UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

# Uma abordagem para apoiar a identificação de não executabilidade no teste estrutural de software

**João Choma Neto**

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

ICMC USP SÃO CARLOS

**João Choma Neto**

# Uma abordagem para apoiar a identificação de não executabilidade no teste estrutural de software

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger de Souza
Coorientador: Profa. Dra. Thelma Elita Colanzi Lopes

**USP – São Carlos**
**Outubro de 2023**

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Choma Neto, João
C548a Uma abordagem para apoiar a identificação de não executabilidade no teste estrutural de software / João Choma Neto; orientadora Simone do Rocio Senger de Souza; coorientadora Thelma Elita Colanzi Lopes. -- São Carlos, 2023.
134 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Teste de Software. 2. Teste estrutura. 3. Programas concorrentes. 4. Problema da Não Executabilidade. I. do Rocio Senger de Souza, Simone , orient. II. Elita Colanzi Lopes, Thelma , coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

**João Choma Neto**

# An approach to support the identification of non-executability in structural software testing

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger de Souza
Co-advisor: Profa. Dra. Thelma Elita Colanzi Lopes

**USP – São Carlos**
**October 2023**

# AGRADECIMENTOS

Quero expressar minha profunda gratidão aos meus pais e minha família pelo apoio incondicional que me ofereceram ao longo de todo o processo que culminou na conclusão desta tese de doutorado. Quero agradecer a Deus por me conceder força e sustento ao longo dessa jornada.

Gostaria de expressar minha sincera gratidão às minhas orientadoras, a professora Doutora Simone de Rocio Senger de Souza e a coorientadora professora Doutora Thelma Elita Colanzi Lopes, pela orientação, formação e pela oportunidade de trabalhar com notáveis pesquisadoras. Agradeço por sua paciência e apoio inabalável, que foram fundamentais para o meu crescimento intelectual e pessoal ao longo deste percurso.

Agradeço aos meus companheiros de pesquisa Ricardo Vilela e Ricardo Chagas que contribuíram de forma ímpar na condução da pesquisa.

Expresso minha gratidão aos meus colegas de pesquisa, cujas contribuições diretas e indiretas desempenharam um papel fundamental no desenvolvimento das atividades relacionadas à minha tese. Em especial, gostaria de mencionar a valiosa colaboração de João Paulo Biazotto e Tiago Piperno Bonetti.

Gostaria de estender meus agradecimentos especiais à FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) pelo apoio financeiro concedido no âmbito do processo 2018/25744-6 que viabilizou a condução desta pesquisa. Além disso, expresso minha gratidão à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro provido sob o código de financiamento PROEX-10633606/D.

A todos, fiz o meu melhor.

# RESUMO

CHOMA NETO, JOÃO. **Uma abordagem para apoiar a identificação de não executabilidade no teste estrutural de software**. 2023. 134 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

A atividade de teste de software é fundamental para garantir a qualidade de um produto de software. No entanto, encontrar um conjunto de casos de teste que satisfaça um determinado critério de teste não é uma tarefa simples, pois o domínio de entrada em geral é vasto e diferentes conjuntos de teste podem ser derivados, com eficácia diferente. No contexto de testes estruturais, a não executabilidade (ou requisitos de teste não executáveis) é uma característica presente na maioria dos programas, o que aumenta o custo e o esforço da atividade de teste. Quando programas concorrentes são testados, novos desafios são enfrentados, principalmente relacionados ao não-determinismo. O não-determinismo pode resultar em diferentes saídas de teste possíveis para a mesma entrada de teste, tornando importante testar todas as situações possíveis. No entanto, a não executabilidade torna-se ainda mais complexa nesse contexto. Para mitigar esse problema, foi desenvolvida uma abordagem chamada Nonexec, que permite a utilização automatizada de propriedades baseadas em código fonte capazes de revelar requisitos de teste não executáveis. Um dos primeiros resultados foi a definição de três conjuntos que classificam os requisitos em: requisitos possivelmente não executáveis, não executáveis e dificilmente executáveis. O processo de detecção das propriedades em código fonte ocorreu sem a utilização de dados de entrada. A abordagem obteve sucesso na identificação das propriedades e conseguiu relacionar a ocorrência de uma propriedade a como ela afeta os requisitos de teste requeridos por um critério de teste. Durante a avaliação experimental da abordagem, foi utilizado um benchmark de programas concorrentes. Foram utilizados os critérios Todos-usos e Todas-sincronizações. Os resultados indicaram que a abordagem auxiliou o testador na atividade de teste estrutural e, com base na taxa de cobertura do critério Todas-sincronizações, a abordagem conseguiu melhorias estatisticamente significativas. Em conclusão, a abordagem proposta auxilia os profissionais de teste na identificação de requisitos de teste que possam apresentar o problema da não executabilidade. As principais contribuições deste trabalho incluem a elaboração de um catálogo de propriedades, a definição da abordagem Nonexec, a implementação da ferramenta Fi-paths e a integração da ferramenta Valipar. Além disso, foram definidos novos termos relacionados ao problema da não executabilidade, tais como “requisito de teste possivelmente não executável” e “requisito de teste dificilmente executável”.

**Palavras-chave:** Teste de Software, Teste estrutural, Programas concorrentes, Problema da Não Executabilidade.

# ABSTRACT

CHOMA NETO, JOÃO. **An approach to support the identification of non-executability in structural software testing**. 2023. 134 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Software testing activity is essential to ensure the quality of a software product. However, finding a set of test cases that satisfies a specific test criterion is not a simple task, as the input domain is generally vast, and different test sets can be derived with different effectiveness. In the context of structural testing, non-executability (or non-executable test requirements) is a feature present in most programs, increasing the testing activity's cost and effort. New challenges are faced when concurrent programs are tested, mainly related to non-determinism. Non-determinism can result in different possible test outputs for the same test input, making it important to test all possible situations, and, therefore, the non-executability becomes even more complex in this context. To mitigate this problem, an approach called Nonexec was developed, which allows for the automated use of source code-based properties capable of revealing non-executable test requirements. One of the first results was the definition of three sets that classify the requirements as possibly non-executable, non-executable, and hardly executable. The process of detecting source code properties occurred without the use of input data. The approach successfully identified the properties and related the occurrence of a property to how it affects the test requirements of each test criterion. During the experimental evaluation of the approach, a benchmark of concurrent programs was analyzed with the all-uses and all-sync testing criteria. The results indicated that the approach assisted the tester in the structural testing activity and, based on the coverage rate of the all-sync criterion, the approach achieved statistically significant improvements. In conclusion, the proposed approach collaborates testing professionals in identifying test requirements that may present the non-executability problem. The main contributions of this work include the development of a catalog of properties, the definition of the Nonexec approach, the Fi-paths tool, and the integration of the Valipar testing tool. In addition, new terms related to the non-executability problem were defined, such as "possibly non-executable test requirement" and "hardly executable test requirement".

**Keywords:** Software Testing, Structural Testing, Concurrent Programs, Non-Executability Problem..

# LISTA DE ILUSTRAÇÕES

# LISTA DE CÓDIGOS-FONTE

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AG | Algoritmo Genético |
| CC | Complexidade Ciclomática |
| CSP | Problema de Satisfação de Restrição |
| GCD | *Greatest Common Divisor* |
| GFC | Grafo de Fluxo de Controle |
| GFCP | Grafo Fluxo de Controle Paralelo |
| MPI | *Message Passing Interface* |
| MS | Mapeamento Sistemático |
| MW | *Master-Worker* |
| P1 | Propriedade 1 |
| P2 | Propriedade 2 |
| P3 | Propriedade 3 |
| P4 | Propriedade 4 |
| P5 | Propriedade 5 |
| P6 | Propriedade 6 |
| P7 | Propriedade 7 |
| PSO | *Particle Swarm Optimization* |
| SBST | *Search Based Software Testing* |

# SUMÁRIO

CAPÍTULO

1

# INTRODUÇÃO

O processo de garantir a qualidade de um software é composto de diversas atividades e dentre elas está a atividade de teste (PRESSMAN; MAXIM, 2016). Inúmeras pesquisas almejam garantir maior qualidade nos sistemas desenvolvidos e, por esse motivo, produzem e aperfeiçoam métodos e técnicas de desenvolvimento a fim de atender aos requisitos de qualidade (VIJAY-KUMAR; SENNE, 2011). A qualidade do software tem como objetivo garantir correções, além de incluir requisitos não funcionais como capacidade, confiabilidade, eficiência, portabilidade, capacidade de manutenção, compatibilidade e usabilidade (HARMAN; MANSOURI; ZHANG, 2009a; SOMMERVILLE, 2007).

O teste de software é uma atividade da Engenharia de Software inserida no processo de validação e verificação, sendo que seus principais objetivos são: revelar a presença de defeitos e avaliar se o software corresponde a sua especificação (MYERS; SANDLER; BADGETT, 2011). Segundo SOMMERVILLE (2007), o objetivo do teste de software é garantir a qualidade do software nos quesitos de: ausência de falhas e realização de ações para as quais o software foi projetado.

A atividade de teste pode ser realizada em todos os níveis do processo de desenvolvimento de software e pode ser considerada uma das mais custosas (AMMANN; OFFUTT, 2016). Estudos indicam que mais de 50 % do custo total de desenvolvimento de um software é gasto no processo de teste (MYERS; SANDLER; BADGETT, 2011; SOMMERVILLE, 2007). Testar todas as possíveis entradas do software é impraticável devido a grande quantidade de possibilidades de dados de entrada que um software pode ter. Quanto maior a complexidade do software mais recursos computacionais e esforço humano são necessários para garantir a qualidade esperada (MYERS; SANDLER; BADGETT, 2011).

Em geral, as seguintes atividades são desenvolvidas durante o teste de software (MYERS; SANDLER; BADGETT, 2011; PRESSMAN; MAXIM, 2016): (i) planejamento dos testes, (ii) projeto de casos de teste, (iii) execução dos testes e (iv) avaliação dos resultados de testes. Todas

essas atividades podem e devem ser exercitadas durante todo o processo de desenvolvimento de um software (MALDONADO *et al.*, 1998).

Buscando sistematizar a atividade de teste, técnicas e critérios são propostos. As técnicas se diferenciam pela origem da informação utilizada, podendo ser: *técnica funcional*, também conhecida como teste de caixa-preta, na qual informações sobre os requisitos e especificação do sistema são utilizados para derivar os casos de teste; *técnica estrutural*, denominada de teste de caixa-branca, na qual são utilizadas informações sobre a estrutura interna do sistema para derivar os casos de teste; e a *técnica baseada em defeitos*, na qual informações sobre os defeitos mais comuns cometidos durante o processo de desenvolvimento são utilizadas para derivar os casos de teste (MYERS; SANDLER; BADGETT, 2011; PERRY; KAISER, 1990). Cada técnica apresenta critérios de teste, os quais sistematizam a atividade de teste, auxiliando em dois aspectos: 1) na seleção e/ou geração de dados de teste, e 2) na decisão de quando essa atividade pode ser finalizada. É importante destacar que as técnicas de teste se complementam e devem ser aplicadas em conjunto de modo a assegurar um teste de boa qualidade (MALDONADO, 1991).

Os critérios da técnica estrutural utilizam uma medida de cobertura dos testes para avaliar a evolução da atividade, auxiliando a decidir se os testes são suficientes. Neste contexto, a análise de cobertura calcula o percentual de requisitos de teste (por exemplo, comandos do código ou caminhos) por um determinado critério de teste que foram executados pelo conjunto de casos de teste. À medida que a atividade de teste é realizada, novos casos de teste são gerados (ou selecionados) visando executar elementos requeridos não cobertos e com isso melhorar a cobertura dos testes. Neste processo, um problema enfrentado é a não executabilidade, que se refere a caminhos (ou requisitos de teste) que, devido a semântica do programa, não são executáveis. Esse problema é categorizado como um problema de difícil solução, além de ser reconhecido como uma limitação da atividade de teste. Atualmente o problema é tratado por abordagens de alto custo computacional ou pela análise manual do testador, e a principal dificuldade é decidir se um requisito de teste é ou não, executável. A tarefa de automatizar completamente a tomada de decisão é de difícil solução (CLARKE, 1976). Como agravante, tem-se que a maioria dos sistemas de software apresentam requisitos não executáveis (BARHOUSH; ALSMADI, 2013; FRANKL, 1987; BUENO; JINO, 2000; NGO; TAN, 2008; YATES; MALEVRIS, 1989), o que interfere na análise de cobertura e na evolução da atividade de teste.

A não executabilidade afeta o esforço de aplicação de critérios de testes estruturais (VERGILIO; MALDONADO; JINO, 2006; YATES; MALEVRIS, 1989; NGO; TAN, 2008; DELAHAYE; BOTELLA; GOTLIEB, 2015). Este fato eleva o custo de geração de dados de teste e uma solução seria remover (ou identificar) os caminhos não executáveis do processo de teste, acarretando melhorias nos resultados de cobertura de código. A geração de dados de teste pode ser dividida em duas etapas: (i) selecionar um conjunto de caminhos que devem ser executados e (ii) encontrar dados de teste para cobrir esse conjunto de caminhos. Neste formato, a seleção

de um caminho pode afetar incisivamente o processo de geração de dados de teste (BUENO; JINO, 2000). A tarefa de identificar e remover esses caminhos não executáveis da etapa de teste estrutural reduziria drasticamente o custo de geração de dados de teste (NGO; TAN, 2007). Vale ressaltar que quaisquer melhorias trazidas com a mitigação de requisitos de teste não executáveis influenciaria diretamente no desempenho da geração automática de dados de teste, uma vez que, durante a geração automática de dados de teste o algoritmo de geração não ficaria preso em requisitos não executáveis e poderia expandir a inspeção por outros valores no espaço de busca. Apesar disso ser desejado, a sua solução não é trivial (LEE; QIN, 2003; BARHOUSH; ALSMADI, 2013; MARASHDIH; ZAABA, 2018). Alguns trabalhos visam amenizar o problema da não executabilidade, considerando, por exemplo: execução simbólica, análise do fluxo de controle e do fluxo de dados, correlação de arestas, chamada polimórfica, dentre outras abordagens. De maneira geral, estes trabalhos, possuem dependência direta de dados de teste e indicam um alto custo computacional relacionado a tentativa de mitigação do problema da não executabilidade (KUNDU; SARMA; SAMANTA, 2015; PAPADAKIS; MALEVRIS, 2010; WANG; XING; ZHANG, 2014; DU; DONG, 2011; PATHADE; KHEDKER, 2018; NGO; TAN, 2008; XIBO; NA, 2011).

Com a evolução do hardware e a necessidade crescente de maior desempenho e redução no tempo de processamento, novos recursos de programação surgiram, dentre eles, a programação concorrente. O problema da não executabilidade pode estar presente tanto em programas sequenciais quanto em programas concorrentes. Diferentemente dos programas tradicionais, a programação concorrente envolve processos (ou *threads*) que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, na qual esses processos podem ou não concorrer pelos mesmos recursos computacionais. O teste de programas concorrentes encontra limitações excedentes causadas principalmente pelo comportamento não determinístico. Um código concorrente pode executar diferentes caminhos com a mesma entrada de teste, gerando um grande número de sequências de sincronização entre os processos as quais podem ser ou não executáveis (PACHECO, 2011).

A proposição do teste estrutural para programas concorrentes leva em conta o conhecimento adquirido no contexto de programas sequenciais, adaptando-o para tratar problemas específicos de software concorrente. As soluções atuais definem modelos de teste e critérios de cobertura para esse contexto, considerando as os paradigmas de passagem de mensagem e variável compartilhada) (SOUZA *et al.*, 2008; SOUZA *et al.*, 2008; SARMANHO *et al.*, 2008; SOUZA; SOUZA; ZALUSKA, 2014; SOUZA *et al.*, 2015b; MELO *et al.*, 2018; KOJIMA *et al.*, 2009; WONG; LEI; MA, 2005; PACHECO, 2011). Um problema que persiste é a existência de requisitos de teste não executáveis, os quais, em programas concorrentes, tornam-se mais complexos devido à interação entre processos (ou *threads*). Soluções como *Reachability testing* (LEI; CARVER, 2006) possibilitam que o teste seja realizado sem a ocorrência de requisitos não executáveis. Isso é possível porque nessa abordagem, cada nova execução considera apenas requisitos possíveis, derivados da análise dinâmica de uma execução anterior. Entretanto, o alto

custo dessa abordagem dificulta sua aplicação em problemas complexos. Outra desvantagem desta abordagem é que ela não possui uma medida de cobertura para indicar a evolução da atividade de teste (LEI; CARVER, 2006).

## 1.1 Contextualização de Motivação

Este trabalho é relacionado às pesquisas sobre teste de software desenvolvidas pelo grupo de Engenharia de Software do ICMC/USP É um trabalho interdisciplinar, envolvendo teste de software, e programação concorrente.

Estudos indicam que caminhos não executáveis estão presentes em todos os sistemas de software, impactando a atividade de teste de software, em especial, na análise de cobertura dos testes e na geração de dados de teste (YATES; MALEVRIS, 1989; GHIDUK, 2014; DELAHAYE; BOTELLA; GOTLIEB, 2015; PINTO; VERGILIO, 2010; HERMADI; LOKAN; SARKER, 2014; PATHADE; KHEDKER, 2018; FRASER; ARCURI; MCMINN, 2015; CHAWLA; CHANA; RANA, 2015).

Os atuais métodos que mitigam requisitos não executáveis podem ser classificados em duas categorias: (i) análise estática, baseada na execução simbólica, muito embora relacione somente os requisitos executáveis, e (ii) análise dinâmica baseada na geração automática de dados de teste, ou seja, análise em tempo de execução (GONG; YAO, 2010). A análise estática é realizada antes da execução do programa, inicialmente é suposto que todos os requisitos do programa são executáveis, contudo, quando um requisito não executável está sendo testado é impossível encontrar um dado de entrada que o exercite. Este fato, faz com que recursos computacionais sejam desperdiçados na tentativa de cobrir tal caminho (MYERS; SANDLER; BADGETT, 2011; GONG; YAO, 2010).

A análise de cobertura de código é uma das limitações do teste estrutural. A busca por alta cobertura dos critérios de teste é custosa e compromete a eficiência da técnica (DELAHAYE; BOTELLA; GOTLIEB, 2015). A geração automática de dados de teste reduz o custo e aumenta a confiabilidade dos testes de software, contudo, o seu principal desafio é a existência de caminhos não executáveis, em razão do desperdício de recursos ao se testar caminhos não executáveis (NGO; TAN, 2008). A geração de dados de teste pode ser dividida em duas etapas: (i) selecionar um conjunto de caminhos que devem ser executados e (ii) encontrar dados de teste para cobrir esse conjunto de caminhos. Neste formato, a seleção de um caminho pode afetar incisivamente o processo de geração de dados de teste (BUENO; JINO, 2000). A tarefa de identificar e remover caminhos não executáveis da etapa de teste estrutural possibilita reduzir o custo de geração de dados de teste (NGO; TAN, 2007).

A análise manual de requisitos de teste é outra estratégia utilizada para identificação de requisitos de teste não executáveis. Um dos problema dessa estratégia é que após a adição de novos casos de teste a cobertura pode não se alterar causando a falsa impressão que existem

requisitos não executáveis, o que leva o testador a fadiga e frustração, resultando em uma análise errônea dos requisitos de teste (SOUZA *et al.*, 2008; VERGILIO; MALDONADO; JINO, 2006; VERGILIO *et al.*, 2006; VERGILIO; MALDONADO; JINO, 1992b). De outro modo, a análise clássica do fluxo de dados pode apresentar imprecisão o que faz caminhos não executáveis serem erroneamente identificados como executáveis (DING; TAN, 2012).

A determinação automática de requisitos de teste não executáveis, apesar de desejada, ainda é um desafio. Atualmente, as principais técnicas que mitigam o problema da não executabilidade não abordam a análise estática e o custo de execução dos programas encarece e atrasa o processo de teste. Este trabalho contribui nesta direção definindo uma abordagem capaz de identificar requisitos de teste não executáveis em programas concorrentes, minimizando o custo do teste estrutural destes programas.

Neste sentido, os principais aspectos que motivaram o desenvolvimento desta tese de doutorado foram:

- A não executabilidade é um problema presente em diversos paradigmas de programação e, no contexto de programas concorrentes, esse problema se intensifica devido à comunicação e sincronização existente;
- Não há uma abordagem atual que seja capaz de detectar adequadamente os requisitos não executáveis de um programa, pois é conhecido que este é um problema de difícil solução e sua solução não é encontrada em tempo polinomial;
- A detecção de caminhos não executáveis antes da atividade de geração automática de dados de teste possibilita reduzir o custo desta atividade, dado que sem os requisitos não executáveis a geração automática de dados de teste consegue evoluir mais satisfatoriamente.

## 1.2 Objetivos e Hipóteses

Considerando o panorama apresentado anteriormente, o objetivo desta tese de doutorado é investigar a viabilidade de determinar automaticamente requisitos de teste não executáveis para auxiliar o teste estrutural de programas sequenciais e concorrentes. Dessa forma, a hipótese desta tese é formulada da seguinte maneira:

**É possível mitigar o problema da não executabilidade por meio da utilização de um catálogo de propriedades de identificação de requisitos não executáveis.**

Para apoiar a hipótese, pretende-se responder a seguinte questão de pesquisa:

- É possível mitigar o problema da não executabilidade por meio do uso de um catálogo de propriedades para identificação de requisitos não executáveis?

Com base na questão de pesquisa principal, este trabalho visa responder as seguintes questões de pesquisa secundárias:

1. Quais propriedades para identificação de requisitos não executáveis podem ser definidas?
2. Quais propriedades para identificação de requisitos não executáveis podem ser aplicadas de forma automática?
3. Quais são os benefícios alcançados com a aplicação das propriedades de identificação de caminhos não executáveis na atividade de teste estrutural?
4. Quais propriedades para identificação de requisitos são viáveis para apoiar o problema da não-executabilidade em programas concorrentes?

Tendo em vista a motivação e a questão de pesquisa, o objetivo geral deste trabalho é apresentar uma abordagem de análise estática denominada *Nonexec*, que visa a utilização de propriedades para a identificação de requisitos de teste não executáveis que podem ser aplicadas sem a execução do programa.

O objetivo ainda pode ser subdividido nos seguintes objetivos específicos:

- Identificar e definir propriedades para identificação de requisitos de teste não executáveis para o teste estrutural;
- Aplicar e estender o conjunto de propriedades de identificação de requisitos não executáveis para o contexto de programas concorrentes;
- Automatizar a aplicação das propriedades para a identificação de requisitos não executáveis;
- Avaliar empiricamente as propriedades propostas.

## 1.3 Organização

Esta tese está organizada da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica utilizada neste trabalho. Nela estão os conceitos necessários para compreensão e embasamento da proposta, bem como, explicações sobre teste estrutural, problema da não executabilidade e conceitos importantes sobre programas concorrentes. No Capítulo 3 são apresentados, por meio de um Mapeamento Sistemático (MS), os trabalhos relacionados ao problema tratado neste trabalho. O Capítulo 4 apresenta a abordagem desenvolvida e o catálogo de propriedades para identificação de requisitos de teste não executáveis. O Capítulo 5 discute a aplicação da abordagem *Nonexec* no contexto de teste estrutural para programas sequenciais e concorrentes. Por fim, o Capítulo 6 apresenta as descobertas alcançadas, as contribuições para o tratamento

do problema da não executabilidade e os potenciais estudos que poderão ser desenvolvidos no futuro, a partir desta tese.

CAPÍTULO

# 2

# REFERENCIAL TEÓRICO

## 2.1 Considerações Iniciais

Neste capítulo são abordados os conceitos relacionados ao tema desta tese de doutorado. De maneira geral os conceitos expostos se concentram em teste de software, problema da não executabilidade e programação concorrente.

## 2.2 Teste de Software

Teste de software é uma das principais atividades empregadas para garantir a qualidade do software em desenvolvimento e, segundo a literatura, pode consumir mais de 50% do custo total de desenvolvimento (MYERS; SANDLER; BADGETT, 2011; SOMMERVILLE, 2007). O objetivo da atividade de teste é garantir que o software realize as atividades para as quais ele foi projetado, com o mínimo de falhas possível (FRANKL, 1987; AMMANN; OFFUTT, 2016).

O processo de teste de software se preocupa em identificar a presença de defeitos por meio de diferentes atividades (MYERS; SANDLER; BADGETT, 2011). Este processo pode ser dividido em quatro atividades principais: (i) planejamento da atividade de teste, (ii) projeto de casos de teste, (iii) execução dos casos de testes e (iv) avaliação da atividade de teste. Todas as atividades de teste podem e devem ser exercitadas durante todo o processo de desenvolvimento de um software visando garantir sua qualidade o início do desenvolvimento (MALDONADO *et al.*, 1998).

Com objetivo de padronizar a atividade de teste foram definidos quatro conceitos básicos pelo padrão IEEE 610.12 (610.12-1990, 1990):.

- (i) **Defeito** (*fault*) é um passo, processo ou definição de dados que está incorreto em um programa de computador;

- (ii) **Engano** (*mistake*) é entendida como uma ação humana que produz um defeito;
- (iii) **Erro** (*error*) é um estado interno incorreto de um programa que é a manifestação de algum defeito; e
- (iv) **Falha** (*failure*) é quando o erro se propaga para a saída do programa, levando a uma saída diferente da esperada, ou seja, o programa produz um comportamento incorreto em relação a sua especificação.

Associar uma falha a um defeito do programa não é uma tarefa trivial, não havendo garantias da existência de uma entrada que alcance um determinado defeito e, ainda, gere uma saída incorreta. Deste modo foi proposto um modelo *Fault / Failure* que indica três condições para que um defeito seja observado. Este modelo, também intitulado RIP (*Reachability, Infection* e *Propagation*), é utilizado para critérios de cobertura, teste de mutação e geração automática de dados de teste (AMMANN; OFFUTT, 2016).

1. ***Reachability:*** O local do defeito deve ser alcançado.
2. ***Infection:*** Depois de alcançar o defeito, o estado do programa deve estar incorreto.
3. ***Propagation:*** O estado incorreto deve se propagar até que uma saída do programa esteja incorreta.

Um caso de teste é um conjunto de condições utilizadas para evidenciar defeitos em sistemas de software, as condições são: entrada do teste, pré e pós condições para execução do teste, computação e resultado esperado. Para que a atividade de teste tenha sucesso, ela deve determinar casos de teste para os quais o programa testado falhe. Quanto maior a qualidade dos casos de teste gerados, maior a chance de que os casos evidenciem defeitos no sistema e menor será o número de casos de teste necessários para garantir a qualidade do software (DELAMARO; JINO; MALDONADO, 2017; MYERS; SANDLER; BADGETT, 2011).

Pelo fato de ser possível aplicar as atividades de teste em diferentes momentos do processo de desenvolvimento do software, o processo de testar um software pode ser dividido em fases. O **Teste de Unidade** tem como objetivo testar as menores unidades que compõem o programa, exercitando cada módulo separadamente e evidenciando defeitos de implementação. O **Teste de Integração** tem o objetivo de assegurar que o sistema irá funcionar como o esperado depois do processo de integração dos seus módulos. O **Teste de Sistema** tem o objetivo de avaliar o sistema integralmente e validar se todos os requisitos foram implementados da forma como foram especificados (DELAMARO; JINO; MALDONADO, 2017).

Durante a atividade de teste, diferentes tipos de informação podem ser empregadas para apoiar a seleção ou geração de casos de teste. As técnicas de teste se diferenciam pela informação utilizada para derivação dos requisitos de teste. Um requisito de teste é uma informação que

deve ser coberta por um caso de teste, por exemplo: funcionalidades, linhas de código, caminhos, dentre outros (DELAMARO; JINO; MALDONADO, 2017). Três técnicas de teste se destacam (PERRY; KAISER, 1990):

- **Técnica Funcional:** a técnica funcional é também denominada teste de caixa-preta. Ela é baseada nas especificações do sistema. Tem como objetivo principal identificar se o programa satisfaz os requisitos funcionais e não-funcionais especificados pelo cliente.
- **Técnica Estrutural:** a técnica de teste estrutural, também denominada teste de caixa-branca, utiliza o código fonte para extrair informações a serem testadas. Os requisitos de teste são definidos com base na implementação do software, isto é, na estrutura interna do software, analisando: caminhos, laços de repetição, definição e utilização de variáveis.
- **Técnica baseada em defeitos:** a técnica baseada em defeitos utiliza informações sobre os defeitos típicos cometidos no processo de desenvolvimento para a geração de casos de teste. Essa técnica tem sido conhecida como Teste de Mutação, devido a esse ser o critério mais conhecido desta técnica.

Cada técnica de teste possui um conjunto de critérios de teste, os quais definem as propriedades que devem ser avaliadas no teste e, assim, quanto melhor o critério de teste maiores são as chances de minimizar o número de casos de teste necessários para garantir a qualidade do software (AMMANN; OFFUTT, 2016). Os critérios de teste ajudam no processo de seleção/geração de dados de teste e em decidir quando parar a atividade de teste. Todo critério de teste é composto por um conjunto de requisitos de teste que devem ser cobertos pelos casos de teste. Quando um requisito de teste (por exemplo, um caminho) não pode ser executado com nenhum caso de teste possível, este requisito é considerado não executável (DELAMARO; JINO; MALDONADO, 2017; MYERS; SANDLER; BADGETT, 2011)

Dado a importância do teste estrutural para este trabalho, ele é detalhado a seguir.

### 2.2.1 Teste Estrutural

A atividade de teste estrutural utiliza a estrutura de dados Grafo para representar um modelo abstrato que extrai requisitos de teste do código fonte. O comportamento do código fonte de um programa pode ser representado por Grafo de Fluxo de Controle (GFC), que é um grafo direcionado composto por nós e arestas/arcos. Os elementos do GFC têm os seguintes significados: um nó corresponde a uma instrução simples ou à avaliação de uma expressão lógica em uma instrução composta; As arestas denotam o potencial fluxo de controle entre as instruções. Um caminho de fluxo de controle do programa é representado por um caminho em um GFC, que inicia a partir do nó de entrada (ZHANG; WANG, 2001; WATSON; WALLACE; MCCABE, 1996).

A Figura 1 representa o código apresentado no Algoritmo 1. Neste figura é apresentado um exemplo de representação abstrata do código fonte em processo de teste estrutural.

Com auxílio do GFC durante o teste estrutural é possível realizar a análise do fluxo de controle relacionado ao comportamento lógico do código e, a análise do fluxo de dados relacionado ao comportamento dos dados atribuídos às variáveis do código. A análise de fluxo de controle e dados pode ser dividida em três etapas: (1) Identificação de um conjunto de caminhos que atenderão ao critério selecionado; (2) Geração de um conjunto de dados de teste que fará com que os caminhos sejam executados; (3) Execução do programa com os dados de teste (NGO; TAN, 2007; SOUZA *et al.*, 2008).

Geralmente os critérios de teste estrutural são divididos em 03 tipos de critérios:

**Critérios baseados na complexidade (ou Critérios de McCabe):** Uma propriedade de grande importância para o teste estrutural é denominada Complexidade Ciclomática (CC). Esta propriedade pode ser definida como $V(G)$ enquanto $V$ significa o número ciclomático na teoria dos grafos e $G$ significa que a complexidade é uma função do grafo. CC é inteiramente baseada na estrutura do GFC e mede a quantidade de lógica de decisão do grafo. Há muitas fórmulas para calcular a complexidade ciclomática e a mais comum é $V(G) = e - n + 2$, onde $e$ representa o número de arestas na GFC e $n$ o número de nós do GFC (WATSON; WALLACE; MCCABE, 1996; HERMADI; LOKAN; SARKER, 2014). Para um programa sem *loops*, o número de caminhos lógicos é igual ao seu número de CC ou número de caminhos independentes / básicos (HERMADI; LOKAN; SARKER, 2014). Este critério também pode ser mensurado da seguinte forma: $caminhos\ independentes = numero\ de\ ns\ predicativos + 1$. Um nó predicativo em um GFC apresenta uma toma de decisão, como um *if* ou *while*.

