UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

PAULO HENRIQUE ALVES KAMMRADT

OPERADORES DE MUTAÇÃO PARA MODELOS DE SISTEMAS REATIVOS

Joinville
2022

PAULO HENRIQUE ALVES KAMMRADT

OPERADORES DE MUTAÇÃO PARA MODELOS DE SISTEMAS REATIVOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Dr. Gian Ricardo Berkenbrock

Joinville
2022

AGRADECIMENTOS

Aos meus pais, por possibilitarem toda a minha trajetória.
Ao Prof. Dr. Gian Ricardo Berkenbrock, pela tutoria, compreensão, pelos encorajamentos e por todas as risadas.
À Rayana, pelo carinho, suporte nos momentos mais críticos e por ser a fonte de minha determinação.
Aos meus amigos e companheiros de curso, por batalharem ao meu lado nessa jornada, rumo à graduação.
À minha família, pelo amor incondicional e pela inspiração.
À Universidade Federal de Santa Catarina, por todas as oportunidades oferecidas.

RESUMO

Com a intensa ascensão do desenvolvimento de software, problemas relacionados a correções de bugs se tornam mais complexos. Uma das etapas do desenvolvimento de software é dedicada a testar e corrigir problemas dessa natureza. O processo de teste de um software pode ser extenso, e propenso a erros humanos. Propostas de automatização da geração de casos de teste são estudadas intensamente a décadas. Uma aproximação frequentemente abordada é a de geração de casos de teste com base em máquinas de estados finitas (MEF). Para tornar a validação de softwares mais eficiente, foram criados critérios diversos de geração de casos de teste. Ferramentas implementam tais critérios para possibilitar a geração automatizada de testes em modelos. Este trabalho propõe o desenvolvimento de operadores de mutação de modelo e de um algoritmo de execução de testes unitários de modelos, utilizando da linguagem de programação C++, integrada à ferramenta PerformCharts. Seu principal objetivo foi analisar a qualidade dos testes, gerados a partir de MEFs pelos critérios implementados ao PerformCharts, utilizando de uma análise de mutação. Os operadores de mutação implementados foram os de omissão de arco, omissão de nó e inversão de arco. O algoritmo de execução de testes unitários foi desenvolvido para percorrer uma MEF, tendo como entrada uma sequência de eventos. Os critérios de geração de testes avaliados foram o Switch-Cover, o método UIO e o método DS. Os casos de teste gerados foram aplicados a uma análise de mutação, a fim de verificar sua capacidade de detecção de modelos defeituosos. Foram utilizadas diferentes MEFs para a análise de cada critério. Por fim, os critérios conseguiram detectar todas as falhas inseridas aos modelos, indicando uma boa qualidade de geração de casos de teste à MEFs simples.

Palavras-chave: Teste de software. Operadores de mutação de modelo. Análise de mutação. Máquina de estados finita. PerformCharts.

ABSTRACT

With the intense rise of software development, problems related to bug fixing become more complex. One of the stages of software development is dedicated to testing and fixing problems of this nature. The process of testing software can be extensive and prone to human error. Proposals for automation of test case generation have been studied intensively for decades. One approach, that is often taken, is that of test case generation based on finite state machines (FSM). To make software validation more efficient, various test case generation criteria have been created. Tools implement such criteria to enable automated test generation on models. This paper proposes the development of model mutation operators and an algorithm to execute unit testing of models, using the C++ programming language, integrated with the PerformCharts tool. Its main objective was to analyze the quality of the tests generated from MEFs by the criteria implemented to PerformCharts, using a mutation analysis. The mutation operators implemented were of arc omission, node omission and arc inversion. The unit testing execution algorithm was developed to traverse a MEF with a sequence of events as input. The test generation criteria were the Switch-Cover, the UIO method, and the DS method. The test cases were applied to a mutation analysis to verify their ability to detect faulty models. Different FEM were used for the analysis of each criterion. Finally, the criteria were able to detect all the faults inserted into the models, indicating a good quality of test case generation for simple FEMs.

Keywords: Software testing. Model based mutation operators. Mutation analysis. Finite state machine. Performcharts.

LISTA DE FIGURAS

Figura 1 – Máquina de Mealy. 13
Figura 2 – Máquina de Moore. 14
Figura 3 – Statechart genérico. 15
Figura 4 – Demonstração de estados OR. 16
Figura 5 – Demonstração de estados AND e OR. 16
Figura 6 – Demonstração de statechart com ortogonalidade e transição entre níveis. 17
Figura 7 – Diagrama da linguagem PcML. 19
Figura 8 – Árvore de estilos de geração de testes. 21
Figura 9 – Visão geral do processo de teste de mutação. 24
Figura 10 – Workflow de um teste de mutação baseado em modelos. 26
Figura 11 – Demonstração de operadores de mutação aplicados. 27
Figura 12 – Diagrama do processo de leitura e geração de grafo 30
Figura 13 – Composição simplificada de GraphBase 30
Figura 14 – Chamada de OMM genérico 33
Figura 15 – OMM de omissão de arco. 34
Figura 16 – OMM de inversão de arco. 35
Figura 17 – OMM de omissão de nó. 36
Figura 18 – Modificação simplificada do método de geração de casos de teste. . 41
Figura 19 – MEF para verificação de operadores de mutação de modelos. . . . 42
Figura 20 – Mutantes gerados pela MEF da Figura 19 43
Figura 21 – Mutante de omissão de arco da MEF da Figura 19. 43
Figura 22 – Mutante de omissão de nó da MEF da Figura 19. 43
Figura 23 – Mutante de inversão de arco da MEF da Figura 19 44
Figura 24 – Resultado dos casos de teste da Tabela 6 45
Figura 25 – MEF para análise de mutação do Switch-Cover 46
Figura 26 – Resultado da análise de mutação do Switch-Cover à Figura 25 . . . 46
Figura 27 – Mutante com melhor desempenho da MEF da Figura 25 47
Figura 28 – Testes aplicados ao mutante da Figura 27 47
Figura 29 – MEF para análise de mutação do método DS 48
Figura 30 – Resultado da análise de mutação do método DS à Figura 29 49
Figura 31 – Mutante com melhor desempenho da MEF da Figura 29 49
Figura 32 – Testes aplicados ao mutante da Figura 31 50
Figura 33 – MEF para análise de mutação do método UIO 50
Figura 34 – Resultado da análise de mutação do método UIO à Figura 33 51

Figura 35 – Mutante com melhor desempenho da MEF da Figura 33 51
Figura 36 – Testes aplicados ao mutante da Figura 35 52
Figura 37 – Resultados das análises dos CGCTs da MEF da Figura 19. 53
Figura 38 – Mutante de omissão de arco 1. 66
Figura 39 – Mutante de omissão de arco 2. 66
Figura 40 – Mutante de omissão de arco 3. 66
Figura 41 – Mutante de omissão de arco 4. 66
Figura 42 – Mutante de omissão de arco 5. 67
Figura 43 – Mutante de inversão de arco 1. 67
Figura 44 – Mutante de inversão de arco 2. 67
Figura 45 – Mutante de inversão de arco 3. 67
Figura 46 – Mutante de omissão de nó 1. 67
Figura 47 – Mutante de omissão de nó 2. 67

LISTA DE TABELAS

Tabela 1 – Métodos e atributos da classe MutationTool 31
Tabela 2 – Métodos e atributos da classe GraphBaseMutant 33
Tabela 3 – Métodos da classe GraphTester 37
Tabela 4 – Possíveis resultados definidos por TestResult. 38
Tabela 5 – Métodos e atributos da classe TestGenerator 40
Tabela 6 – Casos de teste aplicados à MEF da Figura 23. 45
Tabela 7 – Resultados das análises de mutação. 54

LISTA DE ABREVIATURAS E SIGLAS

OMM	Operador de Mutação de Modelo
MEF	Máquina de Estados Finita
FM	Ferramenta de Mutação
CGCT	Critério de geração de casos de teste
SUT	System Under Test
GB	GraphBase

SUMÁRIO

1 INTRODUÇÃO . . . 11
1.1 OBJETIVO . . . 12
1.1.1 Objetivo Geral . . . 12
1.1.2 Objetivo Específico . . . 12

2 FUNDAMENTAÇÃO TEÓRICA . . . 13
2.1 MÁQUINAS DE ESTADOS FINITAS . . . 13
2.2 STATECHARTS . . . 14
2.3 LINGUAGENS DE MARCAÇÃO . . . 17
2.3.1 PerformCharts Markup Language (PcML) . . . 18
2.4 TESTE DE SOFTWARE . . . 19
2.4.1 Critérios para Geração de Casos de Teste . . . 22
2.5 TESTES DE MUTAÇÃO . . . 23
2.5.1 Testes de Mutação Baseado em Modelos . . . 25
2.5.2 Operadores de Mutação de Modelos . . . 26
2.6 TRABALHOS RELACIONADOS . . . 28

3 MATERIAIS E MÉTODOS . . . 29
3.1 ESTRUTURAÇÃO DE UM GRAFO NO PERFORMCHARTS . . . 29
3.2 IMPLEMENTAÇÃO DOS OPERADORES DE MUTAÇÃO DE MODELO 30
3.2.1 Ferramenta de Mutação . . . 31
3.2.2 Modelo Mutante . . . 33
3.2.3 Operador de Omissão de Arco . . . 34
3.3 OPERADORES DE MUTAÇÃO DE MODELO . . . 35
3.3.1 Operador de Inversão de Arco . . . 35
3.3.2 Operador de Omissão de Nó . . . 36
3.4 IMPLEMENTAÇÃO DE TESTE DE MODELO POR SEQUÊNCIA DE EVENTOS . . . 37
3.5 ALTERAÇÕES NOS ALGORITMOS DE GERAÇÃO DE CASOS DE TESTE . . . 39

4 RESULTADOS E DISCUSSÕES . . . 42
4.1 VALIDAÇÃO DOS OPERADORES DE MUTAÇÃO DE MODELO . . . 42
4.2 VALIDAÇÃO DOS TESTES DE MODELO POR SEQUÊNCIA DE EVENTOS . . . 44
4.3 ANÁLISE DE MUTAÇÃO DOS GERADORES DE CASOS DE TESTE 45
4.3.1 Análise do Switch-Cover . . . 46

4.3.2 Análise do Método DS **48**
4.3.3 Análise do Método UIO **50**
4.3.4 Análise Comparativa dos Critérios **52**
4.4 DISCUSSÕES 54

5 CONCLUSÕES **56**

REFERÊNCIAS **58**

APÊNDICE A **61**

APÊNDICE B **66**

1 INTRODUÇÃO

O processo de desenvolvimento de um software é composto por diversas etapas, as quais podem ser sucintamente definidas como, levantamento e análise de requisitos, implementação, testes e implantação do produto (SILVA, 2022). Cada etapa se prova indispensável para a qualidade e confiabilidade final do produto, possuindo também características de desenvolvimento distintas (SILVA, 2022).

Apesar das técnicas, métodos e ferramentas empregadas no processo de desenvolvimento de software, ainda podem existir erros no produto final (DELAMARO; MALDONADO; JINO, 2007). Para minimizar o risco de tais erros, técnicas de verificação e validação são aplicadas, como a atividade de teste, sendo essa uma das mais utilizadas para fornecer evidências da confiabilidade do software (DELAMARO; MALDONADO; JINO, 2007).

Idealmente, os testes de verificação de software devem contemplar todas as possíveis permutações de um sistema. No entanto, essa tarefa normalmente se mostra inviável mesmo em sistemas de mais baixa complexidade, podendo haver milhares de combinações entre suas entradas e saídas (MYERS et al., 2004).

Uma possível solução para tal empasse é o desenvolvimento de algoritmos geradores de casos de teste, os quais possuem a capacidade de automatizar a geração de testes através de critérios de verificação pré-definidos. Um exemplar desse tipo de algoritmo foi desenvolvido por Silva (2022) e integrado à ferramenta PerformCharts, utilizando os métodos Switch-Cover, Método UIO (Unique Input/Output) ou ESU (Entrada/Saída Única) e o Método DS (Sequencia Distinta).

Um problema fundamental dos testes decorre da enorme variedade de possíveis falhas de software a serem consideradas (BELLI et al., 2016). Tal fato compromete a confiabilidade do algoritmo de geração de casos de teste, caso ele não seja propriamente verificado.

Uma maneira de averiguar sua integridade é por testes de mutação. Nesses testes, falhas artificiais simples e significativas são implantadas em uma cópia do programa original, utilizando de operadores de mutação, gerando um novo programa denominado como mutante (BELLI et al., 2016).

Caso um mutante e seu programa original produzam resultados diferentes em pelo menos um caso de teste, uma falha é detectada e o mutante é considerado morto. Caso contrário, o mutante é considerado vivo. Um mutante vivo simboliza uma falha do caso de teste de cobrir todos os possíveis defeitos (ZHU; HALL; MAY, 1997).

Considerando o problema dito, foi proposto como objetivo deste trabalho o desenvolvimento e integração de algoritmos de mutação em modelos descrito em

statecharts no PerformCharts. Com isso, almeja-se verificar a qualidade dos casos de teste por abordagens existentes na ferramenta PerformCharts.

1.1 OBJETIVO

Considerando a necessidade de uma melhoria da qualidade nos testes de software, visando o desenvolvimento de um sistema robusto, o trabalho proposto tem os seguintes objetivos.

1.1.1 Objetivo Geral

Analisar a qualidade dos testes gerados a partir de máquinas de estados finitas pela ferramenta PerformCharts, utilizando de operadores de mutação em modelos e de uma análise de mutação simples.

1.1.2 Objetivo Específico

Para este trabalho tem-se como objetivos específicos:

- Estudar os operadores de mutação para modelos;
- Implementar os algoritmos de execução de testes unitários de modelos;
- Implementar os operadores de mutação para os modelos de máquinas de estados finitas;
- Avaliar casos de testes gerados em conjunto com os operadores de mutação de modelo.

2 FUNDAMENTAÇÃO TEÓRICA

Os conceitos e definições, apresentados no presente capítulo, são referentes à utilização de operadores de mutação de modelos à testes de mutação, aplicados em algoritmos de geração de casos de teste para statecharts e Máquinas de Estados Finitas (MEFs).

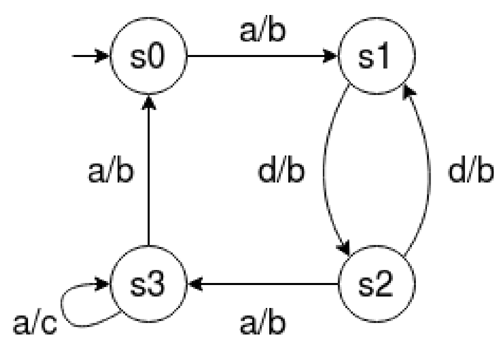
2.1 MÁQUINAS DE ESTADOS FINITAS

As MEFs são abstrações, ou *modelos*, de máquinas de estados concretas, e visam capturar as partes essenciais dos mesmos utilizando de uma estrutura lógica de estados finitos e não vazios. Elas possuem diversas aplicações, podendo ser utilizadas para descrever, por exemplo, sistemas como elevadores, máquinas de vendas e até mesmo softwares (VIEIRA, 2006).

São definidas por um conjunto de elementos, compostos por *estados*, dos quais um é denominado de *estado inicial*, por um conjunto de símbolos de entrada denominado de *alfabeto* e também por *funções de transição*, as quais determinam o próximo estado da MEF usando como base o seu estado atual e seu símbolo de entrada (GRAHAM; JOHNSON, 2014).

As MEFs podem ser classificadas em relação ao seu comportamento de saída, existindo dois modelos distintos das mesmas, as de Moore e as de Mealy, as quais possuem como principal diferença sua dependência com as entradas recebidas. Enquanto nas máquinas de Mealy as saídas dependem tanto de seu estado atual quando da entrada recebida, nas de Moore as mesmas são independentes das estradas, tendo suas saídas unicamente vinculadas ao seu estado atual (FLOYD, 2013).

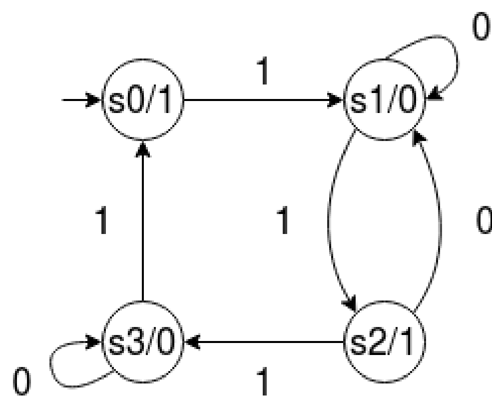
Figura 1 – Máquina de Mealy.



Fonte: elaborado pelo autor.

Na Figura 1 pode-se visualizar um exemplo de uma máquina de Mealy. Já na Figura 2, uma máquina de Moore.

Figura 2 – Máquina de Moore.



Fonte: elaborado pelo autor.

Além dessas, existem outras duas características dignas de nota. A primeira representa seu *determinismo*. Uma máquina de estados é considerada determinística se, para cada par de estados e_1 e e_2, existe no máximo uma transição sob um evento a e, simultaneamente, nenhuma outra transição sob o evento a partindo do estado e_1 (VIEIRA, 2006).

A segunda característica está relacionada a intensidade de sua conexão. Uma MEF pode ser fracamente ou fortemente conectada, dependendo da natureza de suas transições. Se para quaisquer dois estados existir uma sequência de transições que interliguem os mesmos, a MEF é considerada fortemente conectada. Da mesma maneira, caso tal sequência não existir, a MEF é vista como fracamente conectada, pois nem todo estado da mesma oferece a possibilidade de chegar a outro (FANTINATO, 2002).

2.2 STATECHARTS

Um *sistema reativo* é classificado como um modelo que interage *constantemente* com o ambiente, diferentemente de um sistema tradicional, o qual fornece apenas um resultado dada uma entrada (HAREL, 1987). Tal modelo de sistema é, por natureza, complexo em seu desenvolvimento, e em uma tentativa de simplificá-lo, surgiram os *statecharts*.

A denominação de statechart foi desenvolvida por Harel (1987), usada para descrever uma linguagem visual para modelagem e especificação de sistemas. Foi criada com o intuito de descrever estados e transições de forma modular, permitindo agrupamento, ortogonalidade e a fácil locomoção entre níveis de abstração (HAREL, 1987).

Considerado uma extensão da abordagem convencional à diagramas de estado, somado a transições multi-nível e com um mecanismo de comunicação

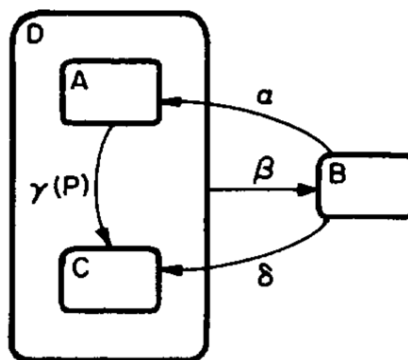
entre componentes simultâneos (HAREL, 1987). Dessa maneira, possibilitam uma representação fiel de um sistema, utilizando de artifícios para a execução de paralelismo, hierarquia e comunicação difusa (LEVI, 2001).

Assim como MEFs, statecharts são baseados em eventos e condições, com a combinação dos mesmos gerando uma transição entre estados. Exitem dois tipos de eventos, sendo esses os de entrada e os de saída, também chamados de ações, as quais ocorrem a partir do estímulo de uma transição (DRUSINSKY; HAREL, 1989).

Os componentes de statecharts podem ser divididos entre aqueles considerados essenciais e optativos. Um exemplo de elementos optativos seriam as ações, condições e probabilidades, em contraste com elementos como estados, eventos e transições, indispensáveis para a fundamentação do mesmo (AMARAL; VELOSO; VIJAYKUMAR, 2005).

A Figura 3, exemplifica o funcionamento de um statechart genérico. Nela, se encontram os estados A, B, e C e, por exemplo, o evento γ, que ocorre no estado A e transfere o sistema para o estado C, mas somente se a condição P for atendida no momento de sua ocorrência (HAREL, 1987).

