UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Uma abordagem para localização de mutantes minimais baseada na estrutura do código fonte

Claudinei Brito Junior

Qualificação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Claudinei Brito Junior

Uma abordagem para localização de mutantes minimais baseada na estrutura do código fonte

Monografia apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, para o Exame de Qualificação, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional.

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
Março de 2019

Claudinei Brito Junior

An approach to minimal mutants localization based on source code structure

Monograph submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, as part of the qualifying exam requisites of the of the Master Program in Computer Science and Computational Mathematics.

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
March 2019

RESUMO

BRITO JUNIOR, C. **Uma abordagem para localização de mutantes minimais baseada na estrutura do código fonte**. 2019. 83 p. Monografia (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Com o objetivo de garantir que um software esteja sendo construído de acordo com suas especificações estabelecidas, a Engenharia de Software dispõe de uma série de atividades, processos e métodos que coletivamente são chamados de Validação, Verificação e Teste. O processo de teste de software pode ser definido como a execução de um programa com determinadas entradas e verificação da consonância das saídas com as saídas esperadas de acordo com as especificações do programa. Na tentativa de cumprir com seu objetivo, o teste de software pode ser dividido em diversas fases e etapas e conta com diversas técnicas e critérios, no qual cada técnica tem um objetivo específico e todas juntas se complementam. Um dos problemas fundamentais do teste de software é saber na prática ou na teoria quando já se testou o suficiente de um programa. O Teste de Mutação é um critério da técnica de Teste Baseado em Defeitos. É reconhecidamente um critério que pode auxiliar na criação de casos de teste com alta efetividade e capacidade de relevar defeitos. Esse critério mede a adequação de um determinado conjunto de casos de teste e assim fornece uma medida qualitativa da assertividade de um conjunto de teste com relação a um programa. A aplicação do teste de mutação se dá por meio de alterações no programa original, a fim de observar se o caso de teste consegue distinguir o comportamento do programa original e do programa alterado. Cada alteração realizada no programa original cria um novo programa chamado de Programa Mutante. Um dos problemas da aplicação do teste de mutação se dá pois normalmente são gerados muitos programas mutantes, mesmo para pequenos programas, o que eleva muito o custo computacional de geração e análise desses programas mutantes. Sendo assim, existem diversas técnicas que visam reduzir o custo computacional do teste de mutação. Essas técnicas normalmente são divididas em: (i) *Do Fewer*; (ii) *Do Smarter*; e (iii) *Do Faster*. Mutante minimal é um conceito pertencente às técnicas *Do Fewer* que procuram meios para reduzir o número de mutantes gerados. Este projeto tem como objetivo propor uma abordagem para identificação de mutantes minimais com base em suas localizações na estrutura do código-fonte e assim evitar a geração e execução de todos os mutantes. Para o desenvolvimento da abordagem, pretende-se utilizar uma representação de programas utilizadas na técnica de teste estrutural, o Grafo de Fluxo de Controle, e assim relacionar os mutantes minimais com lugares específicos do código, como por exemplo, nós essenciais do grafo de fluxo de controle. Uma vez desenvolvida a abordagem, pretende-se avaliá-la por meio de estudos empíricos com métricas já utilizadas na literatura em estudos prévios que visam reduzir o custo do teste de mutação. Como resultado final deste projeto, espera-se avançar o estado da arte na área de teste de software com uma abordagem mais eficiente para aplicação do critério teste de mutação.

Palavras-chave: Teste de Software, Teste de Mutação, Mutantes Minimais.

ABSTRACT

BRITO JUNIOR, C. **An approach to minimal mutants localization based on source code structure**. 2019. 83 p. Monografia (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Aiming ensure that a software has been build in according with established specifications, Software Enginerring has a serie of activities, processes and methods that collectively are called Validation, Verification and Testing. The software testing process can be defined as the program execution with determined inputs and verification of outputs consonances with expected outputs according to program specification. Trying reach his object, software testing can be divided into several phases and steps and has several techniques and criterias, such as each technique has a specified aim and all together complement each other. One of software testing fundamental problem is to know in the practice and theory when have already tested sufficient. Mutation Testing is a criterion of Defect-Based Testing technique. It is recognized as a criterion that can aid in the creation of test cases with high effectiveness and ability to reveal defects. This criterion measures the adequacy of a given set of test cases and thus provides a qualitative measure of the assertiveness of a test set in relation to a program. The application of the mutation testing is done through changes in the original program, in order to observe if the test case can distinguish the behavior of the original program and the altered program. Each change made in the original program creates a new program called Mutant Program. One of the problems of applying the mutation test is that many mutant programs are generated, even for small programs, which greatly increases the computational cost of generating and analyzing these mutant programs. Thus, there are several techniques that aim to reduce the computational cost of the mutation test. These techniques are usually divided into: (i) Do Fewer; (ii) Do Smarter; and (iii) Faster. Minimal mutant is a concept belonging to Do Fewer techniques that seek means to reduce the number of mutants generated. This project aims to propose an approach to identify minimal mutants based on their locations in the source code structure and thus avoid the generation and execution of all mutants. For the development of the approach, we intend to use a representation of programs used in the structural test technique, the Control Flow Graph, and thus to relate the minimal mutants to specific places of the code, such as essential nodes of the control flow graph. Once the approach is developed, it is intended to be evaluated through empirical studies with metrics already used in the literature in previous studies aimed at reducing the cost of mutation testing. As a final result of this project, it is expected to advance the state of the art in the area of software testing with a more efficient approach to applying the mutation testing criterion.

Keywords: Software Testing, Mutation Testing, Minimal Mutants.

LISTA DE ILUSTRAÇÕES

Figura 1 – Notação dos Grafos Causa-Efeito 35
Figura 2 – GFC correspondente ao Source code 1 37
Figura 3 – Aplicação dos operadores de mutaçao AOI e ROR 40
Figura 4 – Programa original e mutante equivalente 41
Figura 5 – Processo genérico do teste de mutação 42
Figura 6 – Exemplo de árvore sintática 53
Figura 7 – Exemplo de grafo de dominância 59

LISTA DE QUADROS

Quadro 1 – Categorias de mutação seletiva - WONG *et al.* (1997) 50
Quadro 2 – Categorias de mutação seletiva - Barbosa, Vincenzi e Maldonado (1998) . . 51
Quadro 3 – Categorias de mutação seletiva - Zhang *et al.* (2013) 53
Quadro 4 – Disciplinas cursadas no programa de mestrado 65
Quadro 5 – Cronograma de atividades do projeto de mestrado 67
Quadro 6 – Operadores de mutação da linguagem Fortran-77 81
Quadro 7 – Operadores de mutação da linguagem C 82
Quadro 8 – Operadores de mutação da linguagem Java 83

LISTA DE ALGORITMOS

Algoritmo 1 – Minimização do conjunto de testes 60
Algoritmo 2 – Minimização do conjunto de mutantes 61

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Subrotina para retornar a situação do aluno com base em sua nota . . 36
Código-fonte 2 – Subrotina para calcular o melhor combustível de acordo com os preços 40

LISTA DE TABELAS

Tabela 1 – Ferramentas que dão suporte à aplicação do teste de mutação 42
Tabela 2 – Visão geral dos resultados obtidos por cada autor 52
Tabela 3 – Escore de mutação de exemplo 57
Tabela 4 – Escore de mutação de exemplo - Mutantes minimais 59
Tabela 5 – Escore de mutação vs Escore de dominância 60

LISTA DE ABREVIATURAS E SIGLAS

ASA	Árvore Sintática Abstrata
ES	Engenharia de Software
GFC	Grafo de Fluxo de Controle
ICMC	Instituto de Ciências Matemáticas e de Computação
USP	Universidade de São Paulo
VV&T	Validação, Verificação e Teste

SUMÁRIO

1 INTRODUÇÃO 23
1.1 Contextualização e Motivação 23
1.2 Objetivos 25
1.2.1 Objetivos gerais *26*
1.2.2 Objetivos específicos *26*
1.3 Organização do trabalho 26

2 FUNDAMENTOS SOBRE TESTE DE SOFTWARE 29
2.1 Fundamentos de Teste de Software 29
2.2 Fases da Atividade de Teste 31
2.3 Técnicas e Critérios de Teste 32
2.3.1 Teste Funcional *33*
2.3.1.1 Particionamento em Classes de Equivalência *33*
2.3.1.2 Análise do valor limite *34*
2.3.1.3 Grafo Causa-Efeito *34*
2.3.1.4 Error guessing *34*
2.3.2 Teste Estrutural *35*
2.3.2.1 Critérios baseados na complexidade *37*
2.3.2.2 Critérios baseados em fluxo de controle *37*
2.3.2.3 Critérios baseados em fluxo de dados *38*
2.3.3 Teste Baseado em Defeitos *38*
2.3.3.1 Ferramentas *42*
2.4 Considerações Finais 46

3 REDUÇÃO DO CUSTO DO TESTE DE MUTAÇÃO 47
3.1 O problema de custo no teste de mutação 47
3.2 Redução do número de mutantes (*do fewer*) 48
3.3 Outras abordagens 53
3.3.1 Árvore sintática abstrata *53*
3.3.2 Identificação de mutantes inúteis *53*
3.3.3 Mutantes de segunda ordem *54*
3.3.4 Mutantes teimosos *54*
3.4 Considerações finais 54

4 MUTANTES MINIMAIS 57
4.1 Definições teóricas e resultados preliminares 57
4.1.1 Conjunto de testes minimais *58*
4.1.2 Mutantes redundantes *58*
4.1.3 Conjunto de mutantes minimais *58*
4.1.4 Grafo de Dominância *58*
4.1.5 Escore de dominância *59*
4.2 Identificação de mutantes minimais 60
4.3 Considerações finais 61

5 PROPOSTA 63
5.1 Contexto do Projeto 63
5.2 Objetivos do Projeto 64
5.3 Metodologia 65
5.4 Plano de trabalho e Cronograma 66
5.5 Resultados Esperados 67
5.6 Considerações Finais 68

REFERÊNCIAS 69

APÊNDICE A OPERADORES DE MUTAÇÃO 81

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização e Motivação

A utilização de softwares no cotidiano de pessoas, empresas e indústrias, faz com que o rigor aplicado no desenvolvimento de software seja cada vez maior. O processo de desenvolvimento de software envolve uma série de etapas, normalmente com sujeitos de diversas áreas de conhecimento e muitas das vezes, boa parte do trabalho é feita de forma remota. São inúmeros os fatores que elevam a complexidade do desenvolvimento de software, portanto, a incidência de defeitos acaba se tornando muito propensa. Dessa forma, é necessário que existam processos, métodos e ferramentas que busquem garantir a qualidade do software. A Engenharia de Software (ES) é uma área do conhecimento do contexto de Ciência da Computação que envolve diversos processos, métodos e ferramentas, que visam garantir a qualidade de software (PRESSMAN, 2016).

De forma a tentar garantir que os artefatos e produtos que sejam desenvolvidos em conformidade com as especificações estabelecidas, a ES dispõe de uma série de atividades que coletivamente são chamadas de Validação, Verificação e Teste (VV&T). Bertolino (2003) define que o processo de teste de software é executar o programa com determinadas entradas e então, verificar se o comportamento do programa condiz com as especificações. Delamaro, Jino e Maldonado (2017) destacam que a atividade de teste consiste de uma análise dinâmica do produto em construção, tornando-se assim, uma atividade indispensável para a identificação de possíveis defeitos que podem ter sido cometidos durante todo o processo de desenvolvimento de software, desde a especificação de requisitos até a implantação. O objetivo geral do teste de software é fornecer evidências sobre a confiabilidade do produto avaliado (MYERS; SANDLER; BADGETT, 2011). Assim sendo, existem diversas técnicas e critérios que são aplicados em conjunto e complementando-se um ao outro, de forma a garantir uma maior completude de verificação do software, vasculhando todo o produto e garantindo que todo o software foi avaliado e que todos os possíveis defeitos foram reportados.

Além das técnicas e critérios do teste de software, a sistematização do teste também é composta por fases e etapas. As etapas são separadas em:

- Planejamento;
- Projeto de casos de teste;
- Execução; e
- Análise.

Independente da técnica, critério ou fase do teste, essas etapas são cruciais para que o teste seja o mais efetivo possível. As fases do teste de software são:

- Teste de Unidade;
- Teste de Integração;
- Teste de Sistemas;
- Teste de Regressão.

O ciclo de teste de um software é a repetição cíclica e coordenada dessas fases, onde cada uma das fases tem um objetivo específico e visa encontrar um tipo de defeito.

Para cada uma das técnicas de teste, a origem da informação utilizada para a derivação dos requisitos de teste é diferente (DELAMARO; MALDONADO; JINO, 2017). O Teste Funcional é uma técnica onde os casos de teste são projetados a partir da consideração que o programa é considerado como uma caixa preta (FABBRI; VINCENZI; MALDONADO, 2017), isto é, não é necessário o conhecimento da estrutura interna do programa para derivação dos casos de teste e avaliação. Os critérios de teste da técnica funcional são:

- Particionamento em Classes de Equivalência;
- Análise do Valor Limite;
- Grafo Causa-Efeito; e
- *Error Guessing*.

O Teste Estrutural estabelece os requisitos de teste com base na estrutura interna do programa, como se fosse uma caixa branca (BARBOSA *et al.*, 2017). Nesta técnica, são testados os caminhos lógicos do programa, pondo em prova conjuntos determinados de laços de repetição e estruturas condicionais. Os critérios de teste da técnica estrutural são:

- Critérios baseados na complexidade;

- Critérios baseados em fluxo de controle: Os principais critérios são Todos-nós, Todas-Arestas e Todos-Caminhos; e
- Critérios baseados em fluxo de dados: Os principais critérios são Rapps e Weyuker e Potenciais-usos.

O Teste Baseado em Defeitos deriva os requisitos de teste a partir de defeitos típicos que são cometidos durante o processo de desenvolvimento de software (DELAMARO *et al.*, 2017). O principal critério do teste baseado em defeitos é o Teste de Mutação. O teste baseado em defeitos visa medir a adequação de um conjunto de casos de teste (DEMILLO; OFFUTT, 1991). O teste de mutação se dá por meio da criação de programas mutantes que são criados com base no programa original, e são realizadas alterações sintáticas no programa, de forma a alterar sua semântica. Portanto, conforme Madeyski *et al.* (2014), os casos de teste que conseguem distinguir o programa original do programa mutante, têm maior probabilidade de revelar defeitos, assim, pode ser considerado mais adequado que outro.

As principais vantagens da aplicação do teste de mutação, segundo Chekam *et al.* (2017) são: (i) apontamento dos elementos que devem ser testados no momento do desenvolvimento dos casos de teste; (ii) fornecimento de critérios para determinação do término do teste; e (iii) quantificação da adequação e qualidade do conjunto de casos de teste. O segundo problema mencionado soluciona um dos problemas fundamentais do teste de software, que é saber na teoria e na prática quando parar de testar (PAPADAKIS *et al.*, 2019).

Ainda que conforme Chekam *et al.* (2017), o teste de mutação tenha muitas vantagens, Wong e Mathur (1995) mencionam que às vezes o teste de mutação se torna impraticável devido ao alto custo da aplicação da técnica. O alto custo computacional do teste de mutação se dá pois normalmente são gerados muitos programas mutantes, e então se torna custoso executar todos esses programas mutantes. O alto custo de recursos humanos do teste de mutação se dá pois são gerados muitos mutantes equivalentes, que são mutantes diferentes do programa original, porém, não alteram o comportamento do programa, dessa forma, os casos de teste não conseguem distinguir o mutante equivalente do programa original, e então é necessária uma intervenção humana.

1.2 Objetivos

Existem várias técnicas e abordagens que visam reduzir o custo do teste de mutação, porém, ainda não há um consenso e muito menos uma definição de uma abordagem ou técnica que efetivamente reduza o custo do teste de mutação e mantenha sua efetividade. Offutt e Untch (2001) definem que as técnicas e abordagens para redução do custo do teste de mutação se restringem à uma das três estratégias mencionadas:

- *Do Fewer*: Procura meios para reduzir o número de mutantes gerados;

- *Do Smarter*: Procura distribuir o custo computacional entre várias máquinas, dividir o custo computacional em diversas execuções, ou, evitar a execução completa de uma só vez; e
- *Do faster*: Procura focar em meios de geração e execução de cada mutante o mais rápido possível.

1.2.1 Objetivos gerais

Existem diversas técnicas que utilizam a estratégia *do fewer* que foram aplicadas para tentar reduzir o custo do teste de mutação (WONG; MATHUR, 1995; MATHUR; WONG, 1993; OFFUTT; ROTHERMEL; ZAPF, 1993; OFFUTT *et al.*, 1996; WONG *et al.*, 1997; BARBOSA; VINCENZI; MALDONADO, 1998).

Just, Kurtz e Ammann (2017) definem que um conjunto de mutantes dominantes (ou minimais) é um subconjunto mínimo do conjunto total de mutantes, tal que qualquer conjunto de testes que seja adequado para o conjunto de mutantes dominantes, também é adequado para o conjunto total de mutantes. Este trabalho tem como objetivo propor e validar uma abordagem capaz de identificar os mutantes minimais com base na estrutura do código fonte, tendo como base a análise da técnica de teste estrutural e seus critérios.