**Critérios Baseados em Fluxo de Controle:** Os critérios de teste baseados em fluxo de controle tem como objetivo avaliar a execução do programa utilizando propriedades de controle da execução do programa como, tomadas de decisão (MYERS; SANDLER; BADGETT, 2011). Dentre os critérios, existem:

- **Todos-Nós:** cada nó do GFC deve ser executado pelo menos uma vez;
- **Todas-Arestas:** cada aresta do GFC deve ser executada ao menos uma vez;
- **Todos-Caminhos:** todo caminho deve ser executados pelo menos uma vez;

**Critérios Baseados em Fluxo de Dados:** O fluxo de dados existente no código do programa é utilizado como fonte de informação para derivar os requisitos de teste. Desta forma, esses critérios baseiam-se nas associações entre a definição de uma variável e seus possíveis usos. Dentre os critérios, existem:

1. **Todas-Definições:** Exige que cada definição de variável seja executada pelo menos uma vez, c-uso ou p-uso;
2. **Todos-Usos (*All-uses*):** Requer que todas as associações entre uma definição de variável e seus usos (c-uso ou p-uso) sejam executadas pelos casos de teste, por pelo menos um caminho livre de definição;
3. **Todos-Potenciais-Du-Caminhos:** Requer que toda associação entre uma definição de variável e seus subsequentes usos, seja executada por todos os caminhos livres de definição e de laços que abrangem esta associação.

Os critérios de teste da técnica de teste oferecem uma medida de cobertura do programa. Com base nos requisitos de teste a análise de cobertura calcula o número de requisitos cobertos, por exemplo, o número de caminhos executados por um conjunto de casos de teste. Essa informação representa o grau de cobertura de um conjunto de testes com relação a determinado critério. A cobertura de 100 % raramente pode ser alcançada em um programa real devido à existência de caminhos não executáveis. Nesta situação, a pergunta que fica é: **Quantos casos de testes são necessários para garantir a qualidade do software?** (NGO; TAN, 2007).

Muitas vezes, mesmo com a utilização de diferentes critérios de teste, não é possível que o desenvolvedor consiga determinar se o comportamento do software está correto ou não (DELAMARO; JINO; MALDONADO, 2017). Por exemplo, não é possível definir casos de teste suficientes para exercitar todos os caminhos possíveis de um software. Dessa forma, o critério de teste Todos-Caminhos acaba não alcançando cobertura satisfatória, fato que ocasiona a falsa sensação de que o software está incorreto (DELAMARO; JINO; MALDONADO, 2017; BARHOUSH; ALSMADI, 2013; FRANKL, 1987; BUENO; JINO, 2000). Este problema é conhecido com o problema da não executabilidade.

Por definição um caminho é dito não executável se não existe um dado de entrada que leve à execução deste caminho. Em outras palavras, um caminho é executável se existir ao menos um caso de teste que ocasione a execução deste caminho. Este problema é considerado como de difícil solução (CLARKE, 1976; BUENO; JINO, 2000) e se apresenta em todos os programas computacionais já desenvolvidos (BARHOUSH; ALSMADI, 2013; FRANKL, 1987; BUENO; JINO, 2000; NGO; TAN, 2008; YATES; MALEVRIS, 1989). A determinação de caminhos não executáveis depende do entendimento da semântica do programa, o que é uma atividade custosa e propensa a erros, normalmente realizada manualmente (BARR *et al.*, 2015).

Durante a atividade de teste, existem três tarefas que necessitam da intervenção humana, sendo elas: (1) a geração (ou seleção) de entradas para o teste, (2) a análise das saídas dos testes para verificar se estão corretas ou se ocorreu um erro e (3) análise da não executabilidade de requisitos de teste. No terceiro caso, a tarefa não pode ser totalmente automatizada, pois é um problema indecidível, aumentando o esforço e custo da atividade de teste de software (DELAMARO; JINO; MALDONADO, 2017).

Embora seja difícil resolver o problema geral de identificar todos os caminhos não executáveis, o conhecimento sobre caminhos não executáveis pode ser usado para melhorar o desempenho da atividade de teste. Quando identificados, os caminhos não executáveis podem ser desconsiderados da atividade de teste contribuindo com o aumento da cobertura de código e, com a redução do esforço de geração de dados de teste (BODÍK; GUPTA; SOFFA, 1997). Em especial, a geração automática de dados de teste pode se beneficiar positivamente se os requisitos de teste não executáveis forem eliminados do conjunto usado para evoluir a geração automática.

A existência de caminhos não executáveis afeta diretamente o processo de geração automática de dados de teste. Devido a existência de caminhos não executáveis, existirão requisitos requeridos por critérios de teste não satisfeitos, o que impedirá a parada da geração automática de dados de teste e, neste caso, outro critério de parada deve ser utilizado o que pode não garantir uma alta eficácia em revelar defeitos (DELAMARO; JINO; MALDONADO, 2017). Na próxima seção mais conceitos sobre o problema da não executabilidade serão apresentados.

### 2.2.2 Problema da Não Executabilidade

Uma questão fundamental no teste de software estrutural é a existência de caminhos não executáveis (BUENO; JINO, 2000). Apesar do grande número de propostas e abordagens para resolução de problemas da engenharia de software, não há algoritmo geral para determinar a execução de um caminho, ou seja, o problema da não executabilidade é definido como uma questão de difícil solução (CLARKE, 1976; FRANKL, 1987).

Os caminhos não executáveis estão presentes em todos os programas, por consequência, o problema da não executabilidade se apresenta em todas as categorias de sistemas de software, sendo imprescindível seu tratamento (NGO; TAN, 2008; YATES; MALEVRIS, 1989). Os métodos comumente usados para determinação de caminhos não executáveis são parciais, fornecendo soluções apenas para casos específicos (BUENO; JINO, 2000). Detectar todos os caminhos não executáveis durante a análise estática resolveria, de forma ideal, o problema da não executabilidade, beneficiando de forma direta o teste estrutural e análise de cobertura (BUENO; JINO, 2000).

Para ilustrar o problema da não executabilidade, considere o Algoritmo 1, *Identifier*. O algoritmo tem como objetivo determinar se um identificador é válido ou não. Um identificador é válido se começar com uma letra e possuir somente letras e dígitos. Além disso, deve ter no mínimo 1 e no máximo 6 caracteres de comprimento. Na sequência, a Figura 1 representa o GFC deste código. Para auxiliar na visualização, cada nó do grafo foi indicado no código com um comentário, desta forma, o fluxo de execução do código pode ser acompanhado no GFC. Nesta análise foi utilizado o teste de *McCabe* para derivar alguns caminhos (conhecidos como básicos) a partir deste programa. Como resultado tem-se 05 caminhos para esta análise:
$C_1 = (1,2,3,4,5,7,4,8,9,11)$
$C_2 = (1,3,4,5,7,4,8,9,11)$

$C_3 = (1,3,4,8,9,11)$
$C_4 = (1,2,3,4,5,6,7,4,8,9,11)$
$C_5 = (1,2,3,4,5,7,4,8,10,11)$

```
#include<stdio.h>
#include<stdlib.h>

char ch;
int valid_s (ch) {
  if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')))
    return (1);
  else
    return (0);
}

int valid_f (ch) {
  if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')) || ((ch >= '0')
   && (ch <= '9')))
    return (1);
  else
    return (0);
}

/*1*/int main() {
/*1*/ char achar;
/*1*/ int length, valid_id;
/*1*/ length = 0;
/*1*/ printf ('Identificador: ');
/*1*/ achar = fgetc (stdin);
/*1*/ valid_id = valid_s(achar);
/*1*/ if (valid_id)
/*2*/   length = 1;
/*3*/ achar = fgetc (stdin);
/*4*/ while (achar != '\n') {
/*5*/   if (!(valid_f(achar)))
/*6*/     valid_id = 0;
/*7*/   length++;
/*7*/   achar = fgetc (stdin);
/*7*/ }
/*8*/ if (valid_id && (length >= 1) && (length < 6) )
/*9*/   printf ('Valido\n');
/*10*/  else
/*10*/    printf ('Invalido\n');
/*11*/}
```

**Código-fonte 1** – Programa *Identifier*.

Os caminhos $C_1$, $C_2$, $C_4$ e $C_5$ são executáveis com os respectivos dados de entrada [1a], [], [a], [1]. O caminho $C_3$ = (1,3,4,8,9,11) é não executável pois para passar por esse caminho só é possível com um identificador inválido e para isso não seria executado o nó 9, que assume um identificador válido. Desta forma, não há um dado de entrada que o exercite e, com isso, o caminho $C_3$ é não executável. Considerando que os caminhos a serem executados são derivados estaticamente pode-se observar que a quantidade de caminhos não executáveis para este exemplo

pode ser grande, pois a execução dos nós 09 e 10 dependem da execução ou não de nós anteriores.

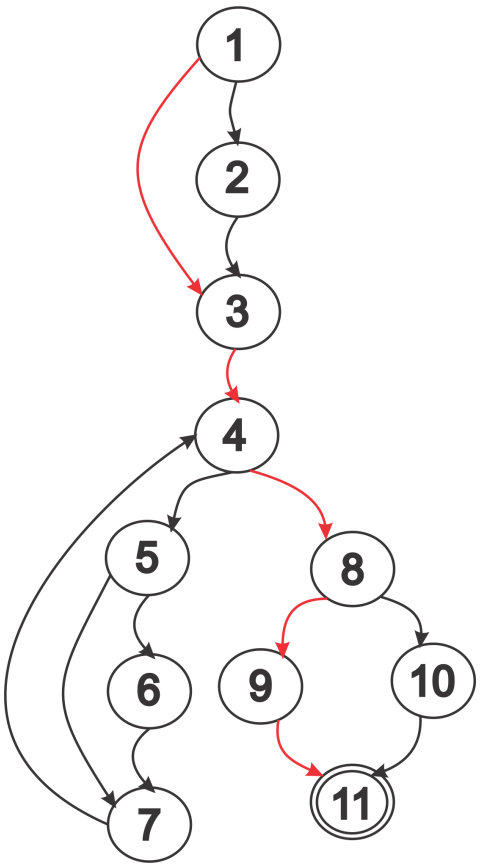


Figura 1 – Grafo de Fluxo de Controle Sequencial com Caminho Não Executável.

Como pode ser percebido no exemplo anterior, apesar de simples, o código apresentou um caminho não executável. Este tipo de caminho pode elevar o custo de geração automática de dados de teste até que se conclua que tal caminho é não executável. Fica claro que o problema da não executabilidade interfere diretamente no custo de geração de dados de teste o que pode impedir que o programa alcance níveis satisfatórios de qualidade durante a análise estática. De forma simplista, para melhorar os resultados de cobertura de código, deve-se remover os caminhos não executáveis existentes no GFC, contudo tal ação não é trivial (LEE; QIN, 2003; BARHOUSH; ALSMADI, 2013; MARASHDIH; ZAABA, 2018).

Geralmente os trabalhos que abordam o problema da não executabilidade são baseados em processamento simbólico e análise de fluxo de dados estático (BUENO; JINO, 2000). Outros pesquisadores consideraram diferentes padrões de não executabilidade, tais como: correlação de arestas, chamada polimórfica, identificação baseada em heurística, automatização da execução simbólica, análise estática e dinâmica automática, remoção manual de caminhos não executáveis e análise de busca de caminho não executável (KUNDU; SARMA; SAMANTA, 2015; MARASHDIH; ZAABA, 2018).

A não executabilidade de um caminho é causada por diferentes razões, por exemplo: presença de código morto, cláusulas conflitantes e declaração condicionais correlacionadas em

termos de uma variável (MARASHDIH; ZAABA, 2018). Por exemplo, um caminho é não executável devido a predicados contraditórios que precisam estar satisfeitos para executar o caminho (HEDLEY; HENNELL, 1985).

A execução simbólica é uma técnica de teste estudada por vários pesquisadores durante os últimos 30 anos, sua ideia básica é executar estaticamente um programa usando valores simbólicos para variáveis. Um problema chave com a execução simbólica é decidir se um conjunto de restrições é satisfatório (ZHANG; CHEN; WANG, 2004). Uma limitação existente nesta técnica está no manuseio de ponteiros, matrizes e chamadas de função que resulta na descoberta de apenas uma pequena porcentagem de caminhos não executáveis (NGO; TAN, 2007). Embora a execução simbólica forneça uma solução parcial para o problema de caminhos não executáveis, ela falha em gerar dados de teste pois o estado de qualquer variável é determinado apenas em tempo de execução (DAHIYA; CHHABRA; KUMAR, 2011).

A detecção manual de caminhos não executáveis é outra abordagem comumente utilizada para tratar o problema da não executabilidade. Apesar disso, nenhuma das heurísticas existentes é capaz de detectar um grande número de caminhos não executáveis (NGO; TAN, 2007). Em uma visão antagônica, os algoritmos de geração automática de dados de teste que podem ser utilizados para detectar caminhos não executáveis, monitorando a execução do programa testado. No entanto, a geração de dados de teste geralmente usa a execução simbólica; assim, é quase tão caro quanto a execução simbólica (NGO; TAN, 2007).

A execução simbólica é uma técnica popular de análise de programas introduzida em meados dos anos 70 para testar se certas propriedades podem ser violadas por um software. Em uma execução concreta, um programa é executado em uma entrada específica e um único caminho de fluxo de controle é explorado. Assim, na maioria dos casos, as execuções concretas podem apenas subestimar a análise do critério de interesse. Em contraste, a execução simbólica pode explorar simultaneamente múltiplos caminhos que um programa pode tomar sob diferentes entradas. A ideia chave é permitir que um programa tome em valores de entrada simbólicos ao invés de concretos. A execução é realizada por uma execução simbólica que mantém para cada caminho um fluxo de controle explorado. O modelo é construído para satisfazer condições existentes no caminho e mapeia variáveis e expressões. O processo se desenvolve por meio de um modelo que indica se há violações e se o caminho é executável. Trabalhando com o conceito de exploração a execução simbólica encontra limitações relacionadas ao tamanho dos programas testados, uma vez que todos os caminhos devem ser verificados para garantir que o programa esteja correto. Além disso, o modelo tem dificuldades com quando encontra ciclos nos programas testados. Por fim, a execução simbólica encontra limitações quando se depara com comportamentos não modelados em diretrizes simbólicas (AVGERINOS *et al.*, 2014; BALDONI *et al.*, 2018; YOUNG; TAYLOR, 1988).

Um dos principais desafios da execução simbólica é o problema da explosão de caminhos: um executor simbólico pode criar um novo estado em cada ramificação do programa, e o

número total de estados pode facilmente se tornam exponenciais no número de ramificações. Acompanhar um grande número de pendências ramificações a serem exploradas, por sua vez, impactam tanto no tempo de execução quanto nos requisitos de espaço de análise do modelo simbólico (AVGERINOS *et al.*, 2014; BALDONI *et al.*, 2018; YOUNG; TAYLOR, 1988).

Como já mencionado, o problema da não executabilidade está presente em todas as categorias de sistemas (NGO; TAN, 2008; YATES; MALEVRIS, 1989), dentre eles os programas concorrentes. Dado que os programas concorrentes são tratados nesta tese, a seguir tem-se detalhes do teste estrutural destes programas.

### 2.2.3 Teste de Programas Concorrentes

Os programas concorrentes podem ser vistos como aplicações divididas em pequenas partes executáveis onde cada parcela resolve uma fração do problema, permitindo assim, uma melhor utilização dos recursos disponíveis e a redução do tempo de processamento (PACHECO, 2011). Muito embora os benefícios da computação concorrente sejam evidentes, devido ao aumento de complexidade são necessários recursos que permitam a execução de processos simultaneamente como, ativação e finalização de processos (PACHECO, 2011; DELAMARO; JINO; MALDONADO, 2017).

Essencialmente a programação concorrente reduz o tempo de execução das aplicações e pode ser utilizada nos mais diversos problemas: previsão do tempo, simulação molecular dinâmica, bioinformática e processamento de imagens (RAPPS; WEYUKER, 1985; PACHECO, 2011; MELO *et al.*, 2018). Estes benefícios exigem altos níveis de confiabilidade requerendo ainda mais da atividade de teste e validação. Características presentes em programas concorrentes como: não determinismo, disputa de recursos computacionais, sincronização e comunicação de processos, torna a atividade de teste mais complexa (SOUZA *et al.*, 2008).

Um processo é definido como um programa em execução sendo composto por um programa executável, dados, contador de instruções e registradores (TANENBAUM, 2009). A concorrência pode ser definida como um instante de tempo em que dois ou mais processos iniciam sua execução, mas ainda não a terminaram. O paralelismo pode ser entendido como a execução de dois processos ao mesmo tempo implicando na existência de mais de um processador. A existência de vários processos em um único processador induz a sensação de paralelismo, contudo, um único processo é executado em cada instante de tempo e os outros aguardam a liberação do processador para continuarem executando. Esta possibilidade pode ser chamada de pseudo-paralelismo ou, de uma forma mais genérica, de concorrência (ALMASI; GOTTLIEB, 1994).

As atividades de comunicação e sincronização são responsáveis por coordenar os processos concorrentes. A comunicação permite que um processo interfira na execução de outro processo e a sincronização garante que o acesso simultâneo não torne os dados compartilhados

inconsistentes (PACHECO, 2011).

A construção de programas concorrentes pode seguir dois diferentes modelos, classificados de acordo com a maneira em que a memória é utilizada. O modelo de **passagem de mensagem** tem sua memória distribuída, ou seja, cada processo possui seu próprio espaço de endereçamento, estabelece a comunicação e a sincronização entre os processos por meio da passagem de mensagem, na qual cada processo é executado por uma unidade de processamento que possui seu próprio espaço de endereçamento e a comunicação acontece por meio das primitivas *send* e *receive*, sendo que os processos continuam a competir por recursos computacionais. No modelo de **memória compartilhada** as variáveis fazem uso do mesmo espaço de endereçamento o que permite a comunicação entre os processos por meio de variáveis compartilhadas e a sincronização é obtida com a utilização de semáforos e monitores (ALMASI; GOTTLIEB, 1994; PACHECO, 2011). Esses dois paradigmas apresentam características diferentes, as quais necessitam ser consideradas durante os testes.

Dentre os desafios encontrados durante a atividade de teste de programas concorrentes, destaca-se o comportamento não determinístico. Este comportamento faz com que um programa apresente comportamentos distintos com a mesma entrada de teste (PACHECO, 2011). Por exemplo, dada uma mesma entrada de teste, o não determinismo existente entre a comunicação dos processos permite que diferentes caminhos sejam exercitados no programa.

No trabalho de Souza SOUZA *et al.* é introduzido o critério de teste de programas concorrentes que interagem através dos paradigmas de passagem de mensagem e memória compartilhada:

1. **Todas-sincronizações (*All-Sync*):** requer que o conjunto de teste execute todos as sincronizações existentes entre os processos.

O não determinismo dificulta a reprodução de uma execução de teste ou a repetição de uma execução para depuração. Isso também implica que um determinado conjunto de dados de teste pode não executar o caminho a ser coberto durante uma execução de teste específica (ZHOU *et al.*, 1998).

O problema da não executabilidade de caminhos foi herdado dos programas sequenciais e se apresenta de forma mais complexa nos programas concorrentes. Neste caso, o problema se estende às sequências de sincronização onde se faz necessário identificar quais combinações de sequências de sincronização são executáveis ou não, para um determinado dado de teste (DELAMARO; JINO; MALDONADO, 2017; DIONNE; FEELEY; DESBIEN, 1996). Na presença do não determinismo, a identificação de caminhos não executáveis é mais desafiadora, pois alguns caminhos executáveis nem sempre executam.

Para ilustrar o problema da não executabilidade em programas concorrentes, considere os Algoritmos 2 e 3, os quais representam códigos de um programa concorrente, paradigma de

passagem de mensagem, implementados em *Message Passing Interface* (MPI) (DELAMARO; JINO; MALDONADO, 2017), *Greatest Common Divisor* (GCD). O programa calcula o máximo divisor comum entre três valores inteiros, utilizando 4 processos (1 *master* e 3 *workers*), os quais se comunicam por passagem de mensagem. O processo principal (*Master*) recebe três números inteiros como entrada e, envia dois números para cada processo (*Worker*) calcular o máximo divisor comum. A ordem em que os processos receberão os valores é definida pelo usuário. Cada processo (*Worker*) calcula o máximo divisor comum entre os dois valores recebidos e retorna esse valor ao *Master*. Dependendo dos valores de entrada fornecidos, dois ou três processos (*Workers*) são utilizados para o cálculo. Além disso, a ordem de recepção dos valores pelo *master* é não determinístico. Independentemente disso, sempre haverá 3 processos escravos em execução.

```
/*1*/void Master(int x, int y, int z)
/*1*/{
/*1*/ int buf[2], l1, l2, l3;
/*1*/ // envia x e y
/*1*/ printf("Ordem de envio(1,2,3)");
/*1*/ scanf("%d%d%d", &l1, &l2, &l3);
/*1*/ buf[0] = x;
/*1*/ buf[1] = y;
/*2*/ MPI_Send(buf,2,MPI_INT,l1,1,MPI_COMM_WORLD);
/*3*/ // envia y e z
/*3*/ buf[0] = y;
/*3*/ buf[1] = z;
/*3*/ MPI_Send(buf,2,MPI_INT,l2,1,MPI_COMM_WORLD);
/*4*/ MPI_Recv(buf,1,MPI_INT,MPI_ANY_SOURCE,2,
      MPI_COMM_WORLD, &status);
/*4*/ x = buf[0];
/*5*/ MPI_Recv(buf,1,MPI_INT,MPI_ANY_SOURCE,2,
      MPI_COMM_WORLD, &status);
/*5*/ y = buf[0];
/*6*/ if (x > 1 && y > 1)
/*6*/ {
/*7*/   buf[0] = x;
/*7*/   buf[1] = y;
/*7*/   MPI_Send(buf,2,MPI_INT,l3,1,MPI_COMM_WORLD);
/*8*/   MPI_Recv(buf,1,MPI_INT,MPI_ANY_SOURCE,2,
        MPI_COMM_WORLD,&status);
/*8*/   z = buf[0];
/*8*/ }
/*9*/ else
/*9*/ {
/*9*/   z = 1;
/*10*/  }
/*10*/  printf("resultado=%d\n", z);
/*10*/}
```

**Código-fonte 2** – Programa GCD em MPI: Processo *Master*.

```
/*1*/void Slave(int rank)
/*1*/{
/*1*/ int buf[2];
/*2*/ MPI_Recv(buf,2,MPI_INT,0,MPI_ANY_TAG,
        MPI_COMM_WORLD,&status);
```

```
/*3*/ while (buf[0] != buf[1])
/*3*/ {
/*4*/ if (buf[0] < buf[1])
/*5*/     buf[1] = buf[1] - buf[0];
/*6*/ else
/*6*/     buf[0] = buf[0] - buf[1];
/*7*/ }
/*8*/ MPI_Send(buf,1,MPI_INT,0,2,
              MPI_COMM_WORLD);
/*9*/}
```

**Código-fonte 3** – Programa GCD em MPI: Processo *Worker*.

Para derivar informações de teste um modelo denominado Grafo Fluxo de Controle Paralelo (GFCP) (SOUZA *et al.*, 2008) é utilizado para representar o programa em teste. Assim, as linhas numeradas nos códigos dos Algoritmos 2 e 3 representam os nós do grafo da Figura 2. Já, as linhas tracejadas no grafo da Figura 2 representam a troca de mensagem entre os processos, ou seja, são arestas de sincronização entre processos. Vale ressaltar que as primitivas de comunicação são destacados em um nó individual.

Por exemplo, as mensagens enviadas pelo processo *Master* ($P^m$) acontecem nos nós 2, 3 e 7. Como o usuário irá informar a ordem em que os processos devem ser executados, o nó 2, por exemplo, pode enviar a mensagem para $P^0$, $P^1$ ou $P^2$, dependendo da sequência definida. Assim, é necessário, durante a atividade de teste, garantir que todas essas possibilidades sejam testadas. A não executabilidade ocorre quando, de maneira estática, define-se quais caminhos deseja-se testar. Para programas concorrentes, a execução de um caminho compreende a execução dos caminhos de cada processo, dado um dado de teste. Nesse contexto, o caminho que deve ser percorrido pelo teste, compreende os caminhos de cada processo (denotados pelos nós $n_p^i$, onde $p$ é o identificador do processo e $i$ o identificador de cada nó), somados às arestas de sincronização. Um exemplo de caminho que seria derivado e é não executável, seria:

> **Nota:** As arestas de sincronização (tracejadas) em vermelho representam o **caminho** não executável. De forma adicional, as arestas de sincronização (tracejadas) em azul representam todas as outras arestas de sincronização possíveis.

$$
\begin{aligned}
&\mathit{Caminho} = (\\
&P^m = \{\ n_m^1, n_m^2, n_m^3, n_8^0, n_m^4, n_8^1, n_8^5\ n_m^6, n_m^9, n_m^{10}\},\\
&P^0 = \{\ n_0^1, n_m^2, n_0^2\ , n_0^3, n_0^4, n_0^5, n_0^6, n_0^7, n_0^3, n_0^8, n_0^9\},\\
&P^1 = \{\ n_1^1, n_m^2, n_1^2\ , n_1^3, n_1^4, n_1^5, n_1^6, n_1^7, n_1^3, n_1^8, n_1^9\},\\
&P^2 = \{\ n_2^1, n_m^3, n_2^2\ , n_2^3, n_2^4, n_2^5, n_2^6, n_2^7, n_2^3, n_2^8, n_2^9\}\ \ )
\end{aligned}
$$

Esse caminho é não executável pois não é possível que $n_m^2$ envie mensagem para os dois processos ($P^0$ e $P^1$), pois envia somente para um a cada execução. Outras combinações dessas possibilidades são derivadas, gerando vários caminhos não executáveis. Por essa razão,

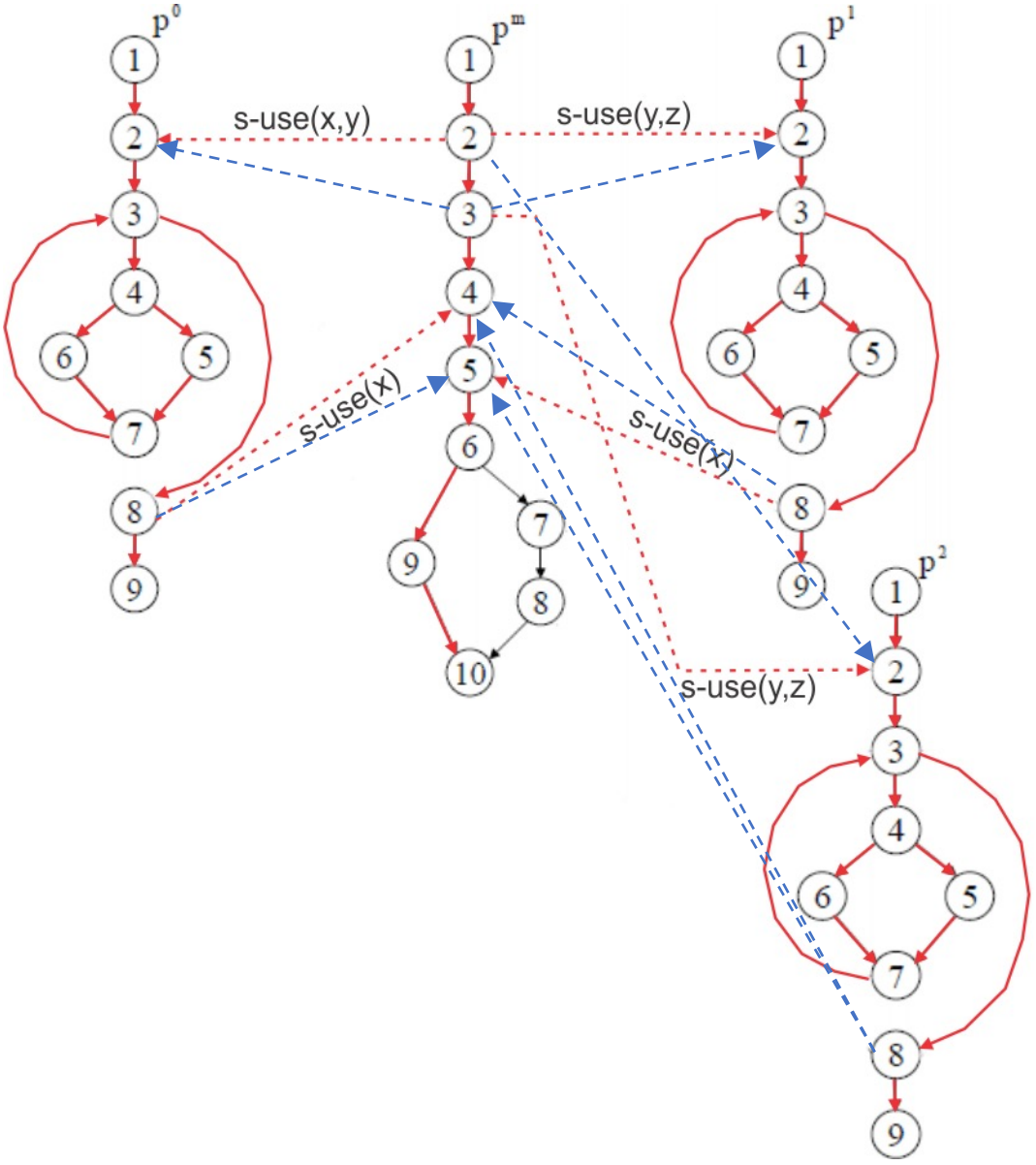


Figura 2 – Exemplo de Grafo de Fluxo de Controle Paralelo.

os caminhos não executáveis tendem a prejudicar a qualidade do teste durante a avaliação da cobertura de critérios de teste, justificando a necessidade de identificar tais caminhos.

Como no exemplo anterior, a análise estática, realizada em programas sequenciais, também pode ser utilizada em programas concorrentes. O custo computacional pode ser reduzido por meio da análise de grafos de fluxo de controle e fluxo de dados garantindo a cobertura de código esperada, apesar disso, o não determinismo cria obstáculos para este tipo de teste (DELAMARO; JINO; MALDONADO, 2017; DIONNE; FEELEY; DESBIEN, 1996; SOUZA *et al.*, 2008).

Em relação a análise dinâmica, existe a abordagem *Reachability Testing* proposta para

identificação de caminhos não executáveis em programas concorrentes. A abordagem trata a intercalação de processos em programas concorrentes de forma determinística e não determinística, garantindo que todas as sincronizações executáveis sejam tratadas (LEI; CARVER, 2005). A análise dinâmica assegura que caminhos não executáveis não sejam testados, contudo, o custo computacional desta abordagem é elevado.