Figura 3 – Statechart genérico.



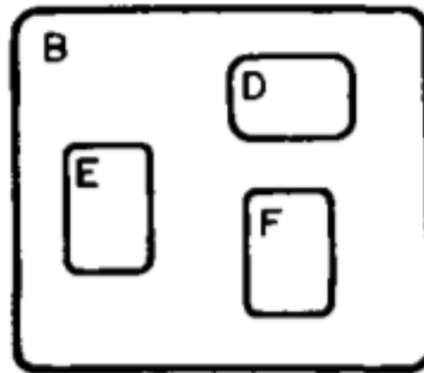
Fonte: (HAREL, 1987, p. 4).

Ainda na Figura 3 o evento β simboliza um evento que levaria o sistema para o estado B partindo tanto de A quanto de C. Dessa maneira, podemos agrupar ambos em um super-estado D, sendo esse uma abstração *exclusive-or* (XOR) dos mesmos, ou seja, para estar no estado D o sistema deve estar no estado A ou no estado C, porém não nos dois.

Tal exemplo também demonstra a representação de paralelismo em statecharts, a qual pode possuir diferentes níveis. A Figura 4 representa uma aglomeração de estados no estilo OR, a qual implica que para o sistema estar no estado B, ele pode estar em qualquer um dos estados D, E ou F.

Já a Figura 5 representa um statechart com aglomeração no estilo AND e OR. Para ele se encontrar no estado A, o mesmo deve estar tanto no estado B quanto no

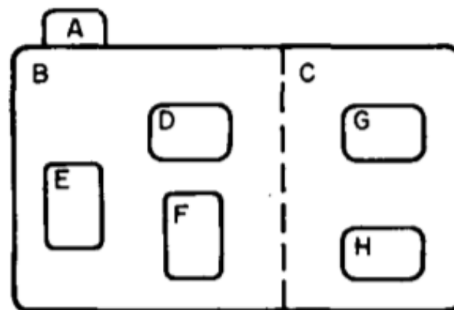
Figura 4 – Demonstração de estados OR.



Fonte: (DRUSINSKY; HAREL, 1989, p. 8).

estado C, representando assim uma arquitetura no estilo AND, e os estados B e C são compostos por uma condição OR de agrupamento. Logo, para o estado do sistema ser A, o estado B deve estar em D, E ou F, e o estado C deve estar em G ou H (SILVA, 2022).

Figura 5 – Demonstração de estados AND e OR.



Fonte: (DRUSINSKY; HAREL, 1989, p. 8).

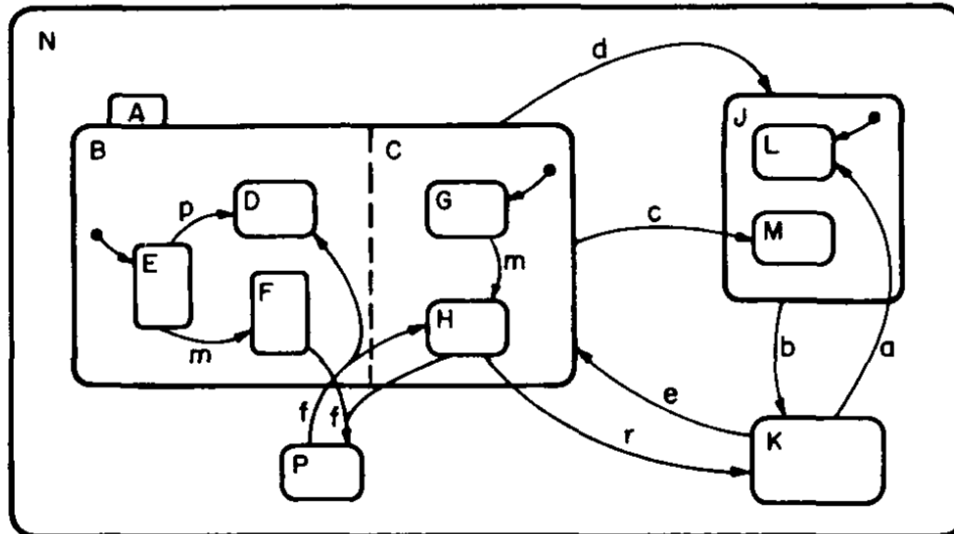
Essas condições de agrupamento definem uma importante parte do sistema de um statechart. Os agrupamentos do tipo OR são ditos como exclusivos e os agrupamentos AND como ortogonais (DRUSINSKY; HAREL, 1989). Dessa maneira, utilizando como exemplo a Figura 5, os estados B e C são ortogonais. Já os estados D, E e F, e os estados G e H são, respectivamente, exclusivos para os estados B e C.

A fim de obter uma visualização mais ampla, traz-se como exemplo a figura 6, na qual é possível verificar importantes recursos de um statechart N.

O sistema pode se encontrar simultaneamente em mais de um estado e também, por um mesmo evento, realizar uma ou mais transições, demonstrando assim a ortogonalidade do sistema. Fato esse pode ser visto na Figura 6, nas transições entre os estados E-F e G-H, ambas ocasionadas pelo evento m.

Além disso, suas transições são independentes do nível dos seus estados, assim demonstrando sua capacidade de realizar transições multi-nível. Isso pode ser observado na Figura 6, na qual ocorre a transição gerada pelo evento f entre os

Figura 6 – Demonstração de statechart com ortogonalidade e transição entre níveis.



Fonte: (DRUSINSKY; HAREL, 1989, p. 8).

estados F-P, ambos localizados em níveis distintos em sua abstração.

Um statechart pode ser também representado por um *grafo*. Segundo Diestel (2007), um grafo consiste em um par ordenado de vértices (correspondem aos estados de uma MEF) V e de um conjunto de arestas (equivalentes às transições de uma MEF) E, as quais conectam os vértices.

Um statechart pode ser novamente abstraído, afim de permitir sua representação no meio da computação. Essa abstração pode ser feita através de linguagens de marcação, possibilitando a um software a leitura, alteração e análise de um statechart (SILVA, 2022).

2.3 LINGUAGENS DE MARCAÇÃO

As linguagens de marcação originalmente surgiram com o aparecimento da Web, na década de 90. Elas possibilitam a construção de padrões, públicos e abertos, permitindo assim maiores avanços no tratamento de informações, minimizando o problema de transferência de um formato de representação para o outro (BAX, 2001).

Uma descrição do princípio de funcionamento das linguagens de marcação foi dado por Bax (2001):

> Estas linguagens identificam, de forma descritiva, cada“entidade informacional” digna de significado presente nos documentos, como, por exemplo, parágrafos, títulos, tabelas ou gráficos. A partir destas descrições, os softwares podem melhor compreender e, em consequência, melhor tratar ou processar a informação contida em documentos eletrônicos (BAX, 2001, p. 33).

Alguns exemplos de linguagens de marcação são: Standard Generalized

Markup Language (SGML), HyperText Markup Language (HTML), eXtensible Markup Language (XML), PerformCharts Markup Language (PcML) entre outras. Para essas linguagens, um documento é composto por três partes, seu conteúdo, sua estrutura e seu estilo.

O conteúdo de um documento é visto como sua informação propriamente dita, a estrutura é o que define como a organização das informações, ou das ideias, se apresentam no documento, e o estilo define a apresentação das informações do mesmo para o usuário (BAX, 2001). Essa distinção promove uma simplificação, possibilitando ao autor do código focar-se exclusivamente ao conteúdo e estrutura do documento.

Cada linguagem de marcação possui especificações voltadas para suas aplicações, as quais podem ser significativamente distintas umas das outras. Um exemplo disso seriam as linguagens HTML e XML, ambas baseadas na SGML, porém com objetivos diferentes.

Enquanto no HTML são utilizadas tags e atributos para representar formas em um navegador, no XML essas tags são utilizadas para organizar um conjunto de dados, deixando a sua interpretação para a aplicação que fizer sua aquisição (AMARAL; VELOSO; VIJAYKUMAR, 2005).

O desenvolvimento do XML é realizado pelo World WideWeb Consortium (W3C), sendo visto como uma gramática usada para definir e descrever estruturas de dados, pois não considera como essas estruturas devem ser criadas ou manipuladas (BATES, 2003). Possui como principais vantagens ser extensível, independente de plataforma e reunir esforços internacionais para sua padronização (AMARAL; VELOSO; VIJAYKUMAR, 2005).

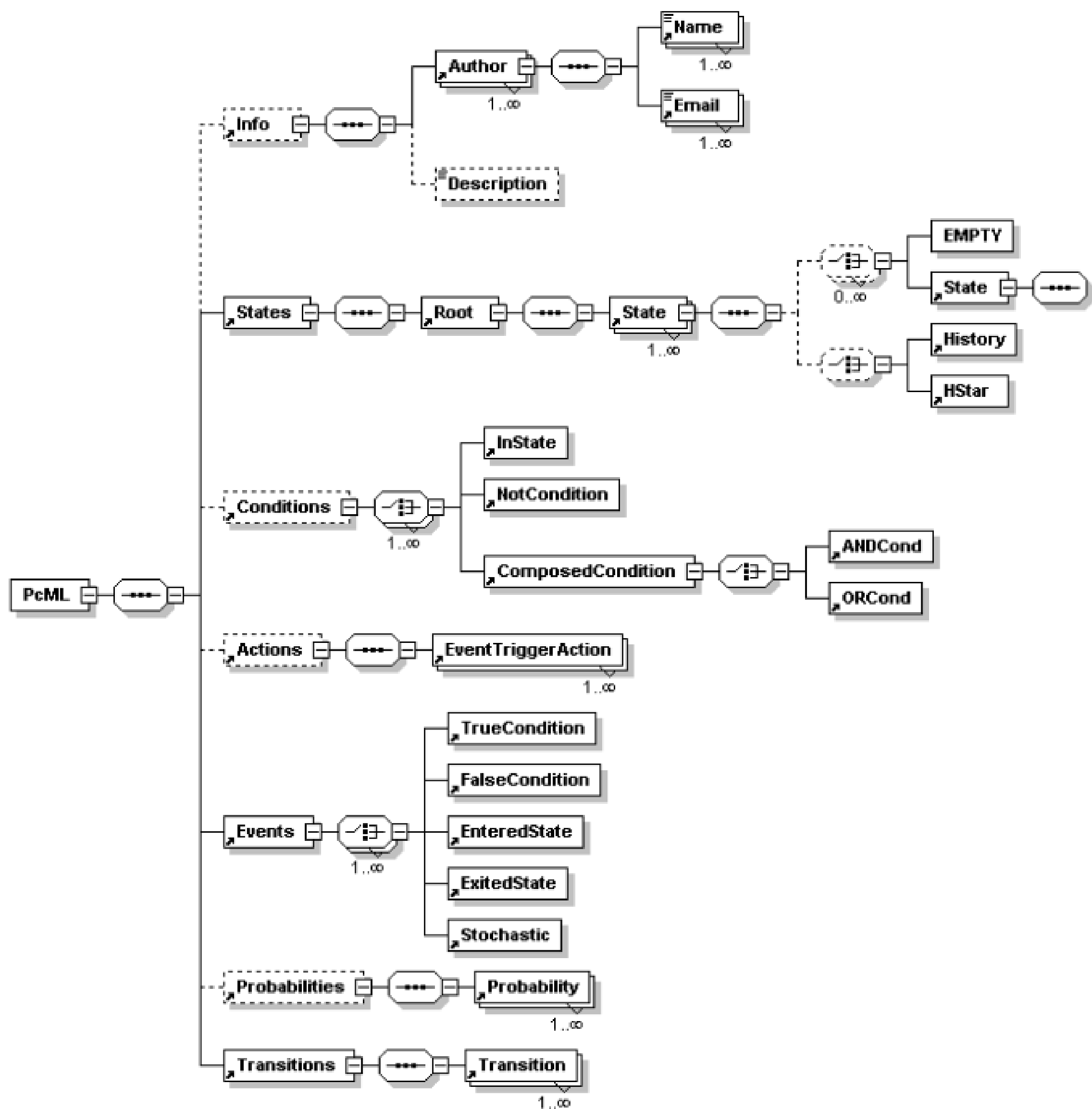
2.3.1 PerformCharts Markup Language (PcML)

Para suprir a necessidade de analisar statecharts de maneira computacional, foi desenvolvido o programa *PerformCharts*. Tal programa utiliza de uma linguagem de marcação própria, desenvolvida para suprir as necessidades específicas de sua aplicação, a qual foi nomeada de PerformCharts Markup Language (PcML).

PcML é uma linguagem baseada no XML, cujas tags, atributos e outras características representam os elementos utilizados em statecharts para especificar sistemas reativos (AMARAL; VELOSO; VIJAYKUMAR, 2005). Na figura 7, são apresentadas tais características por meio de um diagrama.

Como pode ser visto, o diagrama é composto por linhas sólidas e pontilhadas. As linhas pontilhadas representam componentes não mandatórios nas especificações de alguns sistemas, enquanto as sólidas são consideradas indispensáveis (AMARAL; VELOSO; VIJAYKUMAR, 2005). Ainda na Figura 7, nota-se a indicação "$\rightarrow$", destacando os elementos permitidos entre as tags de início e fim das tags de um nível acima (SILVA, 2022).

Figura 7 – Diagrama da linguagem PcML.



Fonte: (AMARAL; VELOSO; VIJAYKUMAR, 2005, p. 11).

Utilizando da linguagem de marcações PcML é possível representar um statechart em um meio computacional, tendo uma forma de descrever seu funcionamento e, consequentemente, torna-se possível implementar testes de software no mesmo.

2.4 TESTE DE SOFTWARE

O teste de software é o processo de excitar um sistema de software usando uma variedade de entradas, com a intenção de validar seu comportamento e descobrir falhas. Estas falhas, também conhecidas como bugs ou defeitos, podem gerar erros

críticos em um software (CHA; TAYLOR; KANG, 2019).

Falhas podem ser geradas devido a erros de lógica ou codificação, muitas vezes sendo relacionados à má interpretação das exigências do usuário, e são em geral inevitáveis devido à natureza humana dos programadores. O objetivo do teste é encontrar tais falhas, sendo um bom teste aquele com uma alta probabilidade de encontrar um erro (PRESSMAN, 2014).

Os conceitos de defeito, erro e falha foram definidos pela IEEE... (2010), e se mostram importantes para a aplicação dos testes. Uma descrição desses princípios foi dada por Silva (2022):

> Um defeito é caracterizado por uma imperfeição ou deficiência no ambiente de trabalho que difere dos requisitos e funcionalidades esperadas. A partir desse defeito, um erro é oriundo de uma ação humana que produz um resultado incorreto. Quando um erro é encontrado, esse pode gerar uma falha, sendo essa um evento em que um sistema ou componente do sistema não executa uma função necessária nos limites especificados (SILVA, 2022, p. 22).

A qualidade do software está fortemente relacionada aos seus testes. A partir das informações obtidas no desenvolvimento dos testes é possível fazer uma estimativa de confiabilidade e manutenção do software. No entanto, por serem uma atividade muito complexa, os testes são separados em fases com objetivos distintos, sendo esses o teste de unidade, o teste de integração e o teste de sistemas (DELAMARO; MALDONADO; JINO, 2007).

O teste de unidade, ou teste unitário, tem como foco as menores unidades de um programa. Com eles, espera-se que sejam identificados os erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas ou erros de programação. Cada unidade é testada separadamente, e pode ser aplicado à medida que ocorre a implementação das unidades (DELAMARO; MALDONADO; JINO, 2007).

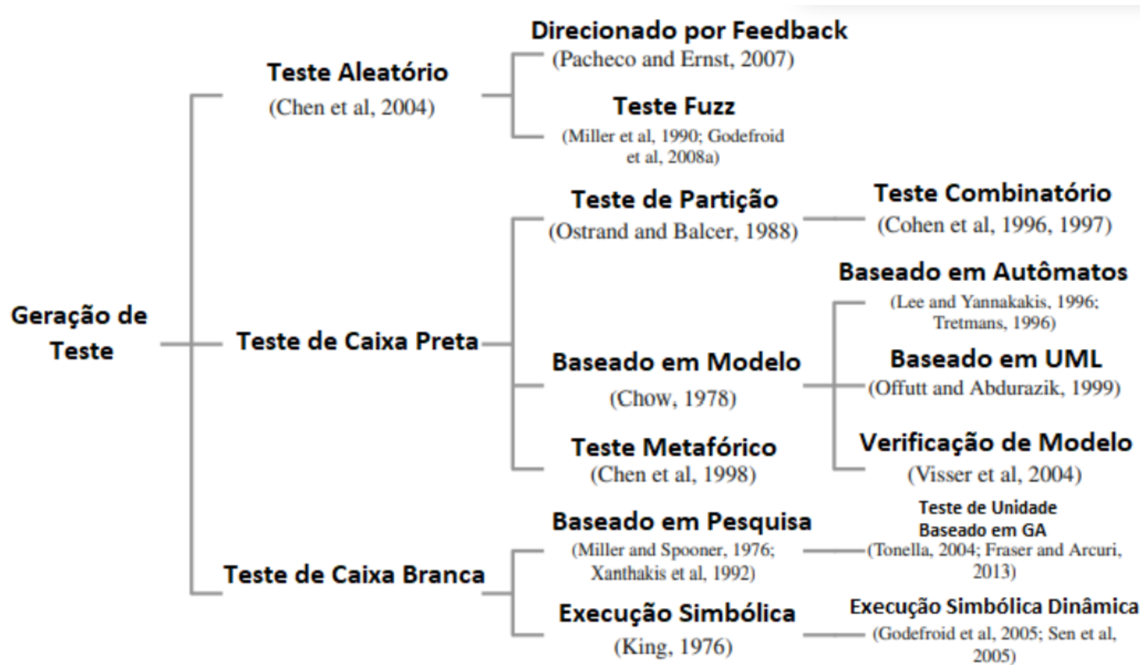
No teste de integração, enfatiza-se construção da estrutura do sistema. Conforme as diversas partes do software são colocadas para trabalhar juntas, surge a necessidade de verificar se a interação entre elas funciona da maneira esperada. Sua eficiência depende do conhecimento das estruturas internas e interações existentes entre as partes do sistema (DELAMARO; MALDONADO; JINO, 2007).

Para se obter o sistema completo e integrado, inicia-se o teste de sistema. O objetivo é averiguar o cumprimento das funcionalidades especificadas nos documentos de requisitos. Aspectos de correção, completude e coerência devem ser explorados, bem como requisitos não funcionais como segurança, desempenho e robustez (DELAMARO; MALDONADO; JINO, 2007).

Apesar de tal separação, a geração manual de testes pode ser um processo tedioso e sujeito as falhas de origem humana. Considerando a importância de um forte conjunto de testes, foram desenvolvidos maneiras de gerar casos de teste

automaticamente (CHA; TAYLOR; KANG, 2019). A Figura 8 apresenta uma árvore com os tipos de geradores de testes.

Figura 8 – Árvore de estilos de geração de testes.



Fonte: (SILVA, 2022, p. 24).

É possível verificar na Figura 8 a existência de três grupos de geração de testes, sendo essas o teste aleatório, o caixa preta e o caixa branca. Desses, o teste aleatório se mostra o mais simples de ser implementado, mas sua falta de sistemática faz com que, para que o mesmo seja confiável, seja necessário um número elevado de amostras (CHA; TAYLOR; KANG, 2019).

Já o teste da caixa branca refere-se a qualquer técnica de teste que considere o código-fonte do programa. Também conhecidos como teste estrutural, ele tende a utilizar as informações do código para medir a adequação e orientação da geração dos casos de teste (CHA; TAYLOR; KANG, 2019).

Por fim, os testes caixa preta, uma aproximação mais sistemática em relação aos testes aleatórios, porém sem ter conhecimento do código-fonte. Apenas agindo sobre as interfaces e especificações do sistema, esse teste tende a focar em suas funcionalidades implementadas, sendo também conhecido como teste funcional (CHA; TAYLOR; KANG, 2019).