1.2.2 Objetivos específicos

A partir do objetivo geral, foram derivados os seguintes objetivos específicos:

1. Proposta de uma abordagem que permita a detecção dos mutantes pertencentes ao conjunto de mutantes minimais com base na localização desses mutantes na estrutura do código fonte;
2. Avaliação da abordagem proposta por meio de estudos empíricos. Os estudos empíricos serão realizados na linguagem *Java*.
    - O estudo realizado na linguagem *Java* utilizará os mesmos programas utilizados por Deng, Offutt e Li (2013), que são diversos programas retirados de livros e projetos *open-source*. A ferramenta utilizada para geração e análise dos mutantes é a *muJava* (MA; OFFUTT; KWON, 2006).

1.3 Organização do trabalho

Este trabalho está organizado em cinco capítulos. Este capítulo introduziu a área de pesquisa investigada apresentando a problemática do trabalho e a motivação necessária para seu desenvolvimento. O Capítulo 2 apresenta os principais conceitos relacionados a Teste de

software. No Capítulo 3 são descritos os principais conceitos referentes à problemática do teste de mutação, bem como estratégias de redução do custo do teste de mutação. No Capítulo 4 são apresentados os conceitos referentes à mutantes minimais, bem como conceitos relacionados e métodos para sua identificação. Por fim, no Capítulo 5 é apresentada a proposta deste trabalho, abordando os objetivos, metodologia, plano de trabalho e cronograma de execução das atividades. Ainda no último capítulo são apresentados os resultados esperados. Por fim, são listadas as referências bibliográficas utilizadas ao longo desta proposta e o glossário.

CAPÍTULO

2

FUNDAMENTOS SOBRE TESTE DE SOFTWARE

Neste capítulo são apresentados os conceitos fundamentais sobre Teste de Software que são necessários para a completa compreensão deste projeto de mestrado. Portanto, serão abordados os Fundamentos de Teste de Software, as Fases da Atividade de Teste, as Técnicas e Critérios de Teste e o processo de Automatização de Teste de Software.

2.1 Fundamentos de Teste de Software

A Engenharia de Software compreende processos, métodos e ferramentas que permitem, possibilitam, facilitam e agilizam a construção de sistemas complexos baseados em computador, dentro do prazo, do custo estimado e com qualidade. Portanto, a base que sustenta a ES é o foco na qualidade, isto é, a ES como um todo, visa garantir a qualidade em todas as etapas do desenvolvimento de software (PRESSMAN, 2016).

Objetivando a garantia de que os métodos e ferramentas utilizadas para o desenvolvimento do software e também o software como produto final estejam em conformidade com as especificações desejadas, a ES dispõe de uma série de atividades que coletivamente são chamadas de Validação, Verificação e Teste. Essas atividades tem o intuito de encontrar as falhas que podem ter sido introduzidas durante o processo de desenvolvimento de software e garantir que não estejam na versão final do produto ou nas versões liberadas para utilização (DELAMARO; JINO; MALDONADO, 2017).

A Validação consiste em garantir que o software que está sendo desenvolvido está em conformidade com as necessidades do usuário, não fugindo do objetivo proposto, portanto, esta etapa verifica se está sendo construído um software que será útil para o usuário. A Verificação, por sua vez, é responsável por assegurar a corretude, consistência e completude do produto que está em construção, durante as fases e etapas do desenvolvimento do projeto de software. Por

último, o Teste é o processo que consiste em várias atividades, que em resumo, executam o software com o objetivo de encontrar o maior número de falhas possíveis no programa, visando que o usuário final não tenha que lidar com estas falhas pois elas já foram identificados, e, portanto, corrigidas.

O objetivo da VV&T é estabelecer confiança e tentar garantir que o sistema de software esteja adequado a seu propósito (SOMMERVILLE, 2011). O Teste de Software pode ser compreendido como um processo ou um conjunto de processos, projetados para garantir que o software faça aquilo que ele foi designado a fazer e não tenha nenhum comportamento inesperado durante sua execução (MYERS; SANDLER; BADGETT, 2011).

As definições e conceitos de alguns termos relacionados à VV&T que são utilizados neste trabalho seguem o que foi definido no padrão IEEE 610.12 (IEEE..., 1990):

- **Defeito** (*Fault*): Um passo, processo ou uma definição de dados incorretos em um programa de computador;
- **Engano** (*Mistake*): Uma ação humana que produz um resultado incorreto;
- **Erro** (*Error*): A diferença entre o valor obtido e o esperado, isto é, algum comportamento inesperado do programa de computador;
- **Falha** (*Failure*): A incapacidade do sistema de produzir resultados que condizem com suas especificações.

De acordo com Pressman (2016), o teste é um conjunto de atividades que pode ser planejado antecipadamente e executado de forma sistemática. O mesmo autor ainda lista algumas diretrizes propostas na literatura que podem ser seguidas para uma efetiva atividade de teste de software:

- Para executar um teste eficaz, devem ser conduzidas eficazes revisões técnicas, desta forma, alguns erros poderão ser eliminados antes do início da atividade de teste;
- A atividade de teste deve começar no teste de componente e deve crescer de dentro para fora, partindo do teste de unidade até o teste de integração[1];
- Diferentes técnicas de teste são apropriadas para diferentes abordagens de ES e em diferentes momentos;
- O teste deve ser conduzido pelos desenvolvedores do software, ou então em projetos maiores, por equipes de teste independentes;
- Teste e depuração[2] são atividades distintas, mas a depuração deve ser integrada em qualquer estratégia de teste.

[1] As fases da atividade de teste são explicadas detalhadamente na Seção 2.2
[2] Detectar, localizar e corrigir falhas em um programa de computador.

2.2 Fases da Atividade de Teste

A atividade de VV&T deve ocorrer durante todo o ciclo de desenvolvimento do software e inclusive, após o desenvolvimento, durante a manutenção do software. Conforme Delamaro, Jino e Maldonado (2017), existem fases e etapas para o teste de software. As fases da atividade de teste de software são geralmente separadas em:

- Teste de unidade;
- Teste de integração;
- Teste de sistemas;
- Teste de regressão.

É necessário compreender que cada fase da atividade de teste tem um objetivo específico (PRESSMAN, 2016). O teste de unidade tem como ponto central as unidades mínimas de um programa (funções, procedimentos, métodos ou classes), portanto, objetiva-se encontrar erros na escolha ou na implementação de algoritmos, estruturas de dados mal formadas ou utilizadas indevidamente, enganos cometidos na entrada ou saída de métodos ou até então simples erros sintáticos ou semânticos. Por se tratar de uma fase da atividade de teste que testa pequenas partes independentes do programa, o teste de unidade pode ser realizado à medida que o software é construído ou até antes da construção do software, de forma que o teste já haverá sido criado quando a implementação for finalizada (DELAMARO; JINO; MALDONADO, 2017).

Finalizado o teste de unidade, o teste de integração concentra-se na construção arquitetural do sistema. Uma vez que as unidades (métodos, funções, classes e camadas) foram construídas, tais começam a interagir entre si e é possível que internamente estejam funcionando corretamente sem nenhum comportamento inesperado, porém, no momento da conexão entre essas unidades é possível que ocorram comportamento inesperados e erros sejam revelados. O responsável por esta fase da atividade de teste deve conhecer o sistema como um todo, conhecendo toda a organização interna, bem como as conexões entre si, portanto, comumente é a própria equipe de desenvolvimento responsável pela projeção e execução destes testes.

Após ter verificado que as unidades funcionam de forma independente e quando se conectam o software continua funcionando como deveria, deve ser executado o teste de sistemas, que pode ser considerado como o teste que é o mais próximo daquilo que será a execução do software por parte do usuário, uma vez que o software como um todo é avaliado nesta fase. Neste momento, são confrontadas as especificações e requisitos do sistema com o software construído, e é também neste momento que são avaliados os requisitos não funcionais, como usabilidade, segurança, performance, entre outros.

O ciclo normal de teste de software se encerra no teste de sistema, que é quando o sistema é testado de forma completa, porém, ainda existe uma outra fase do teste, que é o

teste de regressão. O teste de regressão é executado sempre durante a manutenção do software, normalmente quando novas implementações são realizadas no software, e então, o teste de regressão tem como objetivo identificar possíveis falhas que podem ter sido introduzidas naquilo que até então era estável.

Para que seja possível encontrar o maior número de erros possíveis e entregar um software mais confiável, todas as fases da atividade de teste devem compreender as seguintes etapas:

1. Planejamento;
2. Projeto de casos de teste;
3. Execução;
4. Análise.

A depender do resultado da última etapa que é a análise, é passível que aconteça mais um ciclo com as 4 etapas, até que se alcance o resultado desejado.

2.3 Técnicas e Critérios de Teste

Afirmar que um software está funcionamento corretamente sem nenhum defeito, é impossível, porém, afirmar que um software não está funcionamento corretamente e tem defeitos é possível através da atividade de teste. O objetivo da atividade de teste é revelar defeitos contidos num software para que ele possa então ser corrigido e assim, um software mais confiável será construído, desse modo, um teste bem-sucedido é um teste que revele defeitos. Um caso de teste[3] classificado como bom, é aquele que tem alta probabilidade de revelar defeitos até então não descobertos.

Segundo Myers, Sandler e Badgett (2011), a atividade de teste de software e atividades relacionadas, consomem aproximadamente 50% do tempo de desenvolvimento e mais de 50% do custo de desenvolvimento de software, e uma das atividades que mais demandam tempo dentro da atividade de teste de software, é desenvolver e escolher os casos de teste mais adequados para o programa e que tenha maior probabilidade de revelar defeitos. Ainda que o tempo despendido nessa tarefa seja alto, ainda não existe a garantia de que os casos de teste sejam bons ou que o software esteja num nível confiável de corretude, da mesma forma, é possível garantir que um conjunto de casos de teste tenha testado todos os arcos e todos nós, mas não é possível garantir que este mesmo conjunto de casos de teste seja confiável.

Por se tratar de uma atividade indispensável durante o processo de desenvolvimento de software, assim como qualquer outra atividade, é necessário que o teste de software tenha

[3] Combinação de entrada de dados no programa e saídas correspondes àquelas entradas

o máximo de eficácia e performance e o mínimo de custo. Portanto, é necessário que sejam definidas abordagens sistemáticas e fundamentadas teoricamente e métricas de avaliação para esta atividade. Critérios de teste podem ser utilizados tanto para auxiliar na geração de conjuntos de casos de teste, quanto para auxiliar na avaliação da adequação desses conjuntos (PRESSMAN, 2016).

O que diferencia cada uma das técnicas de teste, é a origem da informação utilizada para que sejam derivados os requisitos de teste (DELAMARO; JINO; MALDONADO, 2017). A seguir são apresentadas três técnicas de teste de software, bem como suas características e seus respectivos critérios.

2.3.1 Teste Funcional

O teste funcional, muitas vezes chamado de teste de caixa preta, é uma técnica de teste que visa avaliar o software a partir do ponto de vista do usuário, isto é, avalia o software a partir de uma visão externa, sem conhecimento algum da estrutura interna do software, bem como detalhes de implementação, isto é, é similar à uma caixa preta onde nada se vê, justificando assim a alcunha atribuída à técnica. Teoricamente, o teste funcional possibilita identificar todos os possíveis defeitos que estejam no software, através do teste exaustivo (MYERS; SANDLER; BADGETT, 2011), que é a exposição do software à todas as possíveis entradas. Porém, para softwares reais, isso se torna impraticável, uma vez que podem existir domínios de entrada gigantescos ou até infinitos, fazendo com que a atividade de teste se torne altamente custosa em termos de tempo empregado e recursos (humanos ou computacionais) usados.

Uma vez que se compreende que é impraticável avaliar todo o domínio de entrada, é necessária a adoção de outras estratégias que possibilitem encontrar o maior número de erros possíveis num software, essas estratégias são critérios que se encaixam dentro do teste funcional. Abaixo são descritos alguns critérios da técnica funcional que são utilizados para revelar defeitos num software.

2.3.1.1 Particionamento em Classes de Equivalência

Sabendo que testar todas as entradas do domínio de entrada é não factível na maioria dos casos, é necessário selecionar um subconjunto de entradas que revelem o maior número de erros possível. O particionamento em classes de equivalência seleciona e agrupa as entradas que de acordo com as especificações do programa, tem comportamento iguais.

Após a separação em classes, é possível afirmar seguramente que cada item dentro da classe pode refletir qualquer outro item da mesma classe, pois eles têm o mesmo comportamento de acordo com as especificações do programa. É importante essa separação, pois quando um item daquela classe revelar um defeito, é sabido que qualquer outro item da mesma classe revelaria o mesmo defeito. Desta maneira, o domínio de entrada é diminuído, fazendo com que seja possível

testar todas (ou uma maior parte) as possibilidades do domínio de entrada.

2.3.1.2 Análise do valor limite

Para ser utilizado em conjunto com o Particionamento em Classes de Equivalência, o critério Análise do Valor Limite busca explorar os valores que estão exatamente sobre ou imediatamente subsequentes às fronteiras das classes de equivalência. Myers, Sandler e Badgett (2011) destacam que a experiência mostra que os casos de teste que analisam condições limites têm maior probabilidade de revelar defeitos. Portanto, o que diferencia o critério anterior deste, é que a análise do valor limite não escolhe aleatoriamente os dados de teste, mas são escolhidos aqueles que explorem o limitante de cada classe de equivalência.

2.3.1.3 Grafo Causa-Efeito

O grafo Causa-Efeito, diferentemente dos outros critérios, explora as combinações dos dados de entrada. Este critério atua na forma de uma linguagem formal, que é o grafo, dessa forma, facilita na identificação de ambiguidades, incompletudes e até inconsistências na especificação do programa. A construção do grafo é separada em causa que corresponde às entradas e efeito que corresponde às saídas esperadas do programa.

Para a construção do grafo, cada nó deve assumir os valores 0 ou 1, 0 indicando ausência do estado, enquanto o 1 representa a presença do estado. A notação utilizada é composta de operadores que são descritos a seguir e depois ilustrados na Figura 1.

- Função identidade: se nó “1” é 1, então, nó “2” é 1. Se nó “1” é 0, então, nó “2” é 0;
- Função *not*: Se nó “1” é 1, então, nó “2” é 0. Se nó “1” é 0, então, nó “2” é 1;
- Função *or*: Se nó “1” ou nó “2” são 1, então, nó “3” é 1. Se nem nó “1”, nem nó “2” são 1, então, nó “3” é 0.
- Função *and*: Se nó “1” e nó “2” são 1, então, nó “3” é 1. Se nó “1” ou nó “2” são 0, então, nó “3” é 0.

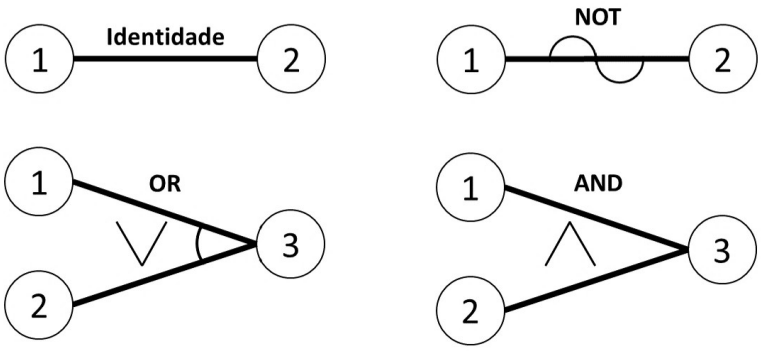
Existem ainda outras notações complementares e definições para a utilização deste critério que não serão abordadas neste trabalho. Para uma discussão mais aprofundada, recomenda-se consultar as referências Delamaro, Jino e Maldonado (2017) e Myers, Sandler e Badgett (2011).

2.3.1.4 Error guessing

Este critério de teste pode ser considerado como uma abordagem *ad-hoc*[4], onde os casos de teste são criados às vezes até de forma involuntária por especialistas da área (não

[4] Abordagem que não é devidamente projetada em razão da necessidade de atender a uma demanda específica

Figura 1 – Notação dos Grafos Causa-Efeito



Fonte: Adaptada de Delamaro, Jino e Maldonado (2017).

necessariamente testadores ou programadores) a partir de suas experiências e intuição, logo, não existe uma forma sistemática de derivar casos de teste a partir desse critério.

2.3.2 Teste Estrutural

Todas as técnicas de teste devem ser tratadas de forma a auxiliarem umas às outras, sendo assim, a técnica estrutural deve ser vista como complementar às demais técnicas existentes, pois cada classe cobre diferentes classes de defeitos.

O teste estrutural, muitas vezes chamado de teste de caixa branca, é uma técnica de teste que define os requisitos de teste baseados na implementação do software, fazendo com que seja necessária a execução de unidades ou componentes elementares do software. A geração, seleção e adaptação dos casos de teste para esta técnica, devem ser feitos levando em consideração o conhecimento da estrutura interna do programa, exatamente o oposto da Técnica Funcional. O artefato de teste nessa técnica são os caminhos lógicos, de forma que são fornecidos casos de teste que exercitem conjuntos específicos de condições, bem como laços de repetição, pares de definição e utilização de variáveis.

