD, oferecendo orientações cruciais para os profissionais que buscam melhorar a precisão na detecção de mutantes equivalentes e reduzir custos.

As contribuições deste trabalho incluem:

- Uma abordagem inovadora para sugerir mutantes equivalentes baseada em testes comportamentais automatizados (Seção 3).
- Desenvolvimento e automação de toda a abordagem através de uma ferramenta chamada NIMROD disponível online em (FERNANDES, 2023) (Seção 3).
- Uma avaliação abrangente da eficácia e eficiência de nossa abordagem, destacando seu desempenho na classificação de mutantes como equivalentes ou não equivalentes (Seções 4 e 5).
- Uma análise aprofundada dos desafios associados à detecção de mutantes equivalentes, com foco nas características dos mutantes e operadores de mutação que levaram a classificações incorretas (Seção 5).
- Implicações para os profissionais sobre como combinar os métodos de *Sugestão* e de *Detecção de Mutantes Equivalentes* para otimizar a gestão de custos ao lidar com o problema da mutação equivalente (Seção 6).

2 REFERENCIAL TEÓRICO

Detectar mutantes equivalentes apresenta um problema desafiador em testes de mutação que já se sabe ser indecidível (BUDD; ANGLUIN, 1982). Abordar do problema não é uma questão recente (JIA; HARMAN, 2011; MADEYSKI *et al.*, 2014; PIZZOLETO *et al.*, 2019; PAPADAKIS *et al.*, 2019). Consequentemente, a tarefa de identificar mutantes equivalentes muitas vezes recai sobre os testadores humanos. No entanto, a detecção manual de mutantes equivalentes é propensa a erros, com julgamentos corretos alcançados em apenas cerca de 80% dos casos (ACREE, 1980), além de ser demorada, levando aproximadamente 15 minutos por mutante equivalente (SCHULER; ZELLER, 2013). Assim, torna-se imperativa a necessidade de heurísticas eficazes para identificar um subconjunto de mutantes equivalentes, a fim de minimizar os custos.

Uma dessas heurísticas, comumente empregada, baseia-se em otimizações de compiladores (OFFUTT; CRAFT, 1994; KINTIS *et al.*, 2018). A ideia básica por trás dessa abordagem é que as otimizações de código podem resultar em códigos compilados idênticos tanto para o programa original quanto para um mutante equivalente. O conceito de Trivial Compiler Equivalence (TCE) já foi introduzido e implementado para linguagens compiladas populares, como C e JAVA, juntamente com ferramentas de teste de mutação como MILU e MUJAVA (KINTIS *et al.*, 2018). A abordagem TCE é sólida, ou seja, se dois binários forem iguais, os programas terão o mesmo comportamento. Consequentemente, ela não gera falsos positivos. No entanto, pode não detectar mutantes equivalentes com o mesmo comportamento, mas com códigos diferentes, o que leva a potenciais falsos negativos.

Para ilustrar esse cenário, considere a classe `Triangle` apresentada na Listagem 1. Esta classe contém um método chamado `classify` que determina o tipo de um triângulo com base nos tamanhos de seus três lados. Geramos quatro diferentes mutantes equivalentes usando a ferramenta de mutação MUJAVA:

Listagem 1 – Um trecho de código extraído da classe `Triangle`.

```
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 }
```

Fonte: Elaborado pelo autor.

- M_1 representa um mutante criado pelo operador AOIU (Inserção de Operador Aritmético - unário), com a transformação: `tri + 1`⇒`-tri + 1`.
- M_2 representa um mutante gerado pelo operador ROR (Substituição de Operador Relacional), com a transformação: `tri == 1`⇒`tri <= 1`.
- M_3 e M_4 são mutantes criados pelo operador AOIS (Inserção de Operador Aritmético - atalho). Ambos inserem um pós-incremento no último acesso à variável local `b`.

Ao executar o TCE contra os mutantes, ele detecta dois dos quatro mutantes: M_1 e M_4. Ao analisar o mutante M_2, pode-se observar que o valor de `tri` começa com zero (linha 4). No entanto, ao chegar na linha mutada (linha 18), o valor de `tri` pode ser apenas um ou maior que um, o que impede a alteração de comportamento para qualquer possível entrada do programa. Para identificar esse mutante equivalente, o compilador precisaria verificar a expressão condicional da instrução `if` na linha 8. Caso a condição seja verdadeira, `tri` é zero e o método retorna.

Ao aplicar o TCE para detectar mutantes equivalentes, ele identifica com sucesso M_1 e M_4 como mutantes equivalentes. No entanto, para M_2, a detecção de equivalência por meio da otimização de código se torna desafiadora. O valor de `tri` é inicializado como zero (linha 4), e na linha mutada (linha 18), `tri` só pode ser um ou maior, dificultando a detecção da mudança de comportamento para qualquer entrada possível do programa. De forma similar, para M_3, aplicar

um pós-incremento ao último acesso de uma variável local dentro de um método não alterará o comportamento do programa (KINTIS; MALEVRIS, 2015; FERNANDES *et al.*, 2017). No entanto, a equivalência desse mutante está oculta na especificação do operador `&&` (GOSLING *et al.*, 2022). O operador condicional `&&` avalia seu operando à direita apenas se o operando à esquerda for `true`. O lado esquerdo da expressão condicional à qual M_3 pertence é mutuamente exclusivo com o lado esquerdo de outra expressão condicional na linha 24. Portanto, a detecção de mutantes equivalentes por meio da TCE se torna complexa e deixa lacunas.

Existem também estudos para evitar mutantes equivalentes mesmo antes de serem gerados. Por exemplo, especificações de heurísticas, baseadas em condições de equivalência, que evitam mutantes equivalentes para operadores de mutação em nível de classe (OFFUTT; MA; KWON, 2006). A aplicação de padrões de fluxo de dados para identificar equivalentes (KINTIS; MALEVRIS, 2015) e a introdução de uma estratégia para ajudar os desenvolvedores a derivar regras que evitem a geração de mutantes inúteis (equivalentes e duplicados) logo antes de sua geração (FERNANDES *et al.*, 2017).

Outros estudos verificam o impacto da execução do mutante. Quanto mais invariantes um mutante viola, mais provável é que ele seja detectado por testes reais (SCHULER; DALLMEIER; ZELLER, 2009). Também foram realizadas análises para verificar se mudanças na cobertura podem ser utilizadas para detectar mutantes não equivalentes (SCHULER; ZELLER, 2013).

Técnicas recentes classificam mutantes equivalentes utilizando aprendizado de máquina (ML), obtendo 80,30% de precisão ao classificar mutantes equivalentes em uma investigação com sete algoritmos tradicionais de ML (BRITO *et al.*, 2020). Outro estudo avaliou um modelo de rede neural de Árvore de Sintaxe Abstrata com dois operadores de mutação e 582 mutantes, resultando em uma precisão de classificação de 90% (PEACOCK *et al.*, 2021). Embora promissoras, essas técnicas ainda precisam ser avaliadas não apenas quanto à eficácia, mas também à eficiência.

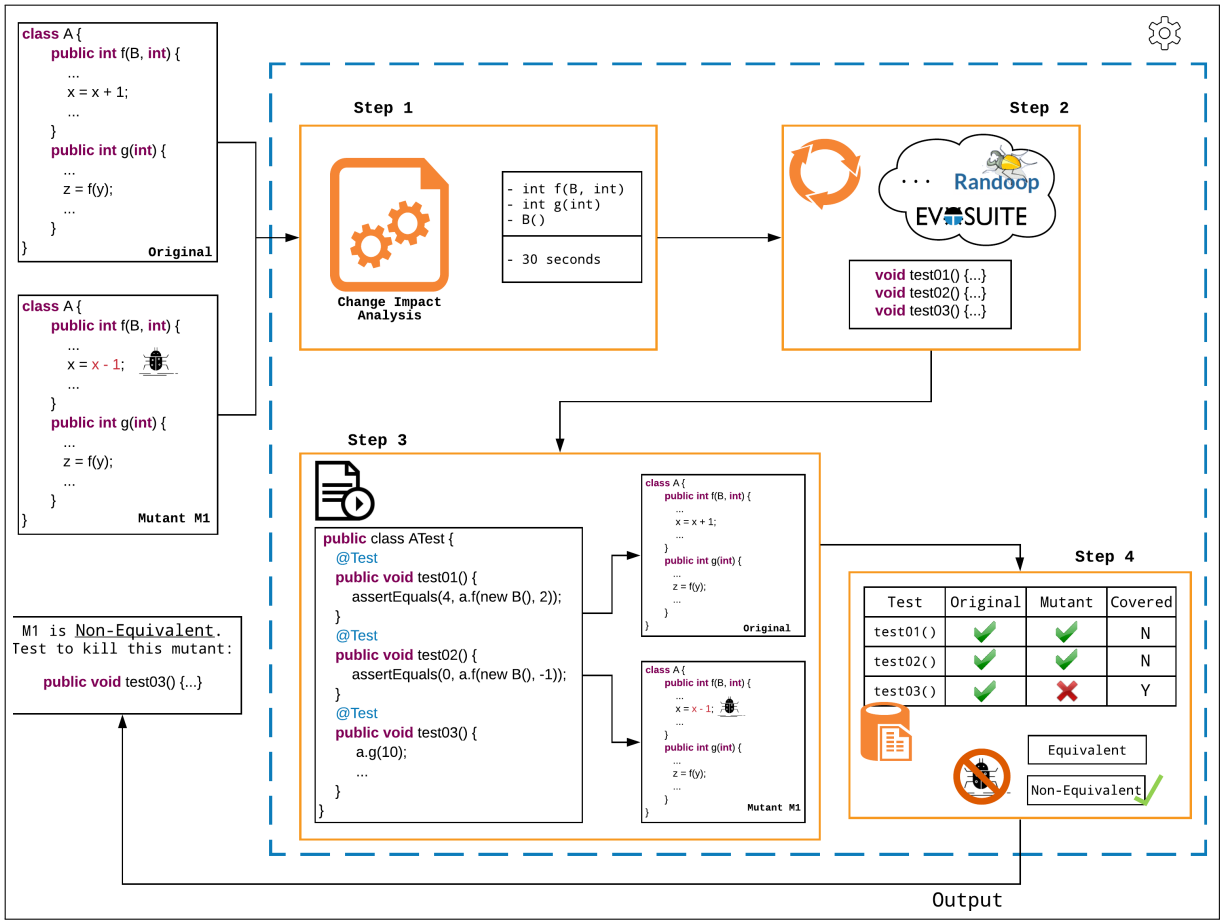
A ideia de gerar automaticamente um conjunto de testes para detectar mudanças comportamentais foi proposta no contexto de refatoração (SOARES *et al.*, 2010). Eles apresentaram o SAFEREFACTOR, uma ferramenta para melhorar a segurança durante atividades de refatoração, que conseguiu identificar bugs no Eclipse e no NetBeans (SOARES; GHEYI; MASSONI, 2013; MONGIOVI *et al.*, 2018).