No *reachability testing* as sequências são derivadas automaticamente e dinamicamente à medida que o processo de teste avança. Nessa estrutura, os eventos de sincronização que ocorrem durante uma execução de teste são registrados em um rastreamento de execução. No final da execução de teste, é analisado quais registros são variantes das sincronizações já executadas. Uma variante de execução representa a parte inicial de uma ou mais sequências de sincronização que definitivamente poderiam ter acontecido, mas não aconteceram, devido à forma como as condições de sincronização foram arbitrariamente resolvidas durante a execução. As variantes são forçadas a serem exercitadas pelo teste, com objetivo de exercitar um comportamento não executado antes. Cada novo comportamento é analisado e novas variantes são geradas até que todas as execuções do programa sejam realizadas com determinada entrada. Desta forma, o *reachability testing* entrega um conjunto parcialmente ordenado de sincronizações cobertas por um dado de entrada definido (HWANG; TAI; HUANG, 1995; TAI, 1997; LEI; CARVER, 2004).

Suponha que toda execução de $P$ com entrada $x$ termina, dessa forma o número de sequências de sincronização distintas de $P$ com entrada $x$ é finito. O objetivo é executar todos as possíveis sequências de sincronização de $P$ com entrada $x$. Como resultado é possível determinar se o conjunto de sequências de sincronização de $P$ com entrada $x$ está correta e se o resultado produzido por cada sequência de $P$ com entrada $x$ está correta. *Reachability testing* permite determinar a exatidão de $P$ com a entrada $x$ (HWANG; TAI; HUANG, 1995; TAI, 1997; LEI; CARVER, 2004).

## 2.3 Considerações Finais

Neste capítulo foram introduzidos os conceitos necessários para uma melhor compreensão do problema da não executabilidade no contexto de teste de software, sequencial e concorrente. Para complementar os conceitos já transmitidos, no próximo capítulo será apresentado o mapeamento sistemático desenvolvido com objetivo de analisar o estado da arte composto por estudos que abordam o problema da não executabilidade. Os estudos relacionados auxiliaram diretamente no desenvolvimento desta tese de doutorado.

CAPÍTULO

3

# O PROBLEMA DA NÃO EXECUTABILIDADE - ESTADO DA ARTE

## 3.1 Considerações iniciais

O mapeamento sistemático é uma técnica de revisão sistemática da literatura que busca identificar, avaliar e sintetizar as evidências disponíveis sobre uma determinada questão de pesquisa. Ele é usado principalmente em áreas como ciência da computação, engenharia e saúde, mas pode ser aplicado em diversas áreas do conhecimento (WOHLIN, 2014). O desenvolvimento de um mapeamento sistemático é importante em várias áreas de pesquisa, pois permite que os pesquisadores identifiquem lacunas no conhecimento, avaliem a qualidade e a relevância dos estudos existentes e gerem novas ideias para futuras pesquisas. Além disso, o mapeamento sistemático pode ajudar a evitar a duplicação de esforços, economizar tempo e recursos e garantir que as decisões sejam tomadas com base em evidências sólidas e confiáveis (WOHLIN, 2014). Para fomentar a proposta original do presente estudo, houve a necessidade de entender como o problema da não executabilidade está sendo abordado na comunidade científica. Com base neste fato, este capítulo apresenta o desenvolvimento de um mapeamento sistemático e seus resultados.

## 3.2 Mapeamento Sistemático

O desenvolvimento de um mapeamento sistemático (MS) foi escolhido devido a ser método sistemático, e então replicável, de alcançar uma visão ampla do tópico em foco, sem se prender a resultados específicos. No que se refere ao método de busca sistemática de estudos, há três métodos principais que podem ser aplicados: busca automática em bases de dados digitais, busca manual e *Snowballing* (PETERSEN; VAKKALANKA; KUZNIARZ, 2015).

A metodologia *Snowballing* é uma boa alternativa para a busca em bases de dados eletrônicas ou pode ser utilizada como ponto de partida para Revisões e Mapeamentos Sistemáticos. A

*Snowballing* é uma abordagem sistemática que utiliza referências e citações de estudos seminais e altamente citados para reunir estudos relevantes para responder as questões de pesquisa. Esta metodologia é muito útil para encontrar estudos relevantes a partir dos estudos selecionados por meio da aplicação de outros métodos, como busca em bibliotecas digitais ou, por meio de *Snowballing* realizado previamente (WOHLIN, 2014). Contudo, esta metodologia é recomendada para casos em que poucos estudos são selecionados com outros métodos, caso contrário pode ser inviável em detrimento do grande número de estudos que podem ser identificados (PETERSEN; VAKKALANKA; KUZNIARZ, 2015).

A primeira e fundamental tarefa da *Snowballing* é definir o conjunto de estudos iniciais que serão utilizados para iniciar as buscas. Este conjunto não deve ser pequeno, ao mesmo tempo que não há necessidade de conter um grande número de estudos. O que se espera é que seja um conjunto compatível com a amplitude da área analisada, desta forma, uma área específica requer menos estudos do que uma área ampla. Uma possibilidade para conceber o conjunto inicial significativo seria identificar estudos seminais e / ou altamente citados (WOHLIN, 2014).

O processo indicado para a metodologia *Snowballing* pode ser classificado em duas etapas: (i) a etapa *Forward Snowballing* realiza a identificação e inclusão de trabalhos científicos que citaram o estudo que está sendo examinado. Nesta etapa, cada trabalho que cita o estudo examinado é analisado; (ii) a etapa *Backward Snowballing* utiliza a lista de referências do estudo examinado para identificar e incluir novos trabalhos científicos ao processo de busca. Nesta etapa, a lista de referências é analisada e os trabalhos que não atendem aos critérios de inclusão são excluídos. O *loop* ou o processo em si da *Snowballing* é composto pela repetição das etapas *forward* e *backward* até que nenhum novo estudo seja encontrado (WOHLIN, 2014).

No contexto desta tese de doutorado houve a necessidade de reunir estudos relevantes ao tema e, para isso, foi proposto o desenvolvimento de um MS. Inicialmente foi utilizada a técnica de busca automática em bases eletrônicas, uma *string* de busca foi desenvolvida e os estudos retornados na busca foram coletados e analisados. Apesar disso, os resultados encontrados não foram significativos, o que motivou a utilização da culminou para utilização da *Snowballing*. Na sequência são apresentados os objetivos, questões de pesquisa, critérios de inclusão e exclusão, dados extraídos e, por fim, a utilização da metodologia da *Snowballing*.

### 3.2.1 Objetivos

O objetivo geral do MS foi identificar os estudos que investigam o problema da não executabilidade de caminhos no teste estrutural.

Como objetivos específicos, tem-se:

- Identificar estudos que investigam o problema da não executabilidade no contexto de programas sequenciais;

Tabela 1 – Estudos seminais.

| Autores | Título |
|---|---|
| Silvia R. Vergilio, José Carlos Maldonado e Mario Jino | Infeasible paths within the context of data flow based criteria (VERGILIO; MALDONADO; JINO, 1996). |
| Paulo Marcos Bueno Siqueira e Mario Jino | Identification of potentially infeasible program paths by monitoring the search for test data (BUENO; JINO, 2000). |
| Silvia R. Vergilio, José Carlos Maldonado e Mario Jino | Constraint based criteria: An approach for test case selection in the structural testing (VERGILIO; MALDONADO; JINO, 2001). |
| Silvia R. Vergilio, José Carlos Maldonado e Mario Jino | Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction (VERGILIO; MALDONADO; JINO, 2006). |

- Identificar estudos que investigam o problema da não executabilidade no contexto de programas concorrentes e;
- Identificar estudos que utilizam meta-heurísticas para solucionar o problema da não executabilidade.

### 3.2.2 Questões de pesquisa

Para alcançar os objetivos propostos foram desenvolvidas quatro questões de pesquisa:

***QP1.*** *Quais abordagens apoiam o tratamento do problema da não executabilidade?*

***QP2.*** *Quais propriedades de identificação de caminhos não executáveis são evidenciadas?*

***QP3.*** *Quais abordagens tratam o problema da não executabilidade no contexto de programas concorrentes?*

***QP4.*** *Quais abordagens utilizam técnicas de Search Based Software Testing para tratar o problema da não executabilidade?*

### 3.2.3 Metodologia

Com base na definição das diretrizes da *Snowballing* foram buscados estudos relevantes a partir de um conjunto de estudos seminais, os quais eram de conhecimento do grupo de pesquisa, além de que, os estudos selecionados como seminais foram os primeiros estudos da área desenvolvidos. Esses estudos estão listados na Tabela 1.

O MS foi dividido em Rodadas, cada rodada era composta por uma fase de *forward* e uma fase de *backward*. Em cada rodada foi realizada uma etapa de seleção de dados, uma

etapa de extração de dados e uma etapa de classificação de dados. Durante as etapas, os estudos coletados foram agrupados em camadas.

As atividades desenvolvidas durante a etapa de seleção de dados (SD) foram:

**Nota:** foi utilizado o termo *Camada* para representar o conjunto de estudos utilizados como *input* das fases *forward* e *backward*. Os numerais: *Camada*01, *Camada*02, representam a evolução cronológica dos conjuntos encontrados durante o *Snowballing*.

- **SD1.** *Forward*: os trabalhos de entrada para a rodada são inseridos na Camada 01 (Tabela 1);
- **SD2.** Os estudos que referenciaram a Camada 01 foram coletados e formaram a Camada 02;
- **SD3.** *Backward*: Os estudos relevantes identificados nas referências dos estudos das Camada 01 e 02 foram coletados e formaram a Camada 03.
- **SD4.** Nesta atividade, as referências foram analisadas com base nos títulos e resumos;

Os critérios de seleção foram aplicados em cada fase de um *loop* (*forward* e *backward*):

- **Inclusão 1 -** Estudos que apresentam o tratamento do problema da não executabilidade de caminhos durante a atividade de teste estrutural.
- **Exclusão 1 -** Estudos não disponibilizados na íntegra.
- **Exclusão 2 -** Estudos não disponibilizados na língua inglesa ou portuguesa.

Atividades desenvolvidas durante a etapa de extração de dados (ED):

- **ED1.** Todos os estudos selecionados com o critério de inclusão e não descartados por nenhum critério de exclusão foram reunidos para extração de dados;
- **ED2.** Os estudos selecionados foram lidos na íntegra e os seguintes dados foram coletados. **Dados gerais:** Título, Autor, *Journal/Conference* publicada, Contexto, Causas / Razões para a proposta, Objetivo. **Dados sobre o desenvolvimento do trabalho:** Algoritmo/Método Utilizado, Heurística (caso exista), Restrições utilizadas (caso exista), Métodos de Avaliação (caso exista).

As seguintes atividades foram desenvolvidas durante a etapa de classificação de dados (CD):

- **CD1.** Os trabalhos selecionados na Etapa E2 foram organizados por: período de publicação, abordagens, *Search Based Software Testing* (SBST) e programas concorrentes;
- **CD2.** As questões de pesquisa foram respondidas com as informações extraídas.

### 3.2.4 Rodadas

O estudo contou com três rodadas, denominadas, **Rodada Seminal**, **Rodada 02** e **Rodada de Atualização**, sendo que cada rodada contou com as etapas descritas na Seção 3.2.3. A Rodada Seminal teve como entrada 04 estudos categorizados como seminais devido ao seu pioneirismo no contexto do problema da não executabilidade. A Rodada 02 contou como entrada os trabalhos selecionados na Rodada Seminal e, por fim, a Rodada de Atualização contou com os estudos selecionados na Rodada 02. As próximas subseções descrevem os resultados de cada rodada.

#### 3.2.4.1 Rodada Seminal

A Figura 3 apresenta as camadas formadas pela execução do primeiro laço do *snowballing*. Em cada camada, em amarelo, estão as quantidades de estudos encontrados em cada camada e, em verde, a quantidade de estudos selecionados.

Essa rodada iniciou no primeiro semestre de 2018 e desenvolveu as atividades de *forward* e *backward*. Os estudos apresentados na Tabela 1 foram inseridos na Camada 01 da fase de *forward*. Segundo o GScholar, esses 04 estudos foram citados por 216 outros estudos que formaram a Camada 02. O total de 69 estudos foram selecionados para leitura integral com aproveitamento de 30 estudos aceitos, baseados nos critérios de inclusão e exclusão, definidos anteriormente. Isso pode ser visto na Camada 02 da Figura 3. Os 30 estudos selecionados serviram como entrada para o *backward* que reuniu 175 estudos referenciados que formaram a Camada 03, desses, 116 eram repetidos da primeira etapa e 59 eram novos estudos para leitura integral. Como resultado, 15 estudos foram aceitos, baseados nos critérios de inclusão e exclusão, definidos anteriormente. Essa rodada foi finalizada ao final do primeiro semestre de 2018, a união dos resultados encontrados na Camada 02 e 03 totalizam 45 estudos aceitos e contemplou publicações no período de 1985 a 2018.

#### 3.2.4.2 Rodada 02

A Figura 4 apresenta as camadas formadas pela execução do segundo laço do *snowballing*. Em cada camada, em amarelo, estão as quantidades de estudos encontrados em cada camada e, em verde, a quantidade de estudos selecionados.

A Rodada 02 teve como entrada os estudos aceitos na rodada anterior, 45 estudos formaram a Camada 01, como pode ser visto na Figura 4. Os estudos da Camada 01 foram citados por 2.110 outros trabalhos. Seguindo os critérios de inclusão e exclusão foram selecionados para

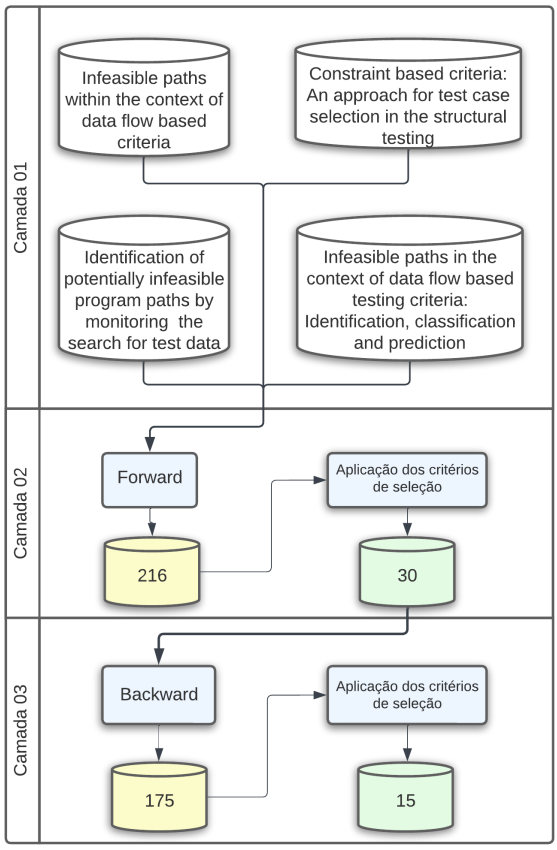


Figura 3 – Rodada Seminal.

Camada 02, 254 estudos sendo 148 repetidos. Dos 106 restantes, 29 já haviam sido analisados na rodada anterior, restando 77 estudos para leitura integral. Como resultado 33 estudos foram aceitos e formaram a Camada 02. O *backward* reuniu 175 referências da Camada 02 com 87 estudos repetidos, sendo 29 analisado na rodada anterior e 59 estudos novos. Como resultados, 22 estudos foram aceitos e formaram a Camada 03. A soma dos resultados das Camadas 02 e 03 totalizam 55 estudos aceitos. Essa rodada foi finalizada ao final do segundo semestre de 2018, contemplando publicações no período de 1985 a 2018.

Os trabalhos selecionados foram publicados no intervalo de 1985 a 2018. Identificou-se que a primeira menção do problema da não executabilidade ocorreu em 1985 por HEDLEY; HENNELL. No trabalho os autores identificaram um problema relacionado a não cobertura de um requisito de teste pela impossibilidade de gerar um dado de teste que o executasse, contudo, até então não haviam estudos mencionando o termo "não executabilidade", ou qualquer tipo de definição formal, por este motivo, neste MS o estudo de HEDLEY; HENNELL foi considerado o primeiro autor a relatar o problema da não executabilidade. Além disso, as duas rodadas analisaram 2.412 estudos e selecionaram 100 estudos. Dentre eles, 7 estudos foram classificados como literatura cinza, apresentados na Tabela 2.

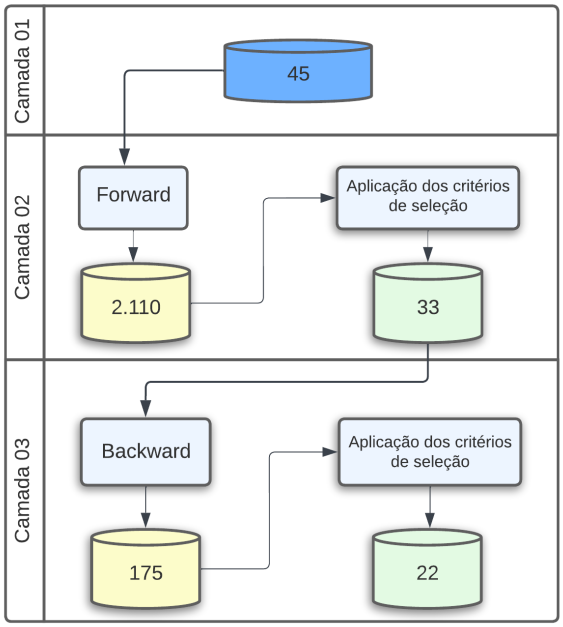


Figura 4 – Rodada 02.

#### 3.2.4.3 Rodada de Atualização

A Figura 5 apresenta as duas camadas formadas pela execução do terceiro laço do *snowballing*. Em cada camada, em amarelo estão as quantidades de estudos encontrados e, em verde, a quantidade de estudos selecionados.

Essa rodada foi executada no primeiro semestre de 2022 com o intuito de atualizar o mapeamento com os estudos que surgiram de 2018 a 2022. Para a última rodada foram reunidos os estudos selecionados na Rodada 02, como pode ser visto na Figura 5. A Camada 01 constituída de 100 estudos, a Camada 02 foi formada por 140 novos trabalhos de 2018 a 2022. O total de 40 estudos foram selecionados para leitura integral e deles, 18 foram aceitos. Como o objetivo dessa rodada foi analisar as atualizações não fazia sentido executar a fase de *backward*, desta forma essa rodada foi encerrada com 18 novos estudos aceitos. Ao final das três rodadas foram incluídos 118 estudos, no período de 1985 a 2022.

### 3.2.5 Resultados

O estudo mais antigo selecionado foi do ano de 1985 e o mais recente do ano de 2018. Antes mesmo da definição do termo "problema da não executabilidade", o MS revelou um estudo que relatou sobre o problema em 1972, por BROWN (1972), indicando requisitos de teste que não podiam ser cobertos pela falta de dados de teste que os exercitassem. Esse fato indicou que o problema da não executabilidade foi identificado e está sendo estudado a cerca de 50 anos. Por ser um problema de difícil solução diversos métodos são propostos para mitiga-lo o que permite que diversos estudos possam ser desenvolvidos nesse contexto abrangendo tanto programação tradicional como concorrente.

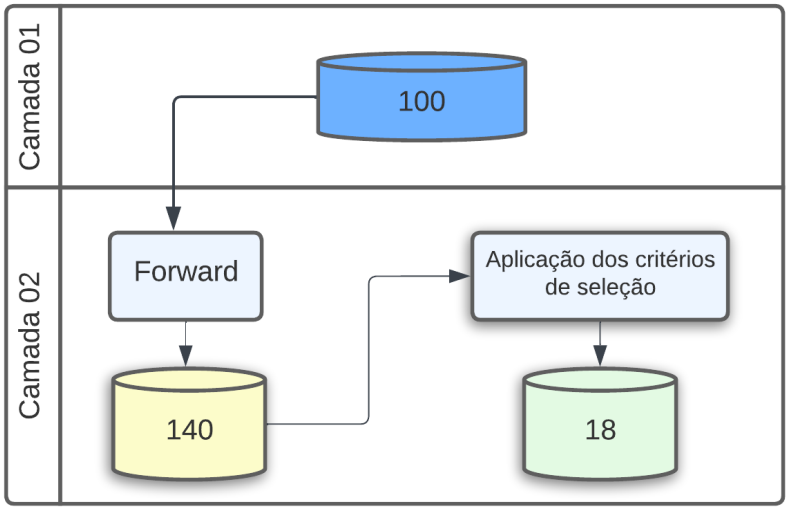


Figura 5 – Rodada de atualização.

Tabela 2 – Literatura cinza identificada.

| Autores | Título do estudo |
|---|---|
| (MAHMOOD, 2007) | A systematic review of automated test data generation techniques. |
| (HARMAN; MANSOURI; ZHANG, 2009b) | Search-based software engineering: A comprehensive analysis and review of trends techniques and applications. |
| (DING; TAN, 2012) | Detection of Infeasible Paths: Approaches and Challenges. |
| (ZHANG; JIANG; HAN, 2015) | Research progress on infeasible path detecting problem. |
| (MARASHDIH; ZAABA, 2018) | Infeasible paths in static analysis: Problems and challenges. |
| (MELO *et al.*, 2018) | Contributions for the structural testing of multithreaded programs: coverage criteria, testing tool, and experimental evaluation. |
| (MARASHDIH; ZAABA; ALMUFTI, 2018) | The Problems and Challenges of Infeasible Paths in Static Analysis. |

Dentre os estudos da literatura cinza apresentada na Tabela 2, destaca-se o estudo (DING; TAN, 2012), um mapeamento sistemático com objetivo de revisar as abordagens utilizadas para detecção de caminhos não executáveis. Como resultado os autores listaram seis abordagens: (1) *data flow analysis*; (2) *path-based constraint propagation*; (3) *property sensitive data flow analysis*; (4) *syntax-based approach*; (5) *infeasibility estimation*; e (6) *generalization of infeasible paths*. Estas abordagens, descritas abaixo, diferem em sua precisão de detecção, custo computacional e aplicações relevantes.

**(1)** ***Data flow analysis:*** A análise clássica de fluxo de dados é uma técnica aplicada sobre o Grafo de Fluxo de Controle (GFC) para calcular e coletar uma lista de informações sobre as variáveis do programa, que mapeia variáveis de programa para valores nos locais requeridos em GFC.

**(2)** ***Path-based constraint propagation:*** As abordagens de propagação de restrições baseadas em caminho aplicam avaliação simbólica a um caminho para determinar sua viabilidade.

**(3)** ***Property sensitive data flow analysis:*** Abordagem híbrida entre a análise de fluxo de dados e a abordagem de propagação de restrições. A abordagem de propagação de restrições: verificação de código e aplicação de critérios de teste, em relação a uma lista de propriedades dadas (Propriedade é um termo abstrato que abrange variáveis, funções ou estruturas de dados especiais, como ponteiro em C / C ++).

**(4)** ***Syntax-based approach:*** A análise de sintaxe é aplicada para detectar caminhos não executáveis usando regras ou reconhecendo padrões. Muitos caminhos não executáveis são causados por conflitos que podem ser identificados usando apenas a análise de sintaxe. As abordagens baseadas em sintaxe aproveitam características (variáveis, funções, estruturas lógicas, estrutura de dados), e definem a sintaxe para que conflitos sejam interpretados como padrões ou regras.

**(5)** ***Infeasibility estimation:*** O número de predicados envolvidos em um caminho é uma heurística para avaliar e estimar a viabilidade de um caminho.

**(6)** ***Generalization of infeasible paths:*** Quando um caminho é não executável em um CFG, todos os outros caminhos que contêm o caminho também são claramente não executáveis. Com base nesse conceito simples, a proposta é gerar todos os caminhos não-executáveis a partir de um determinado conjunto de caminhos não executáveis originais.

Apoiandos-se nas abordagens classificadas por DING; TAN (2012), nesta fase foi realizada uma classificação de acordo as abordagens definidas, na qual cada abordagem foi representada pelos valores (1) a (6). Além dessas, mais duas categorias foram criadas com objetivo de estender a classificação:

**(7)** **SBST:** Utilização de técnicas de busca para apoiar o problema da não executabilidade. Algoritmos de busca são utilizados como tecnologia para geração automática de dados de entrada e, durante o processo alguns requisitos de teste são considerados como não executáveis em razão do processo de busca não conseguir gerar um dado capaz de atender a estes requisitos. Pode-se entender que por meio de força bruta as técnicas de SBST mitigam o problema da não executabilidade.

**(8)** **Programação concorrente:** Abordagem que utiliza técnicas para mitigar o problema da não executabilidade no contexto da programação concorrente.

**(9)** **Estudos sem categoria:** Estudos que não puderam ser categorizados em nenhuma das abordagens anteriores.

O resultado da classificação está apresentado nas Tabelas 3, 4, 5, 6, 7 e 8, a primeira coluna apresenta a referência dos estudos, a segunda coluna o ano de publicação e, nas colunas (1) a (9) foram assinaladas com X indicando os trabalhos que se enquadraram em determinada classificação.

Na sequência são apresentados os resultados encontrados separados por décadas. A Tabela 3 apresenta a classificação dos estudos encontrados na década de 1980, a Tabela 4 dos estudos da década de 1990, a Tabela 5 dos estudos da década de 2000 as Tabelas 6 e 7 dos estudos da década de 2010 e, por fim, dos estudos da década de 2020, Tabela 8.

Tabela 3 – Classificação dos estudos da década de 1980.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (HEDLEY; HENNELL, 1985) | 1985 | x | | | | | | | | |
| (FRANKL, 1987) | 1987 | | | | | x | | | | |
| (YATES; MALEVRIS, 1989) | 1989 | | | | | x | | | | |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

Tabela 4 – Classificação dos estudos da década de 1990.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (MALEVRIS; YATES; VEEVERS, 1990) | 1990 | | | | | x | | | | |
| (VERGILIO; MALDONADO; JINO, 1992a) | 1992 | | | | | x | | | | |
| (VERGILIO; MALDONADO; JINO, 1992b) | 1992 | x | | | x | x | | | | |
| (GOLDBERG; WANG; ZIMMERMAN, 1994) | 1994 | | x | | | | | | | |
| (JASPER *et al.*, 1994) | 1994 | x | | x | | | | | | |
| (MALEVRIS, 1995) | 1995 | | | x | | | | | | |
| (SNELTING, 1996) | 1996 | x | | | | | | | | |
| (FORGÁCS; BERTOLINO, 1997) | 1997 | x | | | | | | | | |
| (BODÍK; GUPTA; SOFFA, 1997) | 1997 | | | | | x | | | | |
| (ZHOU *et al.*, 1998) | 1998 | x | | | | | | | | |
| (YANG, 1999) | 1999 | x | | | | | | | | |
| (NAOI; TAKAHASHI, 1999) | 1999 | | | | | x | | | | |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

A Figura 6 resume a quantitativamente a classificação dos estudos em relação as abordagem divididos nas quadro décadas analisadas. Com esta classificação e com a leitura integral dos estudos selecionados foi possível responder as 4 questões de pesquisa definidas para esse MS.

### 3.2.6 Respostas às questões de pesquisa

**QP1. Quais abordagens apoiam o tratamento do problema da não executabilidade?**

Os 109 estudos foram agrupados nas classes de acordo com os objetivos e métodos apresentados. Em decorrência, da classificação foram alcançados os seguintes resultados: 69 estudos foram classificados como (1) *data flow analysis*; 21 estudos foram classificados como (2) *path-based constraint propagation*; 14 estudos foram classificados como (3) *property sensitive data flow analysis*; 5 estudos foram classificados como (4) *syntax-based approach*; 11 estudos foram classificados como (5) *infeasibility estimation*; 2 estudos foram classificados como (6) *generalization of infeasible paths*; 10 estudos foram classificados como (7) SBST, contudo, houve uma dupla classificação indicando que duas abordagens foram utilizadas de forma conjunta, como resultado, 10 estudos foram classificados como (1) e (7); e, 3 estudos estudos foram classificados como (8) sendo ambos classificados como (1) e (8). Como pode ser visto na

Tabela 5 – Classificação dos estudos da década de 2000.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (BUENO; JINO, 2000) | 2000 | x | | | | | | x | | |
| (ZHANG; WANG, 2001) | 2001 | | | x | | | | | | |
| (VERGILIO; MALDONADO; JINO, 2001) | 2001 | | | x | | | | | | |
| (PERES *et al.*, 2001) | 2001 | | | | | x | | | | |
| (BUENO; JINO, 2002) | 2002 | x | | | | | | x | | |
| (SOUTER; POLLOCK, 2002) | 2002 | x | | | | | | | | |
| (ROBSCHINK; SNELTING, 2002) | 2002 | | | x | | | | | | |
| (SILVA; PREDTOOL, 2003) | 2003 | | | | | x | | | | |
| (DIAZ; TUYA; BLANCO, 2003) | 2003 | x | | | | | | | | |
| (HERMADI; AHMED, 2003) | 2003 | x | | | | | | x | | |
| (ZHANG; CHEN; WANG, 2004) | 2004 | | x | x | | | | | | |
| (WILLIAMS; MARRE; MOUY, ) | 2004 | x | | | | | | | | |
| (XIAOTONG; TAO; PANDE, 2006) | 2006 | x | | | | | | | | |
| (VERGILIO; MALDONADO; JINO, 2006) | 2006 | | | | | x | | | | |
| (YAN *et al.*, 2006) | 2006 | x | | | | | | | | |
| (VERGILIO *et al.*, 2006) | 2006 | x | | x | | | | | | |
| (LEI; CARVER, 2006) | 2006 | x | | | | | | | | |
| (SNELTING; ROBSCHINK; KRINKE, 2006) | 2006 | x | | x | | | | | | |
| (NGO; TAN, 2007) | 2007 | | | | x | | | | | |
| (YAN; ZHANG, 2008) | 2008 | x | | | | | | | | |
| (NGO; TAN, 2008) | 2008 | x | | | | x | | | | |
| (LEE; QIN, 2003) | 2008 | x | | | | | | | | |
| (NAVAS; SANTOSA, ) | 2009 | | x | | | | | | | |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

Tabela 8, somente dois estudo não se enquadraram nas 6 abordagens definidas por (DING; TAN, 2012), esses trabalhos são fruto deste trabalho e tratam o problema da não executabilidade de forma alternativa às abordagens usadas na classificação.

As abordagens identificadas na QP01 apresentaram diversas variações que são significativas. Os estudos classificados como (1) utilizam a atividade de geração de dados de teste para identificar, no GFC, caminhos não executáveis. Os 10 estudos classificados como (1)+(7) utilizam algum algoritmo de busca para auxiliar na geração de dados de teste. Os estudos classificados como (2) e (2)+(3) utilizam os recursos de execução simbólica para identificar caminhos não executáveis. Os estudos categorizados como (3) buscam identificar restrições que determinem a existência de caminhos não executáveis e utilizam a geração de dados de teste para validar essas restrições.

Os estudos classificados como (4) procuram identificar padrões de código que causam o surgimento de caminhos não executáveis, alguns padrões podem ser avaliação por meio de heurísticas. Os estudos classificados como (5) utilizam a abordagem de análise de predicados como forma de avaliar e identificar caminhos não executáveis, quanto maior o número de predicados maior a chance de um caminho ser não executável.