Para abstrair a codificação do programa, a técnica estrutural utiliza uma representação de programa denotada “Grafo de Fluxo de Controle (GFC)” ou “Grafo de Programa”. A representação pode ser abstraída da seguinte forma: $Grafo = (E, N, S)$, onde E é o conjunto de vértices, N o conjunto de arestas e S é o vértice inicial. A construção do GFC é feita através do estabelecimento de relações entre vértices (nós) e arestas (arcos).

Exemplo

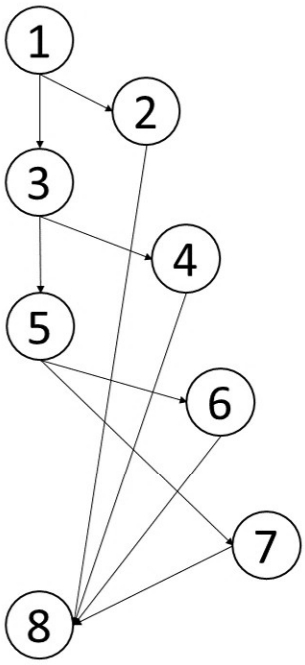
Para o Código-fonte 1 (que representa a implementação do programa “Exibe Notas”), o GFC correspondente é representado na Figura 2.

- As linhas 3 e 5 correspondem ao nó 1 do grafo;
- A linha 6 corresponde ao nó 2 do grafo;
- A linha 7 corresponde ao nó 3 do grafo;
- A linha 8 corresponde ao nó 4 do grafo;
- A linha 9 corresponde ao nó 5 do grafo;
- A linha 10 corresponde ao nó 6 do grafo;
- As linhas 11 e 12 correspondem ao nó 7 do grafo;
- A linha 15 corresponde ao nó 8 do grafo;

Código-fonte 1 – Subrotina para retornar a situação do aluno com base em sua nota

```
 1: public String obtemResultado(int notaFinal){
 2:
 3:     String resultado = "";
 4:
 5:   if ((notaFinal < 0) || (notaFinal > 10)){
 6:     resultado = "Entrada Inválida";
 7:   }else if (notaFinal < 5){
 8:     resultado = "Reprovado";
 9:   }else if (notaFinal < 7){
10:     resultado = "Exame";
11:   }else{
12:     resultado = "Aprovado";
13:   }
14:
15:     return resultado;
16:
17: }
```

Figura 2 – GFC correspondente ao Código-fonte 1



Fonte: Elaborada pelo autor.

Critérios da técnica estrutural

Assim como a técnica funcional, a técnica estrutural também tem alguns critérios específicos que são utilizados para tentar revelar defeitos em um software. Tais critérios são descritos a seguir.

2.3.2.1 Critérios baseados na complexidade

Tais critérios são também chamados como teste do caminho básico ou teste de McCabe. Esses critérios criam os requisitos de teste com base nas informações sobre a complexidade ciclomática do programa (McCabe, 1976). A complexidade ciclomática é uma métrica de software que fornece uma parâmetro quantitativa da complexidade lógica de um programa (DELAMARO; JINO; MALDONADO, 2017).

2.3.2.2 Critérios baseados em fluxo de controle

Os critérios que se baseiam no fluxo de controle do programa, utilizam características inerentes à execução do programa, bem como, desvios e comandos como determinantes na decisão de quais organizações são necessárias. Conforme Myers, Sandler e Badgett (2011) os critérios mais conhecidos são

- **Todos-Nós**: obriga que cada comando (nó) do programa seja executado pelo menos uma vez;

- **Todas-Arestas**: obriga que cada desvio (aresta) do fluxo de controle do programa seja executado pelo menos uma vez;

- **Todos-Caminhos**: obriga que todos os caminhos possíveis do programa sejam executados pelo menos uma vez.

2.3.2.3 Critérios baseados em fluxo de dados

Os critérios baseados em fluxos de dados derivam os casos de teste a partir das relações que tratem das definições de variáveis e as posteriores referências a essas definições. Segundo Ural e Yang (1988), os critérios baseados no fluxo de dados definem que pelo menos uma vez durante o teste, a relação entre a definição e o uso de cada variável seja testada. Os critérios mais conhecidos são:

- **Critérios de Rapps e Weyuker** (RAPPS; WEYUKER, 1982) (RAPPS; WEYUKER, 1985): Os autores propuseram uma extensão ao GFC, chamada "Grafo Def-Uso", que consiste em complementar o GFC com dados referentes ao fluxo de dados do programa, de forma a identificar associações entre pontos do programa onde existe uma atribuição de valor (definição) à variável e pontos onde tal valor é utilizado (referência ou uso da variável);

- **Critérios Potenciais-Usos**: A família de critérios Potenciais-Usos define uma classificação categórica de critérios entre os critérios todos-nós e todas-arestas, e ainda satisfaz o requisito mínimo de cobertura do critério de fluxo de dados, ainda que existam caminhos não executáveis (MALDONADO *et al.*, 1991).

2.3.3 Teste Baseado em Defeitos

O teste de mutação ou análise de mutantes é um critério de teste da técnica de teste baseado em defeitos. Conforme DeMillo e Offutt (1991), esse critério mede a adequação de um dado conjunto de casos de teste. De forma prática, esse critério faz com que o responsável pelo teste interaja com um sistema de mutação automatizado para determinar e melhorar a adequação de um dado conjunto de casos de teste, forçando-o a testar típicos defeitos cometidos na implementação de um software. Papadakis *et al.* (2019) destacam que um dos problemas fundamentais no teste de software é a incapacidade de saber prática ou teoricamente quando se testou suficientemente, enquanto Chekam *et al.* (2017) destacam que existem três vantagens em utilizar o teste de mutação: (i) apontar os elementos que devem ser testados ao desenvolver os casos de teste; (ii) fornecer critérios para o término do teste (quando a cobertura é atingida); e (iii) quantificar a adequação do conjunto de casos de teste.

De acordo com a hipótese do programador competente, os programadores escrevem programas quase perfeitos, e que os erros cometidos são pequenos defeitos sintáticos e que

podem ser corrigidos facilmente (DEMILLO; LIPTON; SAYWARD, 1978). Dessa forma, na prática o teste de mutação cria programas denominados programas mutantes ou alternativos, que são o programa original com algumas alterações sintática. Aceitando a hipótese do programador competente como verdadeira, compreende-se que as alterações feitas no programa original, não inserem erros sintáticos (que fariam que o programa não fosse mais compilável), mas introduzem defeitos artificiais que alterariam a semântica do programa (PAPADAKIS *et al.*, 2019).

Essas alterações no programa não são feitas de maneira aleatória, mas de forma sistemática aplicando determinados operadores de mutação, que nada mais são que a aplicação dos erros de implementação mais corriqueiros. Portanto, os casos de teste capazes de identificar a diferença no comportamento do programa original e do programa mutante, podem ser considerados melhores do que os casos de teste que não identificaram, compreende-se assim, pois, um caso de teste capaz de identificar uma alteração no comportamento do programa devido ao defeito inserido, é capaz também de revelar outros tipos de defeito (MADEYSKI *et al.*, 2014).

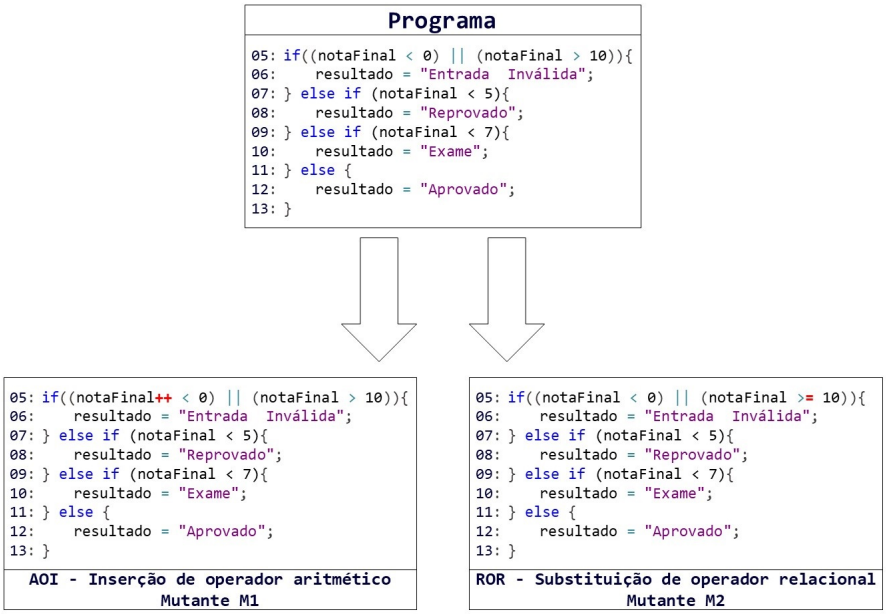
Cada paradigma e linguagem de programação, podem ter seus operadores de mutação específicos. O Apêndice A lista os operadores de mutação das linguagens *Fortran-77*, *C* e *Java*. A Figura 3 mostra como seria a aplicação de 2 operadores de mutação da linguagem *Java* utilizando parte do Código-fonte 1. A ferramenta "MuJava"[5] desenvolvida por Ma, Offutt e Kwon (2006) foi utilizada para gerar programas mutantes e exemplificar seu funcionamento. Como o Código-fonte 1 é um programa simples, não coube a geração de mutantes de classe, sendo assim, foram gerados 34 programas mutantes com base em todos os operadores de mutação tradicionais.

Após os mutantes terem sido gerados, deve-se analisar o resultado da execução de cada um dos mutantes sob o conjunto de casos de teste (denotado por T), espera-se que o comportamento dos programas mutantes sejam diferentes do comportamento do programa original (denotado por P). Se um dado mutante (denotado por M) tem resultado diferente de sucesso/falha do resultado esperado da execução de P, diz-se que M foi morto, portanto, descartado, porém, se o resultado é o mesmo de P, T não conseguiu diferenciar entre M e P e diz-se que M permanece vivo. A existência de mutantes vivos pode indicar que:

- T tem uma vulnerabilidade e pode não conseguir identificar quando um programador cometer um erro igual ao erro introduzido por M; ou
- M é um Mutante Equivalente (denotado por ME), isto é, seu comportamento é igual ao comportamento de P, portanto, M e P são indistinguíveis, fazendo com que seja impossível matar M. Laplante (2010) destaca que decidir se um mutante é ou não um mutante equivalente, é um problema geralmente indecidível, dessa forma, é necessário que o analista de teste ou testador analise todos os mutantes vivos e assim faça a classificação

[5] Informações sobre outras ferramentas que implementam a execução e análise de mutantes podem ser obtidas no Subsubseção 2.3.3.1

Figura 3 – Aplicação dos operadores de mutaçao AOI e ROR



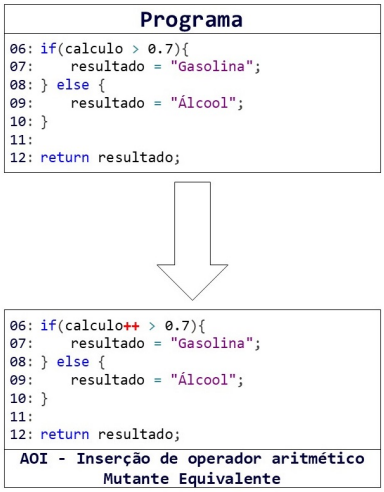
Fonte: Elaborada pelo autor.

daqueles que são equivalentes, a fim de não perder tempo tentando matar esse mutante e também incorporar esse número no cálculo do escore de mutação que será tratado logo adiante. O Código-fonte 2 exibe uma sub-rotina que pode ser utilizada para calcular o melhor combustível de acordo com os valores. A Figura 4 exibe um *ME* ao *P*, tal, é considerado *ME* pois o pós-incremento realizado na variável *calculo*, não altera em nada o comportamento do programa.

Código-fonte 2 – Subrotina para calcular o melhor combustível de acordo com os preços

```
1: public String alcool_ou_gasolina(float valorAlcool, float
     valorGasolina){
2:
3:   float calculo = valorAlcool / valorGasolina;
4:   String resultado = "";
5:
6:   if (calculo > 0.7f){
7:     resultado = "Gasolina";
8:   }else{
9:     resultado = "Álcool";
10:  }
11:
12:  return resultado;
13: }
```

Figura 4 – Programa original e mutante equivalente



Fonte: Elaborada pelo autor.

Uma vez que já se tem o número de mutantes mortos, é possível calcular o nível de adequação de um conjunto de teste através do escore de mutação (*mutation score*). Conforme Ammann e Offutt (2016), o escore de mutação pode ser utilizado como uma métrica de adequação do conjunto de casos de teste com relação ao programa. O escore de mutação é um número entre 0 e 1, portanto, quanto mais próximo do 1 for o escore de mutação, maior é o nível de adequação do conjunto de casos de teste. O cálculo do escore de mutação *ms(P, T)* do programa P com relação ao conjunto de casos de teste T se dá da seguinte forma:

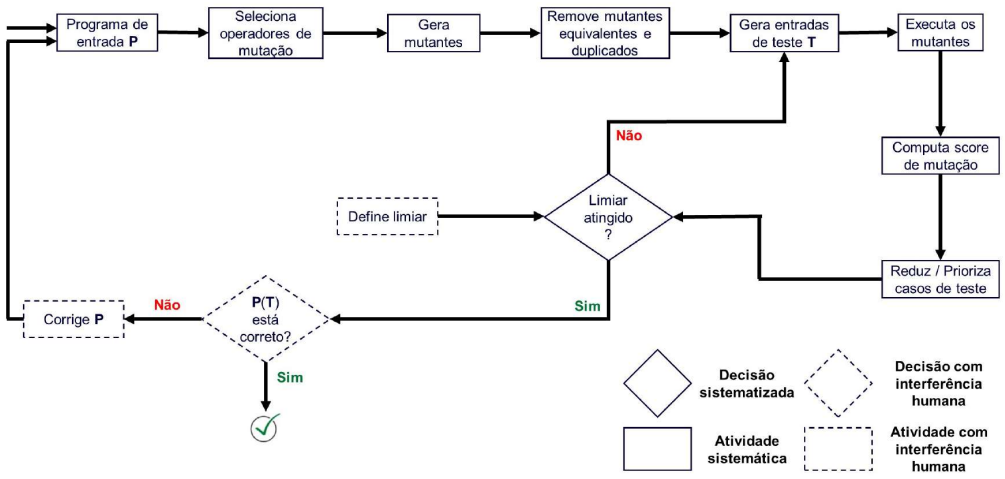
$$ms(P,T) = \frac{MM}{TM - ME}$$

onde:

- ***ms(P, T)***: Escore de mutação do programa P com relação ao conjunto de casos de teste T;
- **MM**: Número de mutantes mortos pelo conjunto de casos de teste;
- **TM**: Número total de mutantes gerados;
- **ME**: Número de mutantes equivalentes.

Offutt e Untch (2001) define um processo genérico do teste de mutação. Devido aos avanços da área, Papadakis *et al.* (2019) adaptou aquele processo, para um processo moderno. O processo moderno envolvido no teste de mutação é representado na Figura 5. Os retângulos e losangos com bordas tracejadas, representam atividades que demandam a interação humana, enquanto os retângulos e losangos com bordas contínuas representam atividades que podem ser sistematizadas.

Figura 5 – Processo genérico do teste de mutação



Fonte: Adaptada de Papadakis *et al.* (2019).

2.3.3.1 Ferramentas

Sem a utilização de ferramentas, o teste de mutação (assim como qualquer outro critério de teste) é afetado devido a uma série de passos complexos envolvidos no processo. Delamaro, Jino e Maldonado (2017) apontam várias ferramentas que foram desenvolvidas desde o desenvolvimento da técnica até hoje. Jia e Harman (2011) e Papadakis *et al.* (2019) apresentam um catálogo com as principais ferramentas utilizadas no suporte à aplicação do teste de mutação. O Subsubseção 2.3.3.1 sumariza as informações exibindo referências, nomes e as linguagens de aplicação das ferramentas. As ferramentas que foram desenvolvidas mas não tiveram um nome definido não foram exibidas no Subsubseção 2.3.3.1. É notável que as linguagens *Java* e *C* tem recebido maior atenção dos pesquisadores, com um vasto número de ferramentas para aplicação do critério, 19 (correspondente à 26,76%) e 15 (correspondente à 21,13%) respectivamente. Ao longo do texto, foi utilizada a ferramenta *MuJava* para demonstração de alguns exemplos e é destacada também na parte final, no Capítulo 5.