Neste trabalho, estendemos a ideia inicial para adaptá-la ao contexto de testes de mutação e adicionamos mais ferramentas para gerar automaticamente os testes. Propomos então uma abordagem alternativa para sugerir mutantes equivalentes através de um ranking (Tabela 3), aproveitando testes comportamentais automatizados e o impacto da cobertura e também fornecemos ao usuário o teste que pode identificar a mudança comportamental. Embora nossa abordagem não possa garantir com absoluta certeza que um mutante seja de fato equivalente, ela pode fornecer sugestões fortes ou fracas com base no ranqueamento dos mutantes sobreviventes. Nossa abordagem identificou corretamente todos os mutantes equivalentes. Os detalhes de nossa abordagem são elaborados na seção subsequente.

3 ABORDAGEM

Nesta seção, explicamos a abordagem proposta para sugerir mutantes equivalentes. Nossa abordagem é baseada em trabalhos anteriores na área de refatoração (SOARES; GHEYI; MASSONI, 2013), porém enquanto a refatoração é uma transformação que preserva o comportamento externo de um programa, um mutante deve transformar um programa de modo que o comportamento externo do programa mude (STEIMANN; THIES, 2010). Adaptamos a solução de refatoração ao contexto de teste de mutação e adicionamos melhorias na fase de análise de impacto, na geração automatizada de testes e na execução pós-teste para melhorar a precisão da classificação de mutantes. A Figura 1 ilustra uma visão geral da abordagem. Ela consiste em quatro etapas principais. Primeiro, realiza uma análise de impacto da mudança causada pela mutação. Em segundo lugar, utiliza a saída da análise de impacto para orientar a geração de testes comportamentais automatizados. Na terceira etapa, executa cada caso de teste gerado contra o programa original e o mutante. Na etapa final, nossa abordagem sugere se o mutante é equivalente ou não e apoia o testador caso um teste para eliminar o mutante seja encontrado. Agora detalharemos cada uma das etapas.

Figura 1 – Nossa abordagem para sugerir mutantes equivalentes.



Fonte: Elaborado pelo autor.

3.1 IDENTIFICANDO ENTIDADES IMPACTADAS

Na Etapa 1, nossa abordagem recebe duas versões do programa como entrada: a original e a mutante. Realizamos um diff entre os dois programas para descobrir onde a mutação ocorreu. Manipulamos essas informações para realizar uma *análise de impacto de mudança* e gerar testes apenas para as entidades impactadas pela transformação. Nossa abordagem é baseada na análise de impacto de mudança proposta por Mongiovi *et al.* (2014). Ela verifica tanto os programas original quanto o mutante, começando pela decomposição de uma transformação de granularidade grosseira em transformações menores. Para cada transformação de granularidade fina, identificamos o conjunto de entidades impactadas e o conjunto de métodos públicos que precisam ser executados para alcançar essas entidades direta ou indiretamente. Além dos métodos públicos, também analisamos os parâmetros de tais métodos para identificar dependências entre métodos.

Para ilustrar o processo de identificação de entidades impactadas, consideramos a classe `FieldUtils` do projeto Joda-time como um exemplo (Listagem 2). O Joda-time é uma biblioteca popular de data e hora em JAVA. A classe `FieldUtils` possui 158 linhas de código e 17 métodos. Introduzimos três mutantes diferentes: $M1$ substitui o operador lógico AND (`&&`) pelo operador lógico OR (`||`) na linha 7, $M2$ insere um pós-incremento (`++`) na variável `total` na linha 17, e $M3$ insere um pré-decremento (`--`) na variável `val2` na linha 24.

Listagem 2 – Trecho extraído da classe `FieldUtils`.

```
1  public class FieldUtils {
2    public static int safeMultiplyToInt(long val1, long val2) {
3      long val = FieldUtils.safeMultiply(val1, val2);
4      return FieldUtils.safeToInt(val);
5    }
6    public static int safeToInt(long value) {
7      if (Integer.MIN_VALUE <= value &&    M1 [ && ⇒ || ]
8        value <= Integer.MAX_VALUE) {
9        return (int) value;
10     }
11     throw new ArithmeticException(...);
12   }
13   public static long safeMultiply(long val1, long val2) { ...
14     long total = val1 * val2; ...
15     return total;      M2 [ total ⇒ total++ ]
16   }
17   public static long safeSubtract(long val1, long val2) {
18     long diff = val1 - val2;
19     if ((val1 ^ diff) < 0 && (val1 ^ val2) < 0) {
20       throw new ArithmeticException
21       ("The calculation caused an overflow: " +
22       val1 + " - " + val2);    M3 [ val2 ⇒ --val2 ]
23     }
24     return diff;
25   } ...
26 }
```

Fonte: Elaborado pelo autor.

$M1$ ocorreu no método `safeToInt`. Note que esse método é chamado por outro método, `safeMultiplyToInt`. Assim, a saída da Etapa 1 é a seguinte.

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

Em relação ao mutante $M2$, a mutação ocorreu no método `safeMultiply`. Esse método é invocado pelo método `safeMultiplyToInt`. Nesse caso, a saída da análise de impacto da mudança é:

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

Em relação ao mutante $M3$, a mutação ocorreu no método `safeSubtract`. Esse método não é invocado por nenhum outro método. Assim, a saída da Etapa 1 é a seguinte:

> m:FieldUtils.safeSubtract(long, long)

Agora, passamos os resultados da análise de impacto da mudança para a etapa de geração de casos de teste (Etapa 2).

3.2 GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE

A geração automatizada de testes é um campo amplo de pesquisa (LAKHOTIA; MCMINN; HARMAN, 2009; SHAMSHIRI *et al.*, 2015; FRASER *et al.*, 2015). Pesquisadores exploraram diferentes abordagens para gerar automaticamente testes unitários, como geração de testes aleatórios, solucionadores de restrições, execução simbólica e algoritmos genéticos.

Ferramentas como EvoSuite (FRASER; ARCURI, 2011), Randoop (PACHECO *et al.*, 2007) e IntelliTest (LI *et al.*, 2016) implementam tais abordagens. Cada ferramenta tem um propósito específico, portanto, diferentes ferramentas gerarão uma sequência diferente de chamadas de método e afirmações. Essas sequências e afirmações dos testes gerados capturam o comportamento atual do programa original em teste. Embora ainda não tenham sido amplamente adotadas pela indústria, essas ferramentas de geração automatizada de testes unitários tornaram-se muito eficazes na geração de dados de entrada que alcançam alta cobertura de código (FRASER *et al.*, 2015) e encontram falhas reais (SHAMSHIRI *et al.*, 2015).

Após coletar as informações da análise de impacto da mudança (Etapa 1), na Etapa 2, usamos ferramentas de geração de testes automatizadas bem conhecidas para gerar um conjunto abrangente de casos de teste, baseado no programa original, para as entidades impactadas identificadas. A ideia é gerar um conjunto massivo de testes apenas para exercitar as entidades afetadas pela mutação, na tentativa de evidenciar a mudança de comportamento causada pela transformação. Observe que podemos instanciar nossa abordagem usando diferentes ferramentas de geração de testes ou até mesmo instanciar a mesma ferramenta mais de uma vez, utilizando diferentes parâmetros de entrada.

Para exemplificar a Etapa 2, retornamos à classe `FieldUtils` (Listagem 2). Na etapa anterior, apenas dois métodos foram impactados pelo mutante $M1$ (`safeToInt` e `safeMultiplyToInt`) e pelo mutante $M2$ (`safeMultiply` e `safeMultiplyToInt`). Para o mutante $M3$, apenas um método foi impactado (`safeSubtract`). A Listagem 3 mostra os casos de teste gerados para os mutantes $M1$, $M2$ e $M3$. Os testes da classe `FieldUtilsTest_M1` serão executados contra o programa original e o mutante $M1$. O mesmo ocorre com os testes da classe `FieldUtilsTest_M2` com o mutante $M2$ e da classe `FieldUtilsTest_M3` com o mutante $M3$.

Listagem 3 – Exemplos de casos de teste gerados para o `FieldUtils`.

```
1  public class FieldUtilsTest_M1{
2    @Test
3    public void test001(){
4      assertEquals(200, FieldUtils.safeMultiplyToInt(10L, 20L));
5    }
6    @Test
7    public void test002(){
8      try {
9        int val = FieldUtils.safeToInt(2147483648L);
10       fail("Failed: Should get an Arithmetic Exception");
11     } catch (ArithmeticException e) { }
12   } ...
13 }
14 public class FieldUtilsTest_M2{
15   @Test
16   public void test001(){
17     assertEquals(200, FieldUtils.safeMultiplyToInt(10L, 20L));
18   }
19   @Test
20   public void test002(){
21     assertEquals(25L, FieldUtils.safeMultiply(5L, 5L));
22   } ...
23 }
24 public class FieldUtilsTest_M3{
25   @Test
26   public void test001(){
27     assertEquals(-1, FieldUtils.safeSubtract(0L, 1L));
28   }
29   @Test
30   public void test002(){
31     try {
32       int val = FieldUtils.safeSubtract(Long.MIN_VALUE, 100L);
33       fail("Failed: Should get an Arithmetic Exception");
34     } catch (ArithmeticException e) { }
35   } ...
36 }
```

Fonte: Elaborado pelo autor.

3.3 EXECUÇÃO DE TESTES

Após a geração dos testes (Etapa 2), executamos os testes contra o programa original e um mutante por vez (Etapa 3). Caso um teste falhe no programa original, o descartamos. Essa não é uma situação comum para ferramentas de geração de testes unitários, pois elas capturam o comportamento atual do programa original. No entanto, a presença de resultados não determinísticos (como *flaky tests* (LUO *et al.*, 2014)) pode dificultar a execução. Dessa forma, terminamos com uma suíte de testes verde para o programa original.

Uma vez que identificamos um teste capaz de expor uma mudança de comportamento

no programa mutante, não executamos os testes subsequentes. Como nosso objetivo é sugerir mutantes equivalentes, não faz sentido continuar executando os testes seguintes, uma vez que o mutante já foi identificado como *não equivalente*.

Durante a etapa de execução dos testes, também registramos a cobertura de execução dos testes do programa original e dos mutantes (SCHULER; ZELLER, 2013). Em outras palavras, registramos a frequência com que cada linha foi executada por todos os testes gerados. Além disso, rastreamos o número de casos de teste que cobrem a instrução onde a mutação ocorreu. Usamos esses dados para criar um ranking dos mutantes sugeridos como equivalentes pela abordagem, como explicamos a seguir.

3.4 SUGESTÃO DE MUTANTES EQUIVALENTES

Na Etapa 4, analisamos os resultados da execução da suíte de testes para sugerir mutantes equivalentes. Mutantes que foram eliminados pelos casos de teste são marcados como *não equivalentes*, pois exibem uma mudança de comportamento em relação ao programa original. Para os mutantes que não foram eliminados, sugerimos que eles são *equivalentes*.

Como não podemos garantir que a sugestão está correta, fornecemos um *ranking* dos mutantes para melhor apoiar os testadores. A parte inferior do ranking inclui mutantes dos quais temos forte confiança de que são equivalentes, enquanto a parte superior inclui mutantes em que temos fraca confiança em nossa sugestão.