Uma das abordagens do estilo caixa preta são os testes baseados em modelos. Esses testes geram um conjunto de casos de teste a partir de um modelo e das exigências passadas a ele. Existem várias maneiras distintas de percorrer um modelo e gerar casos de testes, logo, alguns critérios para a geração dos mesmos foram desenvolvidos (DALAL et al., 1999).

2.4.1 Critérios para Geração de Casos de Teste

Um critério de teste é visto como uma série de etapas, as quais definem como um modelo de software será percorrido e, consequentemente, quais sequências de testes serão geradas pelo algoritmo (CHA; TAYLOR; KANG, 2019). Os critérios de geração de testes implementados ao programa PerformCharts, por (SILVA, 2022), foram o Switch-Cover, o método UIO e o método DS.

O critério de teste Switch-Cover foi proposto por Pimont e Rault (1976). Ele proporciona uma alta cobertura na MEF aplicada, pois segue, como princípio, a ideia de que todas as transições devem ser verificadas (PIMONT; RAULT, 1976). Apesar disso, possui como característica um desempenho dependente dos métodos usados em sua implementação.

A forma de sua implementação é um ponto crítico desse critério, pois é diretamente relacionada a sua capacidade de lidar com MEFs complexas (SANTIAGO et al., 2006). Isso se reflete também na etapa de geração de casos de teste, pois pode gerar casos redundantes, fazendo com que a quantidade necessária de casos de testes para cobrir todas as transições de um grafo seja muito grande, ao até mesmo inviável (SOUZA et al., 2017).

Outro critério é o método UIO, proposto por Sabnani e Dahbura (1988). Nele, é aplicado a cada estado da máquina uma sequência de identificação de estados, com base em suas entradas e saídas. Essa sequência, denominada de *sequência Única de Estrada/Saída* (UES), é utilizada na geração de casos de teste (SABNANI; DAHBURA, 1988).

Para a geração dessa UES, e consequentemente a aplicação do método UIO, a MEF deve ser determinística, fortemente conectada e mínima. No entanto, Sabnani e Dahbura (1988) ressaltam que mesmo cumprindo todos esses pre-requisitos, pode não existir uma UES para determinada MEF. Nesse caso, algumas adequações da MEF são necessárias.

Por fim, o critério DS, proposto originalmente por Gonenc (1970). Similar ao UIO, o método DS se baseia em uma sequência única de entradas que permitem a identificação de cada vértice em uma MEF (PORTO, 2013). Utilizando de uma Sequência de Distinção (SD), a qual aplica um vértice a partir de uma entrada para gerar uma saída distintiva, assim permitindo determinar qual vértice a MEF se encontra (SILVA, 2022).

Outra similaridade ao método UIO são suas restrições. Para o método ser válido, o mesmo deve ser aplicado a uma MEF determinística, fortemente conectada e mínima (GONENC, 1970). Porém, novamente, para certas MEF a SD pode não existir para todos os vértices, mesmo que todas as restrições sejam acatadas (PORTO, 2013).

Os critérios de geração de casos de teste possuem o intuito de assegurar a

qualidade de um software. Contudo, isso se da somente caso a implementação em software dos critérios seja adequada. Um possível meio de inspecionar a qualidade dos geradores de casos de teste é a implementação de testes de mutação aos mesmos.

2.5 TESTES DE MUTAÇÃO

O teste de mutação, também conhecido como análise de mutantes, é um critério de teste baseado em defeitos, introduzido inicialmente por (ACREE, 1980). Nele, o programa testado é alterado diversas vezes, criando um conjunto de programas alternativos chamados de *mutantes* (DELAMARO; MALDONADO; JINO, 2007).

Os testes de mutação, a fim de se mostrarem eficientes, consideram como realidade duas hipóteses, a do *programador competente* e a do *efeito de acoplamento*. A primeira considera que o programador é competente, portanto, com maior probabilidade de produzir um programa que seja correto ou quase correto. Portanto, um programa incorreto pode ser corrigido com uma pequena modificação em sua sintaxe (ALHWIKEM, 2019).

Já a segunda afirma que os casos de teste, os quais possuem a capacidade de distinguir entre programas marginalmente diferentes, são mais sensíveis ao discernimento de programas com grandes diferenças. Ambas as hipóteses implicam que pequenas modificações nos programas são suficientes para revelar falhas complexas (ALHWIKEM, 2019).

Geralmente, cada mutante se diferencia de seu programa original por uma única pequena mudança semântica, chamada de mutação. Essas mutações são geradas de maneira sistemática e automática, utilizando-se de *operadores de mutação*, os quais aplicam padrões de mutação em todo o decorrer do código (CHA; TAYLOR; KANG, 2019).

Os mutantes gerados pela introdução de exatamente uma mudança em um programa são conhecidos como mutantes de primeira ordem. Já os mutantes de segunda ordem são gerados através de duas mudanças simples. Por fim, os mutantes com mais de uma mudança são conhecidos como mutantes de ordem superior (BELLI et al., 2016).

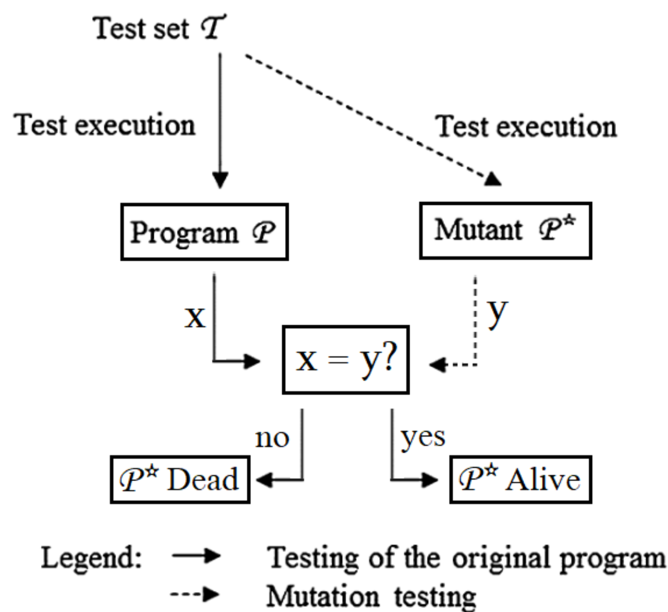
Segundo o efeito de acoplamento, os mutantes de ordem superior provavelmente serão detectados por casos de teste que detectam mutantes de primeira ordem. Portanto, em geral, apenas mutantes de primeira ordem são gerados e usados em testes de mutação (BELLI et al., 2016).

O teste de mutação tem como base a aplicação de um conjunto de testes unitários em um System Under Test (SUT), ou *sistema em teste*, e ao conjuntos de mutantes gerados pelos operadores de mutação. Os resultados dos testes são comparados, e então é analisado a quantidade de mutantes *mortos* pelos testes

(AICHERNIG; JÖBSTL; TIRAN, 2015).

Um mutante é considerado morto se, ao comparar os resultados de todos os testes aplicados a ele e ao SUT, for encontrada ao menos uma discrepância em seus resultados. Caso contrário, o mutante é considerado vivo, indicando a incapacidade do teste de distinguir entre o mutante e o SUT (CHA; TAYLOR; KANG, 2019). A Figura 9 apresenta uma visão geral do processo de teste de mutação baseado em código.

Figura 9 – Visão geral do processo de teste de mutação.



Fonte: adaptado (BELLI et al., 2016, p. 29).

Infelizmente, a verificação do teste de mutação não é tão simples, pois nem todos os mutantes são defeituosos, ou seja, nem todas as falhas injetadas causam discrepâncias observáveis. Por exemplo, operadores de mutação que inserem falhas em locais do código irrelevantes geram mutantes que não podem ser mortos, pois se comportam da mesma maneira que o SUT, e por esse motivo são denominados *mutantes equivalentes* (AICHERNIG; JÖBSTL; TIRAN, 2015).

Os testes de mutação por código sofrem de dois principais defeitos, sendo esses o alto custo computacional e a gestão de equivalências. O primeiro surge quando um conjunto grande de operadores de mutação é aplicado, o que consequentemente produz um grande número de mutantes, podendo exigir um longo do tempo de execução ao ser aplicado um conjunto de teste (ALHWIKEM, 2019).

O segundo exige interferência da ação humana intensiva. Normalmente, o objetivo do teste de mutação é tentar encontrar o maior número possível de mutantes vivos, para assim aperfeiçoar seus casos te teste. No entanto, os mutantes vivos equivalentes não fornecem informações relevantes, e por exigirem esforço humano para serem identificados, consomem muito tempo (ALHWIKEM, 2019).

Uma alternativa para suprir essa deficiência é migrar à aproximação de testes de mutação baseados em modelos, a qual tenta suprir tais necessidades através de operadores de mutação mais abstratos.

2.5.1 Testes de Mutação Baseado em Modelos

O teste de mutação baseado em modelos combina ideias dos testes de mutação com os testes baseados em modelo. Os testes baseados em modelos são uma forma de teste caixa preta, e tentam evitar o trabalho de manualmente escrever centenas de linhas de código através de uma abstração do SUT, em um formato de modelo (AICHERNIG; JÖBSTL; TIRAN, 2015). Isso possibilita aos casos de teste serem automaticamente gerados através do modelo abstraído.

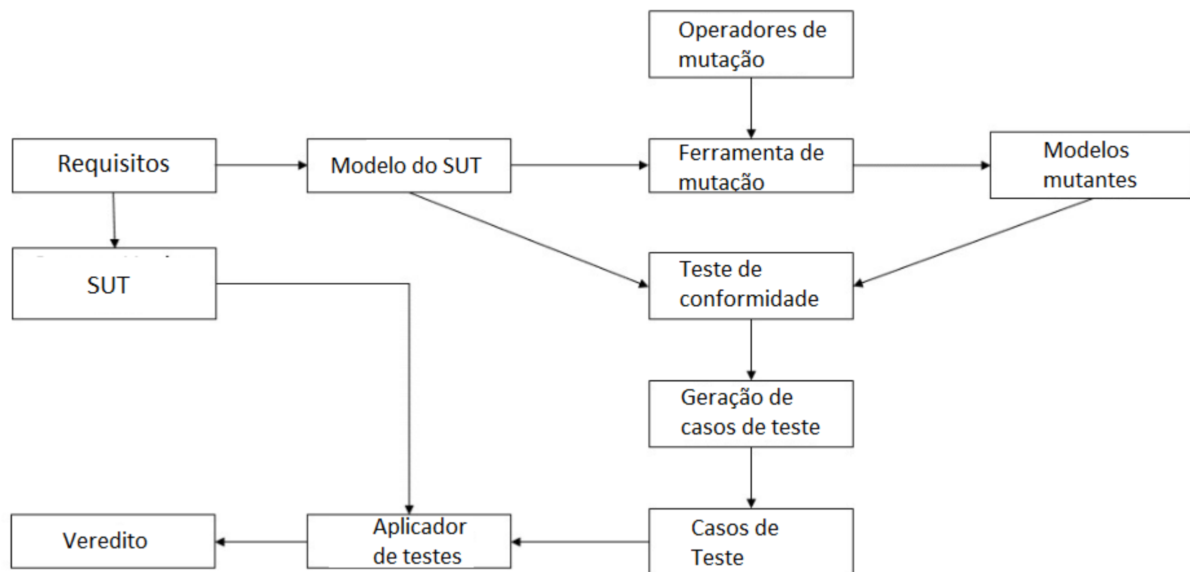
Dessa maneira, o teste de mutação baseado em modelos aplica os conceitos de mutação de código, desenvolvendo operadores de mutação. Os operadores de mutação atuam em cima dos modelos abstraídos do SUT, alterando-o sistematicamente e assim gerando *modelos mutantes*. Aichernig et al. (2014) descreve o princípio do teste de mutação baseado em modelos da seguinte maneira:

> Modelos mutantes representam projetos defeituosos, e o objetivo da abordagem apresentada é gerar casos de teste que podem matar esses projetos defeituosos. [...] busca-se uma sequência de ações que desencadeie comportamentos diferentes entre o modelo original e o modelo mutante. Quando executados no SUT, os testes detectarão se um modelo mutante foi implementado (AICHERNIG et al., 2014, p. 2, nossa tradução).

A Figura 10 apresenta um workflow referente a um teste de mutação de modelo de um SUT. Nele, é possível notar o comportamento do teste de mutação por modelo e suas discrepâncias com o teste de mutação por código. Para exemplificar esse processo, a seguir são apresentadas as etapas executadas pelo mesmo:

1. É gerado um modelo representativo do SUT. Esse é então alterado pela ferramenta de mutação, de acordo com um conjunto pré-definido de operadores de mutação. Uma vez criados todos os mutantes, a geração real do caso de teste começa (LORBER; LARSEN; NIELSEN, 2018).
2. No primeiro passo, o modelo original é comparado com os mutantes. Tradicionalmente, isto é feito através de uma verificação de conformidade entre o original e cada um dos mutantes. Se um mutante e o modelo não possuírem discrepâncias observáveis, ele é classificado como um mutante equivalente, sendo descartado. Caso contrário, o teste de conformidade produz um rastro, cujo é composto do caminho mais curto entre o estado inicial e o estado gerador da não conformidade (LORBER; LARSEN; NIELSEN, 2018).
3. Por fim, os rastros são convertidos em casos de teste, gerando um veredito que pode, ou ser convertido em um formato aceito pela ferramenta de execução

Figura 10 – Workflow de um teste de mutação baseado em modelos.



Fonte: adaptado (LORBER; LARSEN; NIELSEN, 2018, p. 62).

de testes, ou ser transformado em um script executável (LORBER; LARSEN; NIELSEN, 2018).

Seguindo esse processo, certifica-se que se o SUT contiver um bug correspondente a qualquer um de seus mutantes, e o mesmo se comportar de maneira determinística, o conjunto de teste produzido tem a garantia de detectá-lo (LORBER; LARSEN; NIELSEN, 2018).

Visto o workflow de um teste de mutação de modelos, é lógico afirmar que o processo de mutação do modelo é uma etapa imprescindível ao mesmo. Dessa forma, é possível também ressaltar a importância dos operadores de mutação de modelos, já que esses são os elementos que definem as mutações do mesmo.

2.5.2 Operadores de Mutação de Modelos

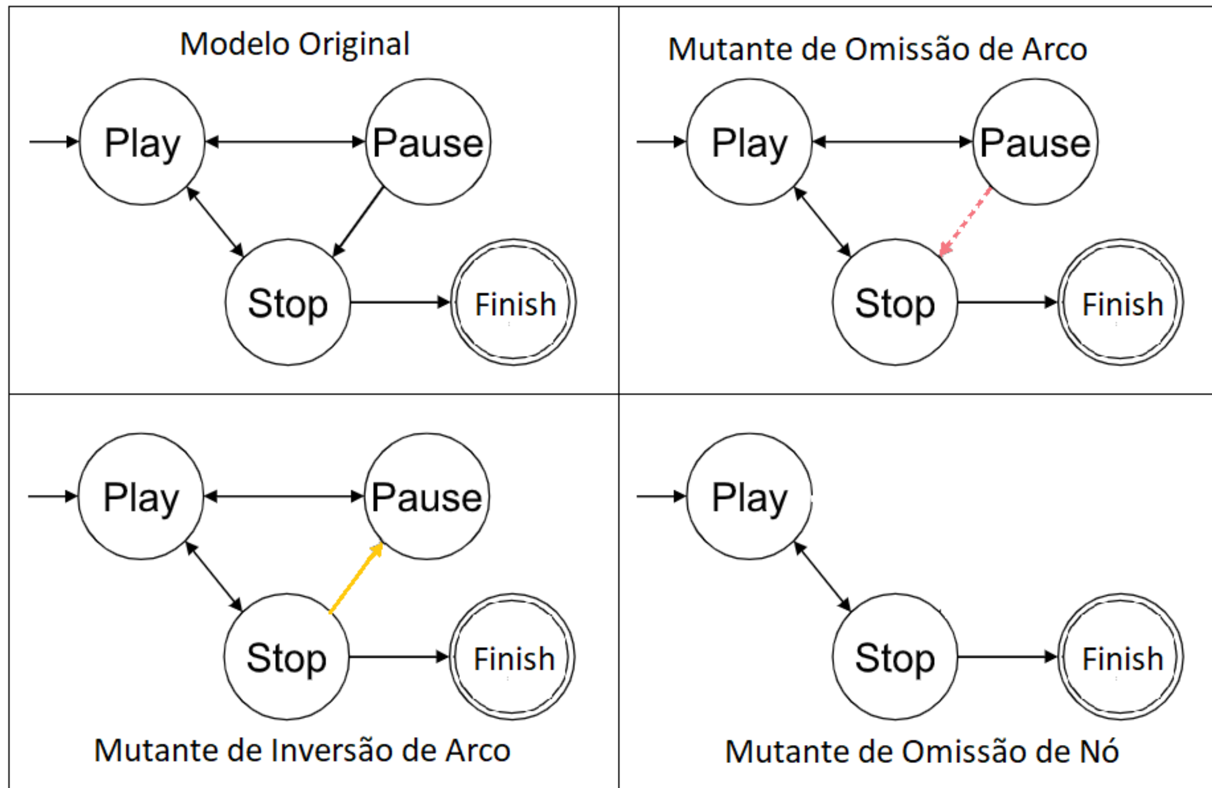
Todas as mutações de modelo são introduzidas por operadores de mutações de modelo, responsáveis por alterá-los de maneira sistemática (AICHERNIG et al., 2014). Cada operador de mutação possui um objetivo distinto, visando a modificação de algum aspecto do modelo aplicado.

Todos os elementos de um modelo podem estar sujeitos a mutações, incluindo nós, arcos, ações, guardas, entre outros. É possível, no entanto, analisar os operadores de mutação de modelo de uma maneira simplificada, resumindo todos os possíveis operadores a uma combinação de dois operadores elementais, sendo esses o de *omissão* e o de *inserção* (BELLI et al., 2016).

Nesse trabalho, foi proposto a implementação de três operadores de mutação de modelos distintos, sendo eles, o de omissão de arco, o de omissão de nó e o

de inversão de arco. A Figura 11 exemplifica o princípio de ação dos operadores mencionados, usando de um modelo genérico como referência.

Figura 11 – Demonstração de operadores de mutação aplicados.



Fonte: adaptado (BELLI et al., 2016, p. 27).

O operador de omissão de arco é derivado direto do operador elemental de omissão, e tem como princípio a remoção de uma única transição entre dois nós de um modelo. Em casos onde a remoção do arco acarreta desacoplamento de um nó do sistema, o operador também remove o nó desacoplado (BELLI et al., 2016).

O operador de omissão de nó também deriva diretamente do operador elemental de omissão, removendo a princípio um único nó do modelo. Nesse caso, o operador também remove todos os arcos previamente conectados ao nó removido, e da mesma maneira que o operador de omissão de arco, todos os nós agora desacoplados do sistema (BELLI et al., 2016).

Vale ressaltar que, operadores de mutação de omissão cuja aplicação gera outras omissões não podem mais serem considerados mudanças de primeira ordem. O resultado disso, considerando o efeito de acoplamento e seus princípios, é uma menor média de eficiência dos operadores de omissão na geração de mutantes significativos quando comparados aos de inserção (BELLI et al., 2016).

Por fim, o operador de inversão de arco, ou operador de corrupção de arco. Em termos de operadores elementais, é visto como uma combinação da omissão seguido de inserção, inserindo um arco com origem e destino invertidos em relação ao arco

omitido. Não possui problemas com relação a sua ordem, porém é particularmente suscetível à geração de mutantes equivalentes, visto o seu comportamento na inversão de arcos com a mesma origem e destino (BELLI et al., 2016).

A forma como os operadores de mutação são descritos e analisados é, em maioria, teórica e analítica, já que se aplicam na mutação de modelos que podem depender de um ambiente de programação específico. Dessa maneira, o método de implementação dos mesmos é deixado a cargo do desenvolvedor.