Tabela 6 – Classificação dos estudos da década de 2010, parte 01.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (GONG; YAO, 2010) | 2010 | x | | | | | | | | |
| (PAPADAKIS; MALEVRIS, 2010) | 2010 | | x | | | | | | | |
| (WILLIAMS, 2010) | 2010 | | x | | | | | | | |
| (JAFFAR; NAVAS; SANTOSA, ) | 2010 | | x | | | | | | | |
| (DELAHAYE; BOTELLA; GOTLIEB, 2010) | 2010 | | x | x | | | | | | |
| (YANO; MARTINS; SOUSA, 2010) | 2010 | x | | | | | | | | |
| (PINTO; VERGILIO, 2010) | 2010 | x | | | | | | x | | |
| (KUMAR; HARISH; KUMAR, 2011) | 2011 | x | | | | | | | | |
| (DELAHAYE, 2011) | 2011 | | x | | | | | | | |
| (ANDALAM; ROOP; GIRAULT, 2011) | 2011 | x | | | | | | | | |
| (KEMPF *et al.*, 2011) | 2011 | x | | x | | | | | | |
| (DU; DONG, 2011) | 2011 | x | | | | | | | | |
| (YANG *et al.*, 2011) | 2011 | x | | | | | | | | |
| (XIBO; NA, 2011) | 2011 | x | | | | | | | | |
| (DAHIYA; CHHABRA; KUMAR, 2011) | 2011 | x | | | | | | | | |
| (SOUZA *et al.*, 2011) | 2011 | x | | | | | | x | | |
| (CHRIST; HOENICKE; SCHÄF, 2012) | 2012 | x | | | | | | x | | |
| (JIANG; ZHANG; YI, 2012) | 2012 | x | | | | | | | | |
| (GONG; TIAN; YAO, 2012) | 2012 | x | | | | | | | | |
| (BERTOLINI; SCHÄF; SCHWEITZER, 2012) | 2012 | x | | | | | | | | |
| (ARLT; SCHÄF, 2012) | 2012 | x | | | | | | | | |
| (WANG *et al.*, 2012) | 2012 | x | | | | | | x | | |
| (JAYARAMAN; TRAGOUDAS, 2013) | 2013 | x | | | | | | | | |
| (LU; MIAO, 2013) | 2013 | x | | | | | | | | |
| (VENGADESWARAN; GEETHA, 2013) | 2013 | | x | | | | | | | |
| (PANDA; SARANGI, 2013) | 2013 | x | | | | | | | | |
| (BARHOUSH; ALSMADI, 2013) | 2013 | x | | | | | | | | |
| (UMESH; SRIVASTAVA, 2013) | 2013 | x | | | | | | | | |
| (DU; LU, 2014) | 2013 | x | | | | | | | | |
| (DING; ZHANG; TAN, 2014) | 2014 | x | | | | | | | | |
| (WANG; XING; ZHANG, 2014) | 2014 | | x | | | | | | | |
| (ELER *et al.*, 2014) | 2014 | | x | | | | | | | |
| (KEMPF; SLOMKA, 2014) | 2014 | x | | x | | | | | | |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

Tabela 7 – Classificação dos estudos da década de 2010, parte 02.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (GHIDUK, 2014) | 2014 | x | | | | | | x | | |
| (HERMADI; LOKAN; SARKER, 2014) | 2014 | x | | | | | | x | | |
| (TAMRAWI; KOTHARI, 2014) | 2014 | x | | | | | | | | |
| (DELAHAYE; BOTELLA; GOTLIEB, 2015) | 2015 | | x | | | | x | | | |
| (RAYCHEV; MUSUVATHI; MYTKOWICZ, 2015) | 2015 | | x | | | | | | | |
| (WANG; JIANG; TIAN, 2015) | 2015 | x | | | | | | | | |
| (RUIZ; CASSÉ, 2015) | 2015 | | | | | | x | | | |
| (DING; TAN; SHAR, 2015) | 2015 | | | | x | | | | | |
| (SOUZA *et al.*, 2015a) | 2015 | x | | | | | | | x | |
| (PIERCE; TRAGOUDAS, 2015) | 2015 | x | | | | | | | x | |
| (AISSAT *et al.*, 2016a) | 2016 | | x | | | | | | | |
| (AISSAT *et al.*, 2016b) | 2016 | | x | | | | | | | |
| (AISSAT; VOISIN; WOLFF, 2016) | 2016 | | x | | | | | | | |
| (SHU *et al.*, 2016) | 2016 | | x | | | | | | | |
| (PANIGRAHI; MALL, 2016) | 2016 | x | | | | | | | | |
| (ANSARI, 2017) | 2017 | x | | | | | | | | |
| (AHMED; TAHA, ) | 2018 | x | | | | | | x | | |
| (PATHADE; KHEDKER, 2018) | 2018 | x | | | | | | | | |
| (QI; LI, 2018) | 2018 | x | | | | | | | x | |
| (GIRGIS; EL-NASHAR; ELSIFY, 2019) | 2019 | x | | | | | | | | |
| (GONG *et al.*, 2019) | 2019 | x | x | | | | | | | |
| (FAZLI; AFSHARCHI, 2019) | 2019 | x | | | | | | | | |
| (PATHADE; KHEDKER, 2019) | 2019 | x | | x | | | | | | |
| (XIA *et al.*, 2019) | 2019 | x | | | x | | | | | |
| (QIAN *et al.*, 2019) | 2019 | x | | | | | | | | |
| (SUN *et al.*, 2019) | 2019 | x | | | | | | | | |
| (ZHU *et al.*, 2019) | 2019 | | | x | x | | | | | |
| (JIANG *et al.*, 2019) | 2019 | x | | x | | | | | | |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

Tabela 8 – Classificação dos estudos da década de 2020.

| Estudo | Ano | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|---|---|---|---|---|---|---|---|---|---|---|
| (SONG; ZHANG; GONG, 2020) | 2020 | | x | | | | | | | |
| (CHERAGHI; HASHEMINEJAD, 2020) | 2020 | x | | | | | | | | |
| (INDUMATHI; AJINA, 2021) | 2020 | x | | | | | | | | |
| (CHOMANETO, 2020) | 2020 | | | | | | | | | x |
| (MARASHDIH; ZAABA; SUWAIS, 2021) | 2021 | | x | | | | | | | |
| (SHU *et al.*, 2021) | 2021 | | x | | | | | | | |
| (DU; LIU; HE, 2021) | 2021 | x | | | | | | | | |
| (CHOMANETO *et al.*, 2021) | 2021 | | | | | | | | | x |

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

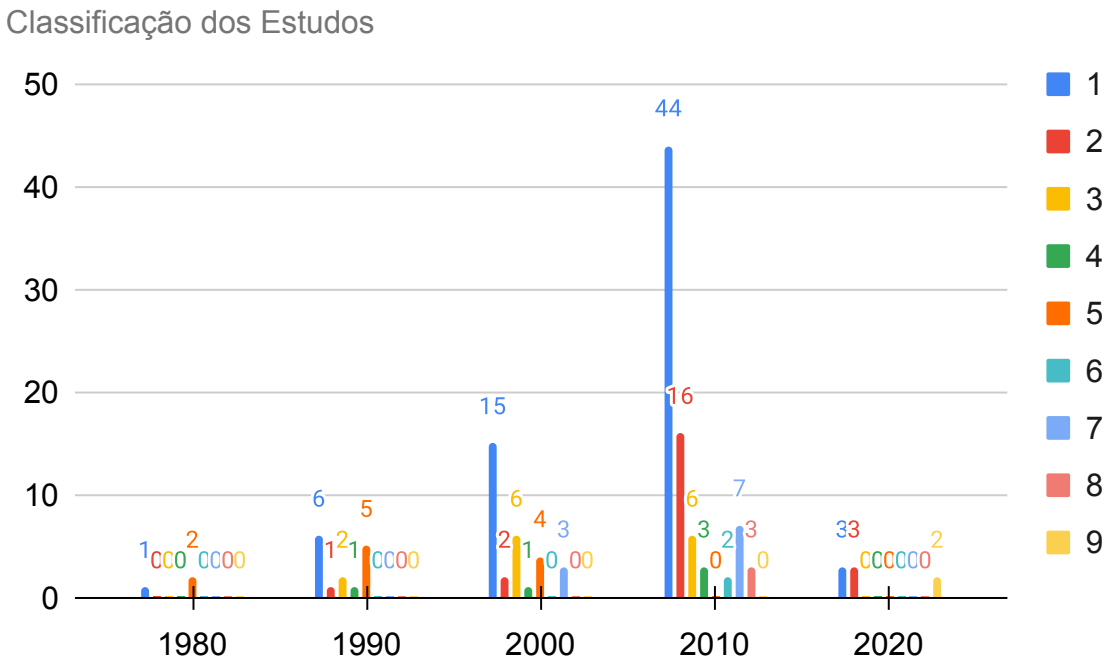


Figura 6 – Classificação resumida das cinco décadas analisadas.

(1) Data flow analysis, (2) Path-based constraint propagation, (3) Property sensitive data flow analysis, (4) Syntax-based approach, (5) Infeasibility estimation, (6) Generalization of infeasible paths, (7) SBST, (8) Programação Concorrente, (9) Nenhuma das categorias.

O que se conclui nesta questão de pesquisa é que diversos estudos trataram o problema da não executabilidade nos últimos 40 anos, contudo, as abordagens podem ser aprimoradas e agregadas com outras soluções originando novas propostas para mitigar o problema da não executabilidade.

**QP2. Quais propriedades de identificação de caminhos não executáveis são evidenciadas?**

Alguns estudos indicaram heurísticas para detecção de caminhos não executáveis. Algumas heurísticas são utilizadas junto com meta-heurísticas, outras junto a solucionadores de restrições e, também, há casos em que são aplicadas de forma manual.

O estudo de BUENO; JINO (2000) entende que o monitoramento do progresso da busca é uma maneira de detectar o mais cedo possível um caminho não executável usando a técnica dinâmica. O estudo de VERGILIO; MALDONADO; JINO (2001) utiliza violações e a distância do caminho testado para identificar sua não executabilidade. HERMADI; LOKAN; SARKER (2014) propuseram um modelo matemático para guiar o AG durante a geração de dados de teste, buscando novos caminhos. DU; LU (2014) realizam a combinação entre a cobertura baseada em caminhos com a lógica de negócio do software. O estudo de DU; DONG (2011) definiu que o grau de saída que é maior ou igual a dois nós, no caminho não executável. ZHOU *et al.* (1998) fizeram uso da abordagem de busca em profundidade e da

utilização de árvores dominadoras e pós-dominadoras. FORGÁCS; BERTOLINO (1997) e SILVA; PREDTOOL (2003) utilizaram a abordagem de Principal *Slicing* reduzindo o número de predicados favorecendo o processo de geração de dados de teste. GONG; YAO (2010) fizeram uso da correlação de caminhos para indicar caminhos não executáveis.

No contexto de análise de restrições para identificação de caminhos não executáveis entende-se que o Problema de Satisfação de Restrição (CSP) abrange um conjunto de variáveis, cada uma das quais pode receber valores de algum domínio. No contexto de viabilidade de caminhos, resolver um CSP significa encontrar um valor para cada variável, de modo que todas as restrições sejam válidas. Neste contexto, uma restrição é formada pelas condições que uma variável precisa atingir para que um caminho seja executado. Por exemplo: dada a expressão $n < 3$, então a restrição no caminho considerada é $n > 3$, posteriormente é realizada uma atribuição do tipo $n = n + 1$, assim a restrição é alterada para $n + 1 < 3$. Se a expressão $n > 9$ for adicionada irá ocorrer uma inconsistência categorizando o caminho como não executável. No contexto de tratamento de restrições o estudo de ZHANG; WANG (2001) utilizou o GFC e averiguar as restrições das variáveis presentes no caminho em questão e, se todas as restrições forem satisfeitas o caminho é considerado executável, caso contrário, não executável. GHIDUK (2014) coleta todas as restrições do caminho gerado e, em seguida, verifica a consistência das restrições. Segundo o estudo de ZHANG; CHEN; WANG (2004) as restrições podem ter operadores lógicos (como AND, OR) e operadores aritméticos (adição e subtração). VERGILIO *et al.* (2006) formularam duas restrições:

- **Critérios Baseados em Fluxo de Controle Restrito:** Os critérios todos os nós restritos e todos os limites restritos são definidos da seguinte forma:
  - ***All-constrained-nodes*:** Todo nó restrito $(i, C)$, onde $i$ é um nó e $C$ é uma restrição, deve ser exercitado. Um nó restrito é exercitado se $C$ for satisfeito durante a execução de um caminho que cubra $i$.
  - ***All-constrained-edges*:** Toda aresta restrita $((i, j), C)$ deve ser exercitada. Uma aresta restrita é exercitada se $C$ for satisfeito durante a execução de um caminho que cubra $(i, j)$.

- **Critérios Baseados em Fluxo de Dados Restritos:** Todos os critérios de usos restritos e todos os potenciais de potencial são definidos da seguinte forma:
  - ***All-constrained-uses*:** Requer que toda associação restrita $((i,k,x), C)$ ou $((i,(j,k),x), C)$ seja exercitada. Uma associação restrita é exercitada se um caminho que cobre a associação for executado e $C$ for satisfeito durante essa execução.
  - ***All-constrained-potential-uses*:** Requer que toda associação potencial restrita $((i,k,x), C)$ ou $((i,(j,k),x), C)$ seja exercitada. Uma associação potencial restrita é executada se

um caminho que cobre a associação potencial for executado e *C* for satisfeito durante essa execução.

Outros estudos não se enquadraram no contexto de utilização de heurística ou análise de restrições, mas seus resultados indicaram propriedades que auxiliam na identificação de caminhos não executáveis.

O estudo de HEDLEY; HENNELL (1985) apresentou duas categorias de propriedades que causam caminhos não executáveis:

1. O número de vezes que os *loops* são executados. *Loops* são reconhecidos como uma das principais fontes de problemas em testes estruturais;
2. Estilo de codificação ruim. Em muitos casos que ocorre não executabilidade, é possível reescrever o código em um estilo mais claro, que é executado com mais eficiência.

O estudo de BODÍK; GUPTA; SOFFA (1997) identificou particularidades que causam a existência de caminhos não executáveis.

1. Uma constante atribuída a uma variável pode implicar uma direção específica da condicional;
2. O resultado da avaliação de um nó predicativo $p_i$ (associado a um comando condicional) pode influenciar o resultado da avaliação de outro nó predicativo $p_j$ do caminho. Isso pode apoiar na definição de propriedade que identifica a não executabilidade pela dependência de nós predicativos;
3. Tipo de conversão: Variáveis que só recebem valores positivos podem ser convertidas para o tipo que permita valores negativos;
4. Referência de ponteiro: O valor de um ponteiro depois que ele é usado para acessar uma célula de memória deve ser diferente de zero, caso contrário, uma exceção teria sido gerada.

O estudo de DING; ZHANG; TAN (2014) definiu, empiricamente, dois padrões de código que podem generalizar a ocorrência de caminhos / ramos não executáveis durante a avaliação simbólica: (i) padrão de valor constante; e (ii) padrão de dominador semântico.

O estudo de VERGILIO; MALDONADO; JINO (2006) indicou cinco propriedades para auxiliar na identificação de caminhos não executáveis:

1. Dependência entre predicados em um caminho: Um caminho pode incluir dois ou mais predicados que são iguais ou opostos; a avaliação de um implica na avaliação do outro.

2. Valores inconsistentes de variáveis em um predicado: Uma variável em um predicado é definida com um valor diferente daquele que faz com que o caminho seja executado. Esta fonte pode ser dividida em dois grupos: (i) o predicado está em um momento ou repete a instrução, neste caso, a variável controla um *loop*; e (ii) o predicado está em uma declaração *if* ou *case*.

3. O número de potenciais-du-caminhos não executáveis está diretamente relacionado com o número de nós, número de variáveis e número de definições de variáveis.

4. Padrões de não executabilidade permitem determinar a inviabilidade de um grande número de requisitos. Eles capturam aspectos semânticos do programa que podem ser usados em testes de regressão e integração.

No estudo NGO; TAN (2007) foram apresentados propriedades de identificação de caminhos não executáveis e padrões de código que apresentam o problema da não executabilidade.

Definição 1 - Padrão de decisão idêntico / complemento. Seja $p$ e $q$ os nós de predicados da construção de seleção em um GFC. Diz-se que $p$ e $q$ seguem padrão de decisão idêntico / de complemento se e somente se as duas condições a seguir forem satisfeitas:

- Todos os caminhos base de $p$ para $q$ são caminhos *def-clear* com relação às variáveis referenciadas em $p$ e $q$; Seja S um conjunto de variáveis. Um caminho *def-clear* do nó p para o nó q em relação a S é um caminho que começa de p e termina em q e nenhuma das variáveis em S é modificada ao longo do caminho.

- Os predicados de ramo $p$ e $q$ são sintaticamente idênticos.

Definição 2 - Padrão de decisão empírica mutuamente exclusiva. Sejam $p1,\ldots,pn$ diferentes nós de predicados da construção de seleção em um GFC. Se esses nós de predicado satisfizerem as seguintes condições, então diz-se que $p1,\ldots,pn$ são empiricamente mutuamente exclusivos (e-mutuamente exclusivos):

- $p_j dominap_j + 1$ e $p_j + 1ps - dominap_j$, $1 <= j <= n-1$.

- $p_j$ possui apenas um sucessor que é dependente do controle de $p_j$, $1 <= j <= n-1$.

- Qualquer caminho de base de $p_1$ a $p_j$, $1 <= j <= n$, é um caminho *def-clear* em relação às variáveis referenciadas em $p_j$.

- Os conjuntos de variáveis primárias referenciados em $p_j$, $1 <= j <= n$, são idênticos.

- Os conjuntos de variáveis externas de todos os $p_j$, $1 <= j <= n$, são idênticos.

Definição 3 - Padrão empírico *check-then-do*. Seja $K$ uma variável em um programa. Seja p um nó de predicado de construção de seleção. Seja u um nó no GFC. Diz-se que u e p seguem o padrão empírico *check-then-do* (*e-check-then-do*) com respeito a $K$ se satisfizerem as seguintes condições:

- $p$ apenas referências a $K$;
- domina $p$ e $p$ pós-domina;
- $u$ atribui $K$ a uma constante que sempre resulta em satisfação do predicado de ramo de um ramo de $p$.

Definição 4 - Padrão empírico de *loop-by-flag*. Seja K uma variável em um programa. Seja $u$, $v$, $p$ e q quatro nós no CFG do programa em que $p$ é um nó de predicado de construção de iteração e $q$ é um nó de predicado de construção de seleção. Diz-se que $u$, $v$, $p$, $q$ seguem o padrão empírico *loop-by-flag* (*e-looping-by-flag*) em relação a $K$ se satisfizerem as seguintes propriedades:

- O predicado em $p$ apenas faz referência a $K$;
- $q$ é transitivamente dependente do controle de $p$;
- $u$ domina $p$ e $p$ pós-dominam $u$;
- $u$ atribui $K$ a uma constante $K_0$, que sempre resulta em satisfação do predicado de ramificação do ramo de entrada de $p$;
- $v$ depende do controle de $q$ e $v$ atribui $K$ a um valor que é sintaticamente diferente de $K_0$.

NGO; TAN (2008) indicou que a principal causa de caminhos não executáveis é a correlação entre algumas declarações condicionais ao longo do caminho. Duas declarações condicionais são correlacionadas se ao longo de alguns caminhos, o resultado do último pode ser implicado a partir do resultado do anterior. Além disso, foi proposto as seguintes propriedades empíricas de identificação de caminhos não executáveis:

Propriedade empírica 1. Seja $x$ e $y$ duas estruturas de decisão em um programa com predicados $p$ e $q$, respectivamente. Se $x$ e $y$ são correlacionados, então provavelmente uma e apenas uma das seguintes expressões podem ser executadas (NGO; TAN, 2008):

- 1. $(p \rightarrow q) \wedge (\neg\, p \rightarrow \neg\, q)$;
- 2. $(p \rightarrow \neg\, q) \wedge (\neg\, p \rightarrow q)$.

Se $p$ e $q$ satisfazem a primeira condição, diz-se que $p$ e $q$ são mutuamente exclusivos, caso contrário, eles são equivalentes.

Propriedade empírica 2. Seja $x$ e $y$ duas estruturas de decisão em um programa. Se $x$ e $y$ são correlacionados então existe pelo menos um caminho não executável que passa por ambos $x$ e $y$ (NGO; TAN, 2008).

YAN; ZHANG (2008) identificou que as dependências de dados entre as estruturas de controle podem impedir que alguns caminhos do GFC sejam exercitados o que causa caminhos não executáveis. Além disso, em alguns casos essas dependências de dados não afetam a complexidade ciclomática.

O estudo de BARHOUSH; ALSMADI (2013) identifica caminhos não executáveis causados pelos predicados logicamente inconsistentes relacionados aos códigos mortos, e pelas declarações condicionais correlacionadas com relação a uma determinada variável. O código morto é descrito como parte do código-fonte de um programa que não é mais necessário ou não é mais usado ou não é mais conexo ao GFC principal, assim, esse trecho de código pode ser confundido com código ativo e relevante, aumentando a complexidade da atividade de teste (BARHOUSH; ALSMADI, 2013).

O estudo VERGILIO; MALDONADO; JINO (1992) identificou algumas propriedades que comumente acarretam caminhos não executáveis. As causas podem acontecer isoladas ou agrupadas.

1. Laços, *flags* e variáveis que controlam laços;
2. Laço e comandos de seleção com predicados dependentes;
3. Teste de variáveis imediatamente após sua definição;
4. Dependência de contexto, a atribuição de variáveis por meio de retorno de funções.

Alguns propriedades de identificação de caminhos não executáveis foram apresentadas, contudo, todas elas são aplicadas para o contexto de programas sequenciais ou tradicionais. Outro fato que vale destacar é a dependência direta entre a geração de dados e a aplicação das heurísticas de identificação de caminhos não executáveis. Dados os fatos, é possível perceber a necessidade de definir um conjunto de propriedades de identificação de caminhos não executáveis, além de definir uma abordagem automatizada para identificação de caminhos não executáveis em programas sequenciais e concorrentes.

Com base na perspectiva das propriedades descritas até aqui, constatou-se a presença de características no código-fonte que ampliam a probabilidade de gerar requisitos de teste não executáveis. Essas características incluem o uso inadequado de variáveis, repetição descuidada de condicionais, bem como o estabelecimento de dependências complexas entre as condicionais,

resultando na criação de estruturas interdependentes. Ao analisar essas características foi possível organizá-las em forma de propriedades que pudessem auxiliar na identificação de requisitos de teste não executáveis. Essas propriedades foram divididas em dois grupos distintos:

- **Grupo 1:** Propriedades nas quais os requisitos de teste não executáveis podem ser revelados por meio da análise de dados de entrada e da execução do programa.
- **Grupo 2:** Propriedades nas quais os requisitos de teste não executáveis podem ser identificados sem depender de dados de entrada ou da execução do programa.

**QP3. Quais abordagens utilizam técnicas de *Search Based Software Testing* para tratar o problema da não executabilidade?**

Para responder essa questão foram reunidos os estudos que abordaram de alguma forma a utilização de meta-heurísticas de busca no contexto de teste de software. A utilização de meta-heurísticas de busca indica uma abordagem que identifica caminhos não executáveis pelo método de força bruta, ao qual realiza inúmeras tentativas de executar um determinado caminho e, pelo fato de não alcançar uma execução acaba identificando aquele caminho como não executável. Esta abordagem utiliza um critério de parada que indica a não executabilidade de determinado requisito de teste. Como resultado foram encontrados 10 estudos.

Nos estudos de BUENO; JINO (2000), XIBO; NA (2011) e HERMADI; LOKAN; SARKER (2014) o Algoritmo Genético (AG) foi utilizado como o principal algoritmo para gerar os dados de teste automaticamente. De forma semelhante, no estudo de BUENO; JINO (2002) o AG foi utilizado para explorar o domínio de entrada do programa, buscando um conjunto de dados de entrada que execute o caminho desejado.

No estudo de HERMADI; AHMED (2003) foi apresentada uma abordagem baseada em AGs que procura gerar um conjunto de dados de teste que devem cobrir um determinado conjunto de caminhos de destino. GONG; TIAN; YAO (2012) utilizaram o AG multiobjetivo para geração de dados para testar caminhos, o fato de haver diversos caminhos permitiu que o AG fosse interpretado como multiobjetivo. O algoritmo multiobjetivo NSGA-II foi utilizado no estudo de PINTO; VERGILIO (2010) com objetivo de gerar dados de teste, este estudo utilizou duas representações diferentes para a população, o que permitiu o teste de código procedural e orientado a objetos. Ainda, neste estudo foi executada uma avaliação experimental com a combinação de três objetivos: cobertura de critérios de teste estrutural, capacidade de revelar falhas e tempo de execução.

O algoritmo de busca utilizado no estudo por DAHIYA; CHHABRA; KUMAR (2011) foi o *Particle Swarm Optimization* (PSO), com objetivo de gerar dados de teste para programas C, atendendo ao critério de teste: todos os caminhos. O estudo de DIAZ; TUYA; BLANCO (2003) utilizou a busca Tabu para auxiliar na geração automática de dados de teste atendendo ao critério de teste: todos as arestas.

Por fim, GHIDUK (2014) utilizou o AG para realizar a avaliação de caminhos gerados automaticamente. Os caminhos foram avaliados com base na função de *fitness* que mensurou a definição de caminhos. A função utilizou como cálculo a probabilidade de adjacência de arestas pertencentes ao caminho.

Todos os estudos de SBST apresentados são exemplos que sofrem com o problema da não executabilidade e propõem uma forma de minimizar o problema aumentando o número de tentativas na busca por dados de entrada. Tal solução encarece o processo de geração automática mas é uma forma de mitigar o problema da não execubilidade. Essa lacuna pode ser explorada para oferecer soluções que possam melhorar o desempenho das técnicas empregadas na geração de dados de teste.

**QP4. Quais abordagens tratam o problema da não executabilidade no contexto de programas concorrentes?**

Durante o desenvolvimento do *Snowballing* não foram identificadas abordagens que tratam o problema da não executabilidade no contexto de programas concorrentes. Dois estudos foram classificados neste contexto, os quais citam trabalhos sobre não executabilidade. Porém, esses trabalhos não tratam o problema da não executabilidade no contexto de programas concorrentes, apenas citam a existência deste problema neste contexto.

Os estudos de ZHOU *et al.* (1998) e SOUZA *et al.* (2008) utilizaram a análise do GFC para avaliar a aplicação de critérios de teste no contexto de programas concorrentes. Muito embora os estudos apliquem critérios de teste para cobertura de caminhos ou arestas, nenhum deles abordou diretamente a identificação de caminhos não executáveis no contexto de programas concorrentes.

## 3.3 Considerações Finais

Este capítulo apresentou os principais trabalhos relacionados ao tratamento do problema da não executabilidade, existente na atividade de teste estrutural de software. Para a identificação de estudos relevantes foi proposto e desenvolvido um MS com a estratégia do *Snowballing*. Como resultado principal foi identificado que o problema da não executabilidade ainda está em aberto tanto para o contexto de programas sequenciais como para programas concorrentes. Além disso, as técnicas de SBST identificadas abordam o problema da não executabilidade de forma indireta, com o auxílio da geração de dados de teste, o que torna a solução cara em relação ao custo computacional. Apesar de inspiradora, a literatura não indicou estudos para o tratamento do problema da não executabilidade sem a utilização de dados de teste, e qualquer tipo de automatização relacionada. Além disso, não foram encontradas propriedades de identificação de caminhos não executáveis para o contexto de programas concorrentes.

A literatura permitiu determinar propriedades que podem ser adaptadas para identificação

de caminhos não executáveis sem a utilização de dados de entrada. Essa perspectiva pode favorecer o processo de geração de dados de teste e mitigar seu custo. Desse modo, motivados pelos resultados encontrados até este ponto, no próximo capítulo será apresentada a proposta de um catálogo de propriedades de identificação de requisitos de teste não executáveis que podem ser utilizadas de forma automática e sem dados de entrada.

CAPÍTULO

4

# NOEXEC - ABORDAGEM PARA IDENTIFICAÇÃO DE REQUISITOS DE TESTE NÃO EXECUTÁVEIS

## 4.1 Considerações Iniciais

A técnica de teste estrutural tem sido amplamente empregada para validar software, pois oferece uma medida de cobertura que permite avaliar a evolução da atividade de teste. Esta técnica examina a estrutura interna do programa e assim assegura que os requisitos funcionais foram testados. Uma limitação desta técnica é o alto número de requisitos de teste a serem cobertos e, consequentemente, a identificação de requisitos de teste não executáveis (VERGILIO; MALDONADO; JINO, 2006; YATES; MALEVRIS, 1989, 1989; NGO; TAN, 2008; DELAHAYE; BOTELLA; GOTLIEB, 2015). Um requisito é não executável se não houver dados de entrada que levem à execução desse requisito (CLARKE, 1976; FRANKL, 1987).

Muitos requisitos de teste não executáveis (por exemplo, caminhos não executáveis) colaboram para aumentar o esforço na geração de dados de teste. A identificação de requisitos de teste não executáveis não é uma tarefa trivial. Trabalhos publicados abordam esse problema e propõem um processo manual para mitigar requisitos de teste não executáveis.

Os critérios de teste estrutural utilizam uma medida de cobertura para avaliar a evolução da atividade de teste, ajudando a decidir se o software está sendo suficientemente testado. À medida que a atividade de teste é executada, novos casos de teste são gerados para executar os requisitos de teste descobertos e, assim, melhorar a cobertura do teste. Nesse processo, um problema enfrentado é o da não-executabilidade, que se refere a requisitos de teste requeridos por um critério de teste (por exemplo, caminhos) que, devido à semântica do programa, são não executáveis.

Alguns trabalhos propuseram abordagens para enfrentar este problema devido ao impacto

de requisitos de teste não executáveis no teste estrutural (FRANKL, 1987; BUENO; JINO, 2000; BARHOUSH; ALSMADI, 2013). O problema da não-executabilidade se apresenta em várias categorias de programas. Tratar e mitigar o problema não é uma tarefa trivial e, por isso, diferentes abordagens têm sido propostas. Diversos estudos se basearam em dados de entrada para encontrar requisitos de teste não executáveis, aumentando o custo e o esforço de teste. Apesar da diversidade de propostas, este trabalho detectou uma lacuna relevante, relacionada a características no código-fonte, para encontrar requisitos de teste não executáveis de forma estática e sem dados de entrada. Fato que minimiza o custo da geração automática de dados de teste, bem como orienta o testador no processo de análise do código-fonte, ao passo que é possível identificar regiões de código que necessitam de atenção.

Nesse contexto, uma abordagem, denominada Nonexec, foi definida para identificar automaticamente requisitos de teste não executáveis e potencialmente não executáveis sem recorrer a utilização de dados de entrada. Conforme modelado, a abordagem indica quais regiões do código-fonte possuem as propriedades e quais requisitos de teste são não executáveis e potencialmente não executáveis. Portanto, a abordagem de identificação mostra ao testador quais regiões de código precisam ser melhor verificadas e sugere um possível critério de parada para a geração automática de dados que podem sofrer com o problema de não executabilidade.

A abordagem Nonexec permite identificar automaticamente requisitos de teste não executáveis, possivelmente não executáveis e dificilmente executáveis, baseando-se no catálogo de propriedades apresentados na próxima seção.

## 4.2 Catálogo de propriedades

Esta seção apresenta um catálogo com propriedades para identificar requisitos de teste não executáveis que não utilizam dados de entrada. As propriedades catalogadas foram retiradas de registros na literatura e observações experimentais baseadas na lógica de programação apresentada no código fonte. De forma complementar serão discutidos as características e o funcionamento das propriedades em detalhes e por meio de exemplos complementares.

### 4.2.1 Distribuições das abordagens

Com base na literatura, nas técnicas existentes e observações experimentais, verificou-se a existência de características presentes no código-fonte que potencializem a geração de requisitos de teste não executáveis, como: utilização inadequada de variáveis, repetição descuidada de predicados condicionantes, concepção de dependências entre predicados condicionantes criando estruturas interdependentes. Essas características podem ser organizadas em propriedades que apoiam a identificação de requisitos de teste não executáveis. As propriedades podem ser separadas em dois grupos:

- **Grupo 1:** propriedades onde os requisitos de teste não executáveis são revelados com auxílio de dados de entrada e com a execução do programa.
- **Grupo 2:** propriedades onde os requisitos de teste não executáveis são revelados sem auxílio de dados de entrada e sem a execução do programa.