Nosso ranking de mutantes depende de duas informações registradas durante a execução dos testes: um valor booleano que indica se a cobertura de execução do mutante mudou em comparação com o programa original (também chamado de *impacto na cobertura*); e o número de casos de teste que atingiram o ponto mutado. Em nosso ranking, priorizamos o impacto na cobertura em relação ao número de testes que exercitaram a mutação.

Caso nenhum teste elimine o mutante, nossa abordagem sugere o mutante como *equivalente*. Como não podemos garantir que a sugestão está correta, fornecemos um *ranking* dos mutantes para melhor apoiar os testadores. Na parte inferior do ranking, colocamos os mutantes em que temos forte confiança de que são realmente equivalentes. Na parte superior, colocamos os mutantes em que temos fraca confiança de que nossa sugestão está correta. Schuler e Zeller (2013) identificaram que o impacto na cobertura fornece um meio eficaz de separar mutações equivalentes de não equivalentes. Eles relataram que, se uma mutação altera a cobertura, o mutante tem 75% de chances de ser não equivalente.

Para explicar melhor a última etapa de nossa abordagem, nos baseamos nos trechos de código apresentados na Listagem 2 e na Listagem 3. Como mencionado, a Etapa 4 analisa a execução da suíte de testes e classifica o mutante como equivalente ou não equivalente.

O teste `FieldUtilsTest_M1.test002` pode expor a mudança de comportamento do

mutante $M1$. A abordagem, então, marca este mutante como não equivalente e informa ao testador que `FieldUtilsTest_M1.test002` é suficiente para eliminar o mutante $M1$. Ao considerar o mutante $M2$, toda a suíte de testes gerada será executada, e o mutante não será eliminado, sendo sugerido como equivalente.

No caso do mutante $M3$, embora este mutante não seja equivalente, nenhum teste gerado identificou a mudança de comportamento. Assim, a abordagem sugere que $M3$ é equivalente. Neste ponto, temos dois mutantes sugeridos como equivalentes. Agora, verificamos as informações coletadas na etapa de execução dos testes para criar o ranking. Ambos os mutantes sugeridos não geraram impacto na cobertura. Em relação à instrução exercitada, o mutante $M2$ teve três casos de teste exercitando a instrução mutada, enquanto o mutante $M3$ teve apenas um caso de teste exercitando a instrução mutada. Dessa forma, o mutante $M3$ vai para o topo do ranking, e o mutante $M2$ fica na parte inferior do ranking. A saída de nossa abordagem seria:

> #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 MELHORIAS

Como explicado, nossa abordagem é baseada em ideias anteriores do campo de refatoração (SOARES; GHEYI; MASSONI, 2013; MONGIOVI *et al.*, 2014). Além de trazer essa ideia para o contexto de testes de mutação, fornecemos várias melhorias em relação ao trabalho relacionado.

Mongiovi *et al.* (2014) apresentaram uma análise de impacto de mudança para ajudar a identificar as entidades impactadas por uma transformação de código. Eles fornecem apenas a opção de análise *interclass*. No entanto, para projetos grandes com dependências complexas, é difícil identificar o que precisa ser testado após uma transformação, especialmente quando buscamos o conjunto de métodos indiretamente impactados que exercitam uma entidade afetada. A lista de métodos a serem testados pode se tornar grande, dificultando a eficácia das ferramentas de geração de testes. Assim, adicionamos a opção de a análise ser *intraclass*, ou seja, a análise de impacto identifica apenas os métodos públicos na classe onde a mutação ocorreu. Também adicionamos a análise dos parâmetros dos métodos. Se algum dos parâmetros não for de um tipo primitivo, classe wrapper ou tipo String, buscamos no *classpath* os construtores necessários para inicializar os objetos e realizar a chamada do método. A saída da análise de impacto de mudança é um conjunto de métodos públicos e, se necessário, um conjunto de construtores para construir as dependências de objetos.

Outras melhorias foram feitas nas Etapas 3 e 4. Durante a etapa de execução dos testes, registramos informações de cobertura para apoiar a fase de sugestão de mutantes equivalentes. Primeiro, seguimos a ideia proposta por Schuler e Zeller (2013) para calcular o impacto na cobertura ao executarmos os testes no programa original e no mutante. Em segundo lugar, usamos as informações de cobertura para contar o número de casos de teste que conseguiram exercitar a instrução onde a mutação ocorreu. Essas informações podem ajudar a avaliar o comportamento da mutação durante a computação. Como não podemos garantir que a sugestão de equivalência esteja correta, essas melhorias foram fundamentais para apoiar os resultados da abordagem.

A natureza da nossa abordagem permite uma redução de custos importante. O custo para o testador projetar e implementar um novo caso de teste para eliminar um mutante sobrevivente identificado como não equivalente. A saída da nossa abordagem indica qual caso de teste pode eliminar o mutante. Dessa forma, o *testador de mutação* pode usar os testes gerados automaticamente para melhorar sua suíte de testes *adequada a mutações*.

A implementação automatizada de nossa abordagem é fornecida através de uma ferramenta chamada NIMROD[1] A ferramenta realiza as etapas necessárias, incluindo análise de impacto de mudança, geração de casos de teste e execução de testes, para sugerir mutantes equivalentes em uma base de código. Seu objetivo é reduzir o esforço do testador, sugerindo potenciais mutantes equivalentes e fornecendo informações relevantes para apoiar o processo de tomada de decisão.

[1]Nimrod é um personagem fictício que aparece em *Uncanny X-Men* (março de 1985). Nimrod é um descendente poderoso e virtualmente indestrutível dos Sentinelas robóticos caçadores de mutantes.

4 AVALIAÇÃO

Nesta seção, apresentamos a avaliação de nossa abordagem automática de teste comportamental, NIMROD, para sugerir mutantes equivalentes sob a perspectiva de testadores de mutação no contexto da análise de mutação. Detalhamos as questões de pesquisa abordadas, os projetos utilizados, o ambiente experimental e o procedimento de avaliação.

4.1 QUESTÕES DE PESQUISA

Para avaliar o NIMROD, abordamos as seguintes questões de pesquisa:

RQ1. *Quão eficaz é o* NIMROD *em sugerir mutantes equivalentes?*

RQ2. *Quanto tempo o* NIMROD *leva para analisar um mutante?*

RQ3. *Quais são as características dos mutantes que o* NIMROD *falhou em classificar?*

RQ4. *Quais operadores de mutação comumente levam o* NIMROD *a falhar?*

Respondemos à RQ1 medindo a **Precisão**, **Recall** e **F-measure** do NIMROD na sugestão de mutantes equivalentes.

- **Precisão** refere-se à proporção de mutantes sugeridos como equivalentes que realmente são equivalentes. Em outras palavras, é a capacidade do sistema de evitar sugerir mutantes não equivalentes. A fórmula para precisão é:

$$\text{Precisão} = \frac{\text{Verdadeiros Positivos}}{\text{Verdadeiros Positivos} + \text{Falsos Positivos}}$$

  onde os *verdadeiros positivos* são os mutantes corretamente identificados como equivalentes, e os *falsos positivos* são os mutantes incorretamente identificados como equivalentes.

- **Recall**, por outro lado, mede a capacidade do sistema de identificar todos os mutantes equivalentes. Ele representa a proporção de mutantes equivalentes corretamente sugeridos em relação ao total de mutantes equivalentes disponíveis. A fórmula para recall é:

$$\text{Recall} = \frac{\text{Verdadeiros Positivos}}{\text{Verdadeiros Positivos} + \text{Falsos Negativos}}$$

  onde os *falsos negativos* são os mutantes equivalentes que não foram identificados como tal.

- **F-measure** é a média harmônica entre a precisão e o recall, fornecendo uma métrica única que combina ambos, equilibrando a necessidade de evitar falsos positivos e falsos

negativos. Isso é especialmente útil quando há um *trade-off* entre as duas métricas. A fórmula para F-measure é:

$$\text{F-measure} = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}}$$

Para estabelecer uma linha de base, usamos um conjunto de mutantes equivalentes identificados manualmente em um trabalho anterior (KINTIS et al., 2018). Isso nos permite calcular *verdadeiros positivos*, *falsos positivos*, *verdadeiros negativos* e *falsos negativos* com base na análise manual. Além disso, comparamos o NIMROD com a ferramenta TCE (KINTIS et al., 2018), que detecta mutantes equivalentes. É importante notar que o NIMROD e o TCE são ferramentas complementares, com o primeiro sugerindo mutantes equivalentes e o segundo detectando-os.

É importante abordar a RQ1, pois nos ajuda a determinar RQ2 entendendo o *tempo* e o *esforço* potencialmente economizados pelo NIMROD em comparação com a análise manual de cada mutante sobrevivente. Para responder à **RQ2**, calculamos o tempo médio que o NIMROD leva para sugerir os mutantes como equivalentes ou não equivalentes.

Uma vez que um teste mata o mutante, confirmamos que tal mutante é não equivalente. Isso significa que o NIMROD não possui falsos negativos, pois todos os mutantes classificados como não equivalentes são mortos por pelo menos um caso de teste. Por outro lado, o NIMROD pode erroneamente classificar mutantes como equivalentes (falsos positivos). Esses mutantes podem ser mutantes teimosos (YAO; HARMAN; JIA, 2014), onde apenas um caso de teste muito específico pode matá-lo. RQ3 analisa melhor os tipos de mutantes que nossa abordagem classifica erroneamente. Para responder à **RQ3**, analisamos o contexto do programa em que a mutação foi inserida.

Testadores de mutação geralmente utilizam um subconjunto dos operadores de mutação para realizar a análise. Portanto, ter informações sobre a relação entre os operadores de mutação e os mutantes equivalentes é útil. Isso nos leva à resposta da **RQ4**. Respondemos à **RQ4** calculando a contribuição de cada operador para a proporção de mutantes equivalentes, como a razão de cada operador para os falsos positivos do NIMROD. Habilitamos todos os 15 operadores de mutação em nível de método disponíveis no MUJAVA (Versão 3).

4.2 PROJETOS AVALIADOS

Para a avaliação, usamos programas e mutantes de um trabalho anterior (KINTIS *et al.*, 2018). Esse conjunto de programas é acompanhado por mutantes equivalentes identificados manualmente, fornecendo uma "verdade base"sobre a indecidibilidade de mutantes equivalentes.

A Tabela 1 detalha os programas JAVA utilizados na avaliação. As três primeiras colunas da tabela apresentam os programas examinados, as classes selecionadas e os métodos conside-

rados. As duas últimas colunas apresentam as linhas de código e o número de mutantes que haviam sido gerados.

Tabela 1 – Projetos JAVA analisados manualmente.

Projeto	Classe	Método	LoC	Total de Mutantes
Bisect	Um programa que calcula raízes quadradas	sqrt	23	135
Commons-lang	Um aprimoramento da biblioteca principal JAVA	capitalize	25	69
		wrap	45	198
Joda-time	Uma biblioteca de manipulação de tempo	add	33	257
Pamvotis	Uma biblioteca de manipulação de tempo	addNode	53	318
		removeNode	18	55
Triangle	Um programa clássico de classificação de triângulos	classify	44	354
XStream	Uma estrutura de serialização de objetos XML	decodeName	40	156
Total			**281**	**1,542**

Fonte: Elaborado pelo autor.