Tabela 1 – Ferramentas que dão suporte à aplicação do teste de mutação

Referência	**Ferramenta**	**Linguagem**
Jabbarvand e Malek (2017)	μDroid	Android apps
Delamare *et al.* (2009) e Delamare, Baudry e Traon (2009)	AjMutator	AspectJ

Continuação da Tabela 1

Referência	Ferramenta	Linguagem
Mateo, Usaola e Offutt (2010)	Bacterio	Java
Usaola *et al.* (2017)	BacterioWeb	Android apps
Do e Rothermel (2006)	ByteME	Java
Kusano e Wang (2013)	CCMUTATOR	C/C++
Acree Jr. (1980) e Hanks (1980)	CMS.1	Cobol
Gligoric *et al.* (2013)	Comutation	Java
Derezinska e Szustek (2008)	Cream	C#
Zip... (2007) e Ellims, Ince e Petre (2007)	CSAW	C
Feng, Marr e O'Callaghan (2008)	ESTP	C/C++
Bradbury, Cordy e Dingel (2006)	ExMan	C/Java
Acree *et al.* (1979), Budd (1981) e Budd, Hess e Sayward (1980)	EXPER	Fortran
Tanaka (1981)	FMS.3	Fortran
Domínguez-Jiménez, Estero-Botaro e Medina-Bulo (2009)	GAmera	WS-BPEL
Zhang *et al.* (2010b)	GenMutants	.Net
Confessions... (2013)	Heckle	Ruby
Omar, Ghosh e Whitley (2014)	HOMAJ	AspectJ/Java
Derezinska e Kowalski (2011)	ILMutator	C#
Parasoft... (2019)	Insure++	C/C++
Schuler e Zeller (2009)	Javalanche	Java
Chevalley e Thévenod-Fosse (2003)	JavaMut	Java
Zhou e Frankl (2009) e Zhou e Frankl (2011)	JDama	SQL/JDBC
Jester (2005)	Jester	Java
Dadeau, Héam e Kheddam (2011)	jMuHLPSL	HLPSL
Madeyski e Radyk (2010)	Judy	Java
Jumble (2015)	Jumble	Java
Parsai, Murgia e Demeyer (2017)	LittleDarwin	Java
Just (2014)	Major	Java
Linares-Vásquez *et al.* (2017)	MDroid+	Android apps
Jia e Harman (2008)	Milu	C
Krenn *et al.* (2015)	MoMut	UML models
DeMillo *et al.* (1987) e DeMillo *et al.* (1988)	Mothra	Fortran

Continuação da Tabela 1

Referência	Ferramenta	Linguagem
MuBPEL... (2017)	MuBPEL	WS-BPEL
Le *et al.* (2014)	MuCheck	Haskell
Smith e Williams (2007)	MuClipse	Java
Delgado-Pérez *et al.* (2017)	MuCPP	C++
Shahriar e Zulkernine (2008b)	MUFORMAT	C
Kim, Harrold e Kwon (2006)	MUGAMMA	Java
Ma, Offutt e Kwon (2005), Ma, Offutt e Kwon (2006) e Offutt, Ma e Kwon (2004)	muJava	Java
Shahriar e Zulkernine (2008a)	Music	SQL
Henard, Papadakis e Traon (2014)	MutaLog	Logic
Mirshokraie, Mesbah e Pattabiraman (2013) e Mirshokraie, Mesbah e Pattabiraman (2015)	Mutandis	JavaScript
Li *et al.* (2015) e GitHub... (2019)	mutant (muRuby)	Ruby
mutatepy... (2017)	mutate	C
GitHub... (2010)	MutateMe	PHP
Andrews e Zhang (2003) e Andrews, Briand e Labiche (2005)	mutgen	C
Gligoric, Jagannath e Marinov (2010)	MutMut	Java
Derezińska e Hałas (2014)	MutPy	Python
Arcaini, Gargantini e Riccobene (2017)	MutRex	Regular expressions
Tokumoto *et al.* (2016)	MuVM	C
Nester... (2013)	Nester	C#
Madiraju e Namin (2011)	Paraμ	Java
Andrés, Merayo e Núñez (2009), Andrés, Merayo e Molinero (2009) e Andrés, Merayo e Núñez (2012)	PASTE	TFSM
Jester (2005)	Pester	Python
Budd *et al.* (1978), Budd e Sayward (1977) e Lipton e Sayward (1978)	PIMS	Fortran
Coles *et al.* (2016)	PIT	Java
PlexTest... (2009)	PlexTest	C/C++

Continuação da Tabela 1

Referência	Ferramenta	Linguagem
Delamaro, Maldonado e Vincenzi (2001)	Proteum	C
Walsh, McMinn e Kapfhammer (2015)	REDECHECK	HTML/CSS
Kapfhammer, McMinn e Wright (2013)	SchemaAnalyst	SQL
Crouzet *et al.* (2006)	SESAME	C/Lustre/Pascal
Dan e Hierons (2012)	SMT-C	C
Gligoric, Badame e Johnson (2011)	SMutant	Smalltalk
Tuya, Suarez-Cabal e Riva (2006)	SQLMutation	SQL
Ghosh, Govindarajan e Mathur (2001)	TDS	CORBA IDL
Polo, Tendero e Piattini (2007)	Testooj	Java
Untch (1992), Untch (1997) e Untch, Offutt e Harrold (1993a)	TUMS	C
Devroey *et al.* (2016) e Kintis, Papadakis e Malevris (2010)	Vibes	Transition systems, Statechart models
Praphamontripong e Offutt (2010)	webMuJava	HTML/JSP
Bertolino *et al.* (2013)	XACMUT	XACML

Fonte: Elaborada pelo autor.

As principais características de ferramenta *MuJava* (do inglês, ***Mu**tation System for **Java***) são destacadas a seguir (MACIEL, 2017):

- Tem três componentes principais: (i) Gerador de mutantes; (ii) Visualizador de Mutantes; e (iii) Executor de Mutantes;
- Implementa dois níveis de operadores de mutação: (i) Operadores tradicionais; e (ii) Operadores de classe;
- Permite escolher quais operadores de mutação serão utilizados para geração.

Deve ser mencionado que as funcionalidades mencionadas acima fizeram com que essa ferramenta fosse escolhida para a execução desse projeto.

2.4 Considerações Finais

As atividades de VV&T são de extrema importância nos processos de ES e desenvolvimento de softwares, a fim de tentar garantir e comprovar a eficácia do software construído, bem como a consonância do requisitos elicitados com o software desenvolvido. Este capítulo definiu os principais conceitos, técnicas e métodos relacionados ao Teste de Software, bem como apresentou as diferenças entre as técnica e critérios de teste e a necessidade de uma ser utilizada em conjunto com a outra.

As definições conceituais encerradas nesse capítulo possibilitam o entendimento do próximo conceito, bem como, dos problemas enfrentados na área de Teste de Software. Os próximos dois capítulos, elucidam uma possível solução para os problemas relacionados ao teste de mutação levantados na Subseção 2.3.3.

CAPÍTULO

3

REDUÇÃO DO CUSTO DO TESTE DE MUTAÇÃO

3.1 O problema de custo no teste de mutação

Mesmo sendo reconhecidamente um critério que pode auxiliar na criação de casos de teste com alta efetividade e capacidade de revelar defeitos (DELAMARO; CHAIM; MALDONADO, 2018), o teste de mutação ainda é considerado muito custoso, isso acontece pois normalmente são gerados muitos mutantes, mesmo para programas simples e pequenos. O problema do alto número de mutantes gerados, normalmente se restringe à: mutantes equivalentes e mutantes duplicados (FERNANDES *et al.*, 2017), dessa forma, eleva-se o custo computacional, na tarefa de executar todos os programas mutantes (incluindo os duplicados) e custo pessoal, na tarefa de identificar quais são os mutantes equivalentes. Chekam *et al.* (2017) destacam que o escore de mutação servem para quantificar a adequação do conjunto de teste, toda via, Papadakis *et al.* (2019) mencionam que o escore de mutação é uma métrica questionável, pois é difícil julgar efetivamente a força do conjunto de casos de teste com base no escore de mutação devido aos mutantes equivalentes e duplicados.

Offutt e Untch (2001) relatam que as abordagens para redução do custo do teste de mutação comumente seguem uma das três estratégias abaixo:

- *Do fewer*: Procura meios para reduzir o número de mutantes gerados, mas sem perda de informações. O presente trabalho aplica esta estratégia. Entre as abordagens desta estratégia estão:
  - Mutação Restrita ou Seletiva[1];
  - Seleção Aleatória ou Mutação por Amostragem[1].

[1] Esta estratégia é detalhada na Seção 3.2.

- *Do smarter*: Procura distribuir o custo computacional entre várias máquinas, dividir o custo computacional em diversas execuções, ou, evitar a execução completa de uma só vez. Entre as abordagens desta estratégia está:

  – Mutação Fraca: Conforme Howden (1982 apud OFFUTT; UNTCH, 2001), a mutação fraca é uma técnica de aproximação que compara os estados internos dos programas mutantes e originais imediatamente após a execução da mutação de determinada parte do programa.

- *Do faster*: Procura focar em meios de geração e execução de cada mutante o mais rápido possível. Entre as abordagens desta estratégia está:

  – Análise de Mutantes Baseada em Esquema: Um esquema de programa é um modelo. De acordo com Baruch e Katz (1988 apud UNTCH; OFFUTT; HARROLD, 1993b), um esquema de programa parcialmente interpretado se assemelha sintaticamente a um programa, porém, contém identificadores livres, que são chamados de entidades abstratas, utilizados no lugar de variáveis, identificadores de tipo de dado, constantes e instruções de programa (UNTCH; OFFUTT; HARROLD, 1993b). Conforme Untch (1992), essa técnica junta todos os mutantes num só programa, denominado "metamutante", e então, esse programa é compilado apenas uma vez sob as mesmas condições de desenvolvimento que o programa original, dessa forma, diminuindo significativamente o custo de análise e execução dos mutantes. O processo de geração de metamutantes começa com a construção de uma árvore sintática abstrata decorada. O conceito e o processo de geração da árvore sintática abstrata decorada pode ser observado na Subseção 3.3.1.

Ferrari, Pizzoleto e Offutt (2018) realizaram uma revisão sistemática da literatura, analisando os trabalhos publicados entre 1988 e 2017 na área de teste de mutação, focando em técnicas para redução do custo. No estudo, foram encontrados 146 trabalhos, desses, 110 (75,34%) utilizavam as técnicas *do fewer* como técnica primária do estudo, 22 (15,06%) utilizavam as técnicas *do smarter* como técnica primária do estudo, e 14 (9,58%) utilizavam as técnicas *do faster* como técnica primária do estudo.

3.2 Redução do número de mutantes (*do fewer*)

Mathur e Wong (1993) utilizam a **Seleção Aleatória** como mecanismo para reduzir o custo do teste de mutação. A seleção aleatória examina uma pequena porcentagem aleatória de cada operador de mutação e então ignora-se o restante dos mutantes, porém, garante-se que todos os operadores de mutação serão utilizados.

Wong e Mathur (1995) avaliaram a aplicação da Seleção Aleatória utilizando sete categorias diferentes, cada categoria representava um percentual[2] de operadores de mutação diferentes. Como esperado, a categoria que teve o maior escore de mutação foi a com 40% de mutantes por operador de mutação, enquanto a que obteve o menor escore de mutação foi a categoria com 10% de mutantes por operador de mutação. O escore de mutação obtido para cada uma das categorias é exibido ao final da seção na Tabela 2.

Uma deficiência da Seleção Aleatória é não levar em consideração as características e a capacidade de revelar defeitos de cada um dos operadores de mutação. Mathur (1991) propôs uma estratégia nomeada Mutação Restrita (*Constrained Mutation*). Offutt, Rothermel e Zapf (1993) adaptaram o termo Mutação Restrita para Mutação Seletiva (*Seletive Mutation*), desde então, alguns autores utilizam o termo Mutação Restrita, enquanto outros utilizam o termo Mutação Seletiva. Devido à pequenas diferenças entre as abordagens, este trabalho baseia-se em Jia e Harman (2011) que compreendem as duas abordagens como Mutação Seletiva[3]. Esta estratégia seleciona apenas alguns operadores de mutação para geração de mutantes e análise, enquanto todos os outros mutantes dos outros operadores de mutação são ignorados.

Mathur e Wong (1993), Wong e Mathur (1995) e Wong e Mathur (1995) através de experimentação para analisar os resultados desta estratégia de redução de mutantes, observaram que através da mutação restrita, escolhendo os operadores *abs* [4] e *ror* [5], executando apenas aproximadamente 18% dos mutantes, puderam alcançar um escore de mutação de aproximadamente 92%, isto é, com uma redução de mais de 80% do custo de teste, sacrificou-se apenas aproximadamente 8% do escore de mutação.

Mathur e Wong (1993) relatam que comparando o tamanho, custo e relativo esforço, não existe diferença significativa entre a mutação restrita (utilizando os operadores *abs* e *ror*) e a mutação aleatória (utilizando 10% de cada um dos operadores de mutação). Zhang *et al.* (2010a) relatam que em termos de efetividade ou estabilidade, as técnicas de seleção de mutantes com base nos operadores experimentadas (Offutt *et al.* (1996), Barbosa, Maldonado e Vincenzi (2001) e Namin, Andrews e Murdoch (2008)) não são superiores à mutação aleatória.

Offutt, Rothermel e Zapf (1993) utilizaram a Mutação Seletiva, testando 10 programas escritos em Fortran-77. Em sua primeira execução, nomeada *2-Selective Mutation*, foram excluídos apenas dois operadores de mutação, *ASR* e *SVR*. Posteriormente, foi feita uma execução semelhante, nomeada *4-Selective Mutation*, excluindo quatro operadores de mutação, os dois anteriores, *CSR* e *SCR*. Na terceira execução, nomeada *6-Selective Mutation*, foram excluídos seis operadores de mutação, os quatro anteriores, *ARC* e *ACR*. Os resultados obtidos para cada

[2] 10%, 15%, 20%, 25%, 30%, 35% e 40%.
[3] Para maiores detalhes sobre a diferença entre as abordagens, consultar Ferrari, Pizzoleto e Offutt (2018)
[4] O operador de mutação *abs* gera mutantes trocando $abs(x)$ por $-abs(x)$ e $zpush(x)$ onde for possível.
[5] O operador de mutação *ror* gera mutantes trocando cada operador relacional por outro operador relacional.

uma das categorias são exibidos ao final da seção na Tabela 2.

Offutt *et al.* (1996) definiram três categorias para utilização da mutação seletiva com base nas classes de operadores de mutação da ferramenta *Mothra*, *ES-Selective* que utiliza operadores *expression/statement*, *RS-Selective* que utiliza operadores *replacement/statement* e *RE-Selective* que utiliza operadores *replacement/expression*. Os resultados obtidos para cada uma das categorias são exibidos ao final da seção na Tabela 2.

Após analisar os resultados e observar que o escore de mutação obtido pela categoria *RS-Selective* foi menor que o escore das outras duas categorias, os autores propuseram uma nova categoria, denominada *E-Selective*, que faz a combinação das categorias de mutação seletiva *ES* e *RE*. Essa nova categoria reduziu em 77,56% o custo da análise de mutantes, mas mantendo um escore de mutação alto, de 99,51%.

WONG *et al.* (1997) utilizaram onze operadores de mutação para avaliar a aplicabilidade da mutação seletiva, entre esses onze operadores, foram selecionadas seis combinações de mutação seletiva. As combinações de mutação seletiva obtiveram um escore de mutação médio de 77,22%. As combinações de mutação restrita e seus respectivos operadores de mutação são exibidos no Quadro 1. Os resultados obtidos para cada uma das categorias são exibidos ao final da seção na Tabela 2.

Quadro 1 – Categorias de mutação seletiva - WONG *et al.* (1997)

Categoria	**Operadores de mutação**
MUT1	olln, olng, orrn
MUT2	olln, olng, orrn, ocng, orln, olrn, olan, oaln
MUT3	vtdr, vtwd
MUT4	strp
MUT5	olln, olng, orrn, vdtr, vtwd
MUT6	olln, olng, orrn, vdtr, vtwd, strp

Fonte: WONG *et al.* (1997).

Barbosa, Vincenzi e Maldonado (1998) definiram seis categorias de mutação seletiva com base nas quatro classes[6] de operadores de mutação da ferramenta *Proteum*. As categorias de mutação seletiva são descritas no Quadro 2. Os resultados obtidos para cada uma das categorias são exibidos ao final da seção na Tabela 2.

Por observarem que as três categorias seletivas com maior escore de mutação contém a classe de operadores de mutação de constantes, foi criada a categoria *Cons* que utiliza apenas os operadores da classe de mutação de constantes. Com esta nova categoria de mutação seletiva, foi obtido um escore de mutação de 97,14% com 78,11% de redução de custo da análise de mutantes.

[6] Mutação de comandos (*statement mutations*), mutação de operadores (*operator mutations*), mutação de variáveis (*variable mutations*) e mutação de constantes (*constant mutations*).

Quadro 2 – Categorias de mutação seletiva - Barbosa, Vincenzi e Maldonado (1998)

Categoria	Descrição
Stat-Oper	Utiliza os operadores das classes *statement* e *operator*
Stat-Var	Utiliza os operadores das classes *statement* e *variable*
Stat-Cons	Utiliza os operadores das classes *statement* e *constant*
Oper-Var	Utiliza os operadores das classes *operator* e *variable*
Oper-Cons	Utiliza os operadores das classes *operator* e *constant*
Var-Cons	Utiliza os operadores das classes *variable* e *constant*

Fonte: Barbosa, Vincenzi e Maldonado (1998).

Os autores Barbosa, Vincenzi e Maldonado (1998) ainda trabalharam com uma segunda definição categoria de mutação seletiva, nomeada $MUT6$, composta pelos operadores de mutação OLLN, OLNG, ORRN, VDTR, VTWD e STRP. Com esta categoria seletiva, obtiveram um escore de mutação médio de 98% e uma redução de custo média de 79,70%. Complementarmente à categoria seletiva MUT6, foram incluídos os operadores de mutação OCNG, ORLN, OLRN, OLAN e OALN, obtendo assim um escore de mutação médio de 98,10% e uma redução de custo média de 73,85%.