2.6 TRABALHOS RELACIONADOS

Esse trabalho é uma continuação de uma implementação feita por Silva (2022), a qual propõe e implementa critérios para geração de casos de teste à plataforma PerformCharts. Seu trabalho implementou os critérios Switch-Cover, método DS e método UIO.

É também importante conhecer trabalhos que tratam de estratégias de mutação de modelos, especialmente em relação a gerações de operadores de mutação. Belli et al. (2016) propõem estratégias de composição de operadores de mutação, baseados na combinação de operadores primários cujo denominou de elementais, testando sua eficácia aplicada em sistemas determinísticos.

A maneira cujos operadores de mutação geram mutantes pode ser influenciada pela estratégia de execução do verificador, a fim de facilitar o processo de integração. Aichernig et al. (2014) propõe estratégias para percorrer, verificar e julgar os mutantes gerados, a fim de optimizar a detecção de mutantes vivos.

Como integração, esse trabalho propõe uma ferramenta para a geração de mutações de modelos a partir de três métodos operadores de mutação diferentes, seguido de sua utilização para uma avaliação inicial da qualidade dos geradores de casos de teste, implementados por Silva (2022) ao software PerformCharts. Os operadores de mutação são construídos sobre estrutura base do programa, podendo ser utilizados livremente em futuras aplicações e projetos da plataforma.

3 MATERIAIS E MÉTODOS

Como mencionado anteriormente na seção 2.5.2, a implementação de como um operador de mutação age em cima de um modelo é fortemente ligada à estrutura na qual o modelo é descrito, permitindo uma abertura de implementação do algoritmo por parte do desenvolvedor.

Este capítulo descreve os métodos de implementação dos operadores de mutação de modelo de omissão de arco, omissão de nó e inversão de arco. Da mesma maneira, serão descritas as implementações das ferramentas de teste de modelos, geradas a fim de averiguar os casos de teste gerados por Silva (2022).

Também é descrita a ferramenta na qual os algoritmos desenvolvidos foram implementados, o PerformCharts. Todos os códigos desenvolvidos e integrados à plataforma nesse trabalho seguem a linguagem C++, no dialeto C++17, e conceito de Orientação a Objetos (OO), com auxílio da ferramenta CMake, para compilação e configuração de arquivos.

3.1 ESTRUTURAÇÃO DE UM GRAFO NO PERFORMCHARTS

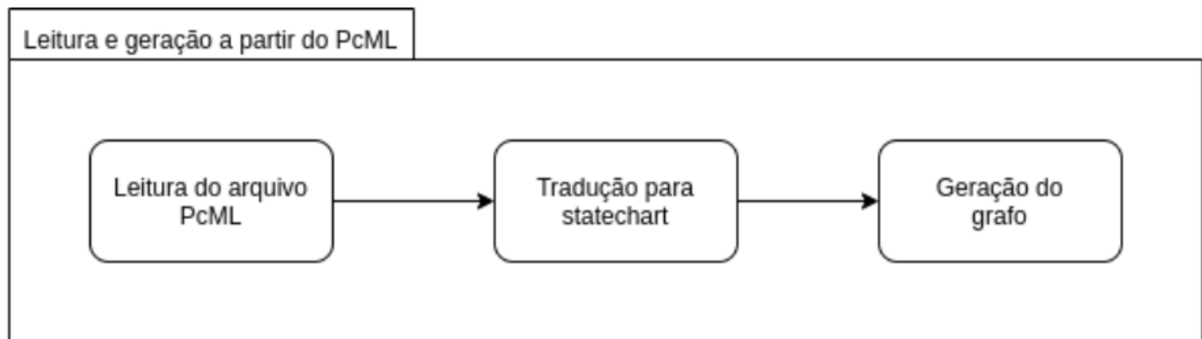
Os algoritmos desenvolvidos nesse trabalho utilizaram, em toda sua extensão, bibliotecas disponibilizadas pelo programa PerformCharts. Esse programa, atualmente *closed source*, também disponibiliza ferramentas para a leitura de uma MEF, a qual é feita utilizando um statechart descrito em um arquivo PcML. A partir desse modelo gerado, o PerformCharts executa funções de criação de estados, transições, eventos, ações e outros elementos de um sistema reativo (AMARAL; VELOSO; VIJAYKUMAR, 2005).

Ainda segundo Amaral, Veloso e Vijaykumar (2005), as transições em um statechart podem ser descritas como *event[condition]/action*. Caso o evento ocorra e a condição seja atendida, é efetuada a transição e após isso a ação é executada. O PerformCharts só possui um tipo de ação, a de evento interno, que ocorre imediatamente após ser estimulada (SILVA, 2022 apud AMARAL; VELOSO; VIJAYKUMAR, 2005).

Após feita a leitura do arquivo em PcML, o PerformCharts utiliza conceitos de orientação a objeto para implementar uma abstração de um statechart, representada pela classe *Statechart*. A classe Statechart é então utilizada para a geração de outra classe, chamada de GraphBase, com o papel de abstrair a estrutura de um grafo, visto na seção 2.2.

A partir dessa etapa, todos os algoritmos de mutação e teste implementados utilizam GraphBase como a estrutura para representar o sistema do modelo. Para

Figura 12 – Diagrama do processo de leitura e geração de grafo

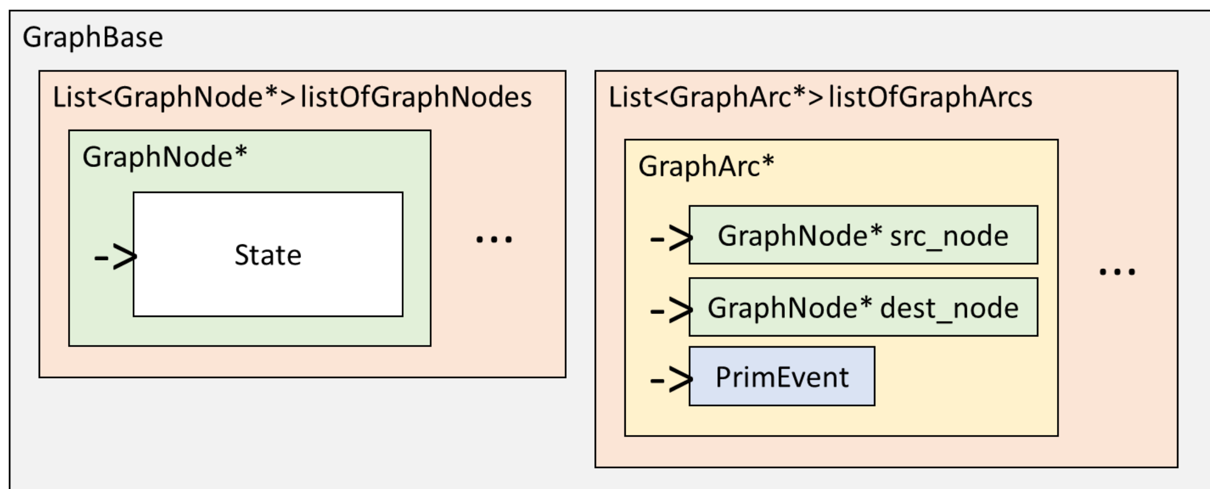


Fonte: (SILVA, 2022, p. 36).

se portar como um grafo, a classe utiliza de abstrações de elementos presentes em modelos, representados pelas classes *GraphNode*, *GraphArc* e *PrimEvent* responsáveis por abstrair, respectivamente, um vértice, uma aresta e um evento.

A Figura 13 apresenta uma simplificação da estrutura da classe GraphBase. Nela, é possível notar uma lista de ponteiros de GraphNode e uma de GraphArc, englobando assim todos os estados e transições do sistema. É também apresentado os conteúdos de GraphNode e GraphArc, o primeiro contendo informações do estado que representa e o segundo apresentando o evento que lhe desencadeia, seu vértice de origem e de destino.

Figura 13 – Composição simplificada de GraphBase



Fonte: elaborado pelo autor.

3.2 IMPLEMENTAÇÃO DOS OPERADORES DE MUTAÇÃO DE MODELO

Esta seção descreve as implementações e concepções dos operadores de mutação de modelo escolhidos: omissão de arco, omissão de nó e inversão de arco. Ela também descreve as classes criadas *MutationTool* e *GrabhBaseMutant*, essas

simbolizando respectivamente a ferramenta de mutação, vista na seção 2.5.1, e os modelos mutantes gerados.

Para fins de simplificação, será usado no restante desse trabalho o termo *OMM* para se referir a um Operador de Mutação de Modelo. Além disso, os modelos mutantes serão referidos apenas como mutantes.

3.2.1 Ferramenta de Mutação

A ferramenta de mutação é representada pela classe *MutationTool*, responsável pelo controle do processo de mutação de um objeto do tipo GraphBase e pela geração de uma lista de mutantes. Todos os OMMs são invocados por métodos dessa classe.

A Tabela 1 apresenta todos os métodos e atributos pertinentes a classe, com os símbolos de + (mais) e - (menos) indicando que o mesmo é público ou privado respectivamente.

Tabela 1 – Métodos e atributos da classe MutationTool

	Atributos e Métodos	Descrição
+	GraphTester(GraphBase)	Construtor da classe.
+	list<GraphBaseMutant> createMutants()	Gera todos os mutantes. Retorna uma lista contendo-os.
+	list<GraphBaseMutant> all_mutants()	Retorna todos os mutantes gerados.
+	list<GraphBaseMutant*> arc_omit_mutants()	Retorna os mutantes de omissão de arco gerados.
+	list<GraphBaseMutant*> arc_invert_mutants()	Retorna os mutantes de inversão de arco gerados.
+	list<GraphBaseMutant*> node_omit_mutants()	Retorna os mutantes de omissão de nó gerados.
-	void create_OmitNodeMutant(NodeIterator)	Invoca OMM de omissão de nó.
-	void create_OmitArcMutant(ArcIterator)	Invoca OMM de omissão de arco.
-	void create_InvertArcMutant(ArcIterator)	Invoca OMM de inversão de arco.
-	GraphBase _gb	GraphBase clonado pelo construtor
-	list<GraphBaseMutant> _all_mutants	Lista de mutantes gerados.
-	list<GraphBaseMutant*> _arc_omit_mutants	Lista mutantes de omissão de arco gerados.
-	list<GraphBaseMutant*> _arc_invert_mutants	Lista de mutantes de inversão de arco gerados.
-	list<GraphBaseMutant*> _node_omit_mutants	Lista de mutantes de omissão de nó gerados.

Nota-se ainda na Tabela 1, no método referente a chamada dos OMMs, a utilização dos iteradores *NodeIterator* e *ArcIterator*. Eles são, respectivamente, apelidos

atribuídos aos tipos *list<GraphNode *>::iterator* e *list<GraphArc *>::iterator*.

No momento de sua instanciação, MutationTool exige como parâmetro de construtor um objeto GraphBase, utilizando uma cópia do mesmo como base para suas mutações. Para fins de explicação, essa cópia será referida a partir deste ponto como *GB*.

Ao ser invocado o método *createMutants*, são aplicados todos as chamadas aos OMMs em cima de GB. Cada chamada gera um novo mutante, adicionado a uma lista interna de MutationTool. O método tenta aplicar os OMMs destinados a arcos e nós em, respectivamente, todos os arcos e nós de GB. O Algoritmo 1 representa o pseudocódigo de *createMutants*.

Algoritmo 1: createMutants()

Dados: Vazio
Resultado: Cria e retorna novos mutantes
$listOfMutants.clear()$
for $node \leftarrow gbNodeFirst$ **to** $gbNodeFinal$ **do**
  $createOmitNodeMutant(node)$
end
for $arc \leftarrow gbArcFirst$ **to** $gbArcFinal$ **do**
  $createOmitArcMutant(arc)$
  $createInvertArcMutant(arc)$
end
return $listOfMutants$

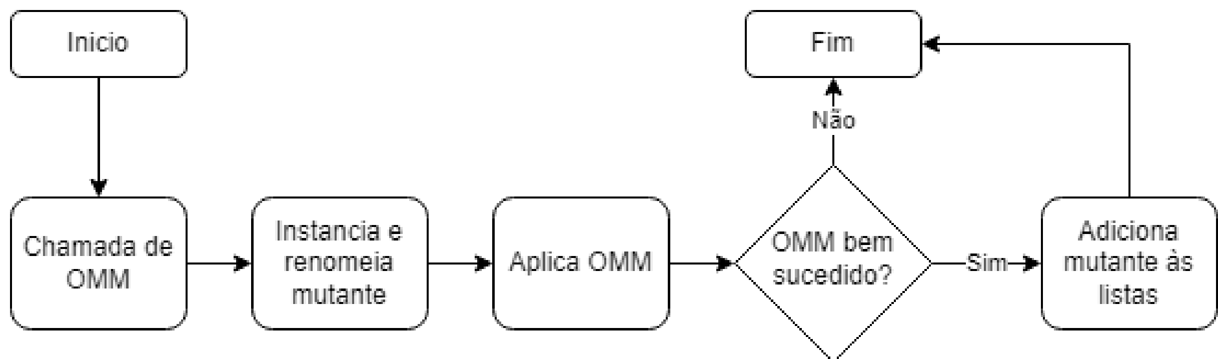
Primeiramente é limpa a lista de mutantes. Após isso, a primeira estrutura de repetição do método percorre todos os nós da lista de nós de GB, utilizando cada um deles como alvo para o OMM de omissão de nó. Já a segunda estrutura de repetição do método realiza um processo semelhante, percorrendo todos os arcos de uma lista de arcos de GB, dessa vez utilizando-os como alvo dos OMMs de inserção e inversão de arco.

Como dito anteriormente, toda chamada de OMM recebe como parâmetro um objeto do tipo *iterador*. Esse iterador aponta para um objeto de uma das listas do GB. Através desse iterador, o OMM é informado qual seu alvo de mutação. O resultado é a criação de um GraphBaseMutant, classe que representa os mutantes. Todo mutante criado pela ferramente de mutação é renomeado conforme o seguinte padrão:

$$"mutant_" + Id_Mutante + Nome_OMM + Id_OMM + Nome_GB$$

Nesse padrão, *Id_Mutante* é o id único do mutante, *Nome_OMM* é a denominação específica dada por cada tipo de OMM, *Id_OMM* é o seu id entre os mutantes gerados pelo mesmo OMM e *Nome_GB* é o nome original do GB. A Figura 14 mostra um fluxograma referente ao processo de chamada de um OMM genérico.

Figura 14 – Chamada de OMM genérico



Fonte: elaborado pelo autor.

Caso o OMM seja bem-sucedido, o mutante resultante é adicionado à lista de mutantes. Nota-se que a ferramenta de mutação faz apenas o papel de gerenciador de mutações, deixando a aplicação dos OMMs para métodos da classe GraphBaseMutant, descrita com mais detalhes na próxima seção.

3.2.2 Modelo Mutante

Um modelo mutante é representado pela classe *GraphBaseMutant*, a qual engloba um objeto do tipo GraphBase, métodos de geração de mutantes e registros de mutação. Ela é responsável por aplicar os OMMs e registrar os arcos e nós modificados. A Tabela 2 apresenta os métodos públicos e privados de GraphBaseMutant utilizados para geração de mutantes.

Tabela 2 – Métodos e atributos da classe GraphBaseMutant

	Atributos e Métodos	Descrição
+	GraphBaseMutant(GraphBase)	Construtor da classe.
+	GraphBase &graph_base()	Retorna GraphBase alocado.
+	void removeGraphArc(ArcIterator)	OMM de omissão de arco. Remove o arco especificado.
+	void removeGraphNode(NodeIterator)	OMM de omissão de nó. Remove o nó especificado.
+	void invertGraphArc(ArcIterator&)	OMM de inversão de arco. Inverte o arco especificado.
-	list<GraphArc*> _omited_arcs	Lista de registro de arcos omitidos
-	list<GraphArc*> _inverted_arcs	Lista de registro de arcos invertidos
-	list<GraphNode*> _omited_nodes	Lista de registro de nós omitidos

Um ponto importante da classe é que ela não se deriva de GraphBase, mas engloba ela. Em seu construtor, recebe um objeto GraphBase e o copia para seu atributo *_graph_base*. Por conta disso, todas os métodos públicos de GraphBase podem ser acessados utilizando do método *graph_base* para acessar o GB.

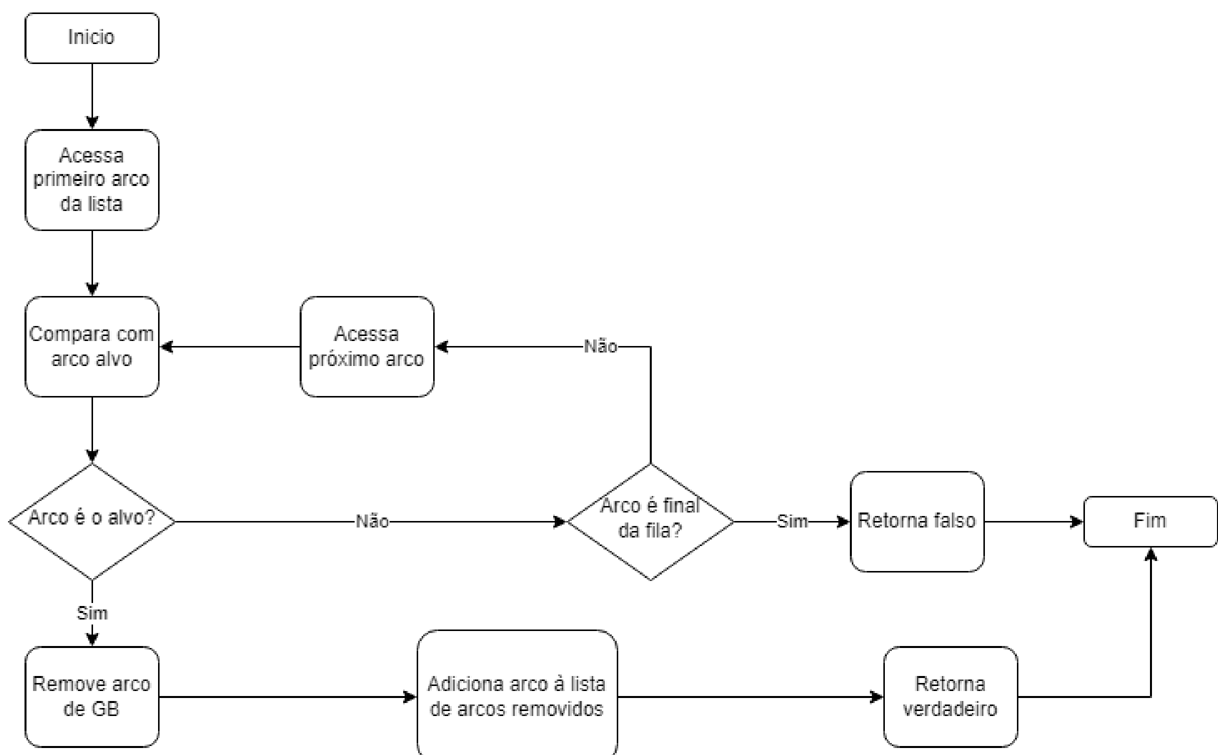
Os métodos mais importantes dessa classe são *removeGraphArc*, *removeGraphNode* e *invertGraphArc*. Responsáveis pelo processo de mutação, é através deles que é feita a integração dos métodos da ferramenta de mutação com o processo de mutação dos modelos. Podem ser considerados parte dos OMMs, e terão seu funcionamento descrito em conjunto com os mesmos nas seções a seguir.

Para fins de simplificação, no restante desse trabalho será utilizado *FM* para se referir à ferramenta de mutação, representada pela classe MutationTool. Além disso, é vinculado o termo *mutante* a GraphBaseMutant.