A lista de projetos avaliados inclui: *Bisect* - um programa simples que calcula raízes quadradas, *Commons-lang* - uma extensão da biblioteca principal do JAVA, *Joda-time* - uma biblioteca de manipulação de tempo, *Pamvotis* - um simulador de LAN sem fio, *Triangle* - um programa clássico de classificação de triângulos, e *XStream* - um framework de serialização de objetos XML.

4.3 CONFIGURAÇÃO EXPERIMENTAL

A avaliação foi conduzida em um PC de quatro núcleos com 2,70 GHz e 16 GB de RAM rodando Ubuntu 20.04.

O NIMROD lida com arquivos de configuração para cada programa a ser analisado, indicando as ferramentas de geração de testes e seus respectivos parâmetros de entrada. Nesta avaliação, utilizamos duas soluções populares de geração de testes: EvoSuite (FRASER; ARCURI, 2011) e Randoop (PACHECO *et al.*, 2007; SOARES; GHEYI; MASSONI, 2013).

O EvoSuite é uma ferramenta baseada em busca que utiliza um algoritmo genético para gerar suítes de teste para classes JAVA. Instanciamos o EvoSuite duas vezes para cada análise de mutantes: uma vez para o EvoSuite Regression (EvoSuiteR), para gerar suítes de teste que revelem diferenças entre duas versões de uma classe JAVA (a original e o mutante), e outra vez com quatro critérios de cobertura (cobertura de Declarações, Linhas, Ramos e Mutação Fraca)

para a geração de testes. Um limite de tempo de 60 segundos foi definido para a geração de testes.

O Randoop gera testes unitários para JAVA utilizando geração de testes aleatória orientada por feedback. Ele gera sequências de invocações de métodos/construtores de forma aleatória para as classes sob teste e cria asserções que capturam o comportamento real do programa. O Randoop normalmente é usado para criar testes de regressão. Também definimos um limite de tempo de 60 segundos para a geração de testes.

Para tornar nossa análise viável, estabelecemos certos limites neste trabalho. Isso inclui um máximo de 60 segundos para gerar testes, 80 segundos para executar a suíte de testes, e um limite de 3.000 testes. Com relação ao EvoSuite, descobrimos que não houve um aumento significativo na suíte de testes além do limite de 60 segundos. Por outro lado, o Randoop geralmente alcança 3.000 testes em menos de 60 segundos. Além disso, o tempo que leva para a suíte de testes mais longa ser executada contra o programa original é de 20 segundos. Como resultado, estabelecemos um tempo garantido de 80 segundos (4 x 20) para executar contra cada mutante, dado nosso escopo.

4.4 PROCEDIMENTO

Para realizar a avaliação, utilizamos os mutantes gerados com a ferramenta MUJAVA [1] (KINTIS *et al.*, 2018) com todos os operadores de mutação disponíveis a nível de método. Em seguida, executamos a análise de equivalência do NIMROD individualmente para cada mutante, comparando-o com o programa original.

Isso ocorre porque a análise de impacto da mudança (Seção 3) relata as entidades que precisam ser exercitadas pelos testes, e a natureza indecidível das ferramentas de geração automática de testes usadas. Assim, o conjunto de testes gerados para analisar um determinado mutante não é necessariamente igual ao conjunto de testes gerados para analisar outro mutante. Para cada mutante, executamos primeiro os testes gerados pelo EvoSuiteR, seguidos dos testes gerados pelo EvoSuite com quatro critérios de cobertura, e, finalmente, os testes gerados pelo Randoop. Essa ordem é escolhida porque o EvoSuite frequentemente pode revelar mudanças de comportamento com menos testes em comparação ao Randoop. Se um caso de teste falhar ou a execução da suíte de testes atingir o tempo limite, o NIMROD suspende a execução dos testes, informa que o mutante não é equivalente e escreve o caso de teste que expôs a mudança de comportamento. Se toda a suíte de testes for executada contra o mutante sem nenhuma falha de caso de teste, nem atingir o tempo limite, o NIMROD termina a análise do mutante, informa que o mutante é *possivelmente equivalente*, e escreve o impacto da cobertura e o número de casos de teste que atingiram a instrução mutada. Após a conclusão da análise para cada projeto, coletamos os resultados e criamos um ranking dos mutantes sugeridos como equivalentes.

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

Para uma avaliação abrangente, também executamos e coletamos dados da ferramenta TCE (KINTIS *et al.*, 2018) nos mesmos mutantes, e a lista de mutantes equivalentes manualmente classificados por Kintis et al. está disponível em nosso site complementar (FERNANDES, 2023).

Na próxima seção, apresentamos os resultados experimentais e discutimos as principais descobertas da avaliação.

5 RESULTADOS E DISCUSSÃO

Nesta seção, apresentamos os resultados e abordamos as questões de pesquisa. A base para nossa análise consiste em 193 mutantes[1] classificados manualmente como equivalentes em pesquisas anteriores (KINTIS *et al.*, 2018). Esses 193 mutantes representam 12,5% do total de 1.542 mutantes analisados.

Antes de detalharmos nossa abordagem, é importante esclarecer a noção de equivalência que adotamos. Um mutante e um programa original são equivalentes se eles apresentarem o mesmo comportamento externamente observável para todas as entradas possíveis. No entanto, podemos dividir essa suposição em dois cenários diferentes: *mundo aberto* e *mundo fechado* (SOARES; GHEYI; MASSONI, 2013). Na suposição de *mundo aberto* (OWA), qualquer caso de teste pode ser gerado para descobrir uma mudança de comportamento, sem considerar os requisitos do projeto ou do código. Na suposição de *mundo fechado* (CWA), os casos de teste devem satisfazer algumas restrições de domínio. Por exemplo: "*todos os métodos testados devem ser chamados através de uma Facade*" ou "*existe uma sequência estrita de chamadas de métodos a ser seguida*". No CWA, um mutante equivalente pode, por exemplo, indicar que o teste está violando um requisito do sistema ou que o sistema possui uma falha de segurança. Nossa abordagem adota a noção de equivalência no mundo aberto, o que significa que não há restrições na geração dos testes.

No NIMROD, existe a possibilidade de nossa abordagem classificar erroneamente um mutante como equivalente quando, na realidade, o mutante é um mutante *teimoso*. Um mutante teimoso é aquele que permanece não detectado por uma suíte de testes de alta qualidade e, portanto, não é equivalente (YAO; HARMAN; JIA, 2014). Isso representa um *falso positivo* em nossa abordagem. Por outro lado, se o NIMROD encontra um teste que mata o mutante, mas a análise manual relata que esse mutante é equivalente, duas situações podem ocorrer: (*i*) a análise manual está incorreta; ou (*ii*) os testes, embora executados corretamente, foram escritos de forma incorreta.

5.1 RQ1. QUÃO EFICAZ É O NIMROD EM SUGERIR MUTANTES EQUIVALENTES?

A Tabela 2 apresenta os resultados gerais da execução do NIMROD nos projetos avaliados. As colunas 1 e 2 mostram, respectivamente, o Programa e o nome do Método. A Coluna 3 indica o número de mutantes gerados para cada método. No total, a ferramenta de mutação MUJAVA gerou 1.542 mutantes. A Coluna 4 mostra os mutantes equivalentes de acordo com a análise manual, a linha de base. As Colunas 5 e 6 mostram o número de mutantes equivalentes (N) de

[1] Inicialmente, o artigo reportou 196 mutantes equivalentes, mas após uma nova análise, os autores atualizaram o site de acompanhamento, e esse número caiu para 193.

acordo com a detecção do TCE e a sugestão do NIMROD, respectivamente. Para os resultados do TCE e do NIMROD, também apresentamos o número de Falsos Positivos (FP) e Falsos Negativos (FN) na tabela.

Para maior clareza, nos referiremos aos projetos pelos nomes únicos dos métodos (Coluna 2) e pelos nomes dos programas (Coluna 1) entre parênteses.

Tabela 2 – Resultados gerais da execução do NIMROD.

Projeto	**Método**	**Total de Mutantes**	**Mutantes Equivalentes**		
			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**

Fonte: Elaborado pelo autor.

A análise manual identificou 193 mutantes equivalentes, que utilizamos como base de comparação com nossos resultados. A ferramenta TCE detectou 109 dos 193 mutantes equivalentes (56%). Como o TCE é uma solução para detectar mutantes equivalentes (MADEYSKI *et al.*, 2014), não há falsos positivos (FP) nos resultados. No entanto, o TCE pode ter falsos negativos (FN), ou seja, pode não identificar todos os mutantes equivalentes. Todos os 84 falsos negativos ocorreram porque a otimização aplicada pelo TCE produziu um bytecode que difere do programa original correspondente. É importante notar que essa situação pode ocorrer mesmo quando o mutante exibe o mesmo comportamento que o programa original. Utilizamos os dados do TCE como referência para entender melhor os resultados do NIMROD. Não é o objetivo deste trabalho sugerir uma solução melhor, pois nossa abordagem, juntamente com o TCE, pode ser complementar do ponto de vista da análise de mutação.

Em contraste com a precisão do TCE na detecção de mutantes equivalentes, nossa abordagem tenta sugerir se um mutante é equivalente por meio de testes comportamentais automatizados. Dos 1.542 mutantes analisados, o NIMROD classificou 449 como equivalentes. Esse número é mais do que o dobro dos 193 mutantes identificados manualmente como equivalentes. De fato, esperávamos que nossa solução sugerisse mais equivalentes do que o número total de mutantes que são realmente equivalentes pois os mutantes que o NIMROD classificou erroneamente como equivalentes representam os falsos positivos (FP).

As Tabelas 4a a 4h apresentam os resultados detalhados da execução do NIMROD e do TCE nos projetos analisados. Cada tabela representa um projeto e mostra o número de mutantes equivalentes identificados manualmente, sugeridos pelo NIMROD e detectados pelo TCE. Para cada tabela, também calculamos a Precisão, Recall e F-Measure para o TCE e o NIMROD. Essas informações nos ajudam a avaliar o desempenho do NIMROD em cada projeto.

A abordagem conseguiu 100% de Recall em todos os projetos avaliados, indicando assim que não gera falsos negativos.

Com base na Precisão e F-measure, em três dos oito projetos avaliados, a saber, *decodeName* (XStream), *add* (Joda-time) e *classify* (Triangle), a abordagem alcançou 100% em abmas. Em dois projetos, *sqrt* (Bisect) e *capitalize* (Commons-lang), o F-measure foi superior a 96% e a Precisão superior a 93%. Nos outros dois projetos, *wrap* (Commons-lang) e *removeNode* (Pamvotis), a abordagem teve um F-measure superior a 82% e Precisão superior a 70%.