Os trabalhos encontrados na literatura não apresentam abordagens para detecção de requisitos de teste não executáveis por meio de propriedades do Grupo 2. Como forma de contribuir com o estado da arte foi construído um catálogo com sete propriedades do Grupo 2. Apesar de termos identificado 7 propriedades, este catálogo não tem a pretensão de ser completo mas sim, oferecer a possibilidade de que novas propriedades possam ser incorporadas conforme forem descobertas. As propriedades P1 a P5 foram extraídas do contexto da programação sequencial e, as propriedades P6 e P7, foram extraídas do contexto da programação concorrente, considerando os modelos de passagem de mensagem e memória compartilhada.

Cada uma das propriedades é descrita nas próximas subseções.

### 4.2.2 Propriedade 1 - Congelamento de variável

**Propriedade 1 (P1) - Atribuição de um valor constante a uma variável** (HEDLEY; HENNELL, 1985; VERGILIO; MALDONADO; JINO, 1992b): Uma constante atribuída a uma variável pode implicar uma direção condicional específica causando um requisito de teste não executável.

P1 é baseada na relação de dependência que pode existir entre duas instruções no código, que são: uma instrução de atribuição constante e a instrução condicional que usa a constante para verificação. O código apresentado no Código-fonte 4 fornece um exemplo dessa propriedade.

Quando uma instrução denota uma constante no código (por exemplo, na linha *n1* (*final int x = 10;*)), seu valor será armazenado na memória no momento da compilação, não sendo possível pois qualquer mudança de definição é permitida durante a execução do código. Portanto, uma instrução do tipo condicional, por exemplo, *IF* ou *WHILE*, que usa a constante *x* (linha *n1* da Código-fonte 4) em sua estrutura de comparação pode criar um requisito de teste não executável. Código-fonte 4, constante *x* tem o valor 10 atribuído, e será fixado na constante no momento da compilação. Mesmo que outra variável use a constante *x*, o valor fixo ainda existirá.

```
/* n1    */ final int x = 10;
/* n...  */ ...
/* n25   */ int age = x;
/* n...  */ ...
/* n175  */ if (name="Joao" && age > 18){
/* n176  */       System.out.printl("True");
/* n177  */ } else {
```

```
/* n178  */          System.out.printl("False");
/* n179  */ }
/* n...  */ ...
/* n575  */ if (x > 21 && height < 1.80){
/* n576  */          System.out.printl("True");
/* n577  */ } else {
/* n578  */          System.out.printl("False");
/* n579  */ }
/* n...  */ ...
/* n1000 */}
```

**Código-fonte 4** – Exemplo da Propriedade P1.

Após algumas linhas de código a variável ***age*** (linha *n25* do Código-fonte 4) está sendo usada na instrução condicional da linha *n175* , não haverá dados de entrada que cubram qualquer requisito que use esse conjunto de instruções (*n175, n176*), uma aresta ou caminho, por exemplo. Da mesma forma, se a constante *x* for usada na linha *n575*, não haverá dados de entrada que cubram qualquer requisito que use o conjunto de instruções (*n575, n576*), uma aresta ou caminho, por exemplo.

> Dada uma variável *x* e uma dada instrução condicional *condition*, onde *condition* faz uso da variável *x* em seu predicado, haverá uma dependência de instruções, que irá ignorar um ou mais caminhos. Os caminhos ignorados serão não executáveis. Logo, quaisquer requisitos de teste relacionados aos caminhos não executáveis serão possivelmente não executáveis.

### 4.2.3 Propriedade 2 - Predicados Opostos e Predicados Iguais

**Propriedade 2 (P2) - Predicados Opostos e Predicados Iguais** (VERGILIO; MALDONADO; JINO, 2006; NGO; TAN, 2007): uma dependência entre predicados opostos ou iguais em um caminho pode levar a caminhos não executáveis devido à avaliação condicional de um predicado interferir na avaliação do predicado seguinte.

P2 é baseada na relação de dependência que pode existir entre duas instruções do tipo condicional, exclusivamente a instrução do tipo *IF*. A dependência pode existir tanto para instruções iguais ($ifa < b$) e ($ifa < b$) quanto para instruções opostas ($ifa < b$) e ($ifa > b$). O Código-fonte 5 fornece um exemplo dessa propriedade.

```
/* n1    */ ...
/* n...  */ ...
/* n175  */ if (a > b){
/* n176  */          System.out.println("A");
```

```
/* n177  */ } else{
/* n178  */       System.out.println("!A");
/* n179  */ }
/* n...  */ ...
/* n275  */ if (a > b){
/* n276  */       System.out.println("B");
/* n277  */ } else{
/* n278  */       System.out.println("!B");
/* n279  */ }
/* n...  */ ...
/* n375  */ if (c > d){
/* n376  */       System.out.println("C");
/* n377  */ } else{
/* n378  */       System.out.println("!C");
/* n379  */ }
/* n...  */ ...
/* n475  */ if (c < d){
/* n476  */       System.out.println("D");
/* n477  */ } else{
/* n478  */       System.out.println("!D");
/* n479  */ }
/* n...  */ ...
/* n1000 */}
```

**Código-fonte 5** – Exemplo da Propriedade P2.

Pode-se associar a relação de dependência denotada por P2 com uma tabela verdade da expressão lógica XOR e XNOR (Tabela 9). Nesta tabela, há sempre duas linhas iguais e duas linhas exclusivas/opostas.

Tabela 9 – Tabela verdade XOR e XNOR.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **XOR** | 0 | 0 | **1** | **XNOR** | 0 | 0 | **0** |
| **XOR** | 0 | 1 | **0** | **XNOR** | 0 | 1 | **1** |
| **XOR** | 1 | 0 | **0** | **XNOR** | 1 | 0 | **1** |
| **XOR** | 1 | 1 | **1** | **XNOR** | 1 | 1 | **0** |

O código-fonte 5 apresenta uma dependência entre duas declarações condicionais iguais nas linhas *n175* e *n275*. Com base nessas duas condicionais, foram derivados quatro caminhos diferentes a serem percorridos. Isso denota as quatro possibilidades da tabela verdade XOR, onde duas delas são não executáveis, neste caso (0 1, 1 0). Os casos possíveis são aqueles em que a condição é falsa ou verdadeira, para ambos os predicados *0 1* e *1 0*, os requisitos são não

executáveis. Qualquer requisito de teste que esteja nos seguintes caminhos: *n175, 176, ..., n277, n278* e *n178, n179, ..., n275, n276*, é não executável. Para o caso de instruções opostas, *n375* e *n475*, o problema é o mesmo, mas, diferentemente do exemplo anterior, a tabela verdade é XNOR. Portanto, qualquer instrução que contemple os casos exclusivos (0 1 e 1 0) será executável, pois os casos de igual predicado (0 0 e 1 1) serão não executáveis. Posteriormente, qualquer requisito de teste que percorra os seguintes caminhos: *n375, n376, ..., n475, n476* e *n378, n379, ..., n478, n479*, é não executável.

> Todo par de instruções condicionais opostas ou iguais criam quatro caminhos sendo dois exclusivos, assim, de quatro caminhos possíveis, dois serão não executáveis. Logo, quaisquer requisitos de teste relacionados aos caminhos não executáveis serão possivelmente não executáveis.

### 4.2.4 Propriedade 3 - Correlação entre declarações condicionais

**Propriedade 3 (P3) - Correlação entre declarações condicionais** (NGO; TAN, 2008; BODÍK; GUPTA; SOFFA, 1997): a correlação entre expressões condicionais ao longo do caminho pode causar requisitos de teste não executáveis. A propriedade P3 é semelhante à P2, porém, com a dependência existente entre uma condicional do tipo *IF* e do tipo *WHILE*.

```
/* n1     */ ...
/* n2     */ b = 100;
/* n...   */ ...
/* n175   */ while (a > b && short=="true"){
/* n176   */        System.out.println("while");
/* n...   */ ...
/* n276   */        if (a < 100){
/* n277   */              System.out.println("A");
/* n...   */              ...
/* n300   */        }
/* n...   */ ...
/* n1000   */ }
```

**Código-fonte 6** – Exemplo da Propriedade P3.

P3 é baseada na relação de dependência que pode existir entre duas instruções no código, que são: uma instrução condicional do tipo *IF* e uma estrutura de repetição (*loop*) do tipo *WHILE*. O objetivo desta propriedade é identificar trechos de código com grande potencial para apresentar um requisito de teste não executável. Se o predicado da instrução *IF* for igual ou oposto ao predicado da instrução WHILE, significa que, como no caso de P2, o comportamento

dos caminhos será de acordo com a tabela verdade XOR ou XNOR . Código-fonte 6 fornece um exemplo para nossa discussão.

As linhas *n175* e *n276* do Código-fonte 6 são dependentes porque usam a mesma variável ***a***. Se a variável *a* não mudar de definição até a linha *n276*, então haverá pelo menos um requisito de teste não executável, aquele em que as condições são opostas. Nos casos de *IF* apresentar um *ELSE* haverá requisitos de teste não executáveis quando as condições forem iguais.

> Todo par de instruções condicionais do tipo IF e WHILE com dependência de predicados, sendo esta, igual ou oposta, criam quatro caminhos sendo dois exclusivos, assim, de quatro caminhos possíveis, dois serão não executáveis. Logo, quaisquer requisitos de teste relacionados ao caminho não executável serão possivelmente não executáveis.

### 4.2.5 Propriedade 4 - Mudança de Definição no caminho

**Propriedade 4 (P4) - Alteração da definição de uma variável durante o caminho analisado** (NGO; TAN, 2007): Alterar a definição de uma variável pode gerar um caminho não executável na presença de *loops*.

P4 foi baseada na relação de dependência que pode existir entre variáveis (uso e definição), estruturas condicionais e estruturas de de repetição. Esta propriedade possui um conjunto de características que devem ser cumpridas:

- (i) a variável deve ser definida antes ou durante o laço de repetição;
- (ii) a variável deve ser utilizada dentro do laço de repetição por uma instrução do tipo condicional e;
- (iii) a variável deve ter uma definição alterada dentro do laço de repetição.

Esses recursos podem ajudar a revelar trechos de código propensos a requisitos de teste não executáveis. No entanto, devido à quantidade de particularidades que esta propriedade trata, não é possível afirmar que as instruções identificadas revelam um requisito de teste não executável. Nesse contexto, a *expertise* do testador deve ser considerada para analisar se a ocorrência de P4 é de fato um requisito de teste não executável ou não.

P4 pode ser ilustrada analisando, no Código-fonte 7, as linhas **n2** (definição de variável), **n175** (loop), **n301** (mudança de definição de variável) e **n476** (condicional). Neste caso, devido à mudança de definição da variável *i* na linha **n301** antes da instrução condicional na linha **n476**, não será possível cobrir a condição positiva da linha **n477**, havendo assim um requisito de teste não executável.

```
/* n1    */ ...
/* n2    */ int i = 11;
```

```
/* n175  */ while (a > b && short=="true"){
/* n176  */        System.out.println("while");
/* n...  */ ...
/* n276  */       if (buyTool){
/* n277  */              System.out.println("A");
/* n...  */              ...
/* n300  */       } else{
/* n301  */               i = -1;
/* n302  */       }
/* n...  */       ...
/* n476  */       if (a < 100 && i > 10){
/* n477  */              System.out.println("A");
/* n...  */              ...
/* n500  */       }
/* n501  */ }
/* n...  */ ...
/* n1000  */ }
```

**Código-fonte 7** – Exemplo da Propriedade P4.

Todo par de instruções dos tipos laço de repetição e estrutura condicional que segue as seguintes restrições: (i) a variável deve ser definida antes ou durante o laço de repetição, (ii) a variável deve ser utilizada dentro do laço de repetição por uma instrução do tipo condicional, e (iii) a variável deve ter uma definição alterada dentro do laço de repetição, será um caminho não executável. Logo, quaisquer requisitos de teste relacionados ao caminho não executável serão possivelmente não executáveis.

### 4.2.6 Propriedade 5 - Código morto

**Propriedade 5 (P5) - Analisa a existência de código morto** (BARHOUSH; ALSMADI, 2013): Predicados logicamente inconsistentes relacionados a códigos mortos causam um requisito de teste não executável devido a não possibilidade de execução do caminho do código. A propriedade P5 foi baseada na identificação de trechos do código que não são utilizados durante a execução do código. Todas as classes existentes no programa devem ser testadas, caso haja uma classe não instanciada, então não será possível cobrir nenhum tipo de critério de teste. Dessa forma, essa propriedade procura instruções do tipo classe que não são chamadas em nenhum momento em todo o programa.

Um exemplo de código morto foi apresentado na classe do Código-fonte 8 e 9, ***Cfc*** (Código-fonte 8) não foi usado no programa principal (Código-fonte 9). A classe ***Cfc*** é um

trecho desconectado do programa principal e, em razão deste cenário, não permite entrada de teste para testar o trecho de código. Então, qualquer requisito existente nas linhas *n1* a *n500* será não executável.

```
/* n1    */ ...
/* n2    */ public class Cfc {
/* n...  */ ...
/* n500  */ }
```

**Código-fonte 8** – Exemplo da Propriedade P5. Classe Cfc.

```
/* n1    */ ...
/* n2    */ public class Main {
/* n...  */ ...
/* n155    */     public static void main(final String[] args)
   {
/* n...  */          ...
/* n...  */          OtherClass other = new OtherClass();
/* n...  */          ...
/* n1000  */    }
```

**Código-fonte 9** – Exemplo da Propriedade P5. Classe main. **Fonte:** Produzido pelo autor.

> Todo código desconectado do programa principal se torna um código morto, gerando caminhos não executáveis. Logo, quaisquer requisitos de teste relacionados ao caminho não executável serão possivelmente não executáveis.

### 4.2.7 Propriedade 6 - Comunicação entre processos Workers

O paradigma *Master-Worker* (MW) é amplamente utilizado em sistemas de computação distribuída, em que vários processadores ou máquinas trabalham juntos para resolver um problema em paralelo (RAUBER; RÜNGER, 2013). O modelo de implementação MW é útil para resolver problemas que podem ser divididos em subproblemas independentes, que podem ser resolvidos de forma paralela e agregados para produzir uma solução final. O paradigma MW pode ser utilizado quando um problema precisa ser executado em fases, onde cada fase deve ser executada de forma independente. Este modelo de implementação pode ser generalizado como modelo *Master-worker* hierárquico ou multinível onde o *master* de nível superior alimenta grandes blocos de tarefas para *masters* de nível inferior, que subdividem ainda mais as tarefas entre seus próprios *workers*, muito embora, parte do trabalho pode ser executado por eles mesmos. Este modelo é adequado para espaço de endereçamento compartilhado ou paradigmas de troca de mensagens, uma vez que a interação é naturalmente bidirecional. O *master* sabe que precisa dar tarefas e os *workers* que precisam receber tarefas do *master* (RAUBER; RÜNGER, 2013).

No paradigma MW existe um processo que controla a execução do programa. O processo *Master* geralmente executa a função principal de um programa paralelo e cria processos *Workers* em pontos específicos para realizar computações/cálculos. Dependendo da aplicação, os processos *Workers* podem ser criados estática ou dinamicamente. A atribuição de tarefas aos processos *Workers* geralmente é feita pelo processo *Master*, mas os processos *Workers* também podem gerar novos processos *Workers* para computação. Quando isso acontece, este *Worker* passa a ser *Master* dos *Workers* que ele criou. Além disso, o processo *Master* "original"fica responsável apenas pela coordenação dos processos que ele criou, permanecendo em um nível acima dos processos *Workers*, e poderia, por exemplo, executar inicializações, temporizações e operações de saída por meio de mensagens a seus *Workers*, sempre respeitando os níveis hierárquicos. O modelo MW está representado na Figura 7, onde processo *Master* realiza as sincronizações (Sync) com os processos *Workers* (RAUBER; RÜNGER, 2013; GRAMA *et al.*, 2003).

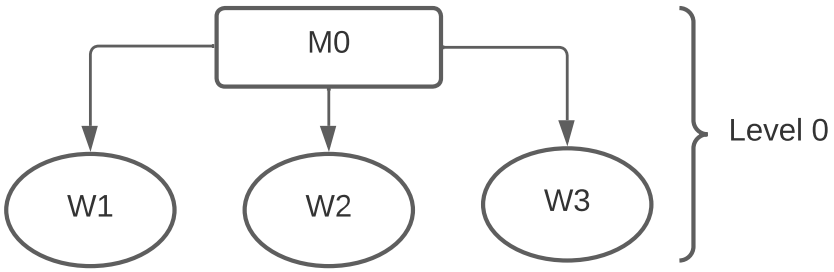


Figura 7 – Modelo Master-Worker.

Formalizando, tem-se um processo *Master M^i* que é responsável por transmitir e receber dados de seus *Workers* $W^1 ... W^j$, onde $j$ é o número de *Workers* criados, em um determinado Level $l_k$. Considerando a Figura 8, o processo $(W^0_{l_0})$, controla os processos *Workers* $(W^1_{l_0})$, $(W^2_{l_0})$ e $(W^3_{l_0})$.

A Figura 8 representa o modelo MW considerando 2 níveis hierárquicos de processos. O *Worker* $(W^2_{l_1})$ controla $(W^4_{l_1})$ e $(W^5_{l_1})$ que ele cria. O *Worker* $(W^3_{N_1})$ controla $(W^6_{l_1})$ que ele cria.

Neste paradigma os processos executam independentemente e se comunicam por passagem de mensagem. O termo *Sync* será utilizado para representar uma aresta de comunicação entre processos por exemplo, $Sync^z = (M^i_{l_k}, W^j_{l_k})$.

**Propriedade 6 (P6) - Aresta de comunicação entre processos *Workers***: Dado o modelo denotado por MW. Se uma aresta de sincronização realiza a comunicação entre dois processos *Workers* de mesmo nível, então esta aresta é não executável.

Toda aresta de comunicação existente entre dois processos do tipo *Worker* (ou seja, um processo *Worker* enviando mensagem para ele mesmo ou para outro processo do tipo *Worker*)

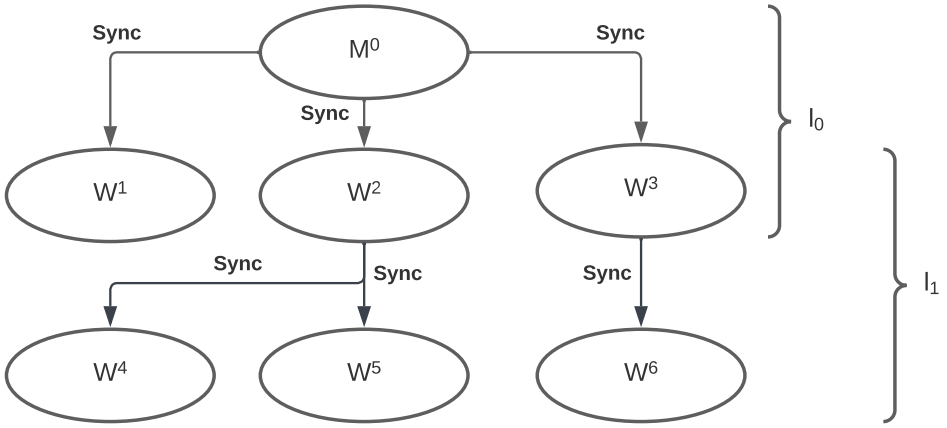


Figura 8 – Modelo Master-Worker com 02 níveis de hierarquia de processos.

acaba sendo limitado pela estrutura do código fonte e pela lógica do modelo. Esse fato impede que um dado de teste execute esse tipo de aresta resultando em uma aresta não executável.

> Toda aresta $Sync^i$ que representa a comunicação entre processos **Workers**, controlados por um mesmo processo **Master** não será executada. Assim, $Sync^i = (W^x_{l_t}, W^y_{l_t})$ sendo $t$ igual para os dois $W$, não será executada, pois não haverá um comando do controlador responsável. Desta forma, $Sync^i$ é não executável.

### 4.2.8 Propriedade 7 - Arestas de comunicação dificilmente executáveis

O conceito de tempo é fundamental para a nossa forma de pensamento e pode ser derivado do conceito mais básico sobre ordem em que os eventos ocorrem. Só é possível dizer que algo aconteceu às 13:15 hs se este ocorreu depois do relógio marcar 13:15 hs e antes de marcar 13:16 hs. O conceito da ordenação temporal eventos na computação permeia esta ideia (LAMPORT, 2019).

Em um programa concorrente, onde existe a disputa por recursos, as vezes é impossível dizer que um dado evento A vai ocorrer antes de um evento B, devido ao não determinismo desses programas. A relação *happens-before* é uma ordenação parcial dos eventos de um programa concorrente (LAMPORT, 2019).

Um evento pode ser um envio ou recebimento de uma mensagem. A relação *happens-before* acontece quando um evento A afeta de alguma forma um evento B (LAMPORT, 2019). Alguns problemas de sincronização surgem justamente porque os desenvolvedores não estão cientes desse fato ou mesmo cientes não se preocupam com a situação negligenciando suas implicações (LAMPORT, 2019). Esta relação temporal tem implicações significativas para a não executabilidade de sincronizações, assim define-se a propriedade abaixo:

**Propriedade 7 (P7) - Aresta dificilmente executável**: Uma aresta é dificilmente executável quando existe uma sincronização *Sync* que não está sendo executada devido a ocorrência de outras sincronizações anteriores.

Dado duas **sincronizações** $Sync^a$ e $Sync^b$ executáveis, em um instante de tempo qualquer $Sync^a$ ocorre, contudo, devido a isso, $Sync^b$ não ocorre. No entanto, em uma nova execução do programa o inverso pode acontecer, $Sync^b$ ocorre e $Sync^a$ não ocorre, e ainda, ambas *Syncs* poderiam ocorrer. O fato da sincronização depender de eventos anteriores, como escalonamento de processos realizado pelo sistema operacional, faz com que as chances de uma comunicação entre nós distantes seja reduzida em relação ao número de possíveis sincronizações anteriores.

Dada uma sincronização $Sync^i = (Z^x, T^y)$ sendo $x$ e $y$ os nomes dos processos e $Z$ e $T$ os nós de ocorrência da sincronização, se $Sync^1 = (2^1, 4^0)$ e $Sync^{30} = (29^2, 4^0)$, poderiam existir 29 sincronizações possíveis antes que a $Sync^{30}$ possa ser escalonada para processamento, contudo, devido ao não determinismo da programação concorrente, a $Sync^{30}$ pode ser executada antes das outras 29 sincronizações possíveis. Logo a $Sync^{30}$ é viável e pode ser executada, mas garantir sua execução é uma tarefa difícil tanto em um processo automatizado como manual. Dessa forma, uma vez identificada a aresta dificilmente executável, o esforço para cobrir a sincronização distante na atividade de teste pode ser reduzido.

Na Figura 9, tem-se três processos onde *P^0* é o *Master*. Considere as seguintes sincronizações executáveis:

$Sync^i = \{(2^1, 4^0), (2^2, 4^0), (8^1, 1^0), (8^2, 1^0)\}$

Suponha que a primeira sincronização do $Sync^1 = (2^1, 4^0)$ irá impossibilitar, devido a relação *happens-before*, que as outras sincronizações $Sync^i = \{(2^2, 4^0), (8^1, 1^0), (8^2, 1^0)\}$ ocorram. Desse modo, para que essas outras três sincronizações sejam cobertas é necessário realizar inúmeras tentativas de cobertura com diferentes casos de teste, sem garantias de que todas as sincronizações sejam executadas em razão do não determinismo.

A detecção da P7 indica a necessidade da execução **controlada** para sincronizações classificadas como P7. Na execução controlada é possível minimizar o efeito da relação *happens-before*. Dessa forma, no contexto de geração automática de dados de teste uma vez identificado esse comportamento o esforço de cobertura das arestas de comunicação poderia ser reduzido, indicando um critério de parada para a geração.

Para toda $Sync^a$ que possua uma dependência com pelo menos uma $Sync^i$, depende da ocorrência da relação *happens-before* entre as sincronizações. Dessa forma, $Sync^a$ é dificilmente executável.

O problema da não executabilidade apresenta-se em várias categorias de programas e mitigar o problema não é uma tarefa trivial e, por esta razão, diferentes abordagens têm sido

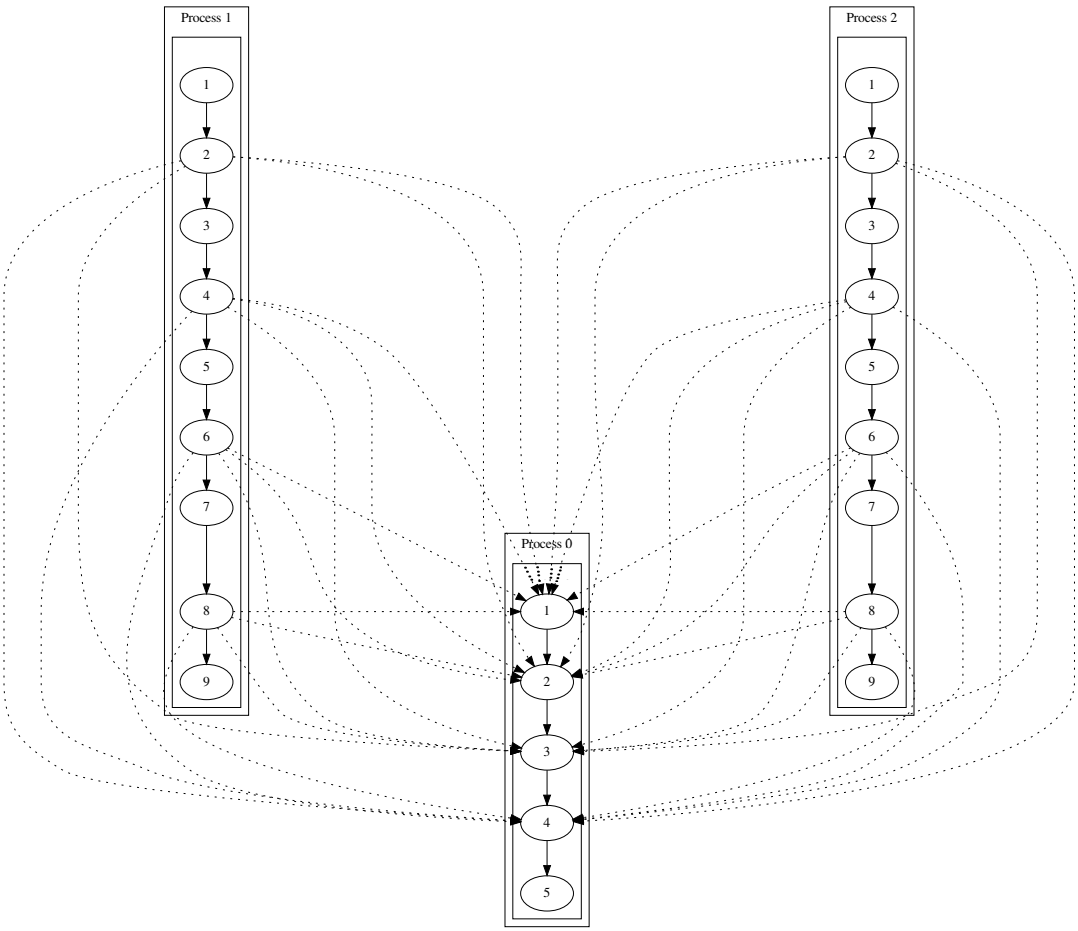


Figura 9 – Arestas de comunicação dificilmente executáveis.

propostas. A importância de se tratar o problema da não-executabilidade está na garantia de qualidade do teste estrutural de software.

Esse seção propôs um catálogo com propriedades que revelam comportamentos no código fonte que favorecem o surgimento de requisitos de teste não executáveis. A aplicação das propriedades, de forma manual ou automatizada, pode reduzir os recursos gastos durante a geração de dados de teste. Salienta-se que da forma como foi proposto esse catálogo pode ser utilizado durante a análise estática do teste estrutural, antes da necessidade de se gerar dados de teste. Esse comportamento refina o conjunto de requisitos de teste exigidos pelo teste estrutural.

Com base neste catálogo de propriedades, na próxima seção será apresentado uma abordagem que propõe a automatização de propriedades das catalogadas com objetivo de mitigar o problema da não-executabilidade sem a execução do programa.

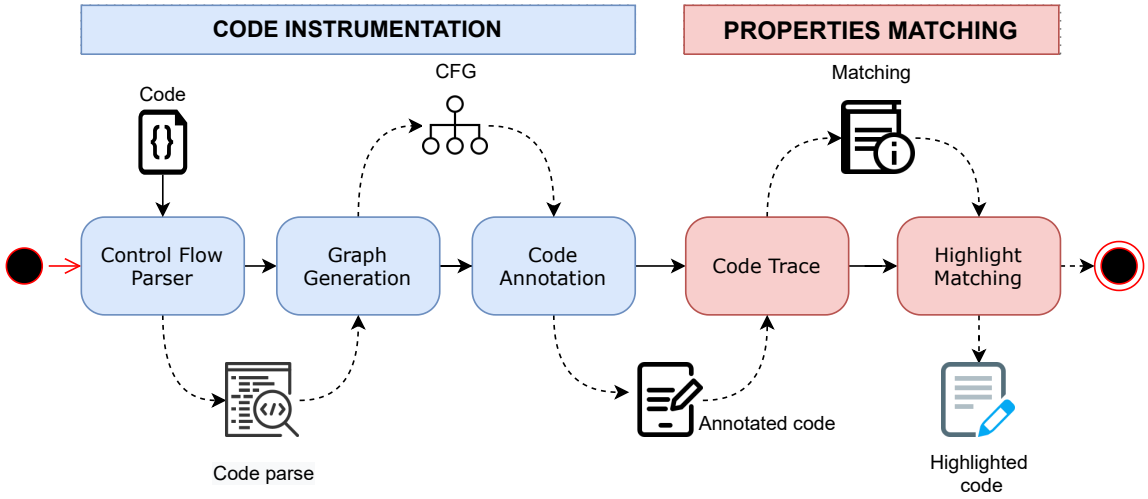


Figura 10 – Fundamentos da abordagem Nonexec.

## 4.3 Abordagem Nonexec

Os resultados alcançados no estudo (CHOMANETO *et al.*, 2021) indicaram a necessidade de sistematizar o processo de aplicação de propriedade. Por essa razão foi proposto uma abordagem, denominada Nonexec, com objetivo de desenvolver um processo de análise de código-fonte capaz de detectar propriedades que evidenciassem comportamentos que levaria a requisitos de teste não executáveis.

No entanto, não foram encontrados ferramentas de apoio que auxiliassem na avaliação do código-fonte e grafo de fluxo de controle. Contar somente com o testador acarretou níveis de exaustão que levaram a avaliações errôneas.

Além disso, uma abordagem de apoio à identificação colabora com a diminuição do custo de geração de dados de entrada, pois retirará do processo da cobertura do código aqueles requisitos de teste identificados como não executáveis.

A Figura 10 apresenta uma visão geral da abordagem, que compreende duas etapas principais: instrumentação de código e correspondência de propriedades, explicados a seguir.

### 4.3.1 Passo 1 - Instrumentação de Código

A atividade de teste estrutural precisa conhecer o código-fonte analisado para interpretar todos os elementos existentes a serem executados. Por esta razão, esta etapa gera um modelo para representar o programa em análise. Esta etapa incorpora três fundamentos: Analisador de Fluxo de Controle, Geração de Grafo e Anotação de Código.