3.2.3 Operador de Omissão de Arco

O operador de omissão de arco é representado pelo método *removeGraphArc*. O parâmetro do tipo ArcIterator recebido informa ao método um arco alvo, para ser removido. A Figura 15 apresenta um fluxograma descrevendo o funcionamento do OMM.

Figura 15 – OMM de omissão de arco.



Fonte: elaborado pelo autor.

O algoritmo percorre a lista de arcos de GB, comparando cada um de seus elementos ao arco alvo. Caso o alvo seja encontrado, ele é então removido da lista de GB e salvo na lista de arcos removidos do mutante. O método retorna *verdadeiro* caso tenha removido o arco com sucesso, do contrário retorna *falso*.

3.3 OPERADORES DE MUTAÇÃO DE MODELO

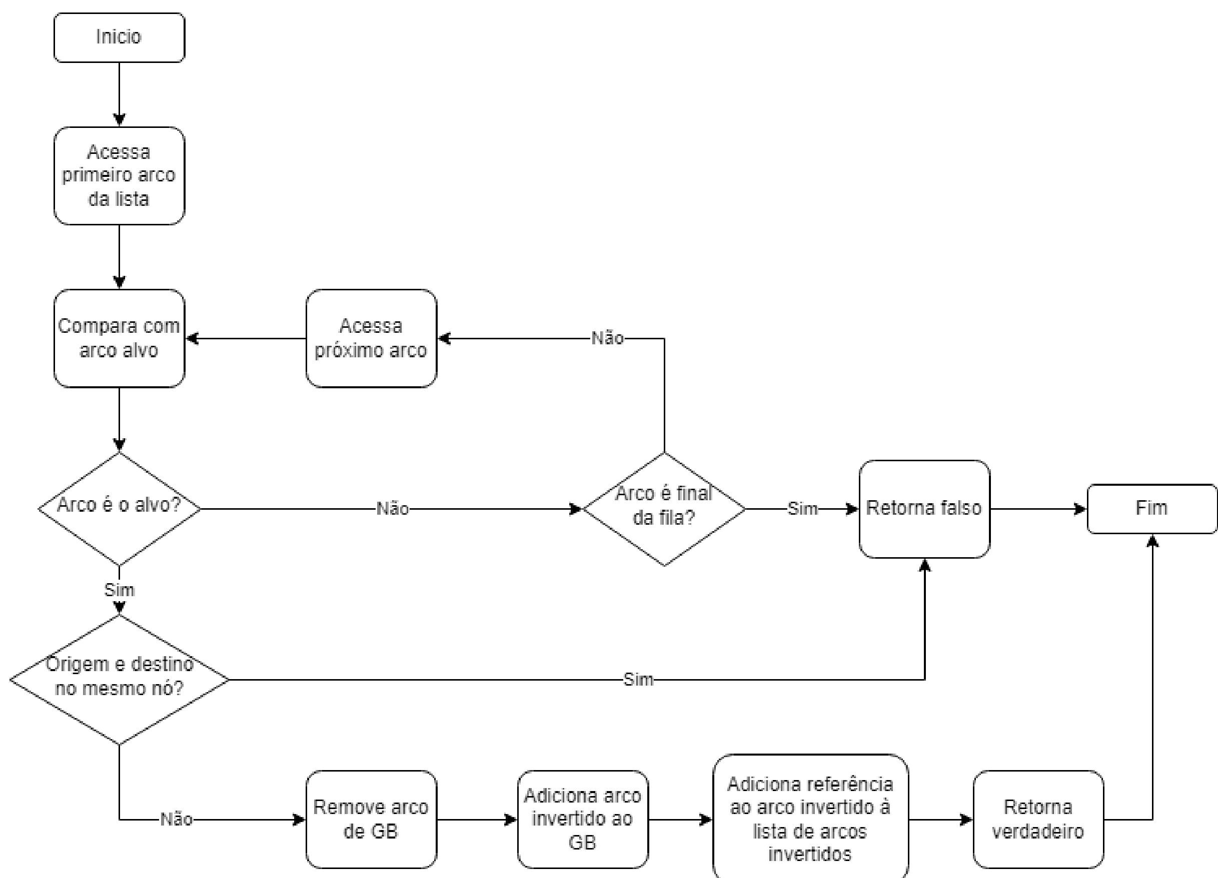
Como mencionado anteriormente, foram implementados três OMMs distintos, sendo esses o de omissão de arco, omissão de nó e inversão de arco. Originalmente, havia sido planejado a implementação de dois OMMs adicionais, os de inserção de arco e inserção de nó. No entanto, ambos foram descartados, devido à atual incapacidade do PerformCharts de identificar quais eventos são válidos para inserção.

Nas seções seguintes, são encontradas as descrições de cada implementação dos OMMs.

3.3.1 Operador de Inversão de Arco

O operador de inversão de arco é representado pelo método *invertGraphArc*. Mais uma vez, ArcIterator informa ao método um arco alvo, dessa vez para ser invertido. A Figura 16 apresenta um fluxograma descrevendo o funcionamento do OMM.

Figura 16 – OMM de inversão de arco.



Fonte: elaborado pelo autor.

Como os operadores de mutação foram implementados seguindo os princípios dos operadores elementais, visto na seção 2.5.2, o processo de inversão/corrupção se torna uma operação de duas etapas. Novamente a lista é percorrida a procura de um

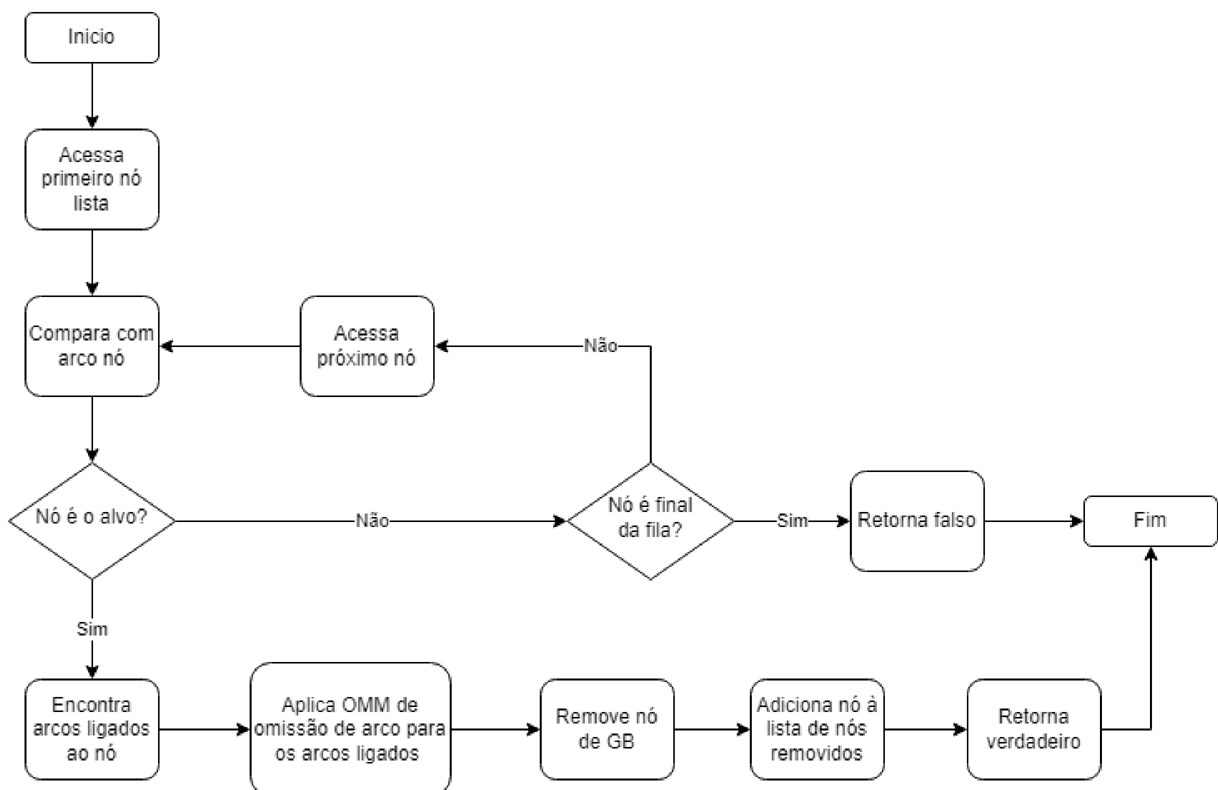
arco alvo, e caso esse seja encontrado e não tiver a origem e destino (operação de inversão redundante) ele é removido da lista de GB.

Após isso, é gerado um arco com parâmetros idênticos ao do removido, com exceção de seus nós de origem e destino, os quais são invertidos. Por fim, o novo arco é inserido na lista de GB, na mesma posição do antigo, e uma referência a ele é inserida na lista de arcos invertidos do mutante. Caso a inversão ocorra é retornado *verdadeiro*, do contrário é retornado *falso*.

3.3.2 Operador de Omissão de Nó

O último OMM, o operador de omissão de nó, é representado pelo método *removeGraphNode*. O parâmetro do tipo NodeIterator recebido informa ao método um nó alvo para ser removido. A Figura 17 apresenta um fluxograma descrevendo o funcionamento do OMM.

Figura 17 – OMM de omissão de nó.



Fonte: elaborado pelo autor.

Como visto na seção 2.5.2, o operador de omissão de nó por natureza não é de primeira ordem. Dessa maneira, ele pode separado em duas etapas, a de omissão de nó e a de omissão de arcos.

Na primeira etapa, o algoritmo percorre a lista de nós de GB, procurando o nó alvo. Caso o nó seja encontrado, ele é então removido da lista de GB e salvo em na lista de nós removidos do mutante. No entanto, também é necessário remover os arcos

ligados previamente ligados ao nó removido.

Assim, na segunda etapa, é percorrida a lista de arcos de GB, a procura daqueles que possuam como origem ou destino o nó omitido. Então, o OMM de omissão de arco é então invocado para cada arco encontrado, assim os removendo. Por fim, é retornado *verdadeiro* caso o nó seja removido, do contrário, retorna-se *falso*.

Com todas as ferramentas de mutação implementadas, é possível continuar com o teste dos geradores de casos de teste, mas para isso se torna necessário algum meio de aplicar os casos de testes gerados aos modelos criados. Por conta disso, foi desenvolvido uma ferramenta de teste de modelos por sequência de eventos.

3.4 IMPLEMENTAÇÃO DE TESTE DE MODELO POR SEQUÊNCIA DE EVENTOS

O objetivo do teste de modelo por sequência de eventos, que será referido como *teste de modelo* pelo resto do texto, é aplicar uma sequência de eventos em um grafo e verificar o seu comportamento. Para representar isso, utilizando das estruturas presentes no PerformCharts, foi desenvolvido uma classe *GraphTester*.

Essa classe tem como premissa navegar nos vértices do grafo, partindo do inicial, utilizando das arestas referentes à lista de eventos recebidos pelos casos de teste. Caso o teste encontre algum problema durante sua execução, ele deve parar. Após o teste ser finalizado, deve ser possível obter o seu resultado e o último vértice visitado pelo teste.

A Tabela 3 apresenta os métodos públicos e privados de GraphTester utilizados para a execução do teste dos modelos.

Tabela 3 – Métodos da classe GraphTester

	Métodos	Descrição
+	GraphTester(GraphBase)	Construtor da classe.
+	TestResult result()	Retorna o resultado do teste.
+	GraphNode* final_state()	Retorna vértice visitado ao final do teste.
+	void test(list<PrimEvent>)	Testa sequência de eventos.

Cada instância de GraphTester é dedicada a testar um único grafo, recebido como seu parâmetro de construção obrigatório. Ela possui três principais métodos, esses sendo *result, final_state* e *test*. O método *final_state* retorna um ponteiro para o último vértice visitado durante o teste, *test* realiza o teste e *result* retorna o seu resultado, cujo é definido por *TestResult*.

A Tabela 4 apresenta os possíveis resultados de GraphTester definidos em TestResult.

Tabela 4 – Possíveis resultados definidos por TestResult.

Resultado	**Descrição**
SUCCESS	Teste bem-sucedido.
INVALID_EVENT	Evento inválido aplicado
NON_DETERMINISTIC_EVENT	Evento não determinístico aplicado
UNKNOWN	Resultado desconhecido

Para o teste, duas situações podem fazer o teste finalizar sem sucesso. A primeira é no caso de um *evento inválido*, a qual ocorre quando algum evento aplicado ao vértice atual não corresponde ao gatilho de nenhuma aresta disponível. Isso simboliza um evento sem efeito, representando uma possível falha no modelo. Nesse caso, o teste tem como resultado *INVALID_EVENT*.

Já a segunda é no caso de *evento não determinístico*. Segundo Silva (2022), os critérios de geração de casos de teste atualmente implementados ao PerformCharts são válidos apenas para modelos determinísticos. Dessa maneira, caso o evento aplicado possibilite o disparo de duas ou mais arestas do mesmo vértice, o teste se encerra e tem como resultado *NON_DETERMINISTIC_EVENT*.

O resultado nomeado de *UNKNOWN* é o retorno padrão do método result() antes de o teste ser realizado. Ele também é retornado no caso de ocorrerem exceções durante a execução do teste, por exemplo, no caso do teste ser vazio. Por fim, caso o teste aplique todos os eventos, percorrendo todos os vértices sem encontrar nenhum problema, o resultado retorna como *SUCCESS*.

Como dito anteriormente, o teste só é executado na chamada do método *test*, o qual aplica os eventos presentes em uma lista de PrimEvent ao grafo GraphBase. Sua implementação pode ser vista no pseudocódigo representado pelo Algoritmo 2.

A primeira estrutura de repetição é responsável por percorrer uma lista de eventos passada ao método. Para cada evento da lista, é feita uma busca dos arcos de GB com origem localizada no nó atual.

A segunda estrutura de repetição percorre todos os arcos encontrados anteriormente, procurando os arcos cujo gatilho seja o evento atualmente em escopo na primeira estrutura de repetição. Sempre que um arco atende essa condição, seu nó de destino se torna um candidato a se tornar o nó atual, e um contador é incrementado.

Após a segunda estrutura de repetição, é feita uma verificação do valor do contador. Caso ele registre zero entradas, significa que nenhum arco correspondeu ao evento aplicado, simbolizando um resultado de *evento invalido*. No entanto, caso o contador registre mais do que uma entrada, mais de um arco correspondeu ao evento aplicado, simbolizando um "evento não determinístico". Ambos os casos simbolizam falhas, tendo o nó atual retornado como estado final.

Por fim, caso o contador registrar exatamente uma entrada, simboliza que o evento é válido e determinístico. Dessa forma, o nó atual é atualizado e a primeira

Algoritmo 2: test(list<PrimEvent>)

Dados: Lista de eventos
Resultado: Testa eventos em modelo, definindo seu resultado e vértice final

```
finalState ← currentNode
counter ← 0
for event ← listOfEventsFirst to listOfEventsFinal do
    arcList ← arcsWithSourceAtCurrentNode
    counter ← 0
    for arc ← arcListFirst to arcListFinal do
        if event == arcEvent then
            counter++
            nextNode ← arc.Destination()
    end
    if counter == 0 then
        finalState ← currentNode
        result ← INVALID_EVENT
        return
    else if counter == 1 then
        currentNode ← nextNode
    else
        finalState ← currentNode
        result ← NON_DETERMINISTIC_EVENT
        return
end
result ← SUCCESS
finalState ← currentNode
```

estrutura de repetição vai para sua próxima iteração. Toda nova iteração dessa estrutura reinicia o contador.

Caso todos os eventos sejam percorridos sem falhas, o resultado é considerado um *sucesso*, e o nó habitado pelo método, em sua última iteração da primeira estrutura de repetição, é atribuído ao *estado final* do teste.

Dessa maneira, se torna possível realizar testes modelos em grafos utilizando de listas de eventos, representadas em C++ por <PrimEvent>. No entanto, os casos de teste gerados pelos critérios de geração de casos de teste do PerformCharts, previamente implementados por Silva (2022), não fornecem nenhuma estrutura de dados desse padrão.

Para solucionar esse empecilho, foram implementadas algumas alterações aos algoritmos de geração de casos de teste, as quais serão detalhadas na próxima seção.

3.5 ALTERAÇÕES NOS ALGORITMOS DE GERAÇÃO DE CASOS DE TESTE

Todos os Critérios de Geração de Casos de Teste (CGCT) gerados por Silva (2022), compostos pelo Switch-Cover, método UIO e método DS, foram implementados separadamente, cada um contido em sua própria classe. No entanto, todos possuem

duas características de implementação em comum, utilizadas como base para as alterações implementadas por esse trabalho.

A primeira característica consiste em sua estrutura de herança. Já que todos os CGCTs são derivados de uma classe base, chamada *TestGenerator*, é possível incorporar facilmente a essa classe funções e atributos comuns a todos os CGCTs.

Já a segunda se refere à forma como os testes são gerados. Segundo (SILVA, 2022), todos os CGCTs possuem um método chamado *generate_test_cases()*, o qual utiliza métodos internos, variando conforme o CGCT, para gerar uma *sequência* de casos de teste em formato de *string*.

Levando esses dois fatores em consideração, foi desenvolvida uma solução baseada na conversão dos casos de teste do formato de string para o de uma lista de listas de eventos, representada em C++ por list<list<PrimEvent>>. A Tabela 5 apresenta todos os novos métodos e atributos implementados a TestGenerator, com os símbolos de + (mais) e # (cerquilha) indicando que o mesmo é público ou protegido, respectivamente.

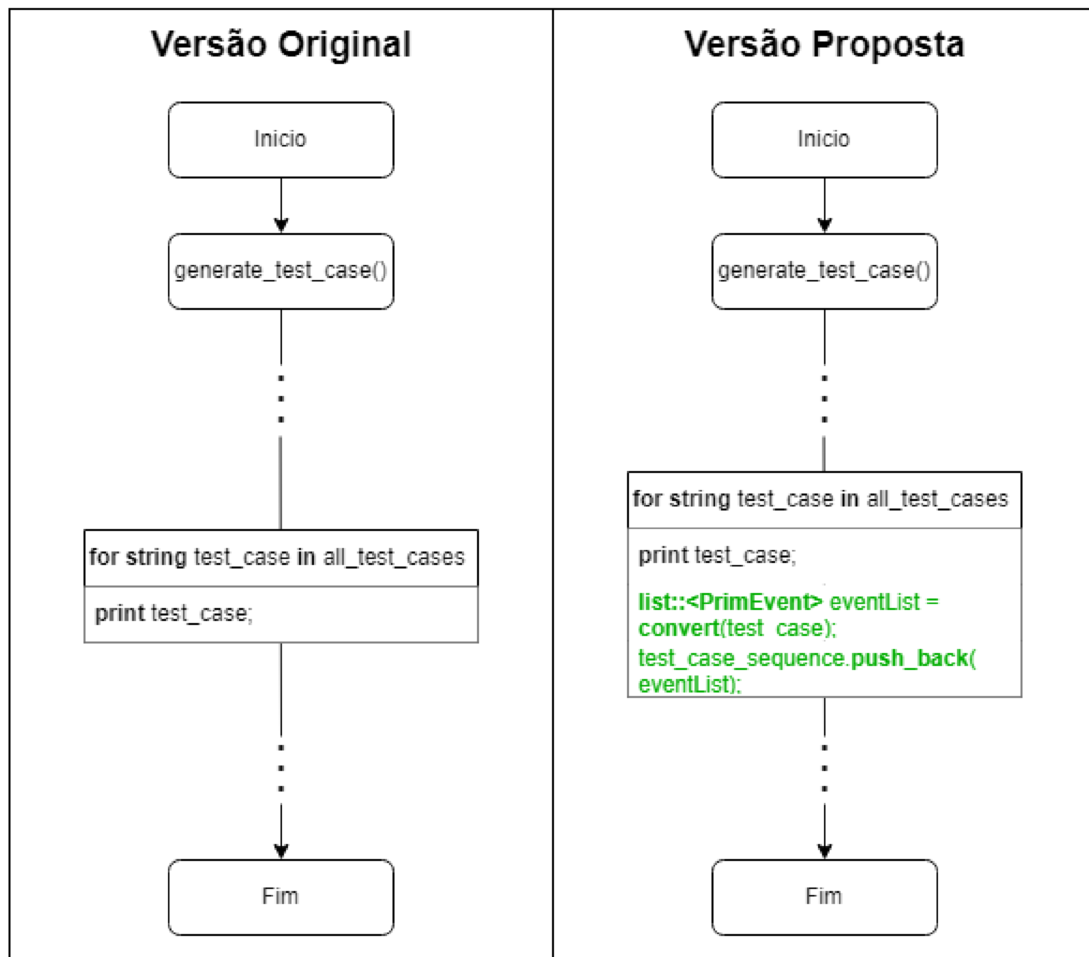
Tabela 5 – Métodos e atributos da classe TestGenerator

	Métodos	Descrição
+	list<list<PrimEvent>> test_cases()	Retorna lista de casos de teste no formato adequado.
+	list<PrimEvent> convert(string)	Converte casos de teste no formato de string para o de lista de eventos.
#	list<list<PrimEvent>> test_case_sequences	Lista de casos de teste.