No entanto, o NIMROD exibiu uma precisão muito baixa no projeto *addNode* (Pamvotis), sugerindo 277 de 318 mutantes como equivalentes. Isso é oito vezes mais do que os 33 mutantes marcados manualmente como equivalentes. Ao analisar os falsos positivos para esse projeto, encontramos algumas características no programa alvo e nos mutantes que podem explicar esse resultado. O método `addNode` tem a seguinte assinatura: `void addNode(int, int, int, int, int, int)`. Ele não retorna um valor, o que requer que o teste use um `assert` que verifique o estado do programa utilizando outro método ou um campo (ou uma exceção para casos excepcionais). Além disso, a maioria dos mutantes marcados erroneamente como equivalentes altera campos que não possuem métodos públicos ou estão localizados em classes diferentes daquela onde a mutação ocorreu (discutiremos esses casos na próxima seção). Em contraste, este foi o único projeto em que o TCE teve 100% de precisão.

Como explicado, o NIMROD calcula duas métricas para criar a classificação e, assim, apoiar o testador: o número de casos de teste que atingiram o ponto mutado e um valor booleano indicando se a execução do teste teve um impacto na cobertura. Classificamos os mutantes usando os seguintes critérios: primeiro, o impacto na cobertura, e depois o número de casos de teste que exercitaram o ponto mutado.

No entanto, não podemos definir um número limite geral que determine quantos mutantes devem ser analisados manualmente em todos os projetos. Cabe ao testador decidir quais mutantes serão revisados manualmente. Neste estudo, definimos o número mediano de casos de teste que tocaram o ponto mutado como o limite para verificar a precisão da classificação. Após isso, verificamos quantos falsos positivos permaneceram antes ou depois da mediana.

A Tabela 3 apresenta os 18 mutantes de *sqrt* (Bisect) sugeridos como equivalentes pelo NIMROD. De acordo com a análise manual, 17 mutantes são equivalentes, o que significa que a classificação do NIMROD teve um falso positivo. O falso positivo é o AOIS_12. Nenhum mutante teve impacto na cobertura. Usando o número de casos de teste que tocaram o ponto mutado, o valor mediano para os 18 mutantes é 204 casos de teste. O mutante AOIS_12 foi exercido por

191 casos de teste. Note que este mutante está abaixo do valor mediano ($191 < 204$). Valores mais baixos podem representar potenciais mutantes não equivalentes. Portanto, mutantes abaixo da mediana poderiam ser selecionados para análise manual.

Tabela 3 – Os mutantes *sqrt* (Bisect) sugeridos como equivalentes. O AOIS_12 é o falso positivo (em negrito) e a linha dupla marca a divisão com base na mediana.

Mutante	Impacto de Cobertura	Num. de Casos de Testes Exercitados
AOIS_43	Não	61
AOIS_48	Não	137
AOIS_60	Não	142
AOIS_31	Não	143
ROR_13	Não	146
AOIS_45	Não	156
AOIU_12	**Não**	**191**
AOIS_74	Não	199
ROR_12	Não	200
AOIS_59	Não	208
AOIU_4	Não	213
ROR_8	Não	221
AOIS_44	Não	235
AOIS_47	Não	245
AOIS_79	Não	311
AOIU_3	Não	329
AOIS_73	Não	386
AOIS_80	Não	465
MEDIAN		204

Fonte: Elaborado pelo autor.

A Tabela 4 apresenta os falsos positivos por projeto. Somente os projetos que tiveram pelo menos um falso positivo estão listados na tabela. Nos projetos *sqrt* (Bisect) e *capitalize* (Commons-Lang), apenas um mutante foi erroneamente classificado. Em ambos os casos, os falsos positivos estavam abaixo da mediana.

No projeto *addNode* (Pamvotis), apesar de haver muitos falsos positivos, 214 (88%) dos 244 estavam abaixo da mediana. Ao examinar os detalhes do resultado, identificamos que 106 (38%) mutantes não foram exercidos por nenhum caso de teste. A maioria desses mutantes estava dentro de uma estrutura `switch-case`, que estava aninhada com uma condicional `if`. O pior caso ocorreu com o *wrap* (Commons-lang). Este projeto possui uma estrutura com três condicionais if aninhadas. Todos os falsos positivos estavam, em algum ponto, nessa estrutura, e o número de casos de teste que tocaram esses mutantes variou de dois a sete. Isso é relativamente baixo, já que uma média de três mil testes foi gerada para esses mutantes.

Tabela 4 – Projetos analisados.

(a) Projeto: *sqrt* (Bisect).

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

(b) Projeto: *classify* (Triangle).

	MANUAL	NIMROD	TCE
EQUIVALENTES	40	40	21
PRECISÃO		100.00%	100.00%
RECALL		100.00%	52.50%
F-MEASURE		100.00%	68.85%

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

	MANUAL	NIMROD	TCE
EQUIVALENTES	28	28	0
PRECISÃO		100.00%	0.00%
RECALL		100.00%	0.00%
F-MEASURE		100.00%	-

(d) Projeto: *add* (Joda-time).

	MANUAL	NIMROD	TCE
EQUIVALENTES	35	35	24
PRECISÃO		100.00%	100.00%
RECALL		100.00%	64.86%
F-MEASURE		100.00%	78.69%

(e) Projeto: *capitalize* (Commons-lang).

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

(f) Projeto: *wrap* (Commons-lang).

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

(g) Projeto: *addNode* (Pamvotis).

	MANUAL	NIMROD	TCE
EQUIVALENTES	33	277	33
PRECISÃO		11.91%	100.00%
RECALL		100.00%	100.00%
F-MEASURE		21.29%	100.00%

(h) Projeto: *removeNode* (Pamvotis).

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

Fonte: Elaborado pelo autor.

Resposta à RQ1: A precisão da nossa abordagem alcançou 100% em três projetos, mais de 93% em dois projetos e mais de 82% em dois outros projetos estudados. Em apenas um projeto, o desempenho foi abaixo de 22%, discutido em RQ3. Definimos o número mediano de casos de teste que tocaram o ponto mutado para distinguir mutantes com uma forte ou fraca chance de serem equivalentes. Em dois casos, os resultados alcançaram 100% de precisão, e no pior caso, a precisão foi de 57%.

5.2 RQ2. QUANTO TEMPO O NIMROD LEVA PARA ANALISAR UM MUTANTE?

Para avaliar a eficiência da abordagem e responder à RQ2, calculamos o tempo médio que o NIMROD levou para sugerir cada mutante como equivalente ou não equivalente. A Tabela 5 apresenta o tempo médio em segundos para cada projeto. O projeto *classify* (Triangle) levou em média 197,10 segundos para sugerir um mutante como equivalente e 11,46 segundos para detectar um mutante não equivalente, sendo o projeto que obteve a melhor eficiência, pois conseguiu o menor tempo médio para avaliar os mutantes equivalentes e não equivalentes e não evidenciou falsos positivos.

Tabela 5 – Tempo médio que o NIMROD levou para analisar cada mutante e distribuição dos falsos positivos de acordo com a mediana.

Projeto	Método	Tempo Médio (segundos)		Falsos Positivos		
		Equiv.	Não Equiv.	Qnt	Mediana $\Longleftarrow$	Mediana $\Longrightarrow$
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		

Fonte: Elaborado pelo autor.

Nossa abordagem possui um tempo de resposta médio rápido para os casos em que o mutante é sugerido como não equivalente. Isso acontece porque, uma vez que um teste mata o mutante, finalizamos a análise. Escolhemos realizar essa fase sequencialmente e definimos o EvoSuite Regression Testing (EvoSuiteR) como a primeira opção. Isso permitiu que mutantes fáceis de matar fossem rapidamente descobertos e eliminados.

Para sugerir um mutante como não equivalente, o projeto *sqrt* (Bisect) teve os piores resultados. Neste projeto, alguns mutantes não equivalentes fizeram com que NIMROD gerasse testes que atingiram o limite de tempo de execução devido a loops infinitos causados pelos mutantes. Portanto, embora essa situação levante uma mudança de comportamento, NIMROD gasta muito tempo até o tempo limite. O projeto *classify* (Triangle) teve o melhor tempo de resposta para detectar mutantes não equivalentes. Este projeto possui estruturas de código relativamente simples quando comparado aos outros projetos do estudo. Por exemplo, sem dependências com classes externas e sem expressões condicionais complexas. Essa condição leva à geração de muitos mutantes fáceis de matar.

Para sugerir um mutante como equivalente, NIMROD precisa gerar e executar todos os testes de todas as ferramentas de geração de testes. Os projetos *classify* (Triangle) e *sqrt* (Bisect) tiveram os melhores resultados, com uma média de 197,10 segundos e 198,37 segundos para analisar um único mutante equivalente. Ambas as classes dos dois projetos não têm dependências com classes externas, isso permite que as ferramentas de geração de testes, especialmente o EvoSuite, não levem tanto tempo para gerar os testes.

Por outro lado, o projeto *addNode* (Pamvotis) levou uma média de 404,47 segundos para sugerir um mutante como equivalente. Como explicado, este projeto possui algumas características que dificultam a geração de testes. Pode-se perguntar se o tempo gasto por NIMROD é aceitável. Observe que os resultados de tempo que relatamos dependem das configurações que usamos nas ferramentas de geração de testes. Por exemplo, configuramos 60 segundos de

limite de tempo para cada instância (Randoop uma vez, EvoSuite duas vezes) para gerar os testes. Essas configurações levaram NIMROD a levar aproximadamente cinco minutos para sugerir um mutante como equivalente. Vale mencionar que a identificação de mutantes equivalentes é uma tarefa manual no último caso.

Classificar manualmente mutantes como equivalentes e não equivalentes leva uma média de 15 minutos por mutante (SCHULER; ZELLER, 2013). Além disso, essa tarefa é propensa a erros: 20% dos mutantes estudados foram classificados erroneamente (ACREE, 1980). NIMROD pode reduzir esse trabalho, pois leva um terço do tempo manual e classifica os mutantes indicando quais deles são mais propensos a serem equivalentes. Além disso, para um mutante não equivalente, o tempo médio é reduzido para 36,57 segundos, 25 vezes menos do que o tempo manual. O TCE pode analisar a equivalência em menos de dois segundos (KINTIS *et al.*, 2018). Na Seção 6, apresentamos uma aplicação prática combinando TCE e NIMROD como uma alternativa para o problema de mutantes equivalentes.

Resposta à RQ2: NIMROD levou uma média de 306,55 segundos por mutante equivalente analisado e 36,57 segundos por mutante não equivalente. Enquanto a análise manual de um mutante para indicar se é equivalente ou não pode levar 15 minutos (SCHULER; ZELLER, 2013), NIMROD leva um terço desse tempo para sugerir mutantes equivalentes e é 25 vezes mais rápido para indicar mutantes não equivalentes.