Barbosa, Maldonado e Vincenzi (2001) definiram um procedimento para a determinação dos operadores suficientes com respeito ao baixo custo e eficiência. Em um primeiro experimento, os autores chegaram nos operadores de mutação SWDD, SMTC, SSDL, OLBN, OASN, ORRN, VTWD, VDTR, Cccr, Ccsr, conjunto denominado *SS-27*. No segundo experimento (dessa vez alterando o conjunto de casos de teste), os operadores de mutação considerados suficientes foram SMTC, SSDL, OEBA, ORRN, VTWD, VDTR, conjunto denominado *SS-5*. Os dados referentes ao escore de mutação e redução do custo obtidos pelos autores pode ser encontrado na Tabela 2.

Zhang *et al.* (2013) propuseram a utilização das estratégias Mutação Seletiva e Seleção aleatória juntas. Os autores definiram oito categorias de Mutação Seletiva. Para cada uma das categorias de mutação seletiva são selecionadores mutantes aleatoriamente com base em um determinado percentual do de determinada origem. As categorias e as origens dos mutantes selecionados são exibidos no Quadro 3. Os dados referentes à média do escore de mutação e da redução do custo médio obtidos pelos autores pode ser encontrado na Tabela 2.

Delamaro *et al.* (2014) avaliaram a utilização de apenas um operador de mutação, o Operador de Delação de Instrução (SSDL - Statement Deletion Operator). O resultado obtido pode ser encontrado no final da seção na Tabela 2.

Uma visão geral dos resultados obtidos por cada autor relatado aqui nesta seção pode ser observada na Tabela 2.

Tabela 2 – Visão geral dos resultados obtidos por cada autor

Autor	**SA**	**MS**	**Categoria**	**EM (%)**	**RC (%)**
Wong e Mathur (1995)	✓		10%	97,56	90,00
	✓		15%	97,77	85,00
	✓		20%	97,95	80,00
	✓		25%	98,68	75,00
	✓		30%	99,10	70,00
	✓		35%	99,26	65,00
	✓		40%	99,39	60,00
Mathur e Wong (1993)		✓	Abs e Ror	92,00	82,00
Offutt, Rothermel e Zapf (1993)		✓	2-SM	99,99	23,98
		✓	4-SM	99,84	41,36
		✓	6-SM	99,71	60,56
Offutt *et al.* (1996)		✓	ES-Selective	99,45	71,52
		✓	RS-Selective	97,31	22,44
		✓	RE-Selective	99,97	6,04
		✓	E-Selective	99,51	77,56
WONG *et al.* (1997)		✓	MUT1	76,67	4,49
		✓	MUT2	70,00	9,05
		✓	MUT3	93,33	20,85
		✓	MUT4	33,33	3,85
		✓	MUT5	93,33	25,34
		✓	MUT6	96,77	29,19
Barbosa, Vincenzi e Maldonado (1998)		✓	Var-Cons	98,82	51,91
		✓	Stat-Cons	98,66	63,28
		✓	Oper-Cons	98,43	41,05
		✓	Stat-Oper	98,40	48,08
		✓	Stat-Var	98,05	58,95
		✓	Oper-Var	97,77	36,73
		✓	Cons	97,14	78,11
Barbosa, Maldonado e Vincenzi (2001)		✓	SS-27	99,66	34,98
		✓	SS-5	99,76	17,95
Zhang *et al.* (2013)	✓	✓	5%	97,77	95,00
	✓	✓	10%	97,95	90,00
	✓	✓	15%	98,68	85,00
	✓	✓	20%	99,10	80,00
	✓	✓	Base	99,74	87,50
	✓	✓	MOp	99,73	87,50
	✓	✓	Class	99,75	87,50
	✓	✓	Meth	99,78	87,50
	✓	✓	Stmt	99,78	87,50
	✓	✓	Class-MOp	99,74	87,50
	✓	✓	Meth-MOp	99,78	87,50
	✓	✓	Stmt-MOp	99,75	87,50
Delamaro *et al.* (2014)		✓	SSDL	96,00	96,74

Legenda – SA: Seleção Aleatória MS: Mutação Seletiva EM: Escore de Mutação RC: Redução de custo

Fonte: Elaborada pelo autor.

Quadro 3 – Categorias de mutação seletiva - Zhang *et al.* (2013)

Categoria	Origem
Base	% do total de mutantes
MOp	% do total de mutantes de cada operador de mutação
Class	% do total de mutantes de uma classe
Meth	% do total de mutantes de um método
Stmt	% do total de mutantes de uma instrução
Class-MOp	% do total de mutantes de cada operador de mutação de uma classe
Meth-MOp	% do total de mutantes de cada operador de mutação de um método
Stmt-MOp	% do total de mutantes de cada operador de mutação de uma instrução

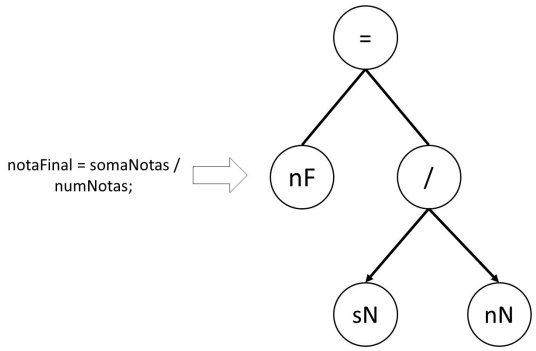
Fonte: Zhang *et al.* (2013).

3.3 Outras abordagens

3.3.1 Árvore sintática abstrata

Em uma Árvore Sintática Abstrata (ASA) cada nó que não é uma folha representa um operador e seus nós filhos representam os operandos. Uma árvore decorada tem atributos tais como o tipo de informação vinculada aos nós. Para exemplificar a utilização de uma ASA, a Figura 6 representa o que seria o comando $notaFinal = somaNotas/numNotas$;.

Figura 6 – Exemplo de árvore sintática



Fonte: Elaborada pelo autor.

3.3.2 Identificação de mutantes inúteis

Fernandes *et al.* (2017) definem uma estratégia para identificarem possíveis candidatos a mutantes inúteis, que segundo os autores, são os mutantes equivalentes e mutantes duplicados. A estratégia é aplicada conforme a seguir: Dado um programa P e um conjunto de teste T, com P não falhando para nenhum caso de teste de T, quando T é executado em um dado mutante

M (gerado a partir de P) e T não consegue distinguir entre o comportamento de M e P, M é classificado como um possível mutante equivalente. Caso dois mutantes gerados a partir de P (M_i e M_j) têm resultados de falha para os mesmos casos de teste em T, a estratégia define M_i e M_j como mutantes duplicados. A partir dessa estratégia, os autores puderam chegar a 37 regras que devem ser aplicadas para que se evite a geração de mutantes que sejam duplicados ou equivalentes.

3.3.3 Mutantes de segunda ordem

Mutantes de primeira ordem são mutantes que contém apenas um erro introduzido no programa original, enquanto mutantes de segunda ordem contém dois erros introduzidos no programa original (POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009). De acordo com os autores, dado um conjunto de mutantes de primeira ordem, ao utilizar a mutação de segunda ordem, o conjunto de mutantes será reduzido pela metade, pois cada programa não conta mais com apenas um mutante, mas sim com dois. Papadakis e Malevris (2010) realizaram um estudo empírico com o objetivo de avaliar o relativo custo e efetividade de mutantes de primeira ordem e mutantes de segunda ordem.

3.3.4 Mutantes teimosos

Yao, Harman e Jia (2014) analisam a relação entre mutantes equivalentes e mutantes teimosos, definidos como difíceis de lidar. Conforme os autores, mutantes teimosos são mutantes que:

- Podem ser mortos;
- Existe pelo menos uma entrada de teste que distingua seu comportamento do comportamento do programa original, porém;
- A entrada de teste que distingue o comportamento do programa original do programa mutante ainda não foi encontrada.

Por meio de um estudo empírico utilizando o conjunto de operadores {ABS, AOR, LCR, ROR, UOI}, os autores encontraram em média 23% de mutantes equivalentes e 7% de mutantes teimosos. Após a realização do estudo empírico, os autores concluíram que a existência de mutantes equivalentes está diretamente relacionada com o tamanho do programa, enquanto a existência de mutantes teimosos não.

3.4 Considerações finais

Após a elucidação dos principais conceitos referentes à teste de software no Capítulo 2, este capítulo apresentou uma possível solução para redução do custo no teste de mutação.

Neste capítulo, foi possível observar a importância da redução do custo no teste de mutação, apresentando diversos estudos realizados entre 1995 e 2014, que em suma buscam apresentar soluções eficientes para o teste de mutação.

As definições conceituais e os trabalhos apresentados nesse capítulo possibilitam o entendimento do próximo conceito, bem como, justificam a abordagem escolhida para este projeto de mestrado. O próximo capítulo, elucida uma possível solução para os problemas relacionados ao teste de mutação e possibilita a compreensão da proposta de trabalho deste projeto.

CAPÍTULO

4

MUTANTES MINIMAIS

4.1 Definições teóricas e resultados preliminares

Para a compreensão do que são mutantes minimais, primeiro, é necessário compreender os conceitos que Ammann, Delamaro e Offutt (2014), Kurtz *et al.* (2014) definiram até chegar nos mutantes minimais. Para facilitar a compreensão, o exemplo retirado de Ammann, Delamaro e Offutt (2014) que está na Tabela 3 será utilizado em todos as seções subsequentes. A Tabela 3 contém cinco casos de teste, $T = \{t1, t2, t3, t4, t5\}$, quatro mutantes, $M = \{m1, m2, m3, m4\}$, escore de mutação de 100% e pode ser lida da seguinte forma:

- O mutante $m1$ é morto pelos casos de teste $t1$, $t3$, $t4$ e $t5$;
- O mutante $m2$ é morto pelos casos de teste $t1$, $t2$, $t4$ e $t5$;
- O mutante $m3$ é morto pelos casos de teste $t2$, $t3$, e $t4$;
- O mutante $m4$ é morto pelos casos de teste $t1$ e $t4$.

Tabela 3 – Escore de mutação de exemplo

	m1	**m2**	**m3**	**m4**
t1	✓	✓		✓
t2		✓	✓	
t3	✓		✓	
t4	✓	✓	✓	✓
t5	✓	✓		

Fonte: Ammann, Delamaro e Offutt (2014).

4.1.1 Conjunto de testes minimais

Um conjunto de teste (denotado $\hat{\mathbf{T}}$) é minimal se e somente se para qualquer $\mathbf{t_i} \in \hat{\mathbf{T}}$, que ao retirar $\mathbf{t_i}$ de $\hat{\mathbf{T}}$, o escore de mutação de M com relação a T é alterado, ou seja, o conjunto de teste só é minimal se caso cada um dos casos de teste forem removidos individualmente, o escore de mutação será alterado em cada remoção, portanto, cada caso de teste é importante para que o escore de mutação seja mantido. Seguindo essa definição, a partir do exemplo da Tabela 3, é possível observar que existem três conjuntos de testes minimais:

$$\bar{\mathbf{T}} = \{\{t4\}, \{t1, t2\}, \{t1, t3\}\}$$

4.1.2 Mutantes redundantes

Um mutante m_j é redundante com respeito ao conjunto de mutantes M e ao conjunto de teste T se e somente se $\bar{\mathbf{T}}_{\mathbf{M}} = \bar{\mathbf{T}}_{\mathbf{M - M_j}}$, ou seja, o mutante só é redundante se a remoção deste mutante não faz alteração nenhuma no conjunto de testes minimais. Abaixo é possível visualizar o método para encontrar os mutantes redundantes do exemplo da Tabela 3:

- $\bar{\mathbf{T}}_{\mathbf{M}} = \{\{t4\}, \{t1, t2\}, \{t1, t3\}\}$;
- $\bar{\mathbf{T}}_{\mathbf{M - m1}} = \{\{t4\}, \{t1, t2\}, \{t1, t3\}\}$;
- $\bar{\mathbf{T}}_{\mathbf{M - m2}} = \{\{t4\}, \{t1, t2\}, \{t1, t3\}\}$;
- $\bar{\mathbf{T}}_{\mathbf{M - m3}} = \{\{t1\}, \{t4\}\}$;
- $\bar{\mathbf{T}}_{\mathbf{M - m4}} = \{\{t4\}, \{t1, t2\}, \{t1, t3\}, \{t2, t5\}, \{t3, t5\}\}$.

É possível visualizar que ao retirar os mutantes $m1$ e $m2$, o conjunto de testes minimais $\bar{\mathbf{T}}$ permanece o mesmo, portanto, esses mutantes são redundantes.

4.1.3 Conjunto de mutantes minimais

Um conjunto de mutantes (denotado por M) é minimal se não contém nenhum mutante redundante. Portanto, com base no exemplo e nos passos anteriores (identificação do conjunto de teste minimal e dos mutantes redundantes), a Tabela 4 exibe os mutantes minimais com relação ao escore de mutação da Tabela 3.

4.1.4 Grafo de Dominância

Ammann e Offutt (2016 apud KURTZ *et al.*, 2014) mencionam que a dominância tem sido usada tradicionalmente para comparar critérios de teste: um critério $C1$ domina um critério $C2$ se todo conjunto de teste que satisfaz $C1$ também satisfaz $C2$. Kurtz *et al.* (2014) estendem o

Tabela 4 – Escore de mutação de exemplo - Mutantes minimais

	m3	m4
t1		✓
t2	✓	
t3	✓	
t4	✓	✓
t5		

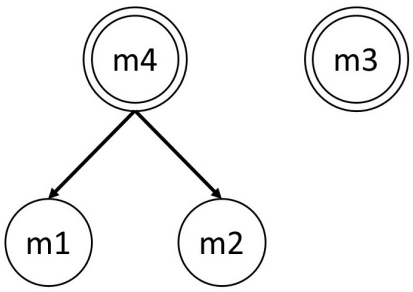
Fonte: Ammann, Delamaro e Offutt (2014).

conceito de dominância para o contexto de mutantes, definindo que: um mutante $M1$ domina outro mutante $M2$ se pelo menos um teste mata $M1$ e se todo teste que mata $M1$ também mata $M2$. Tomando como exemplo a Tabela 3, é possível compreender que:

- Todo teste que mata $m4$ também mata $m1$, portanto, $m4$ domina $m1$: $m4 \rightarrow m1$;
- Todo teste que mata $m4$ também mata $m2$, portanto, $m4$ domina $m2$: $m4 \rightarrow m2$.

A relação de dominância entre mutantes pode ser representada através de um grafo direcionado. A Figura 7 exibe o grafo de dominância referente ao escore da Tabela 3.

Figura 7 – Exemplo de grafo de dominância



Fonte: Elaborada pelo autor.

No grafo de dominância, as relações de dominâncias podem ser visualizadas através das arestas entre os nós. Os duplos círculos ao redor dos nós $m2$ e $m3$ indicam que eles não são dominados por nenhum outro mutante, ou seja, esses mutantes fazem parte do conjunto de mutantes dominantes.

4.1.5 Escore de dominância

Kurtz *et al.* (2016) introduzem um conceito denominado "Escore de dominância"(*dominator mutation score* ou *dominator score*) que segundo os autores, é considerada uma métrica de avaliação mais precisa que o escore de mutação. O escore de dominância de um dado conjunto de

Tabela 5 – Escore de mutação vs Escore de dominância

Conjunto de testes	Escore de mutação	Escore de dominância
{t1}	0,75	0,50
{t2}	0,50	0,50
{t3}	0,50	0,50
{t4}	1,00	1,00
{t5}	0,50	0,00
{t1, t2}	1,00	1,00
{t1, t3}	1,00	1,00
{t1, t2, t3, t4, t5}	1,00	1,00

Fonte: Elaborada pelo autor.

casos de teste é dado pelo número de mutantes pertencentes ao conjunto de mutantes dominantes mortos pelo número total de mutantes no conjunto de mutantes dominantes. A Tabela 5 compara o escore de mutação com o escore de dominância dos casos de teste e mutantes exibidos na Tabela 3, lembrando que o conjunto de mutantes dominantes desse caso é o $\{m3, m4\}$.

4.2 Identificação de mutantes minimais

Kurtz *et al.* (2016) concluem que não existe uma forma de utilizar um conjunto de operadores de mutação e garantir que com aqueles operadores, haverá um baixo custo da análise de mutantes e um alto escore de mutação, portanto, cada programa deve ser analisado individualmente, ou seja, um dado conjunto de operadores de mutação qualquer que é excelente para um dado programa $p1$, não necessariamente será excelente também para um programa $p2$.

A Subseção 4.1.1 define o conceito de conjunto de testes minimais e mostra como obter o conjunto de testes minimais de forma manual. O Algoritmo 1 demonstra de forma sistemática, como se pode obter o conjunto de testes minimais.

Algoritmo 1 – Minimização do conjunto de testes

```
1: procedimento MINTESTSET(M, T)      ▷ Usa como entrada o conjunto de mutantes M e o conjunto de teste T
2:     minSet = T
3:     para todo t em minSet faça      ▷ t é selecionado arbitrariamente
4:         se ms(P, minSet - {t}) == ms(P, T) então      ▷ Verifica se ao retirar t o escore de mutação é mantido
5:             minSet = minSet − {t}
6:         fim se
7:     fim para
8:     retorna minSet      ▷ Obtém o conjunto minimal de testes
9: fim procedimento
```

Fonte: Ammann, Delamaro e Offutt (2014).