**Analisador de fluxo de controle:** Um analisador de fluxo de controle percorre o código para identificar os requisitos de teste. Cada requisito usado no teste estrutural será reconhecido para apoiar os critérios de teste. Esta análise fornece as informações essenciais para a próxima atividade (Geração de Grafos).

**Geração de Grafo:** Na posse das informações extraídas, esta atividade estabelece um modelo que representa o software em teste. O modelo consiste em um Grafo de Fluxo de Controle (Seção 2.2.1) que apresenta o fluxo de dados do programa e as informações do fluxo de controle. O processo de análise é realizado sobre código fonte compilado.

**Anotação de Código:** Essa atividade realiza a instrumentação do programa inserindo chamadas de função no código-fonte do programa em teste. A instrumentação permite a geração de arquivos de rastreamento para esse programa.

### 4.3.2 Passo 2 - Correspondência de Propriedades

A segunda etapa é responsável pelo reconhecimento das propriedades dos requisitos de teste não executáveis. Em seguida, com um algoritmo que interpreta as propriedades, busca-se os artefatos gerados na Etapa 1. Essa etapa incorpora dois fundamentos: Rastreamento de Código e Correspondência de Destaques.

**Rastreamento de Código:** O algoritmo rastreia as propriedades presentes no código-fonte e no grafo de fluxo de controle, detectando qual propriedade afeta as regiões. Uma lista de correspondências dará suporte à adição de informações ao código.

**Realçar Correspondência:** O mesmo processo de análise utilizado para construir o modelo de grafo de fluxo de controle permite instrumentar o código-fonte por meio de sinalizadores que indicam quais origens de regiões de código são afetadas por uma propriedade. Como resultado desta atividade, têm-se um código destacado indicando quais regiões são afetadas por uma propriedade que pode revelar um requisito de teste não executável.

### 4.3.3 Processo automatizado

Em um estudo anterior (CHOMANETO *et al.*, 2021), foi observado que a experiência do testador, inicialmente benéfica e necessária, pode ser uma ameaça na identificação de requisitos, em razão do custo cognitivo pago pelo testador. Esse fato motivou a construção de automatização da abordagem Nonexec. Para isso foi necessário encontrar ferramentas que apoiassem as etapas da abordagem. Pela razão deste trabalho contemplar programação sequencial e concorrente foi necessário utilizar duas ferramentas.

Para avaliar programas sequenciais só foram encontradas ferramentas de acesso aberto que estavam com o projeto descontinuado. Por essa razão, optou-se por desenvolver uma ferramenta de apoio denominada Fi-paths em colaboração a um projeto de mestrado, vinculado ao grupo de pesquisa deste trabalho. Já para avaliar programas concorrentes, optou-se por utilizar a ferramenta Valipar, utilizada pelos membros do grupo de pesquisa. Ambas as ferramentas são descritas nas próximas subseções.

### 4.3.4 Fi-paths

A ferramenta Fi-paths automatiza as etapas da abordagem Nonexec. Fi-paths foi estabelecido para analisar o comportamento de programas Java. Ela fornece ao testador informações de código-fonte que suportam a identificação de requisitos de teste não executáveis e potencialmente não executáveis. Esta seção apresenta a estrutura dessa ferramenta, bem como os desafios, decisões e limitações dessa implementação. A Figura 11 mostra a disposição arquitetural da ferramenta e, auxiliará nas explicações seguintes.

**Detalhes de Implementação:** A ferramenta foi desenvolvida em Java. Ele usa a API *framework* ASM (BRUNETON, 2007) para percorrer o grafo de fluxo de controle de cada método de cada classe. Em seguida, o grafo e os dados gerados por esse processo são analisados para extrair as linhas do código-fonte que correspondem às características das áreas potencialmente problemáticas do programa conforme descrito por cada propriedade (Seção 4.2. Esta análise também é implementada usando o *framework* ASM (BRUNETON, 2007) para identificar padrões no grafo de fluxo de controle do *bytecode* alvo. O *framework* ASM foi escolhido para esta tarefa por causa de sua flexibilidade, estabilidade, a comunidade considerável em torno dele fornecendo suporte aos usuários e a abstração de alto nível do *bytecode* que ele pode fornecer por meio de sua API.

**Entrada e Saída da Ferramenta:** Do ponto de vista do usuário, a única classe diretamente acessível é CorePF que dá acesso à classe estática CorePFBuilder que constrói o objeto com as opções solicitadas. Este objeto é uma instância do CorePF que possui métodos para análise e saída de dados de acordo com as opções fornecidas pelo usuário quando este objeto é construído. Essas opções são os arquivos de bytecode que devem ser analisados, as propriedades a serem procuradas durante a análise, o caminho em que os arquivos resultantes serão gravados e a saída desejada: o bytecode instrumentado e/ou os dados em um formato legível por humanos e por máquina formato que na versão original dos Fi-paths só pode estar no formato JSON.

**Execução da Ferramenta:** Após a construção do objeto CorePF a execução segue os passos abaixo:

1. Para cada classe correspondente a cada bytecode que está sendo analisado, a ferramenta cria uma instância de ClassAnalyser.

2. Cada objeto ClassAnalyser cria uma instância de MethodAnalyser e fornece a ela um PropertyMatcher para cada propriedade que deve ser observada durante a análise e os dados necessários para gerar os resultados desejados.

3. O MethodAnalyser constrói o grafo de fluxo de controle para o método a que corresponde usando o framework ASM (BRUNETON, 2007) abaixo dele e então procura os padrões correspondentes a cada propriedade conforme solicitado pelo usuário. A saída esperada

também é gerada durante esta análise de acordo com os parâmetros fornecidos pelo ClassAnalyser.

**Decisões de Projeto:** Para implementar as novas propriedades da ferramenta Fi-paths, foi necessário projetar a ferramenta de forma a permitir que ela fosse extensível. Disponibilizar apenas um ponto de entrada para o usuário, bem como projetar o comportamento da ferramenta para ser o mais flexível possível, foi necessário para facilitar a adição de novos recursos sem quebrar nenhum código existente que possa eventualmente depender dele. Devido à sua flexibilidade, o Builder Design Pattern foi considerado uma escolha adequada para esta tarefa. O código morto é identificado pela Propriedade 5 e não tem impacto nessa ferramenta, pois esse tipo de código é ignorado durante a compilação do bytecode.

### 4.3.5 ValiPar

A ferramenta ValiPar (SOUZA *et al.*, 2005) destina-se a apoiar a aplicação de modelos e critérios de teste em programas concorrentes. A ferramenta usa a técnica estrutural para testar programas enquanto usa mecanismos de comunicação baseados em passagem de mensagem e de memória compartilhada. A ValiPar possui diferentes versões desenvolvidas para suportar testes de programas concorrentes, em diferentes linguagens de programação, sendo uma delas a linguagem Java, utilizada neste trabalho. Esta versão aborda tanto o modelo de passagem de mensagem, quanto o modelo de memória compartilhada (SOUZA *et al.*, 2005).

A ferramenta ValiPar é composta por cinco módulos, que recebem como entrada artefatos específicos. A Figura 12 apresenta a arquitetura da ferramenta, os módulos são representados pelos retângulos enquanto as entradas são representadas pelas elipses.

O módulo ValiInst recebe como entrada o programa concorrente a ser testado. Este módulo realiza a análise estática do programa e extrai informações de fluxo de controle, fluxo de dados e fluxo de sincronização, que serão usadas por outros módulos no processo de teste. O módulo ValiInst realiza a instrumentação do código-fonte, necessário para construir os rastros de execução dos dados de teste (SOUZA *et al.*, 2005).

O módulo ValiElem tem como objetivo gerar os requisitos de teste para os critérios de teste implementados. Além disso, é gerado um descritor responsável por estabelecer possíveis caminhos que podem cobrir cada requisito de teste. Esse módulo recebe como entrada as informações de fluxo e o PCFG gerado pelo módulo ValiInst (SOUZA *et al.*, 2005).

O módulo ValiExec é responsável pela execução dos casos de teste. O módulo recebe o programa instrumentado e o executa com os dados de teste fornecidos pelo usuário. Para cada execução do programa, este módulo armazena as informações sobre entradas e saídas, a relação de caminhos percorridos por cada processo e a sequência de sincronizações. Também é possível, a execução determinística e não determinística dos programas sob teste, isto é, com a execução não determinística (ou livre) é possível que qualquer aresta de sincronização seja exercitada

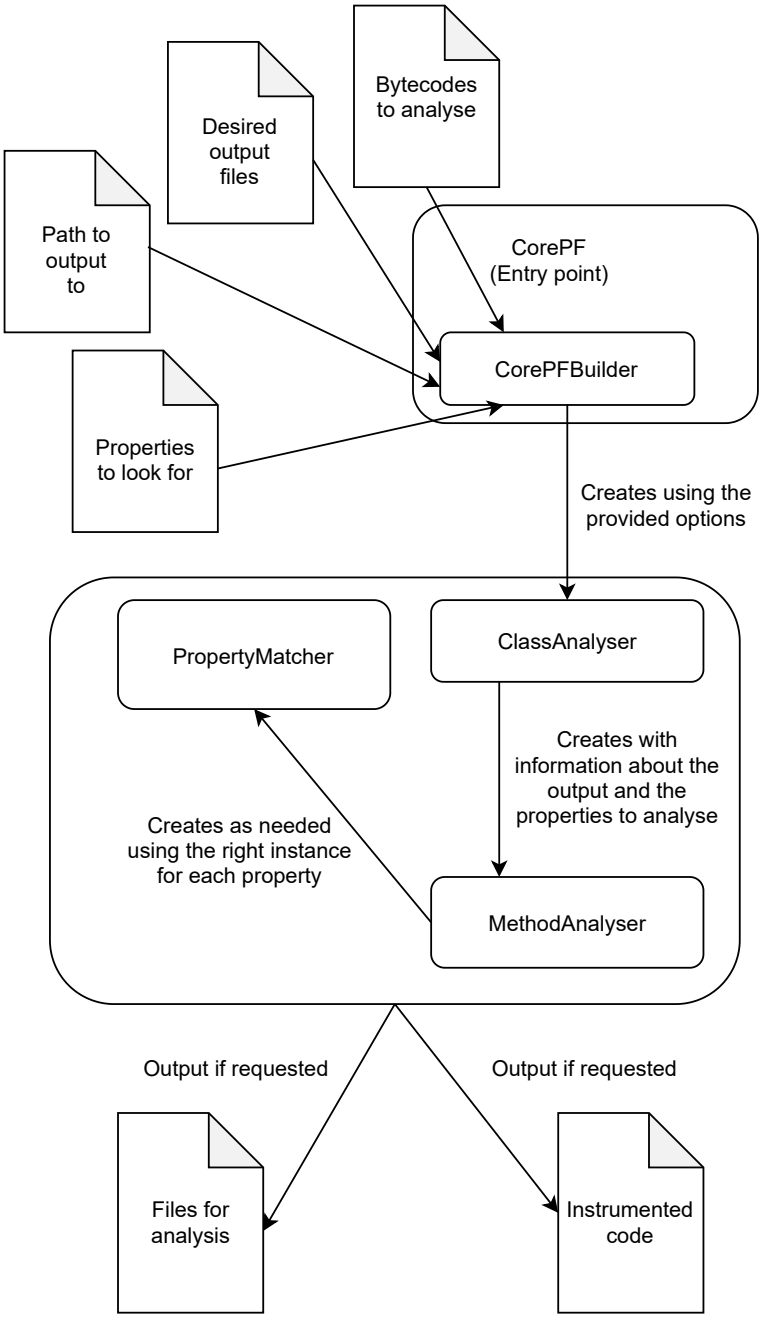


Figura 11 – Arquitetura da Ferramenta Fi-paths.

durante a execução, já com a execução determinística o usuário pode repetir uma sequência de sincronizações executada previamente, considerando os mesmos dados de teste (SOUZA *et al.*, 2005).

O módulo ValiEval avalia a cobertura alcançada com a execução dos casos de teste executados na ValiExec. O módulo utiliza os dados de rastro e os requisitos de teste gerados pelo módulo ValiElem para calcular a cobertura obtida e, então apresenta quais requisitos foram cobertos pelo conjunto de teste utilizado (SOUZA *et al.*, 2005).

Finalmente, o módulo ValiSync tem como objetivo realizar a geração automática de variantes de sequência de sincronização. O módulo impõe a execução de diferentes pares de

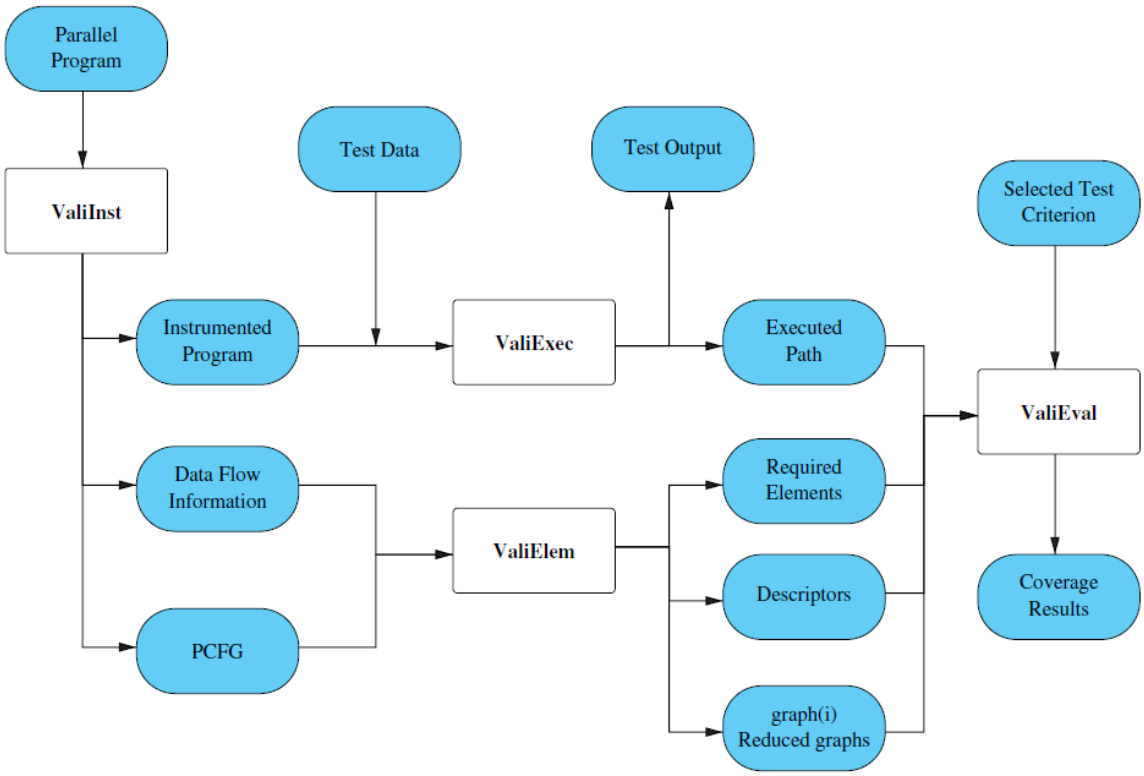


Figura 12 – Arquitetura da Valipar (Fonte:Valipar (SOUZA *et al.*, 2005)).

sincronização com um mesmo dado de teste, aumentando, se possível, a cobertura das arestas de sincronização (BATISTA, 2015).

- Na Etapa 1, o processo automatizado utiliza a ValiInst para realizar a instrumentação do código-fonte e a geração do grafo de fluxo de controle com as informações de comunicação dos processos.
- Na Etapa 2 do processo, o algoritmo de rastreabilidade desenvolvido para aplicação das propriedades P6 e P7 tem como entrada o grafo de fluxo de controle gerado pela ValiInst. O objetivo do algoritmo foi buscar por características no código que revelassem as propriedades P6 e P7. As regiões identificadas indicaram a ValiElem quais requisitos de teste foram afetados, nessa etapa, os requisitos afetados pela P6 foram marcados como não executáveis e os requisitos afetados pela P7 foram marcados como dificilmente executáveis.

Neste ponto, existem informações suficientes para identificar quais requisitos de teste são executáveis, não executáveis e dificilmente executáveis. Com os desafios de encontrar uma ferramenta de apoio para aplicação as propriedades P6 e P7, optou-se pela utilização da Valipar por ser uma ferramenta com diversas funcionalidades e participante do grupo de pesquisa, além de, oferecer as características que este trabalho necessita, análise estática de bytecode Java com geração de GFCP, requisitos de teste e rastreabilidade de execução.

### 4.3.6 Implementação na Valipar

A estrutura da Valipar já oferecia espaço para marcação de requisitos de teste não executáveis. Com essa motivação foi decidido realizar a implementação das propriedades P6 e P7 com a adição de métodos nos módulos da Valipar. Desta forma, as seguintes classes foram modificadas:

- Módulo **models** - interface **RequiredElement** métodos **isHardFeasible()** e **markAsHardFeasible()**. Os métodos foram criados para classificação dos requisitos dificilmente executáveis.
- Módulo **models** - classe **LocalRequiredElement** adição do método **isHardFeasible()** e **markAsHardFeasible()**. Os métodos foram criados para classificação dos requisitos não executáveis.
- Módulo **models** - classe enum **RequiredElementState**, implementação do **HARDFEASIBLE** e método **isHardFeasible()**. Os métodos foram criados para classificação dos requisitos dificilmente executáveis.
- Módulo **clint** - classe **RemoteRequiredElement** adição do método **isHardFeasible()** e **markAsHardFeasible()**. Os métodos foram criados para classificação dos requisitos dificilmente executáveis.
- Módulo **elem** - classe **SyncEdgeRequirementFactory**, adição do método **setPropertySix()** e **setPropertySeven()**. Adição da chama dos novos métodos no método **getRequiredElements()**. Nesse ponto foi realizada a implementação das propriedades P6 e P7, as quais, contribuem com a ValiElem na marcação do status dos requisitos de teste requeridos. Os requisitos eram classificados como não executáveis (P6) ou dificilmente executáveis (P7).
- Módulo **eval** - classe **ValiEval**, adição do método **printSyncs()**, responsável por retornar as sincronizações do programa avaliado e seu status, coberto, não coberto, não executável ou dificilmente executável. Esse método criado para interação com o usuário via terminal.

## 4.4 Considerações Finais

O desenvolvimento de programas sequenciais e concorrentes é fundamental para operacionalidade de sistemas de informação, entretanto a atividade de teste ainda se depara com limitações a serem mitigadas, uma delas é o problema da não-executabilidade de requisitos de teste. Dentre os artifícios utilizados na atividade de teste como, a geração automática de dados de teste que visa facilitar o processo de seleção de entradas e otimizar a verificação dos produtos de software é penalizada pelo mesmo problema.

Este capítulo apresentou a abordagem Nonexec que tem como objetivo mitigar o problema da não-executabilidade em programas sequenciais e concorrentes, por meio da aplicação de propriedades indicadoras de regiões de código que são não-executáveis, possivelmente não-executáveis e dificilmente executáveis. Essas regiões refletem diretamente no comportamento dos requisitos de teste, desfavorecendo a atividade de teste e aumentando seu custo. Para a aplicação automática da abordagem foi desenvolvida a ferramenta Fi-paths que permite a identificação das propriedades P1, P2, P3 e P4 em programas sequenciais e concorrentes. Além disso, a ferramenta ValiPar foi atualizada para aplicar as propriedades P6 e P7 em programas concorrentes. Para avaliar a abordagem Nonexec, o próximo capítulo apresenta dois estudos experimentais.

CAPÍTULO

# 5

# AVALIAÇÃO EXPERIMENTAL

## 5.1 Considerações Iniciais

As percepções e aprendizagens construídas a partir da experimentação permite o avanço no entendimento sobre um tema investigado. Na engenharia de software a análise experimental é utilizada como forma de obtenção de conhecimento e artefatos que corroboram para construção do conhecimento científico(SHULL *et al.*, 2001). Dessa forma, este capítulo apresenta os resultados do processo experimental utilizado para entendimento, predição e avaliação dos conceitos trabalhos até o momento. A análise experimental foi realizada de forma empírica na qual observou-se o comportamento do código fonte com objetivo de identificar comportamentos padronizados que acarretavam o surgimento de requisitos de teste não executáveis e consequentemente caminhos não executáveis.

A definição dos estudos foi inspirada em diretrizes definidas em trabalhos de diferentes domínios (SHULL *et al.*, 2001; Baral; Offutt, 2020; Afzal *et al.*, 2020; Koc *et al.*, 2019; Zhu; Panichella; Zaidman, 2018; RODRIGUES; BRANCHER, 2019). A análise da abordagem foi dividida em: estudo manual relacionado à aplicabilidade das propriedades catalogadas (Capítulo 4) e a análise da abordagem proposta *Nonexec* (Capítulo 4).

## 5.2 Estudo inicial: aplicabilidade do catálogo de propriedades para identificação de requisitos não executáveis

O principal objetivo neste estudo foi analisar o processo de aplicação das propriedades catalogadas para identificação de requisitos de teste não executáveis e medir sua eficácia. Desta forma, respondeu-se as seguintes questões de pesquisa:

| |
|---|
| **QP1:** As propriedades catalogadas podem identificar requisitos de teste não executáveis? |
| **QP2:** Qual o reflexo da aplicação das propriedades catalogadas na atividade de teste? |

Para isso, foi realizado um estudo exploratório com delineamento inspirado em estudos empíricos em diversos domínios (por exemplo, teste de mutação, teste de sistemas robóticos e jogos educativos) (Baral; Offutt, 2020; Koc *et al.*, 2019; RODRIGUES; BRANCHER, 2019). O estudo seguiu uma estratégia baseada nos conceitos de software empírico, e coletou dados quantitativos e qualitativos (SHULL *et al.*, 2001; Baral; Offutt, 2020; Koc *et al.*, 2019; RODRIGUES; BRANCHER, 2019). Como resultado, o estudo experimental foi definido como segue.

### 5.2.1 Conjunto de dados

Para realização dos estudos foi necessário a criação de um conjunto de códigos-fonte para análise e para isso, foi utilizado o trabalho de Ziviani ZIVIANI como inspiração. O conjunto reuniu 19 programas na linguagem Java que foram desenvolvidos para fins acadêmicos. Tabela 10 apresenta um ID para os programas, seu nome e uma breve descrição de sua funcionalidade, Colunas 01, 02 e 03 respectivamente. Esses programas foram selecionados por serem comumente utilizados na literatura e por sua estrutura de dados estar presente na maioria dos sistemas de software corporativo. Além disso, esses programas apresentam características encontradas em programas reais, como definição e uso de variáveis, mudanças na definição de variáveis, estruturas de repetição, estruturas de decisão, classes e métodos. Além deste conteúdo, os códigos-fonte dos programas e uma breve descrição de suas funcionalidades foram disponibilizados no GitHub [1].

Para representar a execução da atividade de teste, foi necessário a aplicação de um critério de teste. Foi escolhido um critério de teste estrutural baseado em fluxo de dados chamado *All-uses* (RAPPS; WEYUKER, 1985). Este critério exige que todas as associações entre definição e uso de uma variável sejam executadas por pelo menos um caso de teste. Cada associação a ser coberta é formada pela definição de uma variável e seu consequente uso. Uma definição ocorre quando um valor é atribuído a uma variável; o uso ocorre quando o valor da variável é empregado na computação ou em uma expressão. O critério *All-uses* foi aplicado com o apoio da ferramenta de teste Baduíno [2].

Como resultado da aplicação do critério de teste, vários requisitos de teste foram criados para apoiar a análise dos programas. Os requisitos foram contabilizados na Tabela 11 na coluna denominada *All-uses*.

### 5.2.2 Procedimento

Inicialmente, foram analisadas as propriedades de identificação e desenvolvidos exemplos, conforme apresentado no Capítulo 4. Com base nos exemplos do catálogo, o autor do estudo realizou a identificação das propriedades manualmente por meio da análise do código-fonte dos programas do **Conjunto de dados**. Quando uma propriedade foi identificada, um sinalizador *(e.g.*

[1] https://github.com/JoaoChoma/iceis2021
[2] https://github.com/saeg/baduino

Tabela 10 – Conjunto de programas analisados.

| Id | Nome | Descrição |
|---|---|---|
| 1 | Max | Programa para obter o valor máximo de um conjunto. |
| 2 | MaxMin1 | Programa para obter os valores máximo e mínimo de um conjunto, de forma não eficiente. |
| 3 | MaxMin2 | Programa para obter os valores máximos e mínimos de um conjunto, de forma mais eficiente que MaxMin1. |
| 4 | MaxMin3 | Programa para obter os valores máximos e mínimos de um conjunto, de forma mais eficiente que MaxMin2. |
| 5 | Mergesort | Classificando o programa pelo método mergesort. |
| 6 | StackAutoRef | Programa para manipular uma estrutura de dados em pilha, implementado através de estruturas de auto-referência. |
| 7 | Sort | Programa para ordenar um array de inteiros. |
| 8 | Fibonacci | Programa para calcular a sequência Fibonacci. |
| 9 | StackArray | Programa para manipular uma estrutura de dados em pilha, implementado através de arranjos. |
| 10 | QueueArray | Programa para manipular uma estrutura de dados de fila, implementado através de arranjos. |
| 11 | HeapSort Max | Programa de manipulação de filas de prioridade, implementado através de arranjos. |
| 12 | HeapSort Max 2 | Programa de manipulação de filas de prioridade, implementado através de arranjos. |
| 13 | HeapSort Min | Programa de manipulação de filas de prioridade, implementado através de arranjos. |
| 14 | HeapSort MinInd | Programa de manipulação de filas de prioridade, implementado através de arranjos. |
| 15 | PartialSorting | Programa de ordenação para obter os primeiros k elementos de um conjunto ordenado de tamanho n. |
| 16 | BinaryTree | Programa que implementa árvore binária e suas operações. |
| 17 | Hashing | Programa que implementa o método de hash, usando listas encadeadas para resolver colisões. |
| 18 | DepthFirstSearch | Programa que implementa uma pesquisa em profundidade em um grafo. |
| 19 | BreadthFirstSearch | Programa que implementa uma pesquisa em largura em um grafo. |
| 20 | Graph | Programa para obter componentes fortemente conectados de um grafo. |
| 21 | PrimAlg | Programa que implementa o algoritmo de Prim para descobrir a árvore geradora mínima em um grafo ponderado. |
| 22 | ExactMatch | Programa que implementa o algoritmo de correspondência exata para pesquisar em strings. |
| 23 | AproximateMatch | Programa que implementa o algoritmo de correspondência aproximada para pesquisar em strings. |

*P1, P2)* foi escrito no código-fonte marcando quais linhas de código foram afetadas/alcançadas por uma propriedade do catálogo.

Na sequência, os requisitos de teste do critério *All-uses* foram analisados para contabilizar quantos requisitos foram afetados por um propriedade, essa análise foi apoiada pelos sinalizadores *(e.g. P1, P2)* inseridos nos programas na etapa anterior.

### 5.2.3 Coleta de dados

Os dados coletados foram todos quantitativos. Números de linhas de código, número de requisitos de teste, número de propriedades identificadas, o número de linhas de código afetadas por uma propriedade e número de requisitos de teste afetados por uma propriedade foram contabilizados. Um requisito de teste afetado por uma propriedade foi definido como **possivelmente não executável** e um requisito de teste que apresentou o comportamento não executável foi definido como **não executável**.

### 5.2.4 Resultados

A Tabela 11 inclui algumas características dos resultados em questão. A primeira coluna mostra o nome do programa analisado, enquanto o número de linhas do código de cada programa na segunda coluna. As colunas P1 a P5 indicam o número de linhas de código afetadas por cada propriedade. A coluna LOCs Afetados mostra a soma das linhas do código afetadas pelas propriedades. A coluna Requisitos de teste exibe o número de requisitos de teste gerados pelo critério de teste *All-uses*. E, por fim, a última coluna indica o número de requisitos de teste afetados pelas propriedades.

As etapas de análise dos códigos fonte e identificação de propriedades foram realizados inteiramente de forma manual, sem qualquer ferramenta de apoio. As propriedades foram aplicadas através da análise sintática e semântica dos programas em busca de características que retratassem as propriedades. A fase de análise do programa foi a que mais demandou tempo e gasto cognitivo do testador.

O desenvolvimento da análise de código se decorreu durante 60 dias ao passo que o testador interpretava o código analisado e o relacionava ao fluxo de dados e de controle. As ocorrências das propriedades foram registradas pelo testador com base nas características definidas no catálogo.

Foram encontradas regiões que, apesar de apresentarem as características que levam a requisitos do tipo não executáveis, acabam não tornando os requisitos não executáveis. Como é o caso das estruturas de repetição que apesar de alterar a definição de uma variável acaba por cobrir o requisito de teste devido às repetições. Neste caso, a experiência do testador foi necessária para tomar a decisão de definir se um requisito de teste seria não executável. Desta forma, foi necessário classificar os requisitos de teste afetados por uma propriedade como requisito de

Tabela 11 – Dados recolhidos durante o estudo.

| Programas | LOC | P1 | P2 | P3 | P4 | LOCs Afetados | *All-uses* | Requisitos Afetados |
|---|---|---|---|---|---|---|---|---|
| MaxMin2 | 14 | 0 | 0 | 0 | 4 | 4 | 38 | 19 |
| MaxMin1 | 13 | 0 | 0 | 0 | 4 | 4 | 38 | 19 |
| MaxMin3 | 25 | 2 | 6 | 0 | 10 | 18 | 100 | 15 |
| MergeSort | 23 | 2 | 0 | 2 | 4 | 8 | 83 | 36 |
| Sort | 71 | 2 | 2 | 2 | 18 | 24 | 244 | 65 |
| Fibonacci | 11 | 2 | 0 | 0 | 4 | 6 | 16 | 12 |
| StackArray | 25 | 4 | 0 | 0 | 8 | 12 | 21 | 10 |
| QueueArray | 27 | 2 | 0 | 0 | 4 | 6 | 29 | 10 |
| HeapSort Max | 25 | 0 | 4 | 10 | 8 | 22 | 57 | 26 |
| HeapSort Max 2 | 62 | 0 | 4 | 10 | 10 | 24 | 145 | 57 |
| HeapSort MinInd | 59 | 0 | 0 | 2 | 12 | 14 | 165 | 56 |
| BinaryTree | 88 | 0 | 16 | 0 | 14 | 30 | 161 | 74 |
| Hashing | 77 | 14 | 10 | 4 | 4 | 32 | 103 | 7 |
| DepthFirstSearch | 42 | 10 | 2 | 0 | 8 | 20 | 77 | 16 |
| BreadthFirstSearch | 57 | 12 | 2 | 0 | 4 | 18 | 105 | 31 |
| Graph | 85 | 2 | 4 | 2 | 40 | 48 | 179 | 79 |
| PrimAlg | 45 | 6 | 0 | 0 | 6 | 12 | 94 | 11 |
| ExactMatch | 54 | 4 | 0 | 6 | 10 | 20 | 210 | 68 |
| AproximateMatch | 28 | 2 | 0 | 0 | 8 | 10 | 85 | 8 |

teste possivelmente não executável. Com essa nova categoria foi possível filtrar o número de requisitos de teste que necessitam dá atenção do testador.

> Um requisito de teste possivelmente não executável é aquele requisito afetado por uma propriedade de requisitos de teste não executáveis.