5.3 RQ3. QUAIS SÃO AS CARACTERÍSTICAS DOS MUTANTES QUE NIMROD NÃO CONSEGUIU CLASSIFICAR?

Para abordar a RQ3, examinamos cuidadosamente todos os falsos positivos de cada projeto. A tabela 6 apresenta as características comuns do código fonte que levaram NIMROD a sugerir mutantes como equivalentes quando, na verdade, não o são. Elencamos os projetos em três características: *Modificador de Nível de Acesso*, *Entidades Externas* e *Valor Muito Restrito*. Dos 1.542 mutantes analisados, um total de 256 (16,60%) mutantes foram classificados incorretamente. Notavelmente, a maioria desses mutantes mal classificados, especificamente 244 (95%), estava associada ao projeto *addNode* (Pamvotis). Nesta seção, apresentamos uma avaliação qualitativa dos falsos positivos para cada uma dessas características.

Tabela 6 – Características comuns (falso positivos) nos resultados do NIMROD.

Problem	Description	Subjects
Modificador de Nível de Acesso	Para matar o mutante, o teste precisa estar na mesma estrutura pacote que o código-fonte do programa.	sqrt (Bisect), addNode (Pamvotis), removeNode (Pamvotis).
Valor Muito Restrito	Para matar o mutante, o teste precisa gerar um valor muito específico.	capitalize (Commons-lang), wrap (Commons-Lang), addNode (Pamvotis), removeNode (Pamvotis).
Entidades Externas	Para matar o mutante, o teste precisa executar e declarar entidades em classes diferentes do local mutado.	addNode (Pamvotis), removeNode (Pamvotis).

Fonte: Elaborado pelo autor.

O problema do *Modificador de Nível de Acesso* ocorre quando o caso de teste e o código fonte precisam estar na mesma estrutura de pacote para que o mutante possa ser eliminado pelo teste. A Listagem 4 apresenta um trecho de código do projeto *sqrt* (Bisect).

No mutante M_1, o operador AOIU (Inserção de Operador Aritmético - Unary) insere um operador de menos no lado direito de uma atribuição a uma variável de campo. Essa transformação altera o valor atribuído ao campo `mResult`. Este campo não tem outro uso ou definição no método `sqrt`. Da mesma forma, não tem outro acesso em qualquer método desta classe que indique uma mudança no comportamento.

Isso nos levou a acreditar que esse mutante poderia ser equivalente; no entanto, como pode ser visto na Listagem 4, esse campo foi declarado como *package-private* (sem modificador explícito). Portanto, para eliminar este mutante, é necessário usar um artifício da linguagem JAVA para contornar a restrição de visibilidade do campo. O teste deve ser criado no mesmo pacote que a classe original sob teste e a asserção deve observar o estado do campo. Assim, terá acesso direto ao campo sem a necessidade de passar por um método de acesso (por exemplo, `getMResult()`) para esse propósito. Conseguimos configurar o EvoSuite para seguir a mesma estrutura de pacotes que o programa original. No entanto, não conseguimos fazer os testes realizarem asserções em campos *package-private*.

No exemplo da classe `Bisect`, se os desenvolvedores do projeto tivessem definido que os testes unitários e o código fonte original deveriam estar em pacotes diferentes, o mutante AOIU seria equivalente (CWA). Como estamos considerando todos os projetos com base na noção de OWA, o mutante AOIU é considerado não equivalente. Portanto, aqui NIMROD falhou.

Para resolver esse problema, os testes do NIMROD devem seguir a mesma estrutura de pacotes da classe sob teste e, além disso, a asserção do teste deve utilizar os campos disponíveis da classe.

Listagem 4 – Trecho de código extraído do projeto `sqrt`.

```
1  public class Bisect {
2    double mEpsilon, mResult;
3    ...
4    public double sqrt( double N ){
5          ...
6          while (Math.abs( diff ) > mEpsilon) {
7              ...
8          }
9          r = x;
10         mResult = r;          M1 [mResult = r; ⇒ mResult = -r;]
11         return r;
12     }
13 }
```

Fonte: Elaborado pelo autor.

O problema do *Valor Muito Restrito* ocorre quando a ferramenta de teste automático não gera um teste com uma entrada que exerça a mudança de comportamento feita pela mutação. Para explicar melhor essa característica, usamos um exemplo extraído do projeto *wrap* (Commons-lang).

Na Listagem 5, o mutante M_2 (linha 21) foi gerado pelo operador de mutação AORB (Substituição de Operador Aritmético). Aqui, ele substitui o operador aritmético + por %. Para alterar o comportamento deste mutante, o teste deve atingir a linha mutada, que está dentro de várias instruções `if` aninhadas. Além disso, a variável `offset` não pode ser redefinida nas repetições subsequentes do while. Durante nosso estudo, este ponto mutado foi exercitado apenas duas vezes, mesmo tendo mais de 3.000 testes gerados por ferramentas de geração automática.

Listagem 5 – Trecho de código extraído do projeto *wrap* (WordUtils).

```
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 }
```

Fonte: Elaborado pelo autor.

Já é sabido pela comunidade de testes de software que criar testes automaticamente para alcançar uma alta cobertura de branches é difícil. Uma possível solução para o NIMROD resolver esse problema é aumentar o limite de tempo das ferramentas para gerar os testes e permitir que um maior número de testes seja gerado (limitemos essas configurações a 60 segundos de limite de tempo e um máximo de 3.000 testes). No entanto, essas decisões implicam diretamente no tempo total para sugerir um mutante como equivalente ou não equivalente. Novas abordagens para resolver esse problema foram apresentadas nos últimos anos (BRAIONE *et al.*, 2017). Assim, as próximas versões das ferramentas de geração automática de testes provavelmente mostrarão melhorias a esse respeito.

O problema das *Entidades Externas* ocorre quando o teste precisa executar ou afirmar entidades que não estão diretamente localizadas na classe alvo onde a mutação ocorreu. A Listagem 6 apresenta um trecho de código extraído do projeto *addNode* (Pamvotis). O método (linhas 4-24) não possui uma instrução de retorno e seu principal objetivo é construir um objeto `MobileNode` e colocar este objeto em um Vector (linha 21). No mutante M_3, o operador AOIS insere um pós-decremento na variável `SpecParams.CW_MAX` (linha 13). Essa variável global é estática, pública e possui apenas um ponto de definição na classe `SpecParams`.

Listagem 6 – Trecho de código extraído do projeto *addNode*.

```
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 }
```

Fonte: Elaborado pelo autor.

Para identificar a mudança de comportamento deste mutante, o teste precisa afirmar o estado da variável `SpecParams.CW_MAX` após a execução do método `addNode`. No entanto, nossa análise de impacto (apresentada na Seção 3 é intraclasse e não considera entidades impactadas em classes/arquivos externos.

Como explicado, a noção de equivalência do NIMROD é baseada no comportamento exposto pelo programa através da execução de testes gerados automaticamente. Portanto, para ter um resultado satisfatório, a classe sob teste deve ser projetada de forma que os testes unitários possam ser executados (BINDER, 1994). No entanto, isso nem sempre é o caso, tornando a redação de um bom teste nesse sentido uma tarefa difícil. Quando isso ocorre, o NIMROD falha em gerar um teste que poderia alterar o comportamento do mutante em comparação com o programa original. Essa situação pode levar o NIMROD a produzir *falsos positivos*. Observamos que o alto número de falsos positivos do projeto *addNode* (Pamvotis) ocorreu porque o sistema não foi projetado para facilitar o uso de testes unitários (o projeto não possui testes unitários escritos pelos desenvolvedores). No futuro, pretendemos realizar ações de refatoração para tornar o código mais testável e, em seguida, repetir o experimento.

Resposta à RQ3: Identificamos com sucesso as três principais características que fazem o NIMROD falhar ao sugerir mutantes equivalentes. Estas incluem o *Modificador de Nível de Acesso, Valor Muito Restrito*, e *Entidades Externas*. No entanto, descobrimos que é possível reduzir significativamente o número de falsos positivos aprimorando a análise estática e ajustando a configuração da geração automática de testes. Dentre as três características, o *Valor Muito*

Restrito é o mais comum e desafiador de abordar. Esse tipo de mutante é notoriamente conhecido como *teimoso* porque requer testes altamente específicos para ser eliminado.

5.4 RQ4. QUAIS OPERADORES DE MUTAÇÃO COMUMENTE LEVAM O NIMROD A FALHAR?

Nesta seção, investigamos a influência dos operadores de mutação no desempenho do NIMROD e determinamos quais deles comumente levaram a classificações incorretas. Para isso, contamos o número de mutantes equivalentes sugeridos por operador que causaram falhas no NIMROD. A Tabela 7 apresenta os resultados para cada operador de mutação, incluindo o número total de mutantes analisados por operador.

Nosso foco está na coluna de falsos positivos, uma vez que representa os mutantes que foram incorretamente classificados como equivalentes pelo NIMROD. Ao analisar os números absolutos, observamos que o operador AOIS (Inserção de Operador Aritmético, atalho) se destaca, gerando 650 (42%) mutantes, o maior entre todos os operadores. Consequentemente, este operador foi responsável por produzir tanto mutantes equivalentes quanto não equivalentes em grandes quantidades. Especificamente, o NIMROD classificou 251 mutantes AOIS como equivalentes, dos quais 125 (50%) foram mal classificados. É importante notar que o operador AOIS é conhecido por gerar inúmeros equivalentes, alguns dos quais poderiam potencialmente ser evitados antes de sua geração (KINTIS; MALEVRIS, 2015; FERNANDES *et al.*, 2017). Além disso, esse operador também gera vários mutantes *teimosos*, que requerem testes específicos para a análise de mutação (YAO; HARMAN; JIA, 2014).

No entanto, ao considerar os números relativos, ou seja, as taxas de falsos positivos para cada operador, descobrimos que o pior desempenho ocorreu com os operadores AOIU (Inserção de Operador Aritmético, unário) e AORB (Substituição de Operador Aritmético, binário), ambos apresentando taxas de falsos positivos próximas a 30%. Notavelmente, mais de 93% dos falsos positivos dos operadores AOIU e AORB se originaram do projeto *addNode* (Pamvotis). A classificação incorreta nesses casos muitas vezes resultou do operador AOIU alterar um campo que pertencia a uma entidade externa ou que tinha um nível de acesso package-private.

Para os demais operadores de mutação (LOI e ASRS), onde o NIMROD também apresentou classificações incorretas, a taxa de acertos na detecção de mutantes não equivalentes (coluna Verdadeiro Negativo) foi superior a 80%, indicando que o NIMROD teve um bom desempenho em distinguir mutantes não equivalentes. Como resultado, não identificamos nenhum operador de mutação específico que estivesse levando intrinsecamente ao fracasso do NIMROD.

Tabela 7 – Resultados do NIMROD por Operador de Mutação.

Operador de Mutação	Descrição	Mutantes	Nimrod		
			Verdadeiro Positivo	Verdadeiro Negativo	Falso Positivo
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**

Fonte: Elaborado pelo autor.