A Subseção 4.1.3 define o conceito de conjunto de mutantes minimais e mostra como obter o conjunto de mutantes minimais de forma manual. O Algoritmo 2 demonstra de forma sistemática, como se pode obter o conjunto de mutantes minimais. Ainda que exista um algoritmo para a obtenção do conjunto mínino de mutantes, Kurtz *et al.* (2014) mencionam que a obtenção do conjunto de mutantes dominantes é um problema indecidível, ou seja, existe um algoritmo que implemente este problema, porém, pode ser que ele entre em um *loop* infinito e nunca pare.

Algoritmo 2 – Minimização do conjunto de mutantes

```
1: procedimento MINMUTANTSET(M, T) ▷ Usa como entrada o conjunto de mutantes M e a
   função de pontuação S
2:     S = S − mutantesVivos
3:     S = S − colunasDuplicadas
4:     minSet = S
5:
6:     dominados = mutantes_dinamicamente_dominados em minSet
7:
8:     retorna (minSet - dominados);          ▷ Obtém o conjunto de mutantes minimais
9: fim procedimento
```

Fonte: Ammann, Delamaro e Offutt (2014).

Um conjunto de mutantes dominantes (denotado por D) é um subconjunto mínimo de um conjunto de mutantes (denotado por M), tal que, qualquer conjunto de teste que seja adequado para D, também é adequado para M (JUST; KURTZ; AMMANN, 2017).

4.3 Considerações finais

Após a compreensão das técnicas para redução do custo do teste de mutação apresentadas no Capítulo 3, este capítulo apresentou os conceitos teóricos referentes a mutantes minimais e como se pode identificar mutantes minimais a partir de um conjunto de casos de teste e um conjunto de mutantes.

CAPÍTULO

5

PROPOSTA

5.1 Contexto do Projeto

O Teste de Mutação tem sido amplamente utilizado e considerado como um efetivo critério de teste que permite que sejam criados conjuntos de testes com alta capacidade de revelar defeitos (AMMANN; DELAMARO; OFFUTT, 2014). A aplicação do teste de mutação se dá por meio da criação programas denominados programas mutantes ou alternativos, que são o programa original com algumas alterações sintáticas. Essas alterações no programa não são feitas de maneira aleatória, mas de forma sistemática aplicando determinados operadores de mutação, que nada mais são que a aplicação dos erros de implementação mais corriqueiros (OFFUTT; ROTHERMEL; ZAPF, 1993). Portanto, os casos de teste capazes de identificar a diferença no comportamento do programa original e do programa mutante, podem ser considerados melhores do que os casos de teste que não identificaram, compreende-se assim, pois, um caso de teste capaz de identificar uma alteração no comportamento do programa devido ao defeito inserido, é capaz também de revelar outros tipos de defeito.

Em contraproposta às vantagens, o teste de mutação às vezes se torna impraticável (WONG; MATHUR, 1995) em sua totalidade devido a alguns fatores: (i) o número de mutantes gerados é muito alto, o que faz com que o custo computacional de sua execução seja alto; (ii) são gerados muitos mutantes duplicados; e (iii) são gerados muitos mutantes equivalentes. Papadakis *et al.* (2015a) relatam que pelo menos 21% do total de mutantes, são duplicados. Os autores definem que mutantes duplicados são os mutantes que são equivalentes entre si, mas não são equivalentes ao programa original.

Baldwin e Sayward (1979), Offutt e Pan (1997), Jia e Harman (2011), Papadakis *et al.* (2015b), Fernandes *et al.* (2017), Kintis *et al.* (2018) propuseram abordagens que podem auxiliar na solução do segundo e do terceiro problema, mas ainda assim, esses problemas são prejudiciais e custosos pois é difícil definir uma forma sistemática de identificar mutantes duplicados e

equivalentes, portanto, esses mutantes inflam o escore de mutação e devem ser identificados manualmente. Quanto ao primeiro problema enumerado, existem diversas técnicas[1] que visam sanar este problema, porém, ainda não existe uma técnica que possa reduzir o número de mutantes e garantir a mesma efetividade do critério como se tivessem sido executados todos os mutantes (KURTZ *et al.*, 2016) .

5.2 Objetivos do Projeto

O presente trabalho tem por objetivo avançar o estado da arte na área de Engenharia de Software, mais precisamente, na área de Teste de Software, por meio de uma abordagem que possibilite a redução do custo de aplicação do critério de Teste de Mutação. Essa redução no custo do teste de mutação se encaixa dentro das técnicas nomeadas *do fewer* por Offutt e Untch (2001).

Delamaro, Chaim e Maldonado (2018) fazem uma análise inicial e propõem a ideia de que o conjunto de mutantes minimais pode estar diretamente relacionado com a localização daquele mutante na estrutura do código fonte. Diferentemente das abordagens e estratégias para redução do custo do teste de mutação apresentadas no Capítulo 3, que em sua maioria propõem a redução do custo por meio da eliminação aleatória de determinados mutantes ou da eliminação de determinados operadores de mutação, este projeto visa reduzir o custo do teste de mutação por intermédio da identificação e aplicação dos mutantes minimais. Mediante a utilização do GFC, representação de programas utilizadas na técnica de teste estrutural, espera-se poder relacionar os mutantes minimais com lugares específicos do código, como por exemplo, nós essenciais do GFC.

Portanto, o presente trabalho é norteado pela seguinte questão de pesquisa:

- *É possível relacionar a presença de mutantes minimais com a localização do mutante no código fonte?*

Por meio da investigação pretende-se propor uma abordagem capaz de possibilitar a identificação dos mutantes a partir da estrutura do código fonte, tendo como base a análise a técnica de teste estrutural e seus critérios. A partir do objetivo principal foram derivados os seguintes objetivos específicos:

1. Proposta de uma abordagem que permita a detecção dos mutantes pertencentes ao conjunto de mutantes minimais com base na localização desses mutantes na estrutura do código fonte;

2. Avaliação da abordagem proposta por meio de estudos empíricos. Os estudos empíricos serão realizados na linguagem *Java*.

[1] A Seção 3.2 menciona as técnicas e mostra suas aplicações

- Estudo e análise das características que diferenciam um conjunto de mutantes comuns e conjunto de mutantes minimais;
- O estudo realizado na linguagem *Java* utilizará como fonte de pesquisa os mesmos programas utilizados por Deng, Offutt e Li (2013), que são diversos programas retirados de livros e projetos *open-source*. A ferramenta utilizada para geração e análise dos mutantes é a *muJava* (MA; OFFUTT; KWON, 2006).

5.3 Metodologia

Com o objetivo de fornecer subsídios teóricos para a conclusão deste trabalho de mestrado, foi necessário um estudo de conceitos básicos relacionados à Engenharia de Software e Teste de Software. Dessa forma, o aluno cursou as seguintes disciplinas oferecidas pelo programa de pós-graduação: (i) Engenharia de Software; (ii) Engenharia de Software Experimental; (iii) Validação e Teste de Software, etc. As disciplinas, os respectivos créditos, semestre e o ano em que foram cursadas estão apresentados no Quadro 4. Cada 1 crédito na disciplina corresponde à 15 horas de carga horária.

Quadro 4 – Disciplinas cursadas no programa de mestrado

Disciplina	**Créditos**	**Conceito**	**Semestre**	**Ano**
Validação e Teste de Software	12	A	1	2018
Revisão Sistemática em Engenharia de Software	6	A	1	2018
Preparação Pedagógica	4	A	1	2018
Engenharia de Software	12	A	1	2018
Engenharia de Software Experimental	6	A	1	2018
Tópicos em Computação e Matemática Computacional II	1	B	2	2018
Metodologia de Pesquisa Científica em Computação	2	A	2	2018
Teoria da Computação	12	B	2	2018

Além das disciplinas cursadas, foi realizado um levantamento bibliográfico em busca de trabalhos que mencionassem ou utilizassem estratégias para redução do custo de aplicação do Teste de Mutação. Além das atividades regulares exigidas pelo programa de mestrado do Instituto de Ciências Matemáticas e de Computação (ICMC)/Universidade de São Paulo (USP), diversas atividades técnicas, investigações e estudos são necessárias para a conclusão desta pesquisa de mestrado. A seguir, tais atividades são descritas por meio da associação de aspectos teóricos e práticos e, assim, configurando um plano de trabalho.

Para o desenvolvimento deste projeto de mestrado estão sendo utilizados materiais teóricos oriundos de conferências, periódicos e materiais disponíveis nas bibliotecas físicas e digitais da USP, bem como motores de busca acadêmicos que permitem acesso a alunos da USP.

5.4 Plano de trabalho e Cronograma

As seguintes atividades foram, estão sendo e serão desenvolvidas durante o mestrado:

1. **Integralização dos créditos**: o regimento do programa de Pós-Graduação do ICMC exige cumprir no mínimo 51 créditos, assim foram cumpridos 55 créditos através das disciplinas;

2. **Exame de proficiência em língua estrangeira**: de acordo com o regulamento do programa de Pós-Graduação do ICMC, o aluno tem 12 meses para realização do exame de proficiência da língua inglesa. Dessa forma, o exame foi realizado em maio de 2018;

3. **Análise das atividades envolvidas no teste de software**: devem ser estudadas e analisadas todas as técnicas e critérios de teste de software, afim de identificar quais critérios e técnicas podem ser utilizados em conjunto com o Teste de Mutação;

4. **Análise das atividades envolvidas no teste de mutação**: devem ser identificados os fatores que fazem o teste de mutação ser uma das técnicas de teste que dão maior confiabilidade ao conjunto de teste. Também foram avaliados os fatores que fazem o teste de mutação ser tão custoso e impraticável em alguns aspectos.

5. **Estudo das técnicas para redução do custo de mutação**: Na literatura existem diversas técnicas (Wong e Mathur (1995), Mathur e Wong (1993), Offutt, Rothermel e Zapf (1993), Offutt *et al.* (1996), WONG *et al.* (1997) e Barbosa, Vincenzi e Maldonado (1998)) que podem ser utilizadas para reduzir o custo de mutação. As técnicas foram estudadas e avaliadas para a definição de qual estratégia seria utilizada neste projeto de mestrado.

6. **Escrita da qualificação**: de acordo com o regulamento do Programa de Pós-Graduação do ICMC, todo aluno de mestrado deve passar por um exame de qualificação, a ser realizado em até 12 meses após o início do curso, neste caso, até março de 2019. Para isso, o aluno irá redigir uma monografia para o Exame Geral de Qualificação, a qual deverá ser iniciada em agosto de 2018 e concluída até fevereiro de 2018.

7. **Defesa da qualificação**: a defesa da qualificação entregue até março de 2019 está prevista para abril de 2019;

8. **Definição e condução de estudos experimentais**: após o desenvolvimento das atividades iniciais do projeto de mestrado, deverão ser conduzidos estudos experimentais baseados nas diretrizes propostas por Wohlin *et al.* (2012) para avaliação da proposta e assim fornecer uma resposta à questão de pesquisa principal da dissertação e posteriormente documentar e publicar devidamente os resultados encontrados.

9. **Escrita de artigos científicos**: o aluno será incentivado a produzir artigos científicos que reflitam os resultados obtidos durante o mestrado de forma a disseminar as contribuições alcançadas. Esta atividade ocorrerá de maneira contínua durante todo o curso. Espera-se

conseguir a publicação de trabalhos em congresso e/ou periódicos que abordem as áreas de Engenharia de Software, Teste de Software e Análise de Mutantes;

10. **Escrita do documento da dissertação**: a redação da dissertação de mestrado será facilitada em parte pela redação da monografia de qualificação, a qual conterá parte inicial da revisão bibliográfica do trabalho. Outras atividades que facilitaram a escrita da dissertação serão a escrita dos artigos previstos e a consolidação do material de pesquisa utilizado no trabalho. Espera-se que o texto produzido permita organizar de maneira clara e sistemática o trabalho desenvolvido suas contribuições; e

11. **Defesa da dissertação**: a defesa da dissertação prevista para ser entregue até fevereiro de 2020 está prevista para março de 2020.

Algumas das atividades listadas já foram concluídas, outras estão em curso e algumas outras devem ser desenvolvidas ainda. O cronograma completo para cada uma das atividades e a descrição das atividades finalizadas pode ser observado no Quadro 5.

Quadro 5 – Cronograma de atividades do projeto de mestrado

Ativ	**2018**										**2019**												**2020**		
	3	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓															
2			✓																						
3						✓	✓	✓	✓	✓															
4								✓	✓	✓	✓	✓													
5										✓	✓	✓													
6						✓	✓	✓	✓	✓	✓	✓													
7													✓												
8													✓	✓	✓	✓									
9				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
10																✓	✓	✓	✓	✓	✓	✓	✓	✓	
11																									✓

5.5 Resultados Esperados

Ao final do projeto de mestrado, espera-se avançar o estado da arte na área de Teste de Software, propondo uma abordagem mais eficiente para aplicação do critério Teste de Mutação.

Dentre as principais contribuições que esse trabalho pode desencadear, é possível destacar:

- Uma abordagem para detecção de mutantes minimais;
- Estudos científicos que corroboram para a eficiência da abordagem.

Tal contribuição pode derivar uma serie de desdobramentos para o contexto da aplicação do teste de mutação, como:

- Redução do número de mutantes gerados.
- Melhora na eficiência da aplicação do critério;
- Nova abordagem para aplicação do teste de mutação.

5.6 Considerações Finais

Este capítulo apresenta a proposta do presenta trabalho, que visa propor e validar uma abordagem para redução do custo no teste de mutação, através da localização de mutantes minimais baseados em sua localização na estrutura do código fonte.

Para avaliação e validação da abordagem proposta, serão realizados estudos empíricos com o objetivo de coletar dados e então compará-los com os resultados de outras abordagens e técnicas disponíveis na literatura.

Conforme foi discutido em diversos momentos neste projeto, a aplicação do teste de mutação possui grande eficácia em avaliar e melhor adequar um conjunto de casos de teste e consequentemente, revelar a presença de defeitos em um produto de software. A principal contribuição deste projeto é a viabilização da aplicação do teste de mutação com o custo reduzido, fazendo assim que sua aplicação se torne irrestrita.

Adicionalmente, os resultados obtidos neste projeto serão disponibilizados nas usuais formas de divulgação de trabalhos científicos, como anais de congressos e revistas científicas que serão utilizadas como forma de avaliação e analise dos resultados obtidos.

REFERÊNCIAS

ACREE, A. T.; BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. **Mutation Analysis**. [S.l.], 1979. Citado na página 43.

ACREE JR., A. T. **On Mutation**. Tese (Doutorado) — Georgia Institute of Technology, Atlanta, GA, USA, 1980. AAI8107280. Citado na página 43.

AGRAWAL, H.; DEMILLO, R.; HATHAWAY, R.; HSU, W.; HSU, W.; KRAUSER, E.; MARTIN, R. J.; MATHUR, A.; SPAFFORD, E. **Design of mutant operators for the C programming language**. [S.l.], 1989. Citado na página 82.

AMMANN, P.; DELAMARO, M.; OFFUTT, J. Establishing theoretical minimal sets of mutants. In: **ICST**. [s.n.], 2014. p. 21–30. Cited By 72. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84903164798&doi=10.1109%2fICST.2014.13&partnerID=40&md5=fa08a86d598c7e4410cc0ccb9208cbc8>. Citado nas páginas 57, 59, 60, 61 e 63.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016. Citado nas páginas 41 e 58.

ANDRÉS, C.; MERAYO, M. G.; MOLINERO, C. Advantages of mutation in passive testing: An empirical study. In: IEEE. **2009 International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.], 2009. p. 230–239. Citado na página 44.

ANDRÉS, C.; MERAYO, M. G.; NÚÑEZ, M. Passive testing of stochastic timed systems. In: IEEE. **2009 International Conference on Software Testing Verification and Validation**. [S.l.], 2009. p. 71–80. Citado na página 44.

____. Formal passive testing of timed systems: theory and tools. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 22, n. 6, p. 365–405, 2012. Citado na página 44.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: ACM. **Proceedings of the 27th international conference on Software engineering**. [S.l.], 2005. p. 402–411. Citado na página 44.

ANDREWS, J. H.; ZHANG, Y. General test result checking with log file analysis. **IEEE Transactions on Software Engineering**, IEEE, v. 29, n. 7, p. 634–648, 2003. Citado na página 44.

ARCAINI, P.; GARGANTINI, A.; RICCOBENE, E. Mutrex: A mutation-based generator of fault detecting strings for regular expressions. In: IEEE. **2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.], 2017. p. 87–96. Citado na página 44.

BALDWIN, D.; SAYWARD, F. **Heuristics for Determining Equivalence of Program Mutations.** [S.l.], 1979. Citado na página 63.

BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao Teste de Software – Capítulo 4 - Teste Estrutural. In: **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda., 2017. p. 47–76. ISBN 9788535283525. Citado na página 24.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the determination of sufficient mutant operators for c. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 11, n. 2, p. 113–136, 2001. Citado nas páginas 49, 51 e 52.