A ação de decidir se um requisito é não executável não pode ser reduzido somente a encontrar uma propriedade no código, ela demanda mais informações e perspectivas humanas. Apesar de não ser possível decidir de forma imediata se um requisito é não executável, identificar quais requisitos são possivelmente não executáveis pode ser compensatório para o processo de teste. O custo de manter um testador analisando o código linha por linha é caro, o custo de utilizar um processo automatizado torna-se caro mediante a requisitos não executáveis, sendo assim, no mínimo, compensaria identificar todas as regiões com possibilidade de serem não executáveis para direcionar a atividade do testador. Nesta perspectiva, indicar os requisitos do tipo possivelmente não executáveis amortiza o custo do testador ou do teste automatizado de analisar todo o conjunto de requisitos as cegas e o direciona às regiões de código que necessitam da perícia humana para ser testada.

Através da análise sintática e semântica do código, foram encontrados trechos de código que apresentam características de propriedades que podem causar um requisito de teste não executável, mas não resultam em um requisito de teste não executável. Essa característica torna

a atividade de teste mais desafiadora. Embora a localização da propriedade possa ser realizada automaticamente, a determinação de requisitos de teste não executáveis é baseada na habilidade do testador. Como é o caso do programa *BinaryTree*. O código apresentou 161 requisitos de teste requeridos pelo critério de teste. Foram encontras 30 ocorrências de propriedades afetando 74 requisitos. Uma possível ocorrência de um requisito não executável está no Código-fonte 10, o qual mostra que não existe um dado de entrada para *p* onde o as variáveis *writea* (linha 0070), no Código-fonte, e *writeb* (linha 0077) recebam o valor de *write* (linha 0068). Como o problema da não executabilidade é baseado na existência de caminhos não executáveis, é possível pensar se não existiria um caminho não executáveis afetando esta estrutura de código, assim, esse caminho interfere na definição de duas variáveis ocasionando a ocorrência das propriedades e trazendo requisitos *All-uses* não executáveis.

```
1 /* 0067 */   private void testa (No p) {
2 /* 0068 */     if (p == null) String write = "p null" return;
    /*(p2)*/
3 /* 0069 */     if (p.esq != null) {
4 /* 0070 */     String writea = write;
5 /* 0071 */       if (p.reg.compara (p.esq.reg) < 0) {
6 /* 0072 */         System.out.println ("Erro: Pai " + p.reg.
    toString() + " menor que filho a esquerda " + p.esq.reg.
    toString());
7 /* 0073 */         System.exit(1);
8 /* 0074 */     }
9 /* 0075 */     }
10 /* 0076 */     if (p.dir != null) {
11 /* 0077 */       String writeb = write;
12 /* 0077 */       if (p.reg.compara (p.dir.reg) > 0 ) {
13 /* 0078 */         System.out.println ("Erro: Pai " + p.reg.
    toString() + " maior que filho a direita " + p.dir.reg.
    toString());
14 /* 0079 */         System.exit(1);
15 /* 0080 */       }
16 /* 0081 */     }
17 /* 0082 */     testa(p.esq);
18 /* 0083 */     testa(p.dir);
19 /* 0084 */   }
```

**Código-fonte 10** – Código parcial do programa BinaryTree.

### 5.2.5 Discussão

A forma como as propriedades foram catalogadas, neste trabalho, permite que a busca de características no código possa ocorrer de forma automatizada. Neste contexto, é possível retirar o testador do processo de identificação e reinseri-lo apenas na necessidade de tomada de decisão. Deste modo, é possível automatizar a aplicação das propriedades devido à forma como as propriedades foram estruturadas no catálogo.

Determinar um requisito de teste não executável é uma tarefa propensa a erros e cara, mesmo para um testador com experiência excepcional. Além disso, uma determinação incorreta de um requisito de teste não executáveis pode não garantir que uma falha possa ser identificada no programa em teste. Tudo isso indica que direcionar o testador a um conjunto reduzido de requisitos de teste possivelmente não executável, reduziria o gasto cognitivo do testador amenizaria o risco da determinação incorreta.

Outra observação feita nesse estudo foi que não existem ferramentas para automação de identificação de requisitos de teste não executáveis sem o uso de dados de teste na literatura. Portanto, existe a necessidade de automatizar e sistematizar a identificação de propriedades.

Uma abordagem pode contribuir para reduzir o trabalho do testador e diminuir o custo de geração de dados de entrada. Essa redução ocorre porque os requisitos de teste não executáveis não serão mais incluídos nos testes estruturais. Apesar de reduzir o trabalho do testador, a automação não o substituirá, pois há propriedades que precisam da contribuição do testador para aumentar sua eficácia.

**Descoberta:** A análise manual em busca de requisitos de teste não executáveis é custosa e desafiadora pois exige tempo e esforço cognitivo do testador. Dessa forma, é mais vantajoso analisar os requisitos de teste possivelmente não executáveis ao invés do conjunto integral de requisitos.

Muito embora as propriedades auxiliem em outros critérios de teste, ela é mais eficaz quando trata o critério Todos-caminhos, uma vez que procura características dependentes do fluxo de dados e de controle. Este critério requer que todos os possíveis caminhos do programa sejam exercitados. Como foi possível ver no exemplo anterior, o fluxo de controle cria requisitos de teste não executáveis que acabam impactando as variáveis.

Por fim, é possível responder às questões de pesquisa que motivaram este estudo.

**QP1:** As propriedades catalogadas podem identificar requisitos de teste não executáveis?

Com base nos resultados do estudo exploratório, a aplicação das propriedades catalogadas identificou poucos requisitos de teste não executáveis. Contudo, foi definido e contabilizado um novo conjunto de requisitos denominados requisitos de teste **possivelmente não executáveis** que necessitam de uma análise refinada para que seja decido sobre sua execução.

**QP2:** Qual o reflexo da aplicação das propriedades catalogadas na atividade de teste?

Como resultado foi identificado que é possível definir um processo automatizado para aplicar as propriedades catalogadas. Outro resultado é que uma vez automatizado, o processo de identificação será capaz de revelar um conjunto de requisitos de teste possivelmente não executáveis. Por último, o conjunto de requisitos possivelmente não executáveis poderá orientar o testador em relação a quais requisitos de teste devem ser analisados de forma refinada e, pode orientar um algoritmo de automatização de teste, como é o caso da geração automática de dados de teste, em como evoluir a busca por melhores casos de teste e evitar gargalos no processo.

Em outra perspectiva, apesar de ser relacionado a caminhos, o problema da não executabilidade pode afetar critérios de teste mais restritos como foi o caso do critério *All-uses*. A identificação de requisitos possivelmente não executáveis irá auxiliar tanto critérios baseados em fluxo de controle como baseados em fluxo de dados.

**Descoberta:** A existência de caminhos não executáveis podem implicar na existência de outros requisitos de teste não executáveis.

### 5.2.6 Ameaças à validade

As seguintes ameaças externas e internas foram reconhecidas e podem ter afetado a validade dos resultados.

Uma possível ameaça à validade externa refere-se à generalização dos resultados porque nossas amostras de conjuntos de dados não foram selecionadas sob nenhum rigor, foram selecionadas devido ao seu baixo nível de complexidade, apesar disso, mesmo nestes programas as propriedades foram identificadas, e por serem uma referência popular. Os programas foram escolhidos por representarem situações típicas e por adequarem-se a essa análise inicial. Além disso, neste estudo foram utilizados programas escritos em Java, portanto não é possível garantir a generalização dos resultados para outras linguagens de programação. Embora as evidências mostrem que as propriedades catalogadas foram modeladas de forma genérica, para aplicá-las em diferentes contextos, novos estudos em outros cenários devem ser realizados para mitigar essa ameaça. Outra ameaça à validade externa refere-se ao tamanho dos programas usados no experimento. Os programas não são tão complexos quanto os programas reais, porém por se tratar de um estudo inicial e exploratório, entende-se que a complexidade do conjunto de dados pode ser aumentada à medida que a análise amadureça.

As ameaças à validade interna incluem a construção do catálogo de propriedades, para mitigar a ameaça, a construção do catálogo foi inspirada em estudos consolidados publicados em bases relevantes. Outra ameaça à validade interna está no processo desenvolvido usado para realizar o estudo exploratório, para mitigar essa ameaça, o estudo foi inspirado em estudos empíricos que exploraram conceitos (Baral; Offutt, 2020; Koc *et al.*, 2019; Zhu; Panichella;

Zaidman, 2018; RODRIGUES; BRANCHER, 2019) e foi baseado no processo exploratório relatado no trabalho de Travassos SHULL *et al.* que define diretrizes para o desenvolvimento de estudos empíricos.

### 5.2.7 Conclusões

O problema da não-executabilidade se apresenta em várias categorias de programas. Tratar e mitigar o problema não é uma tarefa trivial e, por isso, diferentes abordagens têm sido propostas. Apesar da diversidade de propostas, existe uma lacuna de pesquisa relevante relacionada ao uso de um recurso no código-fonte para identificar requisitos de teste não executáveis de forma estática e sem a necessidade de dados de entrada.

Como resultado final deste estudo, a proposta de automatização do processo de identificação das propriedades catalogadas foi concretizado com definição e modelagem da abordagem Nonexec, definida no Capítulo 4. Por se tratar de um estudo inicial, muitas questões permaneceram em aberto, porém, os resultados possibilitaram a construção de uma nova abordagem para auxiliar na atividade de teste estrutural.

Outras questões ainda permaneceram em aberto: *As propriedades podem ser aplicadas em outras linguagens de programação? As propriedades podem ser aplicadas a outros paradigmas de programação?* (por exemplo, programas concorrentes) *Que taxa de requisitos não executáveis são evitados com a aplicação das propriedades? O processo de automação é trivial? Quanto o processo de identificação de requisitos não executáveis reduzirá o custo de geração de dados de teste? Quanto processo de automação reduzirá a carga de trabalho do testador?*

No próximo estudo foi analisado a automatização da abordagem Nonexec com base nas suas implementações nas ferramentas Fi-paths e Valipar.

## 5.3 Segundo estudo: avaliação experimental da abordagem Nonexec

A abordagem Nonexec foi implementada em uma ferramenta, chamada Fi-paths, que dá suporte à abordagem. A ferramenta analisa estaticamente o código-fonte das propriedades para revelar requisitos de teste não executáveis e possivelmente não executáveis. Uma extensão da implementação foi implementada na ferramenta Valipar com a mesma proposta. Para avaliar a abordagem, foi realizado um estudo experimental usando Fi-paths e a Valipar para analisar um conjunto de programas em Java. O principal objetivo neste estudo foi analisar a abordagem em termos de aplicabilidade de propriedades e redução do trabalho do testador ao categorizar o conjunto de requisitos de teste. Especificamente, considerou-se a seguinte questão de pesquisa:

| |
|---|
| **QP1:** A abordagem Nonexec oferece suporte ao testador durante o processo de mitigação de requisitos de teste não executáveis? |

A questão de pesquisa ainda pode ser expressa em termos de hipóteses:

- $H_0$: A abordagem Nonexec **não** oferece suporte significativo ao testador durante o processo de mitigação de requisitos de teste não executáveis.

- $H_1$: A abordagem Nonexec oferece suporte significativo ao testador durante o processo de mitigação de requisitos de teste não executáveis

O projeto deste estudo experimental foi inspirado em outros estudos empíricos usados em outros domínios (ou seja, teste de mutação, teste de sistemas robóticos e jogos educacionais) (Baral; Offutt, 2020; Koc *et al.*, 2019; RODRIGUES; BRANCHER, 2019; SALES; JúNIOR, 2020; SANTOS *et al.*, 2020; SOUZA *et al.*, 2019). Este estudo segue uma estratégia baseada nos conceitos da engenharia de software experimental (SHULL *et al.*, 2001; Baral; Offutt, 2020; Koc *et al.*, 2019; RODRIGUES; BRANCHER, 2019). O estudo experimental foi definido como se segue.

### 5.3.1 Condução

Para analisar a abordagem proposta (Nonexec), o conjunto de dados foi utilizado como entrada para as ferramentas Fi-paths e Valipar. Como resultado, as ferramentas constroem um modelo e aplicam as propriedades ao programa compilado (arquivos *.class*). Quando uma das propriedades aparece no programa, sua localização é marcada com um sinalizador do tipo *flag*. O sinalizador representa qual propriedade foi identificada e quais linhas do código-fonte são afetadas.

Para prosseguir com o experimento, foi necessário usar um critério de teste estrutural. Nesta etapa foram utilizados os critérios *All-uses* e *All-sync*, conforme definido no Capítulo 2. Como as ferramentas de teste realizam uma análise estática do código-fonte, foi utilizado o código-fonte instrumentado da Etapa 2 da abordagem. Com base nas identificações de propriedade sinalizadas por *flags* na etapa anterior da abordagem, foi possível rastrear quais requisitos de requeridos pelos critérios de teste foram afetados por uma determinada propriedade.

O estudo experimental foi dividido em duas fases. Aplicação do processo automatizado para analisar programas concorrentes e, a execução do teste estrutural com a aplicação do critério de teste, geração de dados de teste e cálculo de taxa de cobertura. Os critérios de teste utilizados foram: Critério *All-uses*, este critério exige que todas as associações entre definição e uso de uma variável sejam executadas por pelo menos um caso de teste e, o Critério (*All-sync*) (SOUZA *et al.*, 2013), este critério requer que todas as arestas de sincronização sejam executadas pelo menos uma vez pelo conjunto de teste.

Nota: Um requisito de teste afetado por uma propriedade foi definido como **possivelmente não executável** e um requisito de teste que apresentou o comportamento não executável foi definido como **não executável**.

Dessa forma, todos os requisitos de teste afetados por uma propriedade são denominados possivelmente não executável, pois até o momento não é possível predizer se o requisito é de fato não executável. A aplicabilidade das propriedades indica que o requisito precisa de verificação do testador. Como resultado desta etapa, três conjuntos de requisitos de teste foram formados: *(1)* requisitos de teste **executáveis**, *(2)* requisitos **possivelmente não executáveis** e *(3)* requisitos **não executáveis**. Por fim, calcula-se quantos requisitos de teste são não executáveis e quantos ainda precisam de verificação.

Inicialmente foi executada a ValiElem que contabilizou a quantidade de requisitos de teste existentes em cada programa. Na sequência, o resultado da Etapa 2 do processo automatizado foi utilizado para informar a ValiElem onde as propriedades foram detectadas. Neste ponto foi possível classificar quais requisitos teste foram afetados pelas propriedades, indicando os requisitos não executáveis e possivelmente não executáveis.

A construção dos casos de teste executados na ValiExec contou com a ajuda da ferramenta de geração automática de dados de teste, Bioconcst (VILELA *et al.*, 2023).A abordagem BioConcST explora a geração automática de dados de teste para programas concorrentes usando um algoritmo genético com um novo operador de busca baseado em lógica fuzzy, chamado FuzzyST (VILELA *et al.*, 2023). Com o resultado da ValiExec foi possível aplicar a ValiEval e obter a cobertura do critério de teste *All-uses* e *All-sync*. Os resultados foram agrupados nas Tabelas 13 e 14.

A verificação da hipótese de pesquisa foi realizada com base nos resultados encontrados na aplicação dos critérios de teste antes e depois da utilização da abordagem Nonexec. Duas amostras foram criados sendo: a Amostra 1, as taxas de cobertura encontradas no teste dos programas antes da aplicação da abordagem Nonexec e, Amostra 2, as taxas de cobertura encontradas depois da aplicação da abordagem Nonexec.

Dois teste estatísticos foram utilizados para avaliar a hipótese de pesquisa. O teste de Shapiro-Wilk foi utilizado para avaliar a normalidade das amostras (SHAPIRO; WILK, 1965). O teste t-Student foi utilizado para comparar a média das duas amostras e avaliar se as diferenças entre essas médias são significativas. Ou seja, permitiu avaliar se as diferenças entre as amostras ocorreram por mero acaso ou não (STUDENT, 1908).

As hipótese de teste t-Student são:

- $teste - t - H_0$: A média da amostra é igual à média da referência (ou população).
- $teste - t - H_1$: A média da amostra é diferente à média da referência (ou população).

Dessa forma, com o teste t-Student será utilizado para avaliar a hipótese de pesquisa. Dado a hipótese de pesquisa:

- $H_0$: A abordagem Nonexec **não** oferece suporte significativo ao testador durante o processo de mitigação de requisitos de teste não executáveis.
- $H_1$: A abordagem Nonexec oferece suporte significativo ao testador durante o processo de mitigação de requisitos de teste não executáveis

Se o teste t-Student indicar que não há diferença significativa entre as amostras, então não será possível refutar $H_0$ indicando que a abordagem Nonexec não oferece suporte significativo ao testador. Caso contrário, se o teste t-Student identificar diferença significativa entre as amostras então será possível refutar $H_0$ indicando que a abordagem Nonexec oferece suporte significativo ao testador.

### 5.3.2 Conjunto de dados

O estudo contemplou a execução da Abordagem Nonexec com a aplicação das propriedades definidas no catálogo: P1, P2, P3, P4, P6 e P7.

O conjunto de dados foi composto por programas na linguagem Java que apresentam características encontradas em diversos programas tradicionais, como: definição e uso de variáveis, estruturas de fluxo de controle, estruturas de decisão, classes, métodos e sincronizações. O processo automatizado foi aplicado em um *benchmark* de 13 programas concorrentes, composto de 08 programas implementado no contexto de passagem de mensagem e 04 no contexto de memória compartilhada e 01 implementado em ambos os contextos, passagem de mensagem e memória compartilhada, disponibilizados no *benchmark* TestPar definido por DOURADO. O *benchmark* está apresentado na Tabela 12 indicando o ID e nome programa analisado, LOC (linhas de código), paradigma de comunicação, número de processos, número de *threads* e, o total de requisitos de teste requeridos pelos critérios *All-sync* (AS) e *All-uses* (AU).

### 5.3.3 Coleta de dados

O Código 11 mostra um exemplo de identificação P2 no programa e sinaliza quais linhas estão sendo afetadas. A propriedade P2 foi identificada nas linhas 15 e 17. Essa ocorrência faz com que qualquer requisito de teste que percorra essas linhas de código seja afetado por P2. Quando o critério de teste *All-uses* é aplicado, os seguintes requisitos de teste são afetados por P2:

```
1 package files_java;
2
3 public class Property_2_1{
```

Tabela 12 – Benchmark utilizado no estudo.

| ID | Programa concorrente sob teste | LOC | Paradigma | Processos | Threads | AS | AU |
|---|---|---|---|---|---|---|---|
| **1** | Greatest Common Divisor | 180 | PM | 4 | 4 | 33 | 238 |
| **2** | Greatest Common Divisor Two-Slave | 201 | PM | 3 | 3 | 16 | 210 |
| **3** | Greatest Common Divisor/LeastCommon Multiple | 259 | PM | 4 | 4 | 57 | 619 |
| **4** | Roller Coaster | 374 | PM | 6 | 6 | 206 | 255 |
| **5** | Cigarette Smokers | 253 | MC | 1 | 5 | 42 | 289 |
| **6** | Matrix | 228 | MC | 1 | 5 | 192 | 374 |
| **7** | Jacobi | 411 | MC | 1 | 5 | 532 | 1125 |
| **8** | Token Ring Iterations | 175 | PM | 4 | 4 | 12 | 968 |
| **9** | Token Ring Directions Same Primitives | 221 | PM | 4 | 4 | 12 | 35130 |
| **10** | Token Ring Directions Diferent Primitives | 224 | PM | 4 | 4 | 51 | 5356 |
| **11** | Parallel Sieve of Eratosthenes | 274 | PM | 4 | 4 | 15 | 202938 |
| **12** | Producer Consumer SemaphoreLock Condition Iterations | 241 | MC | 1 | 6 | 20 | 278135 |
| **13** | Token Ring Broadcast | 294 | PM/MC | 3 | 7 | 46 | 6496 |

PM = Passagem de mensagem, MC = Memória compartilhada, AS = *All-sync*, AU = *All-uses*.

```
4
5     public static final int NOT_FOUND = -1;
6
7     public int binarySearch(int a[], int target)
8     {
9         int low =0;
10        int high = a.length-1;
11        int mid;
12
13        while(low <= high){
14            mid = (low + high) / 2;
15            !\color{red}{\textbf{P2}}\colorbox{lightgray}{\
    color{blue}{if}(a[mid] < target)}!
16                low = mid+1;
17            else !\color{red}{\textbf{P2}}\colorbox{lightgray
    }{\color{blue}{if}(a[mid] > target)}!
18                high = mid -1;
19                    else
20                return mid;
21        }
22        return NOT_FOUND;
23    }
24
25 }
```

**Código-fonte 11** – Exemplo de ocorrência da P2.

O requisito de teste derivados de *All-uses* apresentam as informações relacionadas ao

nome da variável *var*, ao nó do GFC de definição da variável e o nó no qual a variável é utilizada $def$ e *use* respectivamente e, a consequência da utilização dessa variável $target$. Nesta configuração tem-se a apresentação formal de requisitos *All-uses* analisados:

<*du covered="0"use="15"def="14"var="mid"target="16"/>*

<*du covered="0"use="15"def="14"var="mid"target="17"/>*

<*du covered="0"use="17"def="14"var="mid"target="18"/>*

<*du covered="0"use="17"def="14"var="mid"target="20"/>*

Neste caso temos uma segunda representação forma dos mesmos quatro requisitos analisados, nó de definição, nó de uso, nome da variável e nó consequência da utilização dessa variável, respectivamente:

$< 14, 15, mid > target16$

$< 14, 15, mid > target17$

$< 14, 17, mid > target18$

$< 14, 17, mid > target20$

Neste exemplo, quando uma variável é utilizada em uma linha afetada significa que os requisitos de teste que requerem alguma informação daquela linha acabam sendo afetados e se tornando possivelmente não executáveis. Com isso, é possível medir a quantidade de requisitos de teste não executáveis e possivelmente não executáveis.

Além disso, o desempenho do processo automatizado foi medido por meio da comparação feita entre os resultados de cobertura sem a aplicação das propriedades e com a aplicação delas.

### 5.3.4 Resultados

As Tabelas 13 e 14 apresentam os resultados obtidos na execução da abordagem Nonexec que aplicou as propriedades P1, P2, P3, P4, P6 e P7 nos programas do *benchmark*. Sua execução foi apoiada pelas ferramentas Fi-paths e Valipar.

Uma nova classe de requisitos foi identificada, pela propriedade P7, denominada **requisitos dificilmente executáveis**, estes requisitos apesar de executáveis exigem atenção do testador e a necessidade de uma execução controlada para serem executados. Os requisito desta nova classe são gerados em razão do comportamento não determinismo existente nas sincronizações de processos concorrentes e da presença do problema de *happen-before* causando um impedimento na execução de alguns requisitos de teste. Durante o processo o processo de teste automatizado uma sincronização já executada interfere em uma sincronização que ocorre em um tempo posterior impedindo sua execução, apesar de executável a sincronização afetada necessita de ajuda do testador para ser testada.

Tabela 13 – Quantidade de ocorrências das propriedades por programas e requisitos *All-uses* afetados.

| Programas | P1 | P2 | P3 | P4 | P6 | P7 | DE | PNE | NE |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 3 | 3 | 2 | 0 | 6 | 15 | 0 | 18/238 (7,56%) | 0 |
| **2** | 27 | 10 | 6 | 6 | 2 | 6 | 0 | 174/210 (82,86%) | 0 |
| **3** | 19 | 10 | 4 | 4 | 26 | 16 | 0 | 115/619 (18,58%) | 0 |
| **4** | 25 | 6 | 3 | 6 | 148 | 11 | 0 | 127/255 (49,8%) | 0 |
| **5** | 12 | 6 | 0 | 0 | 0 | 15 | 0 | 64/289 (22,15%) | 0 |
| **6** | 11 | 4 | 3 | 3 | 0 | 468 | 0 | 70/374 (18,72%) | 0 |
| **7** | 10 | 6 | 4 | 5 | 0 | 172 | 0 | 145/1125 (12,89%) | 0 |
| **8** | 22 | 7 | 2 | 6 | 0 | 0 | 0 | 105/968 (10,85%) | 0 |
| **9** | 14 | 3 | 2 | 7 | 0 | 1 | 0 | 138/35130 (0,39%) | 0 |
| **10** | 31 | 7 | 8 | 6 | 0 | 7 | 0 | 199/5356 (3,72%) | 0 |
| **11** | 29 | 9 | 9 | 15 | 0 | 6 | 0 | 563/202948 (0,28%) | 0 |
| **12** | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 5/278135 (0,00%) | 0 |
| **13** | 6 | 2 | 1 | 1 | 0 | 16 | 0 | 22/6496 (0,34%) | 0 |

DE=Dificilmente Executável, PNE=Possivelmente Não Executável, NE=Não Executável.

**Descoberta:** Um requisito dificilmente executável é um requisito afetado pelo problema do *happen-before* e necessita da atenção do testador para que seja executado.

A Tabela 13 apresenta os resultados referente a aplicação do critério *All-uses*. A Coluna 1 apresenta o ID do programa analisado, as Coluna 2 a 7 apresentam o número de ocorrências das propriedades. A Coluna 8 apresenta o número de requisitos dificilmente executáveis com a sigla (DE), a Coluna 9 apresenta o número de requisitos possivelmente não executáveis com a sigla (PNE) e, a Coluna 10 apresenta o número de requisitos não executáveis com a sigla (NE). De forma complementar foi calculado a taxa de ocorrência de uma categoria (DE, PNE ou NE) dentro do conjunto requisitos de teste requeridos pelo teste.

Com base nos resultados da aplicação do critérios *All-uses*, em média 17,55% dos requisitos de teste requeridos foram classificados como possivelmente não executáveis e a mediana dos dados ficou em 10, 85%. Estes resultados indicam que cerca de 1/10 dos requisitos de teste necessitam de atenção exclusiva do testador, fato que indica quais regiões o teste automatizado pode encontrar gargalos da execução e direcionar os esforços para os outros 90% dos requisitos de teste.

Ao utilizar um critério de teste de fluxo de dados é percebido que uma quantidade considerável de requisitos de teste são afetados pelas propriedades, principalmente pela propriedade P1 que está relacionada diretamente com as variáveis. Por outro lado, apesar da ocorrência, ainda não é possível definir diretamente se um requisito possivelmente não executável (PNE) pode ser categorizado como não executável (NE), o que justifica a coluna NE da Tabela 13 estar com todos os resultados zerados. Outra perspectiva que é possível observar é que as propriedades P6 e P7 por estarem relacionadas exclusivamente as arestas de comunicação dos processos acabam

não afetando os requisitos *All-uses*, o que justifica a coluna DE da Tabela 13 estar com todos os resultados zerados.

Analisando a propriedade P2, observou-se que há casos os quais ela pode mostrar requisitos de teste não executáveis sem a assistência do testador, baseando-se nas diretrizes de propriedade, uma vez que ela é baseada no fluxo de controle. Embora isso não seja possível para as propriedades P1, P3 e P4. Portanto, uma estrutura lógica que representa a primeira percepção do testador durante o processo de avaliação dos requisitos de teste foi reproduzida. Assim, são definidas algumas condições nas quais os requisitos de teste afetados por P2 são não executáveis:

- Condição 1: Se P2 for igual a predicados, então todo requisito de teste afetado pelo fluxo de controle oposto será não executável;
- Condição 2: Se P2 for o predicado oposto, então todo requisito de teste afetado pelo fluxo igual de controle será não executável.

```
1
2 1 package files_java;
3 2
4 3 public class Property_2_3{
5 4
6 5     public static final int NOT_FOUND = -1;
7 6
8 7     public int binarySearch(int a[], int target)
9 8     {
10 9         int low =0;
11 10         int high = a.length-1;
12 11         int mid;
13 12
14 13         while(low <= high){
15 14             mid = (low + high) / 2;
16 15             !\color{red}{\textbf{P2}} {\color{blue}{if (a[mid
    ] > target)}}!
17 16                 low = mid+1;
18 17             else !\color{red}{\textbf{P2}} {\color{blue}{if}
    (a[mid] > target)}}!
19 18                 !\colorbox{lightgray}{high = mid - 1;}}!
20 19                     else
21 20                 return mid;
22 21         }
23 22         return NOT_FOUND;
```

```
24 23        }
25 24
26 25    }
```

**Código-fonte 12** – Exemplo de um requisito de teste não executável.

O Código 12 mostra um programa com a ocorrência de P2 nas linhas 15 e 17 circuladas em azul. O requisito do teste indica que a variável usada na linha 17 e definida na linha 14 deve cobrir a linha 18.

$$< 14, 17, mid > target 18$$

Conforme definido na Condição 1, este requisito de teste não pode ser coberto porque devido à estrutura do código se a condicional na linha 15 for falsa não será possível acessar a condicional na linha 17, e assim não será possível alcançar o alvo na linha 18.

Apesar das definições apresentadas, ainda é recomendado o apoio do testador para avaliar os requisitos afetados pela propriedade P2. Já as propriedades P1, P3 e P4 dependem do fluxo de dados do programa e das habilidades do testador. Embora a localização da propriedade possa ser realizada de forma automática e com baixo custo, o algoritmo de determinação de requisitos de teste não executáveis ainda precisa ser aprimorado.

A Tabela 13 apresenta os resultados referente a aplicação do critério *All-sync*. A Coluna 1 apresenta o ID do programa analisado, as Coluna 2 a 7 apresentam o número de ocorrências das propriedades. A Coluna 8 apresenta o número de requisitos dificilmente executáveis com a sigla (DE), a Coluna 9 apresenta o número de requisitos possivelmente não executáveis com a sigla (PNE) e, a Coluna 10 apresenta o número de requisitos não executáveis com a sigla (NE). De forma complementar foi calculado a taxa de ocorrência de uma categoria (DE, PNE ou NE) dentro do conjunto requisitos de teste requeridos pelo teste.

Com base nos resultados da aplicação do critério *All-sync*, dentre os programas a média de identificação de requisitos não executáveis foi de 37,04% e a mediana foi de 31,90% indicando que em programas de modelo de implementação *Master-Slave* 1/3 dos requisitos tem possuem o problema da não executabilidade. Em relação aos requisitos dificilmente executáveis, a média de ocorrência foi de 26,30% e a mediana foi de 28,07% indicando que aproximadamente 1/3 dos requisitos de teste apresentam algum nível de dificuldade para serem exercitados, em razão do não determinismo existem entre as sincronização dos processos.

As propriedades relacionadas a fluxo de dados (P1, P2, P3 e P4) não afetam os requisitos de teste requeridos pelo critério *All-sync*, critério específico de programas concorrentes, fato que justifica a coluna PNE da Tabela 14 apresentar todos os resultados zerados. Contudo, as propriedades que analisam o problema da não executabilidade nas arestas de comunicação dos processos (P6 e P7) acabam por serem mais efetivas e conseguirem identificar requisitos dificilmente executáveis (DE) e requisitos não executáveis (NE).