Resposta à RQ4: Em nossa investigação, identificamos dez operadores de mutação do MUJAVA, gerando um total de 1.542 mutantes. Seis operadores, notavelmente AOIU e AORB, produziram falsos positivos, erroneamente marcados como equivalentes pelo NIMROD. O operador AOIS, contribuindo com 42% dos mutantes, destacou-se como o mais prolífico tanto nas categorias equivalentes quanto não equivalentes. Apesar da tendência do AOIS em criar muitos mutantes equivalentes, alguns poderiam ser potencialmente evitados antes da geração, como sugerido por pesquisas anteriores (FERNANDES *et al.*, 2017). Além disso, o AOIS gera mutantes *teimosos* que requerem testes muito específicos. Enquanto AOIU e AORB apresentaram as maiores taxas de falsos positivos, próximas a 30%, a maioria se originou no projeto *addNode* (Pamvotis). Para outros operadores de mutação mal classificados (LOI e ASRS), o NIMROD se destacou na detecção de mutantes não equivalentes, com taxas de acerto superiores a 80%. Nenhum operador de mutação específico levou consistentemente ao fracasso do NIMROD; as classificações incorretas estavam ligadas a características específicas do código, detalhadas na Seção 5.3.

5.5 AMEAÇAS À VALIDADE

Nesta seção, discutimos potenciais ameaças à validade de nosso estudo.

Validade Externa: A seleção de projetos usados como projetos neste estudo pode representar uma ameaça à validade externa, especialmente porque nos concentramos em um número limitado de métodos. Para abordar essa preocupação, buscamos diversificar os projetos esco-

lhendo projetos de diferentes sistemas e domínios. Embora os projetos tenham sido selecionados com base em uma análise anterior de equivalência, nossa abordagem não foi testada em relação a métodos com dependências externas complexas, como aqueles envolvendo objetos como *ObjectC func(ObjectA, ObjectB);*. Métodos com tais dependências podem ser desafiadores para ferramentas de geração de testes, pois exigem a criação de mocks (ARCURI; FRASER; JUST, 2017) ou a descoberta de construtores válidos, que podem ter outras dependências. Além disso, dependências de elementos externos, como interfaces gráficas de usuário ou manipulações de arquivos, poderiam limitar a capacidade das ferramentas de geração de testes de gerar casos de teste que expõem mudanças comportamentais (SOARES *et al.*, 2013).

Validade Interna: A análise manual utilizada para classificar os mutantes representa uma potencial ameaça à validade interna. No entanto, o conjunto de mutantes foi analisado manualmente por dois estudos independentes anteriores (KINTIS *et al.*, 2018; KINTIS; MALEVRIS, 2015), além da nossa análise atual. Além disso, foi utilizado TCE para confirmar alguns dos mutantes equivalentes. Para os restantes, verificamos manualmente todos os equivalentes e corroboramos a análise manual anterior. É importante notar que essa ameaça surge devido à indecidibilidade do problema do mutante equivalente e se aplica a todos os estudos relevantes de teste de mutação sobre este tópico.

O conjunto de operadores de mutação selecionados também introduz ameaças à validade interna deste trabalho. No entanto, este conjunto inclui todos os 15 operadores disponíveis no MUJAVA (versão 3), e não excluímos nenhum mutante de nossa investigação. Não avaliamos operadores de mutação relacionados a orientação a objetos, pois pesquisas anteriores (OFFUTT; MA; KWON, 2006) mostraram que eles geram um número pequeno de mutantes e um número relativamente baixo de equivalentes.

A abordagem de classificação que utilizamos conta o número de casos de teste que alcançam o ponto mutado, que depende das informações de cobertura de linha de código. Essa métrica pode não fornecer informações precisas em todas as situações. Por exemplo, dada a expressão `if (a > 0 && b < 10)`, se a mutação ocorrer no lado direito do operador `&&` (por exemplo, b >= 10), todos os testes que alcançam essa linha, mesmo que apenas avaliem o lado esquerdo da expressão, serão computados como alcançando o ponto mutado. Para mitigar essa ameaça, nossa classificação também considera outra métrica: o impacto na cobertura.

A presença de *testes flakey* (LUO *et al.*, 2014) também poderia representar uma ameaça. Por exemplo, o NIMROD pode sugerir um mutante como não equivalente porque há um teste que expõe uma mudança comportamental no programa mutante, enquanto o mutante é, na verdade, equivalente, e a diferença no comportamento ocorreu devido a um teste flakey. Isso levaria a um falso negativo para o NIMROD. Para minimizar essa ameaça, executamos os testes gerados contra o programa original para confirmar que eles capturam o comportamento atual do programa original. Além disso, tanto o EvoSuite quanto o Randoop possuem configurações para evitar testes flakey. Notavelmente, não identificamos falsos negativos entre os mutantes analisados

manualmente e posteriormente avaliados pelo TCE.

Outras ameaças potenciais poderiam surgir de defeitos no software incorporado, como na análise estática ou nas ferramentas de geração automática de testes. Tais defeitos poderiam impactar nossos resultados. No entanto, acreditamos que a influência de tais defeitos seria mínima em nosso estudo.

Validade de Construção: Todos os nossos resultados são observações empíricas, e eles podem não necessariamente se manter em todos os casos. No entanto, disponibilizamos todos os nossos projetos, ferramentas e dados no site complementar[2] deste trabalho, o que permite que pesquisadores independentes verifiquem, repliquem e analisem nossas descobertas. Essa transparência ajuda a mitigar ameaças à validade de construção.

Em conclusão, embora nosso estudo apresente insights valiosos sobre o desempenho do NIMROD para análise de mutação, é essencial estar ciente das potenciais ameaças à validade mencionadas acima. Essas considerações fornecem uma compreensão mais abrangente do escopo e das limitações de nossas descobertas.

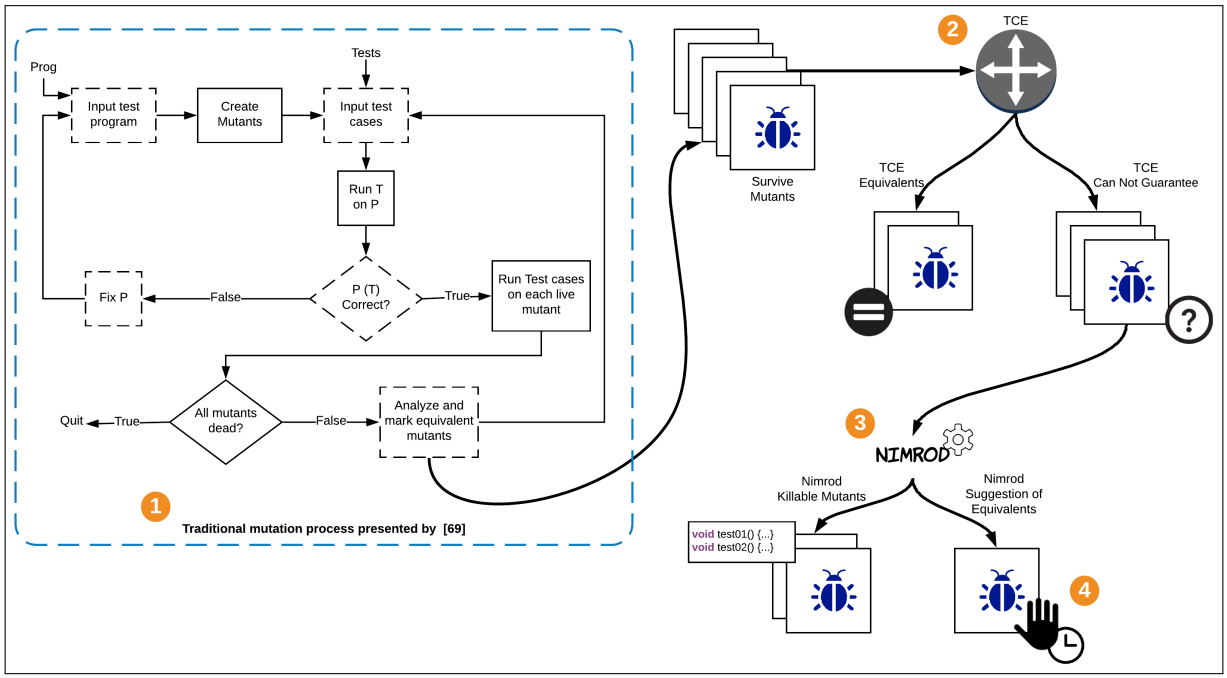
[2]https://github.com/leofernandesmo/nimrod/tree/main

6 ANÁLISE COMPARATIVA

As estratégias para evitar mutantes equivalentes são opções alternativas para reduzir custos e podem ser combinadas com abordagens complementares para detectar e sugerir mutantes equivalentes após a análise de mutação. Assim, TCE e NIMROD podem servir como complementos a essas soluções.

Propomos estender o processo de mutação tradicional incorporando TCE e NIMROD para minimizar o alto custo da análise manual. A Figura 2 apresenta essa extensão. O Passo 1 apresenta a proposta de (OFFUTT; CRAFT, 1994) sobre o processo de mutação comum. As caixas sólidas representam etapas automatizadas por ferramentas como MUJAVA, enquanto as caixas tracejadas representam etapas manuais.

Figura 2 – Combinando TCE e NIMROD para minimizar a análise manual para identificar mutantes equivalentes.



Fonte: Elaborado pelo autor.

No Passo 2, o TCE recebe como entrada um conjunto de mutantes que não foram eliminados pela suíte de testes da aplicação e descarta com segurança um número de mutantes inúteis.

No Passo 3, o NIMROD recebe como entrada os mutantes que o TCE não conseguiu confirmar como equivalentes. Se o NIMROD encontrar um teste que exponha uma mudança de comportamento, indica que o mutante não é equivalente e informa qual teste pode eliminá-lo. Se, após o tempo limite, nenhum teste for capaz de expor uma mudança de comportamento, o NIMROD sugere que o mutante é equivalente. O Passo 4 mostra o processo manual para

identificar mutantes equivalentes. Após utilizar o TCE e o NIMROD, o testador deve considerar agora um número menor de mutantes.

Para ilustrar melhor uma potencial redução de custos, nos referimos à classe `FieldUtils` (projeto Joda-time) apresentada na Seção 3. A Tabela 8 apresenta um resumo da redução de esforço.

Ao executar o MUJAVA com todos os operadores de mutação, 1.339 mutantes são gerados. Após executar a suíte de testes do projeto Joda-time, 543 mutantes foram eliminados e 796 ainda estavam vivos. Isso representa uma taxa de mutação de 40% (sem qualquer análise de equivalência). Usando o processo tradicional, 796 mutantes ainda precisariam ser analisados.

Nosso primeiro passo é considerar os mutantes vivos como entrada para o TCE. O TCE levou 17,33 minutos para analisar os 796 mutantes, com uma média de 1,3 segundos por mutante. O TCE detectou 117 mutantes, reduzindo o número de mutantes a serem investigados para 679. Em seguida, executamos nossa abordagem contra esses 679 mutantes. O NIMROD conseguiu identificar 608 mutantes como não equivalentes (76% de todos os mutantes analisados pelo NIMROD). O tempo total para analisar esses 679 mutantes foi de 24.659 segundos (6,8 horas). No final, o NIMROD sugeriu 71 mutantes como potenciais equivalentes. Em resumo, alcançamos uma redução total de 91,08% no número de mutantes a serem analisados.