BARBOSA, E. F.; VINCENZI, A. M.; MALDONADO, J. C. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas c. **XII Simpósio Brasileiro de Engenharia de Software (SBES 98)**, p. 103–120, 1998. Citado nas páginas 11, 26, 50, 51, 52 e 66.

BARUCH, O.; KATZ, S. Partially interpreted schemas for csp programming. **Science of Computer Programming**, Elsevier, v. 10, n. 1, p. 1–18, 1988. Citado na página 48.

BERTOLINO, A. Software Testing Research and Practice. In: BÖRGER, E.; GARGANTINI, A.; RICCOBENE, E. (Ed.). **Abstract State Machines 2003**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 1–21. ISBN 978-3-540-36498-6. Citado na página 23.

BERTOLINO, A.; DAOUDAGH, S.; LONETTI, F.; MARCHETTI, E. Xacmut: Xacml 2.0 mutants generator. In: IEEE. **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2013. p. 28–33. Citado na página 45.

BRADBURY, J. S.; CORDY, J. R.; DINGEL, J. Exman: A generic and customizable framework for experimental mutation analysis. In: IEEE. **Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)**. [S.l.], 2006. p. 4–4. Citado na página 43.

BUDD, T.; HESS, R.; SAYWARD, F. Exper implementor's guide. **Yale University, New Haven, Connecticut, Technique Report**, 1980. Citado na página 43.

BUDD, T.; SAYWARD, F. Users guide to the pilot mutation system. **Yale University, New Haven, Connecticut, Technique Report**, v. 114, 1977. Citado na página 44.

BUDD, T. A. Mutation analysis of program test data. 1981. Citado na página 43.

BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. The design of a prototype mutation system for program testing. In: **Proceedings of the AFIPS National Computer Conference**. [S.l.: s.n.], 1978. v. 74, p. 623–627. Citado na página 44.

CHEKAM, T. T.; PAPADAKIS, M.; TRAON, Y. L.; HARMAN, M. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: IEEE. **2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)**. [S.l.], 2017. p. 597–608. Citado nas páginas 25, 38 e 47.

CHEVALLEY, P.; THÉVENOD-FOSSE, P. A mutation analysis tool for java programs. **International journal on software tools for technology transfer**, Springer, v. 5, n. 1, p. 90–103, 2003. Citado na página 43.

COLES, H.; LAURENT, T.; HENARD, C.; PAPADAKIS, M.; VENTRESQUE, A. Pit: a practical mutation testing tool for java. In: ACM. **Proceedings of the 25th International Symposium on Software Testing and Analysis**. [S.l.], 2016. p. 449–452. Citado na página 44.

CONFESSIONS of a Ruby Sadist sudo gem install heckle. 2013. <http://ruby.sadi.st/Heckle.html>. (Accessed on 03/07/2019). Citado na página 43.

CROUZET, Y.; WAESELYNCK, H.; LUSSIER, B.; POWELL, D. The sesame experience: from assembly languages to declarative models. In: IEEE. **Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)**. [S.l.], 2006. p. 7–7. Citado na página 45.

DADEAU, F.; HÉAM, P.-C.; KHEDDAM, R. Mutation-based test generation from security protocols in hlpsl. In: IEEE. **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**. [S.l.], 2011. p. 240–248. Citado na página 43.

DAN, H.; HIERONS, R. M. Smt-c: A semantic mutation testing tools for c. In: IEEE. **2012 IEEE Fifth International Conference on Software Testing, Verification and Validation**. [S.l.], 2012. p. 654–663. Citado na página 45.

DELAMARE, R.; BAUDRY, B.; GHOSH, S.; TRAON, Y. L. A test-driven approach to developing pointcut descriptors in aspectj. In: IEEE. **2009 International Conference on Software Testing Verification and Validation**. [S.l.], 2009. p. 376–385. Citado na página 42.

DELAMARE, R.; BAUDRY, B.; TRAON, Y. L. Ajmutator: A tool for the mutation analysis of aspectj pointcut descriptors. In: IEEE. **2009 International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.], 2009. p. 200–204. Citado na página 42.

DELAMARO, M.; CHAIM, M. L.; MALDONADO, J. C. Where are the minimal mutants? In: **Proceedings of the XXXII Brazilian Symposium on Software Engineering**. New York, NY, USA: ACM, 2018. (SBES '18), p. 190–195. ISBN 978-1-4503-6503-1. Disponível em: <http://doi.acm.org/10.1145/3266237.3266241>. Citado nas páginas 47 e 64.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda., 2017. ISBN 9788535283525. Citado nas páginas 23, 29, 31, 33, 34, 35, 37 e 42.

DELAMARO, M. E.; BARBOSA, E. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Introdução ao Teste de Software – Capítulo 5 - Teste de Mutação. In: **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda., 2017. p. 77–116. ISBN 9788535283525. Citado na página 25.

DELAMARO, M. E.; DENG, L.; DURELLI, V. H. S.; LI, N.; OFFUTT, J. Experimental evaluation of sdl and one-op mutation for c. In: IEEE. **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation**. [S.l.], 2014. p. 203–212. Citado nas páginas 51 e 52.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Introdução ao Teste de Software – Capítulo 1 - Conceitos Básicos. In: **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda., 2017. p. 1–8. ISBN 9788535283525. Citado na página 24.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Proteum/im 2.0: An integrated mutation testing environment. In: **Mutation testing for the new century**. [S.l.]: Springer, 2001. p. 91–101. Citado na página 45.

DELGADO-PÉREZ, P.; MEDINA-BULO, I.; PALOMO-LOZANO, F.; GARCÍA-DOMÍNGUEZ, A.; DOMÍNGUEZ-JIMÉNEZ, J. J. Assessment of class mutation operators for c++ with the mucpp mutation system. **Information and Software Technology**, Elsevier, v. 81, p. 169–184, 2017. Citado na página 44.

DEMILLO, R.; GUINDI, D.; KING, K.; MCCRACKEN, W.; MATHUR, A.; OFFUTT, A. An overview of the mothra software testing environment. **Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-3-P**, 1987. Citado na página 43.

DEMILLO, R. A.; GUINDI, D. S.; MCCRACKEN, W.; OFFUTT, A. J.; KING, K. An extended overview of the mothra software testing environment. In: IEEE. **[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis**. [S.l.], 1988. p. 142–151. Citado na página 43.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, April 1978. ISSN 0018-9162. Citado na página 39.

DEMILLO, R. A.; OFFUTT, A. J. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, v. 17, n. 9, p. 900–910, Sep. 1991. ISSN 0098-5589. Citado nas páginas 25 e 38.

DENG, L.; OFFUTT, J.; LI, N. Empirical evaluation of the statement deletion mutation operator. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2013. p. 84–93. ISSN 2159-4848. Citado nas páginas 26 e 65.

DEREZIŃSKA, A.; HAŁAS, K. Analysis of mutation operators for the python language. In: SPRINGER. **Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland**. [S.l.], 2014. p. 155–164. Citado na página 44.

DEREZINSKA, A.; KOWALSKI, K. Object-oriented mutation applied in common intermediate language programs originated from c. In: IEEE. **2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2011. p. 342–350. Citado na página 43.

DEREZINSKA, A.; SZUSTEK, A. Tool-supported advanced mutation approach for verification of c# programs. In: IEEE. **2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX**. [S.l.], 2008. p. 261–268. Citado na página 43.

DEVROEY, X.; PERROUIN, G.; PAPADAKIS, M.; LEGAY, A.; SCHOBBENS, P.-Y.; HEYMANS, P. Featured model-based mutation analysis. In: ACM. **Proceedings of the 38th International Conference on Software Engineering**. [S.l.], 2016. p. 655–666. Citado na página 45.

DO, H.; ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. **IEEE Transactions on Software Engineering**, IEEE, v. 32, n. 9, p. 733–752, 2006. Citado na página 43.

DOMÍNGUEZ-JIMÉNEZ, J. J.; ESTERO-BOTARO, A.; MEDINA-BULO, I. A framework for mutant genetic generation for ws-bpel. In: SPRINGER. **International Conference on Current Trends in Theory and Practice of Computer Science**. [S.l.], 2009. p. 229–240. Citado na página 43.

ELLIMS, M.; INCE, D.; PETRE, M. The csaw c mutation tool: Initial results. In: IEEE. **Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)**. [S.l.], 2007. p. 185–192. Citado na página 43.

FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Introdução ao Teste de Software – Capítulo 2 - Teste Funcional. In: **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda., 2017. p. 9–26. ISBN 9788535283525. Citado na página 24.

FENG, X.; MARR, S.; O'CALLAGHAN, T. Estp: an experimental software testing platform. In: IEEE. **Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008)**. [S.l.], 2008. p. 59–63. Citado na página 43.

FERNANDES, L.; GHEYI, R.; CAVALCANTI, A.; RIBEIRO, M.; MONGIOVI, M.; FERRARI, F.; CARVALHO, L.; SANTOS, A.; MALDONADO, J. Avoiding useless mutants. In: . [s.n.], 2017. p. 187–198. Cited By 5. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85041723246&doi=10.1145%2f3136040.3136053&partnerID=40&md5=221a4b63e393477dd4deb58da64a26a9>. Citado nas páginas 47, 53 e 63.

FERRARI, F. C.; PIZZOLETO, A. V.; OFFUTT, J. A systematic review of cost reduction techniques for mutation testing: Preliminary results. In: IEEE. **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.], 2018. p. 1–10. Citado nas páginas 48 e 49.

GHOSH, S.; GOVINDARAJAN, P.; MATHUR, A. P. Tds: a tool for testing distributed component-based applications. In: **Mutation testing for the new century**. [S.l.]: Springer, 2001. p. 103–112. Citado na página 45.

GITHUB - manuelpichler/mutateme: A PHP 5.3+ Mutation Testing framework. 2010. <https://github.com/manuelpichler/mutateme>. (Accessed on 02/28/2019). Citado na página 44.

GITHUB - mbj/mutant: Mutation testing for Ruby. 2019. <https://github.com/mbj/mutant>. (Accessed on 02/28/2019). Citado na página 44.

GLIGORIC, M.; BADAME, S.; JOHNSON, R. Smutant: a tool for type-sensitive mutation testing in a dynamic language. In: ACM. **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering**. [S.l.], 2011. p. 424–427. Citado na página 45.

GLIGORIC, M.; JAGANNATH, V.; MARINOV, D. Mutmut: Efficient exploration for mutation testing of multithreaded code. In: IEEE. **2010 Third International Conference on Software Testing, Verification and Validation**. [S.l.], 2010. p. 55–64. Citado na página 44.

GLIGORIC, M.; ZHANG, L.; PEREIRA, C.; POKAM, G. Selective mutation testing for concurrent code. In: ACM. **Proceedings of the 2013 International Symposium on Software Testing and Analysis**. [S.l.], 2013. p. 224–234. Citado na página 43.

HANKS, J. M. **Testing COBOL programs by mutation**. Tese (Doutorado) — Georgia Institute of Technology, 1980. Citado na página 43.

HENARD, C.; PAPADAKIS, M.; TRAON, Y. L. Mutalog: A tool for mutating logic formulas. In: IEEE. **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2014. p. 399–404. Citado na página 44.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. **IEEE Transactions on Software Engineering**, SE-8, n. 4, p. 371–379, July 1982. ISSN 0098-5589. Citado na página 48.

IEEE Standard Glossary of Software Engineering Terminology. **IEEE Std 610.12-1990**, p. 1–84, Dec 1990. Citado na página 30.

JABBARVAND, R.; MALEK, S. μdroid: an energy-aware mutation testing framework for android. In: ACM. **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. [S.l.], 2017. p. 208–219. Citado na página 42.

JESTER. 2005. <http://jester.sourceforge.net/>. (Accessed on 02/28/2019). Citado nas páginas 43 e 44.

JIA, Y.; HARMAN, M. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In: IEEE. **Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008)**. [S.l.], 2008. p. 94–98. Citado na página 43.

____. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, v. 37, n. 5, p. 649–678, Sep. 2011. ISSN 0098-5589. Citado nas páginas 42, 49 e 63.

JUMBLE. 2015. <http://jumble.sourceforge.net/>. (Accessed on 02/28/2019). Citado na página 43.

JUST, R. The major mutation framework: Efficient and scalable mutation analysis for java. In: ACM. **Proceedings of the 2014 International Symposium on Software Testing and Analysis**. [S.l.], 2014. p. 433–436. Citado na página 43.

JUST, R.; KURTZ, B.; AMMANN, P. Inferring mutant utility from program context. In: ACM. **Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.], 2017. p. 284–294. Citado nas páginas 26 e 61.

KAPFHAMMER, G. M.; MCMINN, P.; WRIGHT, C. J. Search-based testing of relational schema integrity constraints across multiple database management systems. In: IEEE. **2013 ieee sixth international conference on software testing, verification and validation**. [S.l.], 2013. p. 31–40. Citado na página 45.

KIM, S.-W.; HARROLD, M. J.; KWON, Y.-R. Mugamma: Mutation analysis of deployed software to increase confidence and assist evolution. In: IEEE. **Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)**. [S.l.], 2006. p. 10–10. Citado na página 44.

KING, K.; OFFUTT, A. A fortran language system for mutation-based software testing. **Software: Practice and Experience**, v. 21, n. 7, p. 685–718, 1991. Cited By 214. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-0026185573&doi=10.1002%2fspe.4380210704&partnerID=40&md5=63bb76053636c342797821b6afb78609>. Citado na página 81.

KINTIS, M.; PAPADAKIS, M.; JIA, Y.; MALEVRIS, N.; TRAON, Y. L.; HARMAN, M. Detecting trivial mutant equivalences via compiler optimisations. **IEEE Transactions on Software Engineering**, IEEE, v. 44, n. 4, p. 308–333, 2018. Citado na página 63.

KINTIS, M.; PAPADAKIS, M.; MALEVRIS, N. Evaluating mutation testing alternatives: A collateral experiment. In: IEEE. **2010 Asia Pacific Software Engineering Conference**. [S.l.], 2010. p. 300–309. Citado na página 45.

KRENN, W.; SCHLICK, R.; TIRAN, S.; AICHERNIG, B.; JOBSTL, E.; BRANDL, H. Momut:: Uml model-based mutation testing for uml. In: IEEE. **2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.], 2015. p. 1–8. Citado na página 43.

KURTZ, B.; AMMANN, P.; DELAMARO, M. E.; OFFUTT, J.; DENG, L. Mutant subsumption graphs. In: **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops**. [S.l.: s.n.], 2014. p. 176–185. Citado nas páginas 57, 58 e 61.

KURTZ, B.; AMMANN, P.; OFFUTT, J.; DELAMARO, M. E.; KURTZ, M.; GöKcE, N. Analyzing the validity of selective mutation with dominator mutants. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2016. (FSE 2016), p. 571–582. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950322>. Citado nas páginas 59, 60 e 64.

KUSANO, M.; WANG, C. Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In: IEEE PRESS. **Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.], 2013. p. 722–725. Citado na página 43.

LAPLANTE, P. **Encyclopedia of Software Engineering Three-Volume Set (Print)**. CRC Press, 2010. ISBN 9781351249256. Disponível em: <https://books.google.com.br/books?id=NExnDwAAQBAJ>. Citado na página 39.

LE, D.; ALIPOUR, M. A.; GOPINATH, R.; GROCE, A. Mucheck: An extensible tool for mutation testing of haskell programs. In: ACM. **Proceedings of the 2014 international symposium on software testing and analysis**. [S.l.], 2014. p. 429–432. Citado na página 44.

LI, N.; WEST, M.; ESCALONA, A.; DURELLI, V. H. Mutation testing in practice using ruby. In: IEEE. **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.], 2015. p. 1–6. Citado na página 44.

LINARES-VÁSQUEZ, M.; BAVOTA, G.; TUFANO, M.; MORAN, K.; PENTA, M. D.; VENDOME, C.; BERNAL-CÁRDENAS, C.; POSHYVANYK, D. Enabling mutation testing for android apps. In: ACM. **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. [S.l.], 2017. p. 233–244. Citado na página 43.

LIPTON, R. J.; SAYWARD, F. G. The status of research on program mutation. In: FORT LAUDERDALE FLORIDA, USA. **Digest for the Workshop on Software Testing and Test Documentation**. [S.l.], 1978. p. 355–373. Citado na página 44.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Mujava: an automated class mutation system. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 15, n. 2, p. 97–133, 2005. Citado na página 44.

MA, Y.-S.; OFFUTT, J.; KWON, Y.-R. Mujava: A mutation system for java. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: ACM, 2006. (ICSE '06), p. 827–830. ISBN 1-59593-375-1. Disponível em: <http://doi.acm.org/10.1145/1134285.1134425>. Citado nas páginas 26, 39, 44, 65 e 83.

MACIEL, A. C. **Avaliação da qualidade de oráculos de teste utilizando mutação**. Tese (Doutorado) — Universidade de São Paulo, 2017. Citado na página 45.

MADEYSKI, L.; ORZESZYNA, W.; TORKAR, R.; JOZALA, M. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 1, p. 23–42, 2014. Citado nas páginas 25 e 39.

MADEYSKI, L.; RADYK, N. Judy–a mutation testing tool for java. **IET software**, IET, v. 4, n. 1, p. 32–42, 2010. Citado na página 43.