A inserção desses métodos em cada CGCT teve de ser feita *individualmente*, porém seus princípios de aplicação ao método *generate_test_cases* foram idênticos. Para todo CGCT, *generate_test_cases* eventualmente gera seus casos de teste, individualmente, no formato de string. Nesse momento, se utiliza do método *convert* para criar uma instância de casos de teste, representado por lista de elementos, e então adicioná-la à test_case_sequences.

A Figura 18 demonstra uma simplificação do processo acima, aplicada a um CGCT genérico.

Figura 18 – Modificação simplificada do método de geração de casos de teste.



Fonte: elaborado pelo autor.

Para qualquer um dos CGCTs implementados, a adição dos métodos destacados em verde, na Figura 18, aplicados à string de caso de teste, irá gerar ao final da execução de *generate_test_cases* uma lista de casos de teste, no formato list<list<PrimEvent>>. Essa lista é então acessível através do método *test_cases*.

Dessa forma, se torna possível a utilização dos casos de teste gerados pelos CGCTs nos testes de modelo por sequência de eventos, e assim, averiguar a qualidade dos mesmos ao aplicá-los em modelos mutantes e verificar a quantidade de mutantes vivos, cujos testes foram incapazes de detectar.

4 RESULTADOS E DISCUSSÕES

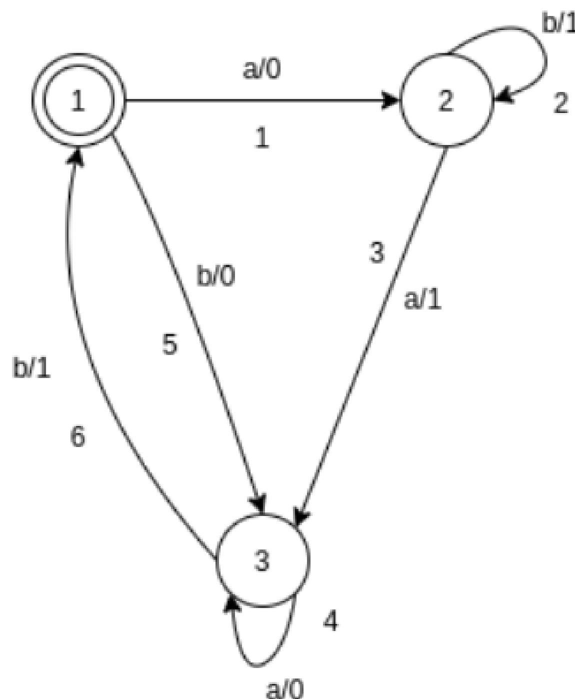
Como resultado, esse trabalho trouxe a implementação de três operadores de mutação de modelos e de algoritmos de execução de testes unitários por sequência de eventos, bem como uma análise de mutação dos geradores de casos de testes presentes no PerformCharts.

Para as validações feitas nas próximas seções, foram usadas MEFs presentes na literatura e utilizadas por Silva (2022) na validação dos geradores de casos de teste do PerformCharts. Os arquivos PcML que representam os statecharts estão no Apêndice A. Os modelos apresentados nesse capítulo foram gerados com estados iniciais simbolizados por círculos duplos.

4.1 VALIDAÇÃO DOS OPERADORES DE MUTAÇÃO DE MODELO

A fim de validar o funcionamento dos operadores de mutação de modelo, foi utilizada a ferramenta de mutação no statechart representante da MEF da Figura 19.

Figura 19 – MEF para verificação de operadores de mutação de modelos.



Fonte: (SILVA, 2022 apud CHOW, 1978, p. 65).

Segundo as premissas da FM, devem ser aplicados OMMs a todo arco e nó compatíveis ao operador. No caso da Figura 19, a qual possui 3 nós e 6 arcos, é esperada a geração um total de 3 mutantes de omissão de nó, 6 de omissão de arco e

4 de inversão de arco[1], resultando em um total de 13 mutantes.

As identificações pertencentes aos mutantes gerados por essa MEF são exibidas pela Figura 20.

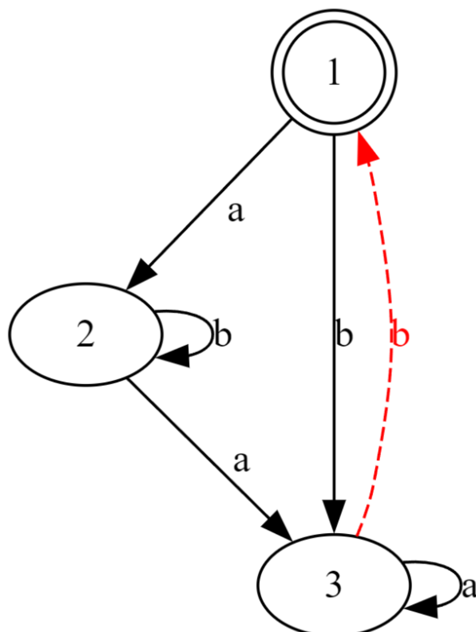
Figura 20 – Mutantes gerados pela MEF da Figura 19

```
mutant_0_omitArc_0_Example_All_Cases
mutant_1_omitArc_1_Example_All_Cases
mutant_2_omitArc_2_Example_All_Cases
mutant_3_omitArc_3_Example_All_Cases
mutant_4_omitArc_4_Example_All_Cases
mutant_5_omitArc_5_Example_All_Cases
mutant_6_invertArc_0_Example_All_Cases
mutant_7_invertArc_1_Example_All_Cases
mutant_8_invertArc_2_Example_All_Cases
mutant_9_invertArc_3_Example_All_Cases
mutant_10_omitNode_0_Example_All_Cases
mutant_11_omitNode_1_Example_All_Cases
mutant_12_omitNode_2_Example_All_Cases
```

Fonte: elaborado pelo autor.

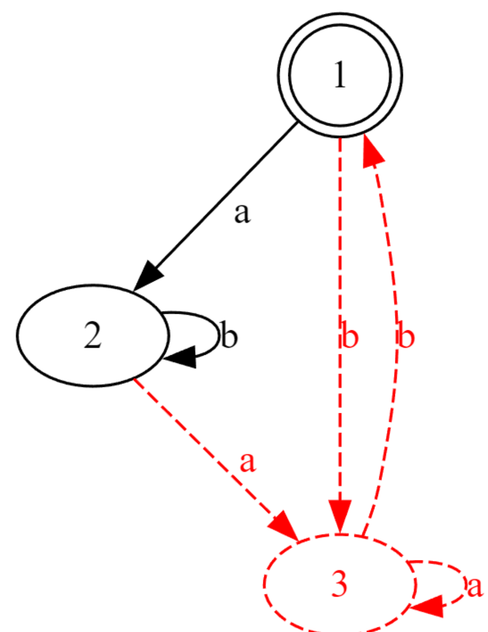
Como esperado, foram aplicados OMMs a todos os elementos possíveis da MEF. As Figuras 21, 22 e 23 demonstram, respectivamente, um mutante da MEF de omissão de arco, um de omissão de nó e um de inversão de arco. Os demais mutantes gerados podem ser encontrados no Apêndice B.

Figura 21 – Mutante de omissão de arco da MEF da Figura 19.



Fonte: elaborado pelo autor.

Figura 22 – Mutante de omissão de nó da MEF da Figura 19.

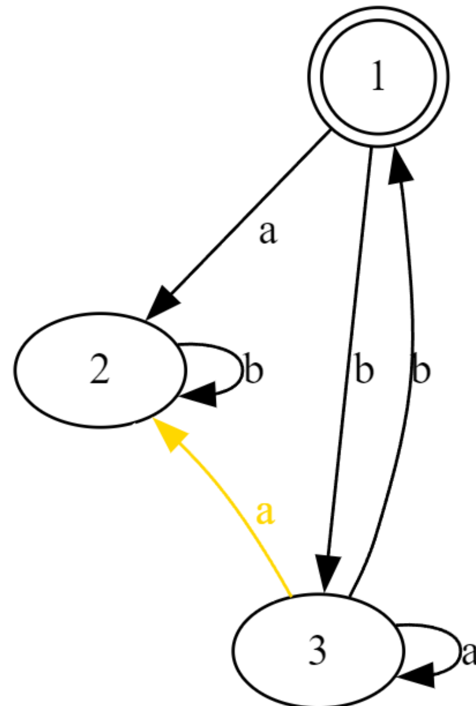


Fonte: elaborado pelo autor.

[1] OMM de inversão de arco não é compatível com arcos congruentes.

Essas imagens foram geradas automaticamente pela plataforma PerformCharts, a partir da leitura direta da estrutura dos grafos mutantes. Na Figura 21 e na Figura 22, são apresentados em vermelho os *elementos removidos* pelos OMMs de omissão. Já na Figura 23, é apresentado em amarelo o arco invertido pelo OMM de inversão de arco.

Figura 23 – Mutante de inversão de arco da MEF da Figura 19



Fonte: elaborado pelo autor.

Nota-se que todos os OMMs são aplicados a um único elemento do statechart, afetando somente a ele e aos elementos dependentes dele. Esse comportamento condiz com o resultado esperado.

4.2 VALIDAÇÃO DOS TESTES DE MODELO POR SEQUÊNCIA DE EVENTOS

Para validar a implementação do algoritmo de teste de modelo por sequência de eventos, foi aplicado ao mutante de inversão de arco visto na seção anterior, representado pela Figura 23, um teste utilizando um conjunto específico de casos de teste. Esse conjunto foi escolhido com objetivo de desencadear todos os possíveis tipos de resultados comportados pelo teste, bem como testar seu comportamento após a detecção de uma falha.

Tabela 6 – Casos de teste aplicados à MEF da Figura 23.

Caso de teste	Resultado esperado	Estado final esperado
b b a b	SUCCESS	2
a b a	INVALID_EVENT	2
b b a a b	INVALID_EVENT	2
b a	NON_DETERMINISTIC_EVENT	3
b a b a b	NON_DETERMINISTIC_EVENT	3
teste vazio	UNKNOWN	1

A Tabela 6 apresenta o conjunto de casos de teste aplicados à MEF, o resultado esperado de cada caso e o estado que a MEF se encontrava no final do teste. O objetivo da aplicação de cada sequência pode ser visto na lista a seguir:

- {*b b a b*} para testar um caso bem-sucedido.
- {*a b a*} para testar o evento inválido [a] no estado 2.
- {*b b a a b*} para testar a parada de um teste após o evento inválido no estado 2.
- {*b a*} para testar o evento não determinístico [a] no estado 3.
- {*b a b a b*} para testar a parada do teste após o evento não determinístico no estado 3.
- { } para testar uma lista de eventos vazia.

A Figura 24 mostra saídas dos testes implementados. Nota-se, ao compará-los com os valores da Tabela 6, que os resultados condizem com o esperado, indicando que execução do algoritmo de testes por sequência de eventos teve o comportamento adequado.

Figura 24 – Resultado dos casos de teste da Tabela 6

```
mutant_8_invertArc_2_Example_All_Cases
b a b a b -> Result: NON_DETERMINISTIC_EVENT -> Final State: 3
b b a a b -> Result: INVALID_EVENT -> Final State: 2
b b a b -> Result: SUCCESS -> Final State: 2
a b a -> Result: INVALID_EVENT -> Final State: 2
b a -> Result: NON_DETERMINISTIC_EVENT -> Final State: 3
teste vazio -> Result: UNKNOWN -> Final State: 1
```

Fonte: elaborado pelo autor.

4.3 ANÁLISE DE MUTAÇÃO DOS GERADORES DE CASOS DE TESTE

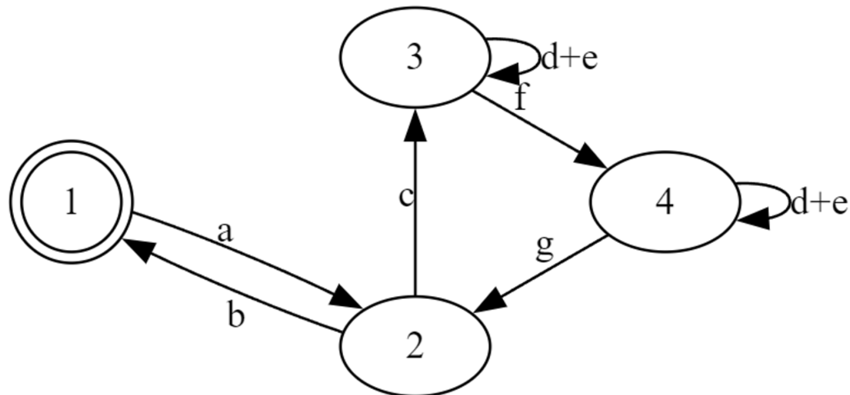
Com as duas ferramentas implementadas e verificadas, foi possível realizar uma análise de mutação nos critérios geradores de casos de teste implementados no PerformCharts. Para isso, cada critério foi utilizado para a geração de casos de teste de duas MEFs, uma específica ao critério e a outra comum entre eles, utilizada para

comparar suas habilidades de encontrar falhas.

4.3.1 Análise do Switch-Cover

A Figura 25 representa a MEF utilizada para a análise de mutação do critério *Switch-Cover*.

Figura 25 – MEF para análise de mutação do Switch-Cover



Fonte: elaborado pelo autor.

O resultado da análise de mutação realizada para o critério Switch-Cover é apresentado pela Figura 26. Segundo ele, foram aplicados 26 casos de teste a cada mutante, os quais conseguiram matar *todos* os 16 mutantes criados pelos OMMs. Para isso, foram executados 416 testes de sequência de eventos, dos quais apenas 16 não detectaram as falhas do modelo.

Figura 26 – Resultado da análise de mutação do Switch-Cover à Figura 25

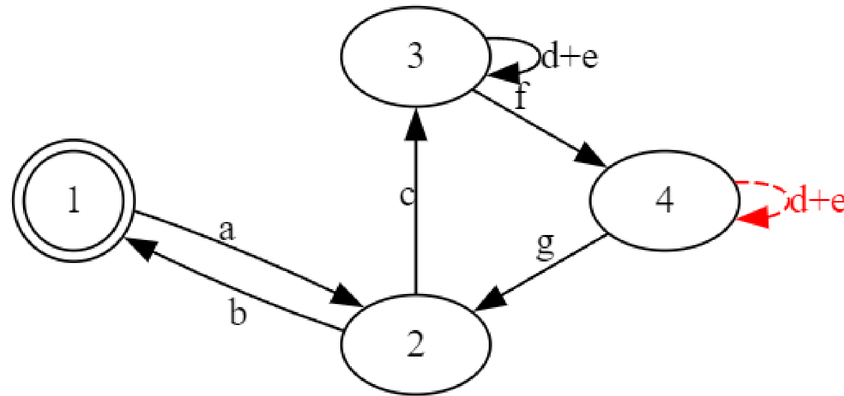
```
----------------------------------------
MUTATION TEST CONCLUDED - Switch-Cover
Number of living mutants:    0
Number of dead mutants:      16
Test cases per mutant:       26
Total of tests applied:      416
Total of passed tests:       16
Mutant closest to live:      mutant_3_omitArc_3_Example_SwitchCover (4/26)
----------------------------------------
Average of passed tests [invertion arc mutants]:   0.600000
Average of passed tests [omission arc mutants]:    2.750000
Average of passed tests [omission node mutants]:   0.285714
----------------------------------------
```

Fonte: elaborado pelo autor.

Também se encontram, no bloco inferior da Figura 26, as médias do número de testes, por tipo de mutante, que não conseguiram detectar a mutação implementada. Mutantes com maior chance de não serem detectados são vistos como mais significativos à análise de mutação.

Nesse caso, o mutante com mais relevância ao teste foi o de omissão de arco, e o de menor foi o de omissão de nó. O mutante mais próximo de viver, passando em 4 dos 26 testes aplicados a ele, pode ser visto na Figura 27. Esse mutante foi gerado pelo OMM de omissão de arco.

Figura 27 – Mutante com melhor desempenho da MEF da Figura 25



.

Fonte: elaborado pelo autor.

O relatório resultante dos testes aplicados ao mutante da Figura 27 é apresentado na Figura 28.

Figura 28 – Testes aplicados ao mutante da Figura 27

```
mutant_5_omitArc_5_Example_SwitchCover -> TEST CASES SUCCEDED: 4/26
Result:SUCCESS    LastState:1   TestCase:a b
Result:SUCCESS    LastState:1   TestCase:a c f g b
Result:SUCCESS    LastState:1   TestCase:a c d+e f g b
Result:SUCCESS    LastState:1   TestCase:a c d+e d+e f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f d+e g c f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f g c f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e g c d+e f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f g c d+e f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f d+e g c f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f g c f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f d+e d+e g c f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e f g c f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e d+e g c d+e f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e g c d+e d+e f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f g c d+e d+e f d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f g c d+e f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f d+e d+e g c f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c d+e d+e f g c f d+e d+e g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f d+e d+e g c d+e d+e f g b
Result:INVALID_EVENT   LastState:4   TestCase:a c f g c d+e d+e f d+e d+e g b
```

.

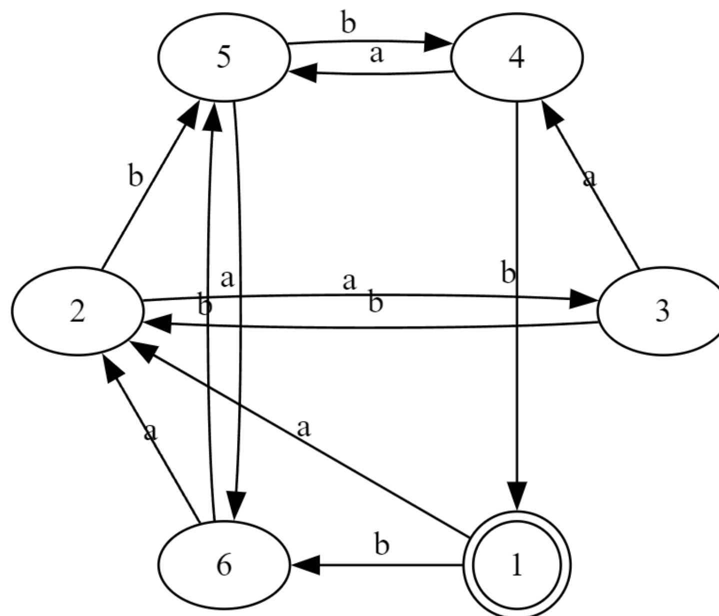
Fonte: elaborado pelo autor.

O resultado da análise de mutação do critério Switch-Cover comprova sua a capacidade de encontrar falhas em um modelo. No entanto, também indica certa ineficiência, devido ao excesso de testes aplicados, considerando que para o mutante ser morto basta apenas detectar uma única falha.

4.3.2 Análise do Método DS

A Figura 29 representa a MEF utilizada para a análise de mutação do critério *método DS*.