Tabela 8 – Redução do esforço ao combinar TCE e NIMROD para a classe `FieldUtils` do projeto Joda-Time.

Mutantes	**Descrição**	
1,339	Total de mutantes no FieldUtils	
-543	Mortos pela suite de testes do joda-time	
796	Mutantes sobreviventes	100.00%
-117	Equivalentes pelo TCE	17.33 minutos
679	Mutantes sobreviventes	⇓ 14.69%
-608	Não equivalentes pelo Nimrod	410.99 minutos
71	Mutantes sobreviventes	⇓ 91.08%

Fonte: Elaborado pelo autor.

Não pretendemos aqui generalizar as reduções de custos de forma alguma. Em vez disso, pretendemos mostrar que nossa abordagem pode potencialmente reduzir custos ao analisar e marcar mutantes equivalentes.

7 CONCLUSÕES

Neste trabalho, introduzimos NIMROD, uma abordagem baseada em testes comportamentais automatizados, para mitigar o impacto de mutantes equivalentes em testes de mutação. Ao sugerir automaticamente mutantes equivalentes e gerar testes para eliminar mutantes não equivalentes, NIMROD reduz o trabalho manual exigido na análise de mutação.

Nossos resultados demonstram que NIMROD alcançou uma alta taxa de sucesso na sugestão de mutantes equivalentes, identificando corretamente 100% dos equivalentes em três dos oito projetos avaliados, e acima de 93% em dois projetos. Apenas em um projeto a performance caiu abaixo de 50%. Além disso, NIMROD reduziu significativamente o tempo necessário para identificar mutantes não equivalentes, com uma média de 36,57 segundos por mutante não equivalente. Para os mutantes sugeridos como equivalentes, onde as ferramentas de geração de testes precisavam executar todos os testes gerados, NIMROD levou uma média de 306,55 segundos (aproximadamente 5,1 minutos). Isso é consideravelmente mais rápido do que os 15 minutos tipicamente exigidos para análise manual. Ao automatizar esse processo, NIMROD economiza tempo e esforço valiosos dos testadores na criação de casos de teste para identificar mutantes não equivalentes.

A escalabilidade da abordagem apresentada depende significativamente das ferramentas de geração de testes automatizadas empregadas na Etapa 2 (Figura 1). Dada a impraticidade de realizar testes exaustivos para programas mais complexos, essas ferramentas utilizam uma configuração de tempo como critério de parada. Uniformizamos a alocação de um tempo fixo para todos os projetos, independentemente da complexidade do método testado, uma prática que não deve ser considerada padrão. Extrair informações do programa sob teste através de análise estática e, em seguida, configurar a ferramenta de geração de testes de acordo com o contexto pode ser uma opção para melhorar o desempenho.

Além disso, investigamos as características dos mutantes que NIMROD classificou incorretamente como equivalentes (falsos positivos). Três classes de características foram identificadas: *Modificador de Nível de Acesso*, *Entidades Externas* e *Valor Muito Restrito*. Melhorias na análise estática e nas ferramentas de geração de testes automáticas podem ajudar a reduzir as instâncias de falsos positivos, especialmente projetando classes testáveis para auxiliar na análise de equivalência do NIMROD. Notavelmente, cerca de 69% dos erros ocorreram devido a mudanças de comportamento nos testes que exigiam acesso a entidades externas ou correspondência de estruturas de pacotes entre os testes e o programa sob teste. Esses problemas podem ser mitigados por meio da evolução das ferramentas de teste automático, empregando análise de impacto entre classes e ajustando geradores de testes para corresponder às estruturas de pacotes e acessar elementos privados de pacote e protegidos.

Além disso, avaliamos o impacto dos operadores de mutação na performance do NIMROD.

Enquanto os operadores AOIU e AORB apresentaram as maiores taxas de falsos positivos, suas taxas de acerto no NIMROD superaram as taxas de erro, indicando que não há um conjunto definitivo de operadores que leva a classificações incorretas. O operador AOIS gerou o maior número de mutantes equivalentes, mas manteve uma alta taxa de acerto no NIMROD. Assim, não identificamos um conjunto específico de operadores responsável pela classificação incorreta.

Trabalhos futuros incluem a realização de novos experimentos para validar estatisticamente as descobertas deste trabalho, utilizando outras ferramentas de mutação, como Major e PIT, bem como diferentes operadores, como operadores de mutação em nível de classe.

Planejamos também incluir projetos com diferentes níveis de complexidade, por exemplo, métodos com dependências externas complexas para aprimorar ainda mais a análise. Novas estruturas de benchmark podem apoiar esse plano.

Como uma melhoria na abordagem, pretendemos incrementar a ferramenta para reduzir o número de falsos positivos para os casos que identificamos, incluindo melhorias na análise de impacto. Na versão atual, temos apenas duas opções na análise de impacto: *intraclasse* e *interclasse*. No caso de intraclasse, direcionamos os testes apenas para as entidades impactadas na própria classe mutada, o que pode ser insuficiente. No caso de interclasse, direcionamos os testes para as entidades impactadas em todo o projeto, o que pode ser muito grande. Talvez um meio-termo traga mais benefícios ao nosso contexto.

Além disso, ao compreender os diferentes casos de mutantes persistentes, podemos orientar melhor os testes automáticos ou até mesmo mudar essas ferramentas para algo como geração de testes baseada em mutantes.

Para resumir, nossa pesquisa propõe o uso do NIMROD como uma solução para abordar mutantes equivalentes em testes de mutação. Ao automatizar a identificação de equivalentes e produzir testes para revelar não equivalentes, o NIMROD oferece assistência valiosa aos testadores de software, reduzindo significativamente a necessidade de trabalho manual e tempo, enquanto garante a eficácia do processo de análise de mutação.

Gostaríamos também de mencionar que esse trabalho obteve apoio do PIBIC IFAL 2023-2024, o qual durante também desenvolvemos uma parceria com um docente da University of Twente e um doutorando da Universidade Federal de Pernambuco.

REFERÊNCIAS

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

ANDREWS, J.; BRIAND, L.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: **ICSE**. [*S.l.: s.n.*], 2005. p. 402–411.

ARCURI, A.; FRASER, G.; JUST, R. Private api access and functional mocking in automated unit test generation. In: **ICST**. [*S.l.: s.n.*], 2017. p. 126–137.

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

BRAIONE, P. *et al.* 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**. [*S.l.: s.n.*], 2017. p. 90–101.

BRITO, C. *et al.* A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants. In: **ICSTW**. [*S.l.: s.n.*], 2020. p. 304–313.

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

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

FERNANDES. **Nimrod - Experimental Pack Repository**. [*S.l.*]: GitHub, 2023. Disponível em: https://github.com/leofernandesmo/nimrod/tree/main. Acesso em: 18 jan. 2024.

FERNANDES, L. *et al.* Avoiding useless mutants. In: **GPCE**. [*S.l.: s.n.*], 2017. p. 187–198.

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: **ESEC/FSE**. [*S.l.: s.n.*], 2011. p. 416–419.

FRASER, G. *et al.* Does automated unit test generation really help software testers? a controlled empirical study. **ACM Transactions on Software Engineering and Methodology**, ACM, v. 24, n. 4, p. 23:1–23:49, 2015.

GOSLING, J. *et al.* **The Java Language Specification**. 2022. Disponível em: https://docs.oracle.com/javase/specs. Acesso em: 12 jul. 2022.

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

JUST, R. *et al.* Are mutants a valid substitute for real faults in software testing? In: **ESEC/FSE**. [*S.l.: s.n.*], 2014. p. 654–665.

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

KINTIS, M. *et al.* Detecting trivial mutant equivalences via compiler optimisations. **TSE**, v. 44, n. 4, p. 308–333, 2018.

LAKHOTIA, K.; MCMINN, P.; HARMAN, M. Automated test data generation for coverage: Haven't we solved this problem yet? In: **Testing: Academic and Industrial Conference-Practice and Research Techniques**. [*S.l.: s.n.*], 2009. p. 95–104.

LI, S. *et al.* Measuring code behavioral similarity for programming and software engineering education. In: **Proceedings of the 38th International Conference on Software Engineering Companion**. [*S.l.: s.n.*], 2016. p. 501–510. ISBN 978-1-4503-4205-6.

LUO, Q. *et al.* An empirical analysis of flaky tests. In: **ESEC/FSE**. [*S.l.: s.n.*], 2014. p. 643–653.

MADEYSKI, L. *et al.* Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. **TSE**, v. 40, n. 1, p. 23–42, 2014.

MONGIOVI, M. *et al.* Making refactoring safer through impact analysis. **SCP**, Elsevier, v. 93, p. 39–64, 2014.

MONGIOVI, M. *et al.* Detecting Overly Strong Preconditions in Refactoring Engines. **TSE**, v. 44, n. 5, p. 429–452, 2018.

NAEEM, M. R. *et al.* A machine learning approach for classification of equivalent mutants. **Journal of Software: Evolution and Process**, v. 32, n. 5, 2020.

OFFUTT, J.; CRAFT, M. Using compiler optimization techniques to detect equivalent mutants. **STVR**, Wiley Online Library, v. 4, n. 3, p. 131–154, 1994.

OFFUTT, J.; MA, Y.-S.; KWON, Y.-R. The class-level mutants of mujava. In: **AST**. [*S.l.: s.n.*], 2006. p. 78–84.

PACHECO, C. *et al.* Feedback-directed random test generation. In: **ICSE**. [*S.l.: s.n.],* 2007. p. 75–84.

PAPADAKIS, M. *et al.* Mutation testing advances: an analysis and survey. In: **Advances in Computers**. [*S.l.*]: Elsevier, 2019. v. 112, p. 275–378.

PAPADAKIS, M. *et al.* Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In: **ICSE**. [*S.l.: s.n.*], 2018.
p. 537–548.

PEACOCK, S. *et al.* Automatic equivalent mutants classification using abstract syntax tree neural networks. In: **ICSTW**. [*S.l.: s.n.*], 2021. p. 13–18.

PIZZOLETO, A. V. *et al.* A systematic literature review of techniques and metrics to reduce the cost of mutation testing. **JSS**, v. 157, 2019.

SCHULER, D.; DALLMEIER, V.; ZELLER, A. Efficient mutation testing by checking invariant violations. In: **ISSTA**. [*S.l.: s.n.*], 2009. p. 69–80.

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

SHAMSHIRI, S. *et al.* 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)**. [*S.l.: s.n.*], 2015. p. 201–211.

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

SOARES, G. *et al.* Comparing approaches to analyze refactoring activity on software repositories. **JSS**, v. 86, n. 4, p. 1006–1022, 2013.

SOARES, G. *et al.* Making program refactoring safer. **IEEE software**, v. 27, n. 4, p. 52–57, 2010.

STEIMANN, F.; THIES, A. From behaviour preservation to behaviour modification: Constraint -based mutant generation. In: **ICSE**. [*S.l.: s.n.*], 2010. p. 425–434.

VOAS, J.; MCGRAW, G. **Software fault injection**: inoculating programs against errors. [*S.l.*]: John Wiley & Sons, Inc., 1997.

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