Tabela 14 – Quantidade de ocorrências das propriedades por programas e requisitos *All-sync* afetados.

| Programas | P1 | P2 | P3 | P4 | P6 | P7 | DE | PNE | NE |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 3 | 3 | 2 | 0 | 6 | 15 | 15/33 (45,45%) | 0 | 6/33 (18,18%) |
| **2** | 27 | 10 | 6 | 6 | 2 | 6 | 6/16 (37,50%) | 0 | 2/16 (12,50%) |
| **3** | 19 | 10 | 4 | 4 | 26 | 16 | 16/57 (28,07%) | 0 | 26/57 (45,61%) |
| **4** | 25 | 6 | 3 | 6 | 148 | 11 | 11/206 (5,34%) | 0 | 148/206 (71,84%) |
| **5** | 12 | 6 | 0 | 0 | 0 | 15 | 15/42 (35,71%) | 0 | 0 |
| **6** | 11 | 4 | 3 | 3 | 0 | 468 | 468/1944 (24,07%) | 0 | 0 |
| **7** | 10 | 6 | 4 | 5 | 0 | 172 | 172/352 (48,86%) | 0 | 0 |
| **8** | 22 | 7 | 2 | 6 | 0 | 0 | 0/12 (0,00%) | 0 | 0 |
| **9** | 14 | 3 | 2 | 7 | 0 | 1 | 1/12 (8,33%) | 0 | 0 |
| **10** | 31 | 7 | 8 | 6 | 0 | 7 | 7/51 (13,73%) | 0 | 0 |
| **11** | 29 | 9 | 9 | 15 | 0 | 6 | 6/15 (40,00%) | 0 | 0 |
| **12** | 0 | 2 | 0 | 0 | 0 | 4 | 4/20 (20,00%) | 0 | 0 |
| **13** | 6 | 2 | 1 | 1 | 0 | 16 | 16/46 (34,78%) | 0 | 0 |

DE=Dificilmente Executável, PNE=Possivelmente Não Executável, NE=Não Executável.

Tabela 15 – Taxa de coberturas e melhorias alcançadas com a aplicação das propriedades P6 e P7 em relação ao critérios *All-sync*.

| | **Sem aplicação** | | **Com P6** | | | **Com P7** | | |
|---|---|---|---|---|---|---|---|---|
| **Programas** | Requisitos | Cobertura | Requisitos | Cobertura | Melhoria | Requisitos | Cobertura | Melhoria |
| **1** | 7/33 | 21% | 7/27 | 39,39% | 18,39% | 22/27 | 81,48% | 63,09% |
| **2** | 8/16 | 50% | 8/14 | 57,14% | 7,14% | 12/14 | 85,71% | 28,57% |
| **3** | 14/57 | 24% | 14/31 | 45,26% | 21,26% | 30/31 | 96,77% | 75,51% |
| **4** | 25/206 | 12% | 25/58 | 43,10% | 31,10% | 36/58 | 62,07% | 30,97% |
| **5** | 7/42 | 16,67% | 15/42 | 16,67% | 0 | 22/42 | 52,38% | 35,71% |
| **6** | 21/1944 | 1,08% | 468/1944 | 1,08% | 0 | 489/1944 | 25,15% | 24,07% |
| **7** | 32/352 | 9,09% | 172/352 | 9,09% | 0 | 204/352 | 57,95% | 48,86% |
| **8** | 2/12 | 16,67% | 2/12 | 16,67% | 0 | 2/12 | 16,67% | 0 |
| **9** | 4/12 | 33,33% | 4/12 | 33,33% | 0 | 5/12 | 41,67% | 8,33% |
| **10** | 7/51 | 13,73% | 7/51 | 13,73% | 0 | 14/51 | 27,45% | 13,73% |
| **11** | 11/15 | 73,33% | 0/15 | 73,33% | 0 | 15/15 | 100,00% | 26,67% |
| **12** | 10/20 | 50,00% | 10/20 | 50,00% | 0 | 14/20 | 70,00% | 20,00% |
| **13** | 30/46 | 65,22% | 30/46 | 65,22% | 0 | 46/46 | 100,00% | 34,78% |

As propriedades P6 e P7 foram evidenciadas na Tabela 15 em razão da sua novidade no tratamento de programas concorrentes. A tabela apresenta os resultados obtidos na aplicação das propriedades P6 e P7 em relação a taxa de cobertura alcançada no critério *All-sync* durante a execução do teste. A Coluna 1 apresenta o ID do programa analisado, a Coluna 2 e 3 apresentam a quantidade de requisitos cobertos e a taxa de cobertura, respectivamente, sem a aplicação das propriedades. As Colunas 4, 5 e 6 apresentam a quantidade de requisitos cobertos, a taxa de cobertura e a melhoria alcançada em relação a última taxa de cobertura, respectivamente, com a aplicação da propriedade P6. As Colunas 7, 8 e 9 apresentam a quantidade de requisitos cobertos, a taxa de cobertura e a melhoria acumulada em relação a última taxa de cobertura, respectivamente, com a aplicação da propriedade P7.

A propriedade P6 foi efetiva na identificação de requisitos não executáveis em programas

que apresentam o modelo de implementação *Master-Slave*. Um detalhe pode ser ressaltado, quanto maior o número de requisitos de teste maior é a expressividade da melhoria trazida pela aplicação da propriedade P6.

A propriedade P7 evidência quais requisitos de teste precisam de atenção do testador. Esse caso particular impede que o requisito seja exercitado mesmo que seja executável, desse modo com execução controlada é possível executá-lo. Esses requisitos denominados de Dificilmente Executáveis (DE), devido a dificuldade em realizar sua cobertura relacionada ao problema *happens-before*. A propriedade P7 identifica os requisitos que exigem tratamento especial na atividade de teste, uma vez feita a execução controlada do teste foi possível exercitar os requisitos DE. Um caso que pode ser evidenciado é Programa 2, nele houve a ocorrência de 6 requisitos ED, mas 2 ED já haviam sido cobertos no teste automatizado, fato que evidência a possibilidade de cobertura automática desse tipo de requisito mesmo com a dificuldade de sincronização. Outro exemplo que pode ser evidenciado é o Programa 8 que não apresentou a ocorrência de P7, muito embora, o Programa 11 mesmo apresentando poucos requisitos como o Programa 8, apresentou 6 ocorrências de P7, isto indica que a implementação do código corrobora com a ocorrência da propriedade.

A abordagem não afirma que todos os requisitos possivelmente não executáveis são de fato não executáveis, mas indica ao testador quais regiões do código devem ser inspecionadas com mais rigor quando houver requisitos de teste possivelmente não executáveis. Isso significa que durante o teste estrutural, o testador pode concentrar esforços em requisitos que apresentam característica que podem levar ao problema da não executabilidade e deixar o teste automatizado responsável por trabalhar com os outros requisitos ainda não cobertos. Analisando essa perspectiva, a identificação de requisitos possivelmente não executáveis pode ser um indicativo na priorização requisitos de teste.

Devido à complexidade envolvendo fluxo de dados das propriedades P1, P2, P3 e P4, será necessário construir uma estrutura de tomada de decisão inteligente baseada na capacidade do testador de identificar um requisito de teste não executável. Os resultados deste estudo experimental indicam a necessidade de *frameworks* inteligentes para compensar as habilidades do testador na fase de decidir se um requisito possivelmente não executáveis é não executáveis e eliminar falsos positivos.

A propriedades P6 e P7 quando utilizadas pela abordagem Nonexec melhoram os resultados trazidos pelo teste estrutural no contexto de programas concorrentes. Por fim, as propriedades P1 a P4 não refletem melhorias explícitas no teste estrutural contudo indicam regiões do código que apresentam comportamento que exigem atenção do testador. Referente ao critério *All-sync*, poucos requisitos foram afetados por uma propriedade, contudo, analisando o critério *All-uses* é possível ver uma quantidade representativa de requisitos que exigem atenção do testador.

### 5.3.5 Discussão

Analisando o estado da arte, a literatura até agora não apresentou uma estratégia/técnica para automatizar a identificação de requisitos de teste do tipo não executáveis sem o uso de dados de entrada. Como resultado, este estudo inicia a discussão neste cenário, alguns *insights* inesperados foram identificados.

Devido à complexidade do problema da não-executabilidade em testes estruturais, o impasse de decidir se um requisito de teste é não executáveis vai além do processo de encontrar propriedades para apoiar essa identificação. No entanto, o processo de automação permitiu o desenvolvimento da ferramenta Fi-paths e a incorporação da Nonexec na ferramenta Valipar. A automação viabilizou o processo de identificação de propriedades e como resultado ajudou a abordagem a revelar quais requisitos de teste podem apresentar o problema de não executabilidade. A abordagem permitiu a identificação de várias regiões do código que apresentam requisitos possivelmente não executáveis. Essa filtragem informa ao testador quais regiões do código devem ser priorizadas uma vez que elas apresentam comportamento que pode evidenciar o problema da não-executabilidade.

Em contribuição com o desenvolvimento de código, a identificação dos requisitos de teste possivelmente não executáveis pode indicar regiões do código fonte que podem ser refatoradas ou otimizadas. Por exemplo, a utilização recorrente de estruturadas de decisão (*if*) de forma indiscriminada pode indicar a construção de um código inexperiente levando ao problema da não executabilidade. Outro fato que indica más práticas de programação é a utilização ineficaz de variáveis globais que levam a interdependência de estruturas de decisão como *if* e *while*, levando o código a apresentar o problema da não executabilidade.

Fazendo referência a pesquisa que identifica *bad smells* em códigos é possível fazer algumas correlações. A utilização de variáveis congeladas ou globais (propriedade P1), pode indicar rigidez no código. A utilização indiscriminada e inexperiente de estruturas de decisão (propriedades P1 e P2) pode indicar o *bad smell* relacionado a complexidade indicando que o código não tem um fluxo adequado e interdependente. Ainda relacionado a ocorrência das propriedades P2 e P3 é possível identificar a ocorrência do *bad smell* de duplicação de código onde o trechos de código aparecem repetido em algumas regiões do programa. Essa são algumas correlações que é possível fazer com a identificação de propriedades e a observação de regiões para refatoração de código.

O desenvolvimento deste estudo identificou uma clara contribuição para os conceitos de fluxo de dados e fluxo de controle. As propriedades que identificam comportamentos do tipo não executáveis dependentes do fluxo de dados precisam de suporte na tomada de decisão, pois existe a probabilidade de que o requisito analisado seja não executável, mas devido à complexidade da escolha das possibilidades, é necessária tecnologia inteligente para apoiar a abordagem. Por outro lado, as propriedades dependentes do fluxo de controle podem relevar com mais facilidade

quais requisitos de teste são não executáveis além de revelar possivelmente não executáveis. Por outro lado, as propriedades dependentes das sincronizações entre processos são mais efetivas na identificação de requisitos de teste não executáveis.

O problema de tomada de decisão pode ser automatizado com uma proposta de recomendação de que um requisito de teste é possivelmente não executáveis. Isso foi alcançado porque as propriedades P1, P3 e P4 evidenciam os caminhos de fluxo não executáveis do programa e, quando coletadas, podem indicar qual ocorrência de propriedade tem maior probabilidade de causar requisitos de teste não executáveis.

Esperava-se que o custo computacional para execução da abordagem Nonexec seria alto, porém, devido ao processo ter uma estrutura sem implicações semânticas, a execução da abordagem levou alguns segundos. Durante a aplicação manual da abordagem, a análise de 30 programas durou cerca de 60 dias, enquanto que a automatização reduziu esse tempo para 1 dia. Muito embora, seja necessário um novo estudo com foco na avaliação do desempenho computacional da abordagem Nonexec.

**QP1:** A abordagem Nonexec oferece suporte ao testador durante o processo de mitigação de requisitos de teste não executáveis?

Sim, a abordagem possibilitou identificar requisitos de teste não executáveis, possivelmente não executáveis e dificilmente executáveis, reduziu o esforço do testador facilitando a identificação de regiões do código que precisam de atenção na etapa de verificação. Aproximadamente 1/10 dos requisitos de teste necessitam da atenção do testador ficando os outros 90% favoráveis a utilização do teste automatizados. Em relação as propriedades específicas de programas concorrentes, os resultados indicam que a abordagem alcançou o objetivo de auxiliar o testador no processo de teste estrutural revelando requisitos não executáveis e dificilmente executáveis.

Para avaliar a hipótese definida para esse estudo foi utilizado o Teste t de Student que avalia se existe diferença significativa entre duas amostras (STUDENT, 1908). A primeira amostra analisada foi o conjunto de taxas de cobertura encontrado pelo teste em relação ao critério *All-sync* e a segunda amostra foi o conjunto de taxas de cobertura encontrado depois da aplicação das propriedades conforme Tabela 15, colunas 03 e 09 respectivamente. Antes da aplicação do teste t as amostras foram avaliadas pelo teste Shapiro-wilk que mensura se as amostras possuem distribuição normal. Ambas as amostras apresentaram distribuição normal dos resultados com amostra 1 *p-value* = 0,1312 e amostra 2 *p-value* = 0.179, possibilitando a utilização do teste t para avaliação da hipótese do estudo.

O resultado da aplicação do Teste T encontrou o *p-value* = 0.003663 que aponta uma diferença significativa entre as amostras. Esse resultado indica que a melhoria alcançada na cobertura do teste com a aplicação da abordagem Nonexec foi estatisticamente significativa, que baseado no experimento desenvolvido sob o *benchmark* apresentado anteriormente. Dessa forma, é possível refutar a hipótese $H_0$ e aceitar a hipótese alternativa $H_1$ a qual define que "*A*

*abordagem Nonexec oferece suporte significativo ao testador durante o processo de mitigação de requisitos de teste não executáveis*". O boxplot apresentado na Figura 13 mostra que a mediana das coberturas era aproximadamente 20% e passou para 60% com ajuda da abordagem Nonexec.

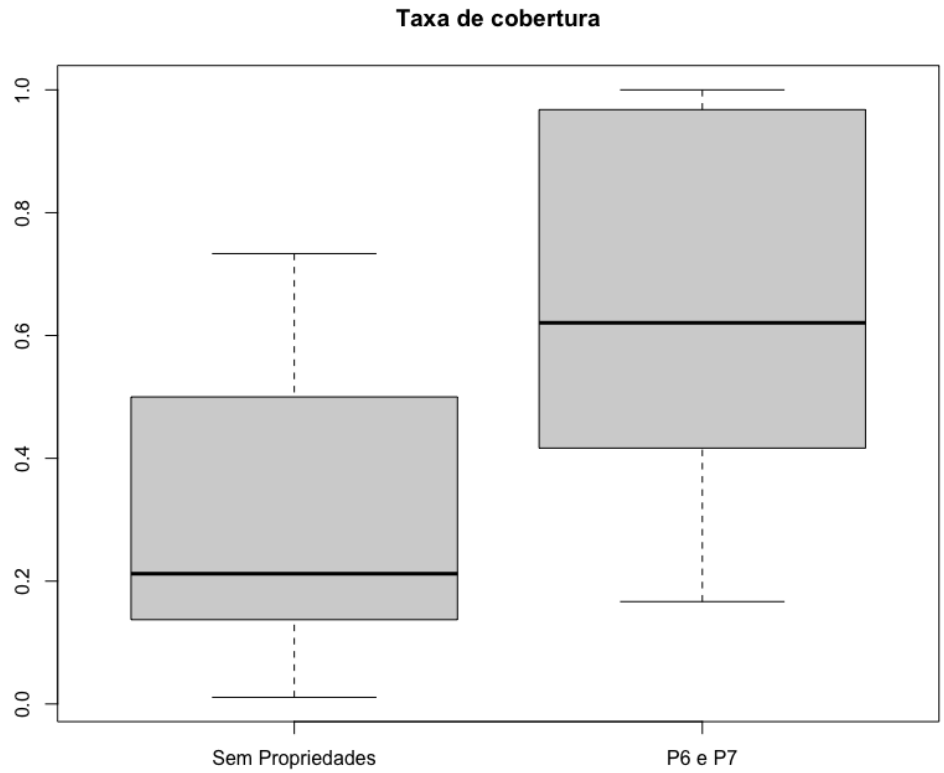


Figura 13 – Resumo das taxas de cobertura encontradas para o critério de teste *All-sync*.

### 5.3.6 Ameaças à Validade

As seguintes ameaças externas e internas são reconhecidas e podem ter afetado a validade dos resultados.

Uma possível ameaça à validade externa refere-se à generalização dos resultados. Essa ameaça foi mitigada com a utilização de amostras utilizadas em outros experimentos do grupo de pesquisa. O uso do conjunto de dados conforme foi utilizado reduz essa ameaça. Neste estudo foram utilizados programas escritos em Java, portanto não é possível garantir a generalização dos resultados para outras linguagens de programação. Outra ameaça à validade externa refere-se ao tamanho dos programas utilizados no experimento, contudo, os programas diferem em nível de complexidade indo do mais básico até programas que exigem custo computacional elevado. É evidente que a complexidade do conjunto de dados pode ser aumentada à medida que novas rodadas experimentais sejam executadas. Para mitigar essas ameaças, foram usados 13 programas de diferentes funcionalidades e complexidades. Além disso, o comportamento dos programas é suficientemente distinto livre de viés e garante representatividade significativa para o estudo.

As ameaças de validade interna incluem a construção de um estudo exploratório baseado em propriedades previamente utilizadas manualmente, mas automatizadas na sequência. Outra ameaça à validade interna está no processo desenvolvido usado para realizar o estudo exploratório, para mitigar essa ameaça, o processo foi desenvolvido após estudos empíricos que exploram esses conceitos (Baral; Offutt, 2020; Koc *et al.*, 2019; Zhu; Panichella; Zaidman, 2018; RODRIGUES; BRANCHER, 2019) e o processo exploratório foi baseado no estudo (SHULL *et al.*, 2001) que define diretrizes para o desenvolvimento de estudos experimentais.

### 5.3.7 Conclusões

A abordagem Nonexec foi utilizada para identificar automaticamente as propriedades no código-fonte que revelam requisitos de teste não executáveis, possivelmente não executáveis e dificilmente executáveis. Inicialmente, um catálogo de propriedades para identificar requisitos de teste não executáveis foi utilizado. Em seguida, adotou-se um processo que não utiliza dados de entrada para minimizar o custo de geração de dados de entrada. Além disso, um analisador foi incorporado para automatizar a aplicação do catálogo de propriedades com a possibilidade de ser extensível, fato que irá permitir a Fi-paths e a Valipar possa ser utilizada para outros paradigmas de programação.

Acredita-se que a abordagem pode beneficiar o testador em seu trabalho diário. Ao aplicar uma ferramenta de teste de cobertura, requisitos não executáveis atrapalham o processo de cobertura e dificultam a decisão se a cobertura máxima não for atingida. Portanto, ter informações sobre requisitos não executáveis, possivelmente não executáveis e dificilmente executáveis podem ajudar o testador a decidir quando interromper o processo de teste ou quais requisitos priorizar. Embora fosse esperado que o custo computacional para identificar as propriedades fosse alto, isso não foi confirmado.

O processo de identificação das propriedades por si só não foi suficiente para decidir se um requisito é não executável ou não, esta etapa precisa ser aprimorada devido ao alto nível de complexidade necessário para decidir se um requisito possivelmente não executável é não executável. Com base nos resultados, o testador é necessário para melhorar o processo de identificação de requisitos de teste não executáveis, por isso uma opção favorável é usar *insights* humanos para auxiliar o processo de tomada de decisão na identificação de requisitos de teste não executáveis. A evolução deste trabalho analisará a escalabilidade da abordagem, considerando programas mais complexos e o impacto de requisitos possivelmente não executáveis durante a aplicação de testes estruturais.

A abordagem Nonexec foi modelada para auxiliar o processo de teste estrutural mitigar o problema da não executabilidade. A abordagem foi instanciada em duas ferramentas possibilitando sua aplicação no teste de programas sequenciais concorrentes. A abordagem possibilitou a utilização de propriedades exclusivas para o contexto de programas concorrentes analisando suas sincronizações. A utilização da Nonexec possibilitou a identificação requisitos de teste

dificilmente executáveis, possivelmente não executáveis e não executáveis. Como resultado duas novas classes de requisitos foram identificadas durante a análise experimental denominadas requisitos dificilmente executáveis e requisitos possivelmente não executáveis.

## 5.4 Considerações finais

O desenvolvimento de programas sequenciais e concorrentes é fundamental para operacionalidade de sistemas de informação, entretanto a atividade de teste ainda se depara com limitações a serem mitigadas, uma delas é o problema da não-executabilidade de requisitos de teste. Dentre os artifícios utilizados na atividade de teste como, a geração automática de dados de teste que visa facilitar o processo de seleção de entradas e otimizar a verificação dos produtos de software é penalizada pelo mesmo problema.

Este capítulo apresentou a abordagem Nonexec que tem como objetivo mitigar o problema da não-executabilidade durante o teste estrutural, por meio da aplicação de propriedades indicadoras de regiões de código que são não-executáveis, possivelmente não-executáveis e dificilmente executáveis. Estas regiões refletem diretamente no comportamento dos requisitos de teste, desfavorecendo a atividade de teste e aumentando seu custo.

Neste capítulo foram apresentados a análise de dois estudos experimentais que estudaram a aplicação das propriedades catalogadas e a avaliou a abordagem Nonexec. Ainda, neste capítulo foram apresentados o projeto experimental do estudo e os resultados obtidos por meio da condução dos experimentos. Algumas descobertas foram feitas e novos conjuntos foram definidos para classificar requisitos de teste.

Considerando os resultados apresentados, foi possível observar contribuições significativas da abordagem Nonexec, a qual oferece suporte ao testador no processo de mitigar o problema da não executabilidade na atividade de teste estrutural. Direciona o testador à regiões do código que possam apresentar o problema da não executabilidade. Apesar dos resultados serem positivos a abordagem ainda pode ser melhorada incluindo um processo de decisão aprimorado que carece ser, ainda, modelado e desenvolvido.

No próximo capítulo serão apresentadas as conclusões encontradas com o desenvolvido teórico e prático desta tese.

CAPÍTULO

6

# CONCLUSÕES E TRABALHOS FUTUROS

## 6.1 Caracterização da Contribuição

Programas sequenciais e concorrentes são cada vez mais relevantes na vida humana. Essas aplicações melhoraram significativamente o desempenho das tarefas, das mais simples às mais complicadas. A qualidade desses produtos afeta diretamente como os usuários finais percebem o desempenho e a assertividade na execução das tarefas. Diante disso, é primordial que a atividade de teste seja executada de forma constante e a baixo custo durante todo o ciclo de desenvolvido do software. Técnicas diversas são utilizadas como por exemplo teste funcional, teste estrutural e teste de mutação. O problema da não executabilidade afeta diretamente o desempenho do teste estrutural e aumenta o esforço cognitivo do testador, além de prejudicar o desempenho da geração automática de dados de teste.

O estudo desenvolvido nesta tese empenhou-se em mitigar o problema da não executabilidade em teste estrutural oferecendo uma abordagem a ser utilizada em programas concorrentes, sob a seguinte hipótese **É possível mitigar o problema da não executabilidade por meio da utilização de um catálogo de propriedades de identificação de requisitos não executáveis**.

De forma geral a pesquisa desenvolvida resultou na construção de uma abordagem denominada Nonexec que realiza a identificação de propriedades baseadas no código fonte capazes de revelar requisitos de teste não executáveis, possivelmente executáveis e dificilmente executáveis. Desta forma, é possível responder a questão de pesquisa principal.

"É possível mitigar o problema da não executabilidade através do uso de um catálogo de propriedades de identificação de requisitos não executáveis?"Sim é possível utilizar um catálogo de propriedades de identificação de requisitos não executáveis e mitigar o problema da não executabilidade identificando regiões do código fonte geradoras de requisitos de teste não executáveis, possivelmente não executáveis e dificilmente executáveis.

Com base nos resultados da análise experimental é possível responder as questões de pesquisa secundárias:

1. **Quais propriedades para identificação de requisitos não executáveis podem ser definidas?**

   - Propriedade 1 (P1) - Congelamento de variável. Seção 4.2.2.
   - Propriedade 2 (P2) - Predicados Opostos e Predicados Iguais. Seção 4.2.3.
   - Propriedade 3 (P3) - Correlação entre declarações condicionais. Seção 4.2.4.
   - Propriedade 4 (P4) - Mudança de Definição no caminho. Seção 4.2.5.
   - Propriedade 5 (P5) - Código morto. Seção 4.2.6.
   - Propriedade 6 (P6) - Comunicação entre processos *Workers*. Seção 4.2.7.
   - Propriedade 7 (P7) - Arestas de comunicação dificilmente executáveis. Seção 4.2.8.

2. **Quais propriedades para identificação de requisitos não executáveis podem ser aplicadas de forma automática?** Todas a propriedades catalogadas podem ser utilizadas de forma automatizada. A propriedades Atribuição de um valor constante a uma variável (P1), Predicados Opostos e Predicados Iguais (P2), Correlação entre declarações condicionais (P3) e Mudança de Definição no caminho (P4) detectam requisitos possivelmente não executáveis. A propriedades Predicados Opostos e Predicados Iguais (P2) e Comunicação entre processos *Workers* (P6) detectam requisitos de teste não executáveis. E, por fim, a propriedade Arestas de comunicação dificilmente executáveis (P7) detecta requisitos dificilmente executáveis.

3. **Quais são os benefícios alcançados com a aplicação das propriedades de identificação de caminhos não executáveis na atividade de teste estrutural?** A identificação das propriedades no código fonte possibilita a classificação dos requisitos de teste requeridos pelos critérios de teste estrutural. Esse fato oferece mais informações ao testador referente à dificuldade de cobertura de alguns requisitos. A classificação indica ao testador quais requisitos necessitam de atenção e quais serão desconsiderados (os não executáveis) da atividade de teste. Além disso, a utilização das propriedades reduz a atividade cognitiva do testador pois indica onde o problema da não executabilidade tem a possibilidade de ocorrer e pode ser utilizado para reduzir o custo durante a geração automática de dados de teste.

4. **Quais propriedades de identificação de requisitos são viáveis para apoiar o problema da não-executabilidade em programas concorrentes?** Para o contexto de programas concorrentes o estudo identificou e definiu três propriedades para mitigar o problema da não executabilidade, sendo elas a Comunicação entre processos *Workers* (P6), Arestas de comunicação dificilmente executáveis (P7), as duas primeiras relacionadas a programas com passagem de mensagem e a última relacionada a programas de memória compartilhada.

## 6.2 Contribuições Principais

O principal objetivo desta tese de doutorado foi a definição da abordagem Nonexec e o catálogo de propriedades de identificação de requisitos de teste não executáveis, a qual tem por finalidade contribuir com teste estrutural de software sequencial e concorrente sinalizando regiões geradoras de requisitos de teste não executáveis, possivelmente não executáveis e dificilmente executáveis. Essa abordagem auxilia o testador de software a encontrar requisitos de teste que podem apresentar o problema da não executabilidade e assim direcionar seus esforços durante a análise.

De forma secundária, a abordagem pode oferecer suporte à geração de dados de teste, ao indicar regiões problemáticas e possibilitar a definição de um critério de parada para o algoritmo de geração de dados. Além disso, poderá auxiliar o rastreamento de *code smell* por meio da identificação de regiões do código que apresentam o comportamento das propriedades catalogadas, uma vez que algumas regiões identificadas surgem devido a problemas na lógica de programação.

Além de contribuições conceituais, as atividades desenvolvidas nesta tese permitiram colaborar com as atividades que o mestrando Ricardo Chagas, matriculado no programa CCMC, está realizando para a construção de sua dissertação de mestrado intitulada "Identificação automatizada de caminhos não-executáveis para programas concorrentes com passagem de mensagens". De forma complementar, as principais contribuições deste trabalho são descritas a seguir:

- Construção de um catálogo extensível de propriedades de identificação de requisitos de teste não executáveis;
- Definição da abordagem Nonexec;
- Definição da ferramenta Fi-path;
- Incorporação da ferramenta Valipar;
- Avaliação experimental da Abordagem Nonexec e catálogo de propriedades;
- Definição de novos termos relacionados ao problema da não executabilidade: requisito de teste **possivelmente não executável** e requisito de teste **dificilmente executável**.

### 6.2.1 Publicações resultantes

Durante o período de construção da tese foram produzidos e publicados os seguintes estudos:

**a)** Publicações relacionadas à teste:

- **Choma Neto, João**, Chagas, R., Mori, A., Vilela, R., Colanzi, T., Souza, S. (2022, October). A Strategy to Support the Infeasible Test Requirements Identification. In Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing (pp. 29-38). DOI: https://doi.org/10.1145/3559744.3559748.

- **Choma Neto, João;** Mori, Allan; Vilela, Ricardo; Colanzi, Thelma. and Souza, Simone Do Rocio Senger de. How to Identify the Infeasible Test Requirements using Static Analyse? An Exploratory Study. DOI: 10.5220/0010497107820789. In Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 1, pages 782-789 ISBN: 978-989-758-509-8. Copyright c 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved.

- **Choma Neto, João.** Automatic support for the identification of infeasible testing requirements. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020), 2020. Association for Computing Machinery, New York, NY, USA, 587–591. DOI: https://doi.org/10.1145/3395363.3402646.

**b)** Produções relacionadas à tese, submetidas:

- **Choma Neto, João**; Chagas, Ricardo; Vilela, Ricardo F.; Colanzi, Thelma E.; Souza, Paulo S.L.; Souza, Simone R.S. Mitigating the problem of non-executability during the testing of concurrent programs. Concurrency and Computation: Practice and Experience. Submetido.

**c)** Atividades como estágio PAE e apoio a outros membros do grupo de pesquisa geram publicações não relacionadas à tese geradas durante o doutorado:

- Ferreira Vilela, R., **Choma Neto, João**, Santiago Costa Pinto, V. H., Lopes de Souza, P. S., do Rocio Senger de Souza, S. (2023). Bio-inspired optimization to support the test data generation of concurrent software. Concurrency and Computation: Practice and Experience, 35(2), e7489. DOI: https://doi.org/10.1002/cpe.7489.

- **Choma Neto, João;** Bento, Luiz Henrique; OliveiraJr, Edson; Souza, Simone Do Rocio Senger de. Are we teaching UML according to what IT companies need? A survey on the São Carlos-SP region. In: SIMPÓSIO BRASILEIRO DE EDUCAÇÃO EM COMPUTAÇÃO (EDUCOMP), 1., 2021, On-line. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021. p. 34-43. DOI: https://doi.org/10.5753/educomp.2021.14469.

- Souza, Simone Do Rocio Senger de; **Choma Neto, João;** PASCHOAL, Leo Natan; Hernandes, Elis. Ensino Remoto Emergencial de Engenharia de Software com PBL: um relato de experiência. In: WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO(WEI),

29., 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021. p. 31-40. ISSN 2595-6175. DOI: https://doi.org/10.5753/wei.2021.15894.

- OliveiraJr, Edson; Colanzi, Thelma; Amaral, Aline; Cordeiro, André Felipe; **Choma Neto, João;** Souza, Simone Do Rocio Senger de. Ensino, Aprendizagem e Uso Profissional da UML em Maringá e Região. In: WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO (WEI), 29., 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021. p. 328-337. ISSN 2595-6175. DOI: https://doi.org/10.5753/wei.2021.15924.

## 6.3 Trabalhos Futuros

Durante a construção de conhecimento relacionada a esta tese de doutorado, foi possível identificar novas lacunas de pesquisa que, se solucionadas, tem possibilidade de contribuir e melhorar diretamente a abordagem Nonexec e, consequentemente, contribuir com a atividade de teste estrutural. Abaixo são apresentados os possíveis trabalhos futuros que podem ser originados com base nas contribuições trazidas neste tese.

- Definição de novas propriedades para programas concorrentes. Outros modelos de comunicação utilizados na programação concorrente serão analisados em busca de novas propriedades e, com base nos conceitos de programação concorrente será possível definir novas propriedades de identificação;
- Utilizar a abordagem Nonexec para apoiar a geração da dados de teste automática por meio da identificação de requisitos de teste não executáveis e, a definição de um critério de parada baseado nas propriedades de identificação;
- Utilização de um algoritmo de recomendação para mensurar a probabilidade de um requisito possivelmente não executável se tonar não executável e predizer o comportamento de requisitos;
- Construção de um estudo experimental, com a participação de testadores humanos, para avaliar quais requisitos classificados como possivelmente não executáveis são não