Figura 29 – MEF para análise de mutação do método DS



Fonte: elaborado pelo autor.

O resultado da análise de mutação realizada para o critério método DS é apresentado pela Figura 30. Segundo ele, foram aplicados 12 casos de teste a cada mutante, os quais conseguiram matar *todos* os 30 mutantes criados pelos OMMs. Para isso, foram executados 360 testes de sequência de eventos, dos quais apenas 119 não detectaram as falhas do modelo.

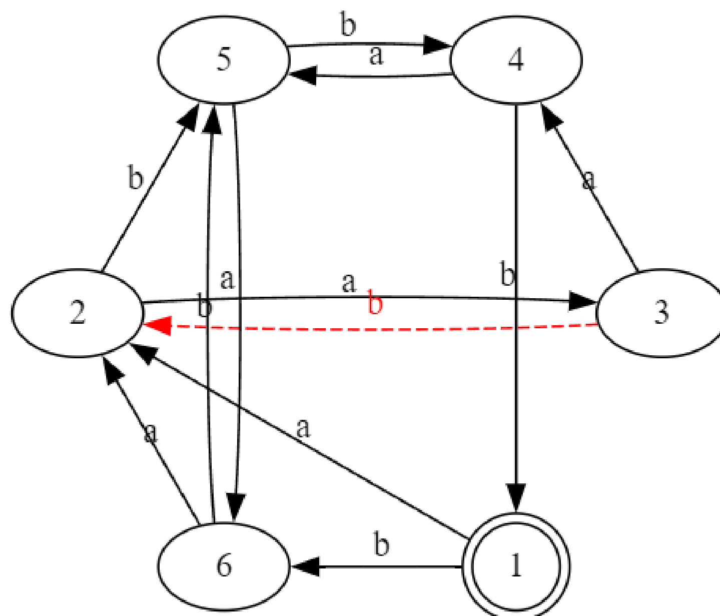
Figura 30 – Resultado da análise de mutação do método DS à Figura 29

```
----------------------------------------
MUTATION TEST CONCLUDED - DS
Number of living mutants:    0
Number of dead mutants:      30
Test cases per mutant:       12
Total of tests applied:      360
Total of passed tests:       119
Mutant closest to live:      mutant_6_omitArc_6_Example_DS (10/12)
----------------------------------------
Average of passed tests [invertion arc mutants]:    3.250000
Average of passed tests [omission arc mutants]:     11.833333
Average of passed tests [omission node mutants]:    0.750000
----------------------------------------
```

Fonte: elaborado pelo autor.

Novamente, o mutante com mais relevância ao teste foi o de omissão de arco, e o de menor foi o de omissão de nó. O mutante mais próximo de viver, passando em 10 dos 12 testes aplicados a ele, pode ser visto na Figura 27. Esse mutante foi gerado pelo OMM de omissão de arco.

Figura 31 – Mutante com melhor desempenho da MEF da Figura 29



Fonte: elaborado pelo autor.

O relatório dos testes aplicados ao mutante da Figura 31 é apresentado na Figura 32.

Figura 32 – Testes aplicados ao mutante da Figura 31

```
mutant_6_omitArc_6_Example_DS -> DS -> TEST CASES SUCCEDED: 10/12
Result:SUCCESS   LastState:6   TestCase:a b b a a
Result:SUCCESS   LastState:6   TestCase:b b b a a
Result:SUCCESS   LastState:3   TestCase:a b b b a a
Result:SUCCESS   LastState:6   TestCase:b a b b a a
Result:SUCCESS   LastState:3   TestCase:b b b b a a
Result:SUCCESS   LastState:3   TestCase:a a a b b a a
Result:SUCCESS   LastState:3   TestCase:a b b b b a a
Result:SUCCESS   LastState:6   TestCase:a b a b b a a
Result:SUCCESS   LastState:2   TestCase:a a a b b b a a
Result:SUCCESS   LastState:3   TestCase:a a a a b b a a
Result:INVALID_EVENT   LastState:3   TestCase:a a b b a a
Result:INVALID_EVENT   LastState:3   TestCase:a a b b b a a
```
.

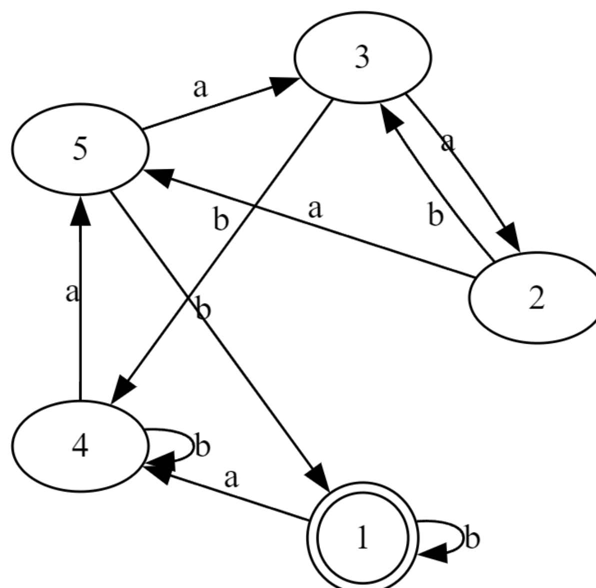
Fonte: elaborado pelo autor.

O resultado da análise de mutação do critério do método DS comprova sua a capacidade de encontrar falhas em um modelo. Além disso, a quantidade de testes responsáveis por detectar a mesma falha em seu pior caso foi de 2, representando apenas 1/6 do total de testes, indicando uma boa eficiência de geração para essa MEF.

4.3.3 Análise do Método UIO

A Figura 33 representa a MEF utilizada para a análise de mutação do critério do *método UIO*.

Figura 33 – MEF para análise de mutação do método UIO



Fonte: elaborado pelo autor.

O resultado da análise de mutação realizada para o critério do método UIO é apresentado pela Figura 34. Segundo ele, foram aplicados 10 casos de teste a cada mutante, os quais conseguiram matar *todos* os 23 mutantes criados pelos OMMs. Para

isso, foram executados 230 testes de sequência de eventos, dos quais apenas 114 não detectaram as falhas do modelo.

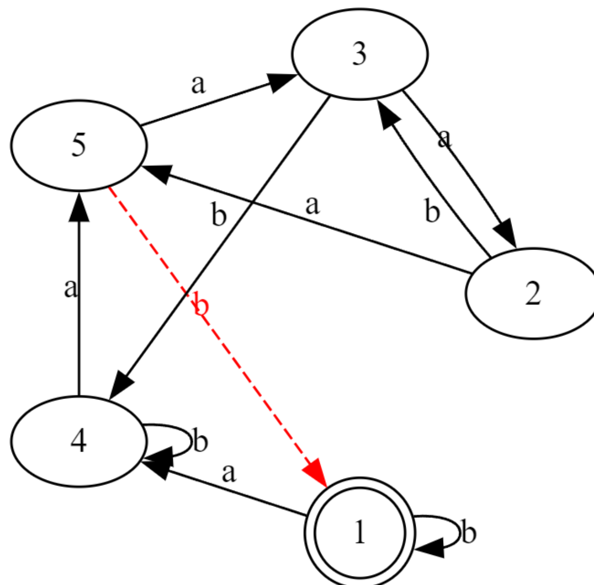
Figura 34 – Resultado da análise de mutação do método UIO à Figura 33

```
----------------------------------------
MUTATION TEST CONCLUDED - UIO
Number of living mutants:    0
Number of dead mutants:      23
Test cases per mutant:       10
Total of tests applied:      230
Total of passed tests:       114
Mutant closest to live:      mutant_4_omitArc_4_Example_UIO (9/10)
----------------------------------------
Average of passed tests [invertion arc mutants]:   4.875000
Average of passed tests [omission arc mutants]:    12.400000
Average of passed tests [omission node mutants]:   1.300000
----------------------------------------
```

Fonte: elaborado pelo autor.

Mais uma vez, o mutante com mais relevância ao teste foi o de omissão de arco, e o de menor foi o de omissão de nó. O mutante mais próximo de viver, passando em 9 dos 10 testes aplicados a ele, pode ser visto na Figura 27. Esse mutante foi gerado pelo OMM de omissão de arco.

Figura 35 – Mutante com melhor desempenho da MEF da Figura 33



Fonte: elaborado pelo autor.

O relatório dos testes aplicados ao mutante da Figura 35 é apresentado na Figura 36.

Figura 36 – Testes aplicados ao mutante da Figura 35

```
mutant_4_omitArc_4_Example_UIO -> UIO -> TEST CASES SUCCEDED: 9/10
Result:SUCCESS   LastState:1   TestCase:b b
Result:SUCCESS   LastState:4   TestCase:a b b
Result:SUCCESS   LastState:4   TestCase:a b b b
Result:SUCCESS   LastState:2   TestCase:a a a a
Result:SUCCESS   LastState:3   TestCase:a a a a b
Result:SUCCESS   LastState:3   TestCase:a a a a a a
Result:SUCCESS   LastState:4   TestCase:a a a b b b
Result:SUCCESS   LastState:3   TestCase:a a a a b a b
Result:SUCCESS   LastState:2   TestCase:a a a a a a a
Result:INVALID_EVENT   LastState:5   TestCase:a a b b
```

Fonte: elaborado pelo autor.

O resultado da análise de mutação do critério do método UIO, assim como o dos demais critérios, comprovou sua a capacidade de encontrar falhas em um modelo. A quantidade de testes responsáveis por detectar a mesma falha, em seu pior caso, foi de 1. Esse resultado é visto como ideal, novamente indicando uma boa eficiência de geração para essa MEF.

4.3.4 Análise Comparativa dos Critérios

As análises anteriores, embora válidas, não fornecem meios de comparar os resultados de diferentes CGCTs, visto que eles foram implementados à MEFs distintas. Para obter resultados que possibilitem suas comparações, foi feita uma análise de mutação utilizando casos de teste gerados pelos CGCTs aplicados à MEF da Figura 19, localizada na seção 4.1, a qual é compatível com todos os critérios testados. O resultado das análises pode ser visto na Figura 37.

Figura 37 – Resultados das análises dos CGCTs da MEF da Figura 19.

```
MUTATION TEST CONCLUDED - DS
Number of living mutants:    0
Number of dead mutants:      13
Test cases per mutant:       6
Total of tests applied:      78
Total of passed tests:       28
Mutant closest to live:      mutant_2_omitArc_2_Example_All_Cases (5/6)
----------------------------------------
Average of passed tests [invertion arc mutants]:   2.000000
Average of passed tests [omission arc mutants]:    6.000000
Average of passed tests [omission node mutants]:   0.333333
----------------------------------------



----------------------------------------
MUTATION TEST CONCLUDED - UIO
Number of living mutants:    0
Number of dead mutants:      13
Test cases per mutant:       6
Total of tests applied:      78
Total of passed tests:       30
Mutant closest to live:      mutant_2_omitArc_2_Example_All_Cases (5/6)
----------------------------------------
Average of passed tests [invertion arc mutants]:   2.000000
Average of passed tests [omission arc mutants]:    6.666667
Average of passed tests [omission node mutants]:   0.333333
----------------------------------------



----------------------------------------
MUTATION TEST CONCLUDED - Switch-Cover
Number of living mutants:    0
Number of dead mutants:      13
Test cases per mutant:       41
Total of tests applied:      533
Total of passed tests:       54
Mutant closest to live:      mutant_1_omitArc_1_Example_All_Cases (18/41)
----------------------------------------
Average of passed tests [invertion arc mutants]:   1.250000
Average of passed tests [omission arc mutants]:    15.000000
Average of passed tests [omission node mutants]:   0.666667
----------------------------------------
```

Fonte: elaborado pelo autor.

O comportamento dos testes, quando aplicados a essa MEF, é muito semelhante ao dos testes analisados de previamente. Todos os CGCTs conseguiram matar todos os mutantes, demonstrando uma consistência no comportamento dos algoritmos de geração de casos de teste.

A Tabela 7 apresenta o conjunto de todos os resultados obtidos nas análises

realizadas aos critérios de geração de casos de teste. Nela, MEF1 é representada pela Figura 25, MEF2 pela Figura 29, MEF3 pela Figura 36 e MEF4 pela Figura 19.

Tabela 7 – Resultados das análises de mutação.

	Switch-Cover		**DS**		**UIO**	
	MEF 1	MEF 4	MEF 2	MEF 4	MEF 3	MEF 4
Mutantes vivos	0	0	0	0	0	0
Total de mutantes	16	13	30	13	23	13
Testes por mutante	26	41	12	6	10	6
Total de testes aplicados	416	533	360	78	230	78
Total de testes que não detectaram mutantes	16	54	119	28	114	30
Resultado do teste do mutante mais próximo de permanecer vivo	4/26	18/41	10/12	5/6	9/10	5/6
Média de testes que não detectaram uma falha (omissão de arco)	2.75	15.00	11.83	6.00	12.40	2.00
Média de testes que não detectaram uma falha (inversão de arco)	0.60	1.25	3.25	2.00	4.88	6.67
Média de testes que não detectaram uma falha (omissão de nó)	0.29	0.67	0.75	0.33	1.3	0.33

4.4 DISCUSSÕES

Este trabalho realizou a implementação de uma ferramenta de mutação de modelos, utilizando três operadores de mutação de modelo, os de omissão de arco, omissão de nó e inversão de arco. Também foi desenvolvido uma ferramenta de teste de modelo, utilizando uma sequência de eventos.

Essas implementações possibilitaram a realização de uma análise de mutação dos critérios geradores de casos de testes disponíveis no PerformCharts. Dessa forma, se tornou possível avaliar a integridade dos geradores de teste utilizados, bem como sua eficiência e qualidade durante o processo.

Para realizar o teste de um modelo, é utilizada uma lista de eventos. Como os geradores de casos de teste não disponibilizam seus eventos nesse formato, é feita uma conversão. Essa conversão pode acarretar problemas de desempenho e aumentar o risco de bugs. Logo, uma integração mais adequada pode ser necessária para o teste de modelos mais complexos.

Todos os CGCTs foram completamente aprovados pela análise de mutação, reforçando uma ideia de confiabilidade nos algoritmos quando aplicados a modelos

mais simples. Apesar disso, esse método de análise não pode afirmar a inexistência de falhas nos critérios, apenas encontrar as mesmas. A implementação de mais operadores de mutação *relevantes* e a utilização de MEFs mais complexas pode melhorar a credibilidade das análises.

O principal objetivo desse tipo de análise de mutação é encontrar as falhas e possíveis limitações dos CGCTs. A incapacidade das análises realizadas de encontrar alguma falha nos CGCTs é, possivelmente, resultado de suas atuais limitações como, por exemplo, a incapacidade de gerar OMMs dependentes da inserção ou corrupção de elementos.

Outras limitações da implementação incluem, mas não se limitam, a incapacidade de analisar de MEFs não determinísticas, de detectar modelos equivalentes, de aplicar estilos de OMMs individualmente e de detectar as *ações/saídas* do statechart, tanto no processo de mutação quando no teste do sistema.

Dentre os possíveis OMMs a serem implementados futuramente, é destacado o potencial do operador de inserção de arco, o qual, segundo o *princípio do acoplamento*, deve ser o mutante com a maior eficiência média. No entanto, ressalta-se que a geração de mutantes de inserção só se mostra viável quando aplicada em conjunto a um método de detecção de equivalências.

Já entre os atuais operadores, o operador de omissão de arco se mostrou o mais eficaz, sendo responsável por *todos* os mutantes mais próximo de ficarem vivos encontrados. Esse resultado, novamente, condiz com o princípio do acoplamento, e levanta o questionamento sobre a necessidade da utilização do OMM de omissão de nó, operador de ordem múltipla que mostrou o maior custo de operação e a menor eficiência na análise de mutantes.

Os algoritmos implementados também disponibilizam a base para a implementação de outras ferramentas de testes dependentes do processo de mutação de modelo, como o critério de geração de casos de teste de mutação de modelo, se mostrando expansível e compatível com os métodos da plataforma PerformCharts.

5 CONCLUSÕES

Este trabalho propôs a implementação de algoritmos de teste de modelos por sequência de eventos e de uma ferramenta de mutação, composta por três operadores de mutação. Essa implementação foi desenvolvida visando uma integração às bibliotecas da ferramenta PerformCharts, responsável pela implementação de testes do ambiente WebPerformCharts.

A ferramenta de teste de modelos por sequência de eventos obteve sucesso em seus testes, conseguindo percorrer um grafo através de uma sequência de eventos e retornando os resultados esperados. Caso o resultado seja uma falha, ela consegue retornar o nó do ocorrido e também o tipo de falha, seja essa um caso de *evento inválido*, *não determinismo* ou algum outro tipo de exceção.

Para serem aplicados aos testes de modelo, foi desenvolvida uma maneira de extrair uma lista de eventos dos Critérios Geradores de Casos de Teste (CGCT). Para isso, os casos de teste foram convertidos de um formato de texto para uma classe de evento interna ao PerformCharts. Essa conversão não é ideal, pois aumenta o custo de processamento em MEFs muito complexas e abre margem para bugs desconhecidos.

Também foi implementada a ferramenta de mutação, a qual aplica os operadores de mutação de omissão de arco, omissão de nó e inversão de arco em um grafo. A forma de mutação dos OMMs foi desenvolvida de maneira a evitar a geração de mutantes equivalentes. Os resultados observados das mutações dos OMMs foram os esperados, gerando todos os mutantes não equivalentes possíveis a cada OMM.

Ao ser feita a análise de mutação nos CGCTs do PerformCharts, foi constatada a sua capacidade de gerar casos de teste capazes de matar todos os mutantes. Enquanto esse resultado indica uma qualidade promissora aos critérios, também mostra uma ineficiência em encontrar possíveis bugs e inconsistências dos CGCT.

Os resultados de cada CGCT, quando comparados entre si, indicaram uma boa eficiência de geração de casos de teste dos métodos UIO e DS, especialmente quando comparado ao Switch-Cover. No entanto, se ressalta a necessidade da aplicação de análises em MEFs mais complexas, a fim de verificar se essa aproximação não resultará em alguma falha de detecção de mutante.

Uma análise feita nos OMMs, utilizando os resultados das análises dos CGCTs, indicou uma maior eficiência do OMM de omissão de arco e, em contrapartida, uma grande ineficiência do critério de omissão de nó. Esse resultado, segundo o princípio de acoplamento, também era esperado, e traz o questionamento da necessidade da implementação do operador de omissão de nó.

Para trabalhos futuros, é sugerido uma reformulação dos operadores de

mutação, ressaltando o potencial da implementação do OMM de inserção de arco. Para assegurar a qualidade averiguada dos CGCTs, pode-se aplicar novas análises de mutação em MEFs mais complexas, com ortogonalidade. Buscando uma melhoria na manipulação de modelos mutantes, é sugerido a implementação de algoritmos de verificação de equivalência entre MEFs. Para a aplicação de outros testes, é sugerido implementar aos teste um registro de trajetória percorrida, assim como uma forma de considerar os efeitos de ações em sua execução.

REFERÊNCIAS

ACREE, A. T. J. **On Mutation**. Tese de Doutorado em Computação e Programação — Atlanta School of information and Computer Science, Georgia Inst. of Tech, Atlanta, 1980.

AICHERNIG, B. et al. Killing strategies for model-based mutation testing. **Software Testing, Verification and Reliability**, v. 25, 02 2014.

AICHERNIG, B. K.; JÖBSTL, E.; TIRAN, S. Model-based mutation testing via symbolic refinement checking. **Science of Computer Programming**, Elsevier B.V., v. 97, p. 383–404, 2015.

ALHWIKEM, F. H. M. **A Novel Approach toMutation Operator Design forMDE Languages**. Tese de Doutorado de Ciência da Computação — University of York, York, 2019.