MADIRAJU, P.; NAMIN, A. S. Paraμ–a partial and higher-order mutation tool with concurrency operators. In: IEEE. **2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2011. p. 351–356. Citado na página 44.

MALDONADO, J. C. *et al.* Critérios potenciais usos: Uma contribuição ao teste estrutural de software. **(Publicação FEE)**, [sn], 1991. Citado na página 38.

MATEO, P. R.; USAOLA, M. P.; OFFUTT, J. Mutation at system and functional levels. In: IEEE. **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.], 2010. p. 110–119. Citado na página 43.

MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing. In: IEEE. **1991 The Fifteenth Annual International Computer Software & Applications Conference**. [S.l.], 1991. p. 604–605. Citado na página 49.

MATHUR, A. P.; WONG, W. E. Evaluation of the cost of alternate mutation strategies. **VII Simpósio Brasileiro de Engenharia de Software**, p. 320–334, 1993. Citado nas páginas 26, 48, 49, 52 e 66.

McCabe, T. J. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, Dec 1976. ISSN 0098-5589. Citado na página 37.

MIRSHOKRAIE, S.; MESBAH, A.; PATTABIRAMAN, K. Efficient javascript mutation testing. In: IEEE. **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.], 2013. p. 74–83. Citado na página 44.

____. Guided mutation testing for javascript web applications. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 5, p. 429–444, 2015. Citado na página 44.

MUBPEL - WS-BPEL Testing Tools - Redmine. 2017. <https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL/>. (Accessed on 02/28/2019). Citado na página 44.

MUTATEPY - | Pierre-Cyrille Héam. 2017. <http://members.femto-st.fr/pierre-cyrille-heam/mutatepy>. (Accessed on 02/28/2019). Citado na página 44.

MYERS, G.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. Wiley, 2011. (ITPro collection). ISBN 9781118133156. Disponível em: <https://books.google.com.br/books?id=GjyEFPkMCwcC>. Citado nas páginas 23, 30, 32, 33, 34 e 37.

NAMIN, A. S.; ANDREWS, J. H.; MURDOCH, D. J. Sufficient mutation operators for measuring test effectiveness. In: ACM. **Proceedings of the 30th international conference on Software engineering**. [S.l.], 2008. p. 351–360. Citado na página 49.

NESTER - free software that helps to do effective unit testing in C#. 2013. <http://nester.sourceforge.net/>. (Accessed on 02/28/2019). Citado na página 44.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 5, n. 2, p. 99–118, abr. 1996. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/227607.227610>. Citado nas páginas 26, 49, 50, 52 e 66.

OFFUTT, A. J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. **Software testing, verification and reliability**, Wiley Online Library, v. 7, n. 3, p. 165–192, 1997. Citado na página 63.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: **Proceedings of the 15th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. (ICSE '93), p. 100–107. ISBN 0-89791-588-7. Disponível em: <http://dl.acm.org/citation.cfm?id=257572.257597>. Citado nas páginas 26, 49, 52, 63 e 66.

OFFUTT, A. J.; UNTCH, R. H. Mutation 2000: Uniting the orthogonal. In: ____. **Mutation Testing for the New Century**. Boston, MA: Springer US, 2001. p. 34–44. ISBN 978-1-4757-5939-6. Disponível em: <https://doi.org/10.1007/978-1-4757-5939-6_7>. Citado nas páginas 25, 41, 47, 48 e 64.

OFFUTT, J.; MA, Y.-S.; KWON, Y.-R. An experimental mutation system for java. **ACM SIGSOFT Software Engineering Notes**, ACM, v. 29, n. 5, p. 1–4, 2004. Citado na página 44.

OMAR, E.; GHOSH, S.; WHITLEY, D. Homaj: A tool for higher order mutation testing in aspectj and java. In: IEEE. **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2014. p. 165–170. Citado na página 43.

PAPADAKIS, M.; JIA, Y.; HARMAN, M.; TRAON, Y. L. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: IEEE PRESS. **Proceedings of the 37th International Conference on Software Engineering-Volume 1**. [S.l.], 2015. p. 936–946. Citado na página 63.

____. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: . [s.n.], 2015. v. 1, p. 936–946. Cited By 74. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84942646591&doi=10.1109%2fICSE.2015.103&partnerID=40&md5=67d0bb7b9f0dc30d3b772caeb23ed0c7>. Citado na página 63.

PAPADAKIS, M.; KINTIS, M.; ZHANG, J.; JIA, Y.; TRAON, Y. L.; HARMAN, M. Chapter six - mutation testing advances: An analysis and survey. In: MEMON, A. M. (Ed.). **Advances in Computers**. Elsevier, 2019, (Advances in Computers, v. 112). p. 275 – 378. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0065245818300305>. Citado nas páginas 25, 38, 39, 41, 42 e 47.

PAPADAKIS, M.; MALEVRIS, N. An empirical evaluation of the first and second order mutation testing strategies. In: IEEE. **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.], 2010. p. 90–99. Citado na página 54.

PARASOFT Insure++ Memory Debugging | Parasoft. 2019. <https://www.parasoft.com/products/insure>. (Accessed on 02/28/2019). Citado na página 43.

PARSAI, A.; MURGIA, A.; DEMEYER, S. Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems. In: SPRINGER. **International Conference on Fundamentals of Software Engineering**. [S.l.], 2017. p. 148–163. Citado na página 43.

PLEXTEST | ITRegister. 2009. <http://www.itregister.com.au/products/plextest>. (Accessed on 02/28/2019). Citado na página 44.

POLO, M.; PIATTINI, M.; GARCÍA-RODRÍGUEZ, I. Decreasing the cost of mutation testing with second-order mutants. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 19, n. 2, p. 111–131, 2009. Citado na página 54.

POLO, M.; TENDERO, S.; PIATTINI, M. Integrating techniques and tools for testing automation. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 17, n. 1, p. 3–39, 2007. Citado na página 45.

PRAPHAMONTRIPONG, U.; OFFUTT, J. Applying mutation testing to web applications. In: IEEE. **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.], 2010. p. 132–141. Citado na página 45.

PRESSMAN, R. S. **Engenharia de Software: Uma Abordagem Profissional**. [S.l.]: AMGH Editora Ltda., 2016. ISBN 9788580555332. Citado nas páginas 23, 29, 30, 31 e 33.

RAPPS, S.; WEYUKER, E. Data flow analysis techniques for test data selection. In: . [s.n.], 1982. p. 272–278. Cited By 85. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84987194972&partnerID=40&md5=9925cc8f46f1d4fe8e1c2408e9f16765>. Citado na página 38.

____. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, SE-11, n. 4, p. 367–375, 1985. Cited By 587. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-0022043004&doi=10.1109%2fTSE.1985.232226&partnerID=40&md5=ef12c556c24e8948d36e62565ce133bf>. Citado na página 38.

SCHULER, D.; ZELLER, A. Javalanche: efficient mutation testing for java. In: ACM. **Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering**. [S.l.], 2009. p. 297–298. Citado na página 43.

SHAHRIAR, H.; ZULKERNINE, M. Music: Mutation-based sql injection vulnerability checking. In: IEEE. **2008 The Eighth International Conference on Quality Software**. [S.l.], 2008. p. 77–86. Citado na página 44.

____. Mutation-based testing of format string bugs. In: IEEE. **2008 11th IEEE High Assurance Systems Engineering Symposium**. [S.l.], 2008. p. 229–238. Citado na página 44.

SMITH, B. H.; WILLIAMS, L. An empirical evaluation of the mujava mutation operators. In: IEEE. **Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)**. [S.l.], 2007. p. 193–202. Citado na página 44.

SOMMERVILLE, I. **Engenharia de software**. PEARSON BRASIL, 2011. ISBN 9788579361081. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>. Citado na página 30.

TANAKA, A. **Equivalence Testing for FORTRAN Mutation System Using Data Flow Analysis.** [S.l.], 1981. Citado na página 43.

TOKUMOTO, S.; YOSHIDA, H.; SAKAMOTO, K.; HONIDEN, S. Muvm: Higher order mutation analysis virtual machine for c. In: IEEE. **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.], 2016. p. 320–329. Citado na página 44.

TUYA, J.; SUAREZ-CABAL, M. J.; RIVA, C. D. L. Sqlmutation: A tool to generate mutants of sql database queries. In: IEEE. **Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)**. [S.l.], 2006. p. 1–1. Citado na página 45.

UNTCH, R. H. Mutation-based software testing using program schemata. In: **Proceedings of the 30th Annual Southeast Regional Conference**. New York, NY, USA: ACM, 1992. (ACM-SE 30), p. 285–291. ISBN 0-89791-506-2. Disponível em: <http://doi.acm.org/10.1145/503720.503749>. Citado nas páginas 45 e 48.

____. Schema-based mutation analysis: A new test data adequacy assessment method. 1997. Citado na página 45.

UNTCH, R. H.; OFFUTT, A. J.; HARROLD, M. J. Mutation analysis using mutant schemata. In: ACM. **ACM SIGSOFT Software Engineering Notes**. [S.l.], 1993. v. 18, n. 3, p. 139–148. Citado na página 45.

____. Mutation analysis using mutant schemata. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 18, n. 3, p. 139–148, jul. 1993. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/174146.154265>. Citado na página 48.

URAL, H.; YANG, B. A structural test selection criterion. **Information Processing Letters**, v. 28, n. 3, p. 157–163, 1988. Cited By 11. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-0024277796&doi=10.1016%2f0020-0190%2888%2990162-7&partnerID=40&md5=b18f8f92a161cb7769aae0ce941c3dd5>. Citado na página 38.

USAOLA, M. P.; ROJAS, G.; RODRÍGUEZ, I.; HERNÁNDEZ, S. An architecture for the development of mutation operators. In: IEEE. **2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.], 2017. p. 143–148. Citado na página 43.

WALSH, T. A.; MCMINN, P.; KAPFHAMMER, G. M. Automatic detection of potential layout faults following changes to responsive web pages (n). In: IEEE. **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2015. p. 709–714. Citado na página 45.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012. Citado na página 66.

WONG, W.; MATHUR, A. P. Reducing the cost of mutation testing: An empirical study. **Journal of Systems and Software**, v. 31, n. 3, p. 185 – 196, 1995. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/0164121294000980>. Citado nas páginas 25, 26, 49, 52, 63 e 66.

WONG, W. E.; MALDONADO, J. C.; DELAMARO, M. E.; SOUZA, S. D. R. S. D. A comparison of selective mutation testing in c and fortran. **Workshop do Projeto Validação e Teste de Sistemas de Operação**, p. 71–84, jan 1997. Citado nas páginas 11, 26, 50, 52 e 66.

WONG, W. E.; MATHUR, A. P. Fault detection effectiveness of mutation and data flow testing. **Software Quality Journal**, v. 4, n. 1, p. 69–83, Mar 1995. ISSN 1573-1367. Disponível em: <https://doi.org/10.1007/BF00404650>. Citado na página 49.

YAO, X.; HARMAN, M.; JIA, Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: ACM. **Proceedings of the 36th International Conference on Software Engineering**. [S.l.], 2014. p. 919–930. Citado na página 54.

ZHANG, L.; GLIGORIC, M.; MARINOV, D.; KHURSHID, S. Operator-based and random mutant selection: Better together. In: IEEE PRESS. **Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.], 2013. p. 92–102. Citado nas páginas 11, 51, 52 e 53.

ZHANG, L.; HOU, S.-S.; HU, J.-J.; XIE, T.; MEI, H. Is operator-based mutant selection superior to random mutant selection? In: ACM. **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1**. [S.l.], 2010. p. 435–444. Citado na página 49.

ZHANG, L.; XIE, T.; ZHANG, L.; TILLMANN, N.; HALLEUX, J. D.; MEI, H. Test generation via dynamic symbolic execution for mutation testing. In: IEEE. **2010 IEEE International Conference on Software Maintenance**. [S.l.], 2010. p. 1–10. Citado na página 43.

ZHOU, C.; FRANKL, P. Mutation testing for java database applications. In: IEEE. **2009 International Conference on Software Testing Verification and Validation**. [S.l.], 2009. p. 396–405. Citado na página 43.

____. Jdama: Java database application mutation analyser. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 21, n. 3, p. 241–263, 2011. Citado na página 43.

ZIP of files used to build the Csaw mutation tool. 2007. <http://www.skicambridge.com/papers/Csaw_v1_files.html>. (Accessed on 02/28/2019). Citado na página 43.

APÊNDICE

A

OPERADORES DE MUTAÇÃO

Quadro 6 – Operadores de mutação da linguagem Fortran-77

Classe	Sigla	Descrição
cca	aar	referência de *array* para substituição de referência de *array*
	acr	referência de *array* para substituição constante
	asr	referência de *array* para substituição de variável escalar
	car	constante para substituição de referência de *array*
	cnr	substituição de nome de *array* comparável
	csr	constante para substituição de variáveis escalares
	sar	variável escalar para substituição de referência de *array*
	scr	escalar para substituição constante
	src	substituição constante de origem
	svr	substituição de variável escalar
pda	abs	inserção de valor absoluto
	aor	substituição de operador aritmético
	crp	substituição constante
	dsa	alterações na instrução de dados
	icr	substituição do conector lógico
	ror	substituição de operador relacional
	uoi	inserção do operador unário
sal	der	substituição final do DO
	glr	substituição de marcação GOTO
	rsr	substituição de instrução RETURN
	san	análise de instrução (substituição por TRAP)
	sdl	instrução de deleção

Fonte: King e Offutt (1991).

Quadro 7 – Operadores de mutação da linguagem C

Classe	Sigla	Descrição
Mutação de Instrução	STRP	Laço em execução de instrução
	STRI	Laço em condição *if*
	SSDL	Instrução de deleção
	SRSR	Substituição de instrução de Retorno
	SGLR	Substituição de marcação goto
	SCRB	Substituição do *continue* por *break*
	SBRC	Substituição do *break* por *continue*
	SBRn	Saída do enésimo nível de enclausuramento
	SCRN	Continuação até o enésimo nível de enclausuramento
	SWDD	Substituição do *while* por *do-while*
	SDWD	Substituição do *do-while* por *while*
	SMTT	Multiplos saltos de laço
	SMTC	Múltiplos saltos de *continue*
	SSOM	Mutação de operador de sequência
	SMVB	Movimento de chave para cima ou para baixo
	SSWM	Mutação de instrução *Switch*
Mutação de Operadores	Obom	Mutação de operadores binários
	Ouor	Mutação de operadores unários
	Oido	Incremento/Decremento
	OLNG	Negação lógica
	OCNG	Negação de contexto lógico
	OBNG	Negação de operador *bitwise*
	OIPM	Mutação de precedência de operador de indicação
	OCOR	Substituição de operador de conversão
Mutação de variável	Vsrr	Substituição de referência de variável escalar
	Varr	Substituição de referência de *array*
	Vtrr	Substituição de referência de estrutura
	Vprr	Substituição de referência de ponteiro
	Vscr	Substituição de componente da estrutura
	Vasm	Mutação de índice de referência de *array*
	Vdtr	Laço de domínio
	Vtwd	Mutação de giro
Mutação de constante	Crcr	Substituição de constante requerida
	Cccr	Substituição de constante por constante
	Ccsr	Substituição de constante por escalar

Fonte: Agrawal *et al.* (1989).

Quadro 8 – Operadores de mutação da linguagem Java

Classe	Sigla	Descrição
Nível de método	AOD	Exclusão de operador aritmético
	AOI	Inserção de operador aritmético
	AOR	Substituição de operador aritmético
	ASR	Substituição de operador de atribuição
	COD	Exclusão de operador condicional
	COI	Inserção de operador condicional
	COR	Substituição de operador condicional
	LOD	Exclusão de operador lógico
	LOI	Inserção de operador lógico
	LOR	Substituição de operador lógico
	ROR	Substituição de operador relacional
	SOR	Substituição de Operador de Mudança
Nível de classe	EAM	Alteração no método de acesso
	EMM	Alteração no método de modificação
	EOA	Substituição de atribuição de referência e valor
	EOC	Substituição de atribuição de referência e valor
	IHD	Ocultar Exclusão de variável
	IHI	Ocultar inserção de variável
	IOD	Sobrescrita de método de deleção
	IOP	Sobrescrita de posição de chamada de método
	IOR	Sobrescrita de renomeio de método
	IPC	Chamada explícita da Exclusão de construtor pai
	ISD	Exclusão da palavra-chave *super*
	ISI	Inserção da palavra-chave *super*
	JDC	Criador padrão suportado por Java
	JID	Exclusão de inicialização de variável de membro
	JSD	Exclusão de modificador estático
	JSI	Inserção de modificador estático
	JTD	Exclusão da palavra-chave *this*
	JTI	Inserção da palavra-chave *this*
	OAC	Mudança de ordem de argumento
	OMD	Exclusão de sobrecarga de método
	OMR	Alteração no conteúdo do método de sobrecarga
	PCC	Alteração do tipo de conversão
	PCD	Inserção de operador de conversão de tipo
	PCI	Inserção de operador de conversão de tipo
	PMD	Declaração de variável de instância com tipo de classe pai
	PNC	Nova chamada de método com tipo de classe filho
	PPD	Declaração de variável de parâmetro com tipo de classe filho
	PRV	Atribuição de referência com outro tipo compatível

Fonte: Ma, Offutt e Kwon (2006).


ICMC
USP
SÃO CARLOS