AMARAL, A. S. M. S.; VELOSO, R. R.; VIJAYKUMAR, N. L. **ON PROPOSING A MARKUP LANGUAGE FOR STATECHARTS TO BE USED IN PERFORMANCE EVALUATION**. Relatório Técnico — Instituto Nacional de Pesquisas Espaciais,São José dos Campos, 2005.

BATES, C. **Linguagens e Máquinas: Uma Introdução aos Fundamentos da Computação**. 1. ed. Nova Jersey: John Wiley & Sons, Inc., 2003.

BAX, M. P. Introdução às linguagens de marcas. **Ciência Da Informação**, v. 30, n. 1, p. 32–38, 2001.

BELLI, F. et al. Model-based mutation testing - approach and case studies. **Science of Computer Programming**, Elsevier B.V., v. 120, p. 25–48, 2016.

CHA, S.; TAYLOR, R. N.; KANG, K. **Handbook of Software Engineering**. 1. ed. Suiça: Springer, 2019.

CHOW, T. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, SE-4, n. 3, p. 178–187, 1978.

DALAL, S. et al. Model-based testing in practice. In: **Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)**. [S.l.: s.n.], 1999. p. 285–294.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. 1. ed. São Paulo: Elsevier, 2007.

DIESTEL, R. **Graph Theory**. 5. ed. [S.l.]: Springer, 2007.

DRUSINSKY, D.; HAREL, D. Using statecharts for hardware description and synthesis. **IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN**, v. 8, n. 7, p. 798–807, 1989.

FANTINATO, M. Teste de software baseado em máquinas de estados finitos. In: **VIII Congreso Argentino de Ciencias de la Computación**. [S.l.: s.n.], 2002.

FLOYD, T. L. **Digital Fundamentals: A Systems Approach**. 1. ed. Londres: Pearson Education Limited, 2013.

GONENC, G. A method for the design of fault detection experiments. **IEEE Transactions on Computers**, C-19, n. 6, p. 551–558, 1970.

GRAHAM, R. D.; JOHNSON, P. C. Finite state machine parsing for internet protocols: Faster than you think. In: **2014 IEEE Security and Privacy Workshops**. San Jose, CA, USA: IEEE, 2014. p. 185–190.

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, n. 1, p. 231–274, 1987.

IEEE Standard Classification for Software Anomalies. **IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)**, p. 1–23, 2010.

LEVI, F. Compositional verification of quantitative properties of statecharts. **Journal of Logic and Computation**, v. 11, n. 6, p. 829–878, 2001.

LORBER, F.; LARSEN, K. G.; NIELSEN, B. Model-based mutation testing of real-time systems via model checking. In: **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2018. p. 59–68.

MYERS, G. J. et al. **The Art of Software Testing**. 2. ed. Nova Jersey: John Wiley & Sons, Inc., 2004.

PIMONT, S.; RAULT, J.-C. A software reliability assessment based on a structural and behavioral analysis of programs. In: **Proceedings of the 2nd InternationalConference on Software Engineering**. Washington, DC, USA: [s.n.], 1976.

PORTO, F. R. **Estratégia para geração de sequencias de verificação para máquinas de estados finitos**. Dissertação de Mestrado em Ciências de Computação e Matemática Computacional — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2013.

PRESSMAN, R. S. **Software Engineering a Practitioner's Approach**. 7. ed. Nova York: McGraw-Hill, 2014.

SABNANI, K.; DAHBURA, A. A protocol test generation procedure. **Computer Networks and ISDN Systems**, v. 15, n. 4, p. 285–297, 1988. ISSN 0169-7552. Disponível em: https://www.sciencedirect.com/science/article/pii/0169755288900645.

SANTIAGO, V. et al. A practical approach for automated test case generation using statecharts. In: **30th Annual International Computer Software and Applications Conference (COMPSAC'06)**. [S.l.: s.n.], 2006. v. 2, p. 183–188.

SILVA, B. A. d. **Critérios para geração de casos de teste baseados em modelos descritos em PcML**. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) — Centro Tecnológico de Joinville, Universidade Federal de Santa Catarina, Joinville, 2022.

SOUZA, E. F. et al. H-switch cover: A new test criterion to generate test case from finite state machines. Kluwer Academic Publishers, USA, v. 25, n. 2, 2017. Disponível em: https://doi.org/10.1007/s11219-015-9300-8.

VIEIRA, N. J. **Linguagens e Máquinas: Uma Introdução aos Fundamentos da Computação**. 1. ed. Belo Horizonte: Cengage Learning, 2006.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. **ACM Computing Surveys**, v. 29, n. 4, p. 366–427, 1997.

APÊNDICE A - ARQUIVOS PCML DAS MEFS UTILIZADAS.

Estão disponibilizados, a seguir, todos os arquivos PcML utilizados para representar as MEFs ao PerformCharts.

Código 1 – Arquivo PcML referente à MEF da Figura 19

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PcML Title="Example_All_Cases" Date="2021-11-23" Version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="pcml.xsd">
  <Info>
    <Author>
      <Name>Gian</Name>
      <Name>Paulo</Name>
    </Author>
    <Description>
      Exemple test all cases.
       </Description>

  </Info>
<States>
  <Root Name="FSM" Type="XOR" Default="1">
    <State Name="1" Type="BASIC"/>
    <State Name="2" Type="BASIC"/>
    <State Name="3" Type="BASIC"/>
  </Root>
</States>

<Outputs>
  <Output Name="O0" Value="0" />
  <Output Name="O1" Value="1" />
</Outputs>

<Actions>
  <OutputTriggerAction Name="o0" Output="O0" />
  <OutputTriggerAction Name="o1" Output="O1" />
</Actions>

<Events>
  <Stochastic Name="a" Value="1.0"/>
  <Stochastic Name="b" Value="1.0"/>
</Events>
```

```
<Transitions>
  <Transition Source="1" Event="a"  Destination="2" Action="o0"/>
  <Transition Source="1" Event="b"  Destination="3" Action="o1"/>
  <Transition Source="2" Event="b"  Destination="2" Action="o1"/>
  <Transition Source="2" Event="a"  Destination="3" Action="o1"/>
  <Transition Source="3" Event="b"  Destination="1" Action="o1"/>
  <Transition Source="3" Event="a"  Destination="3" Action="o0"/>
</Transitions>
</PcML>
```

Código 2 – Arquivo PcML referente à MEF da Figura 25

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PcML Title="Example_SwitchCover" Date="2021-11-23" Version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="pcml.xsd">
  <Info>
    <Author>
      <Name>Gian</Name>
      <Name>Paulo</Name>
    </Author>
    <Description>
      Exemple test Switch-Cover cases.
       </Description>
  </Info>
<States>
  <Root Name="FSM" Type="XOR" Default="1">
    <State Name="1" Type="BASIC"/>
    <State Name="2" Type="BASIC"/>
    <State Name="3" Type="BASIC"/>
        <State Name="4" Type="BASIC"/>
  </Root>
</States>
<Outputs>
  <Output Name="00" Value="0" />
  <Output Name="01" Value="1" />
    <Output Name="03" Value="3" />
    <Output Name="04" Value="4" />
    <Output Name="05" Value="5" />
    <Output Name="06" Value="6" />
</Outputs>
<Actions>
  <OutputTriggerAction Name="o0" Output="00" />
  <OutputTriggerAction Name="o1" Output="01" />
    <OutputTriggerAction Name="o3" Output="03" />
  <OutputTriggerAction Name="o4" Output="04" />
    <OutputTriggerAction Name="o5" Output="05" />
  <OutputTriggerAction Name="o6" Output="06" />
```

```
</Actions>
<Events>
  <Stochastic Name="a" Value="1.0"/>
  <Stochastic Name="b" Value="1.0"/>
    <Stochastic Name="c" Value="1.0"/>
  <Stochastic Name="d+e" Value="1.0"/>
  <Stochastic Name="f" Value="1.0"/>
  <Stochastic Name="g" Value="1.0"/>
</Events>
<Transitions>
  <Transition Source="1" Event="a"  Destination="2" Action="o0"/>
  <Transition Source="2" Event="b"  Destination="1" Action="o1"/>
  <Transition Source="2" Event="c"  Destination="3" Action="o3"/>
  <Transition Source="3" Event="d+e"  Destination="3" Action="o4"/>
  <Transition Source="3" Event="f"  Destination="4" Action="o5"/>
    <Transition Source="4" Event="d+e"  Destination="4" Action="o6"/>
    <Transition Source="4" Event="g"  Destination="2" Action="o5"/>
</Transitions>
</PcML>
```

Código 3 – Arquivo PcML referente à MEF da Figura 29

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PcML Title="Example_DS" Date="2021-11-23" Version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="pcml.xsd">
  <Info>
    <Author>
      <Name>Gian</Name>
      <Name>Paulo</Name>
    </Author>
    <Description>
      Exemple test DS cases.
       </Description>

  </Info>
<States>
  <Root Name="FSM" Type="XOR" Default="1">
    <State Name="1" Type="BASIC"/>
    <State Name="2" Type="BASIC"/>
    <State Name="3" Type="BASIC"/>
    <State Name="4" Type="BASIC"/>
    <State Name="5" Type="BASIC"/>
    <State Name="6" Type="BASIC"/>
  </Root>
</States>

<Outputs>
```

```
  <Output Name="O0" Value="0" />
  <Output Name="O1" Value="1" />
</Outputs>

<Actions>
  <OutputTriggerAction Name="o0" Output="O0" />
  <OutputTriggerAction Name="o1" Output="O1" />
</Actions>

<Events>
  <Stochastic Name="a" Value="1.0"/>
  <Stochastic Name="b" Value="1.0"/>
</Events>

<Transitions>
  <Transition Source="1" Event="a"  Destination="2" Action="o0"/>
  <Transition Source="1" Event="b"  Destination="6" Action="o0"/>
  <Transition Source="2" Event="a"  Destination="3" Action="o0"/>
  <Transition Source="2" Event="b"  Destination="5" Action="o1"/>
  <Transition Source="3" Event="b"  Destination="2" Action="o0"/>
  <Transition Source="3" Event="a"  Destination="4" Action="o0"/>
  <Transition Source="4" Event="b"  Destination="1" Action="o0"/>
  <Transition Source="4" Event="a"  Destination="5" Action="o0"/>
  <Transition Source="5" Event="b"  Destination="4" Action="o1"/>
  <Transition Source="5" Event="a"  Destination="6" Action="o0"/>
  <Transition Source="6" Event="a"  Destination="2" Action="o1"/>
  <Transition Source="6" Event="b"  Destination="5" Action="o1"/>
</Transitions>
</PcML>
```

Código 4 – Arquivo PcML referente à MEF da Figura 33

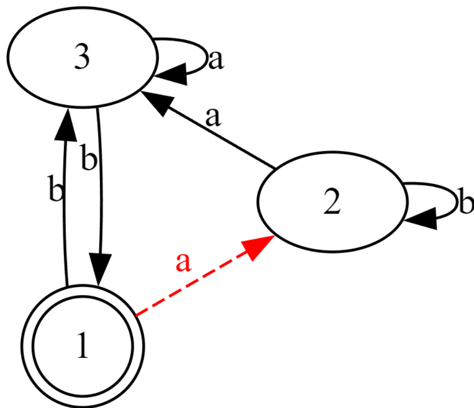
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PcML Title="Example_UIO" Date="2021-11-23" Version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="pcml.xsd">
  <Info>
    <Author>
      <Name>Gian</Name>
      <Name>Paulo</Name>
    </Author>
    <Description>
      Exemple test UIO cases.
       </Description>

  </Info>
<States>
  <Root Name="FSM" Type="XOR" Default="1">
```

```
    <State Name="1" Type="BASIC"/>
    <State Name="2" Type="BASIC"/>
    <State Name="3" Type="BASIC"/>
    <State Name="4" Type="BASIC"/>
    <State Name="5" Type="BASIC"/>
  </Root>
</States>

<Outputs>
  <Output Name="O0" Value="0" />
  <Output Name="O1" Value="1" />
  <Output Name="O2" Value="2" />
</Outputs>

<Actions>
  <OutputTriggerAction Name="o0" Output="O0" />
  <OutputTriggerAction Name="o1" Output="O1" />
  <OutputTriggerAction Name="o2" Output="O2" />
</Actions>

<Events>
  <Stochastic Name="a" Value="1.0"/>
  <Stochastic Name="b" Value="1.0"/>
</Events>

<Transitions>
  <Transition Source="1" Event="b"  Destination="1" Action="o2"/>
  <Transition Source="1" Event="a"  Destination="4" Action="o0"/>
  <Transition Source="2" Event="b"  Destination="3" Action="o0"/>
  <Transition Source="2" Event="a"  Destination="5" Action="o1"/>
  <Transition Source="3" Event="a"  Destination="2" Action="o0"/>
  <Transition Source="3" Event="b"  Destination="4" Action="o0"/>
  <Transition Source="4" Event="b"  Destination="4" Action="o1"/>
  <Transition Source="4" Event="a"  Destination="5" Action="o0"/>
  <Transition Source="5" Event="b"  Destination="1" Action="o1"/>
  <Transition Source="5" Event="a"  Destination="3" Action="o1"/>
</Transitions>
</PcML>
```

APÊNDICE B - MODELOS MUTANTES GERADOS

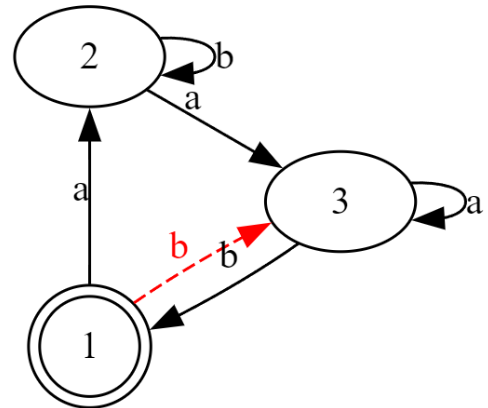
Estão disponibilizados, a seguir, todos os mutantes, ainda não apresentados, gerados pela ferramenta de mutação da aplicada à MEF da Figura 19.

Figura 38 – Mutante de omissão de arco 1.



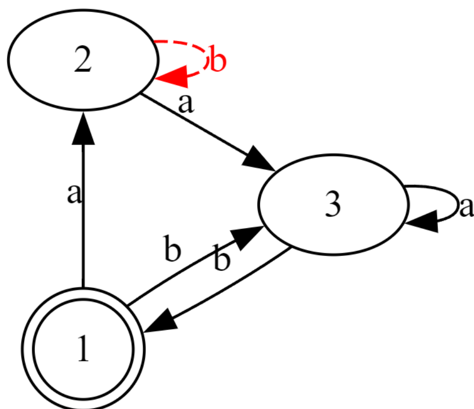
Fonte: elaborado pelo autor.

Figura 39 – Mutante de omissão de arco 2.



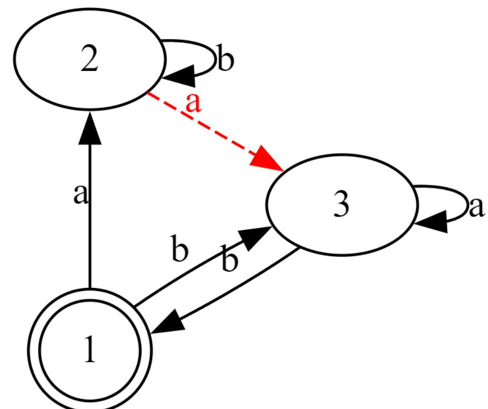
Fonte: elaborado pelo autor.

Figura 40 – Mutante de omissão de arco 3.



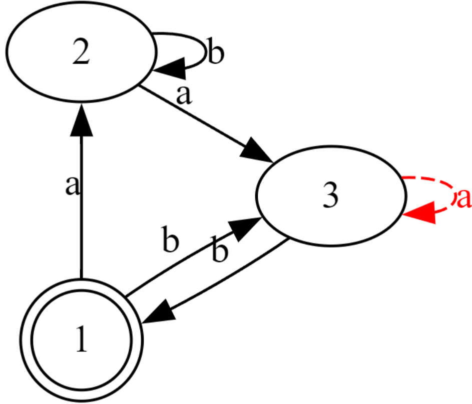
Fonte: elaborado pelo autor.

Figura 41 – Mutante de omissão de arco 4.



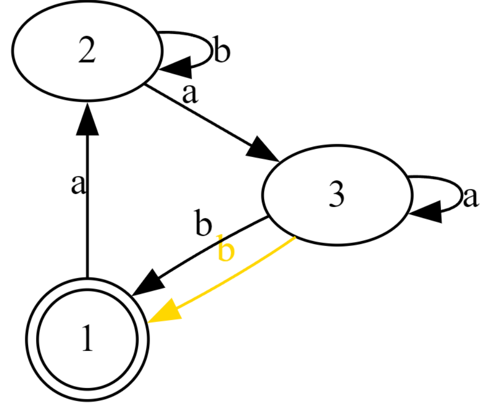
Fonte: elaborado pelo autor.

Figura 42 – Mutante de omissão de arco 5.



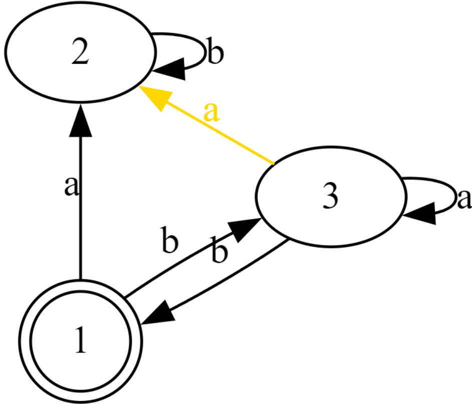
Fonte: elaborado pelo autor.

Figura 43 – Mutante de inversão de arco 1.



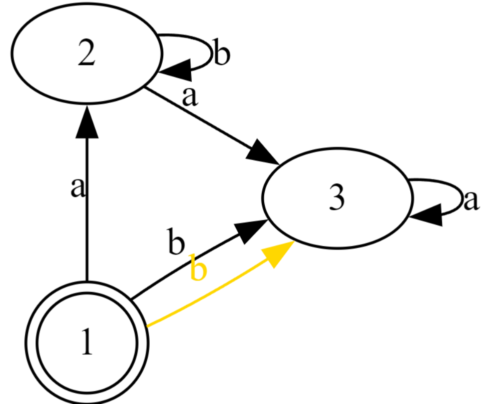
Fonte: elaborado pelo autor.

Figura 44 – Mutante de inversão de arco 2.



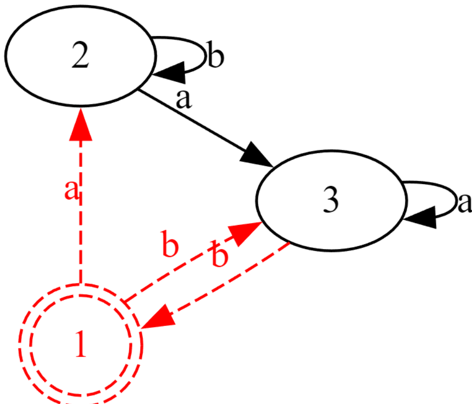
Fonte: elaborado pelo autor.

Figura 45 – Mutante de inversão de arco 3.



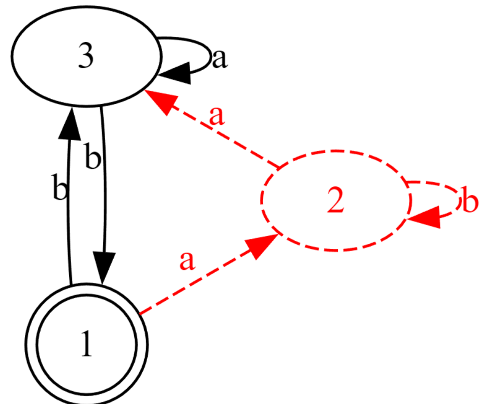
Fonte: elaborado pelo autor.

Figura 46 – Mutante de omissão de nó 1.



Fonte: elaborado pelo autor.

Figura 47 – Mutante de omissão de nó 2.



Fonte: elaborado pelo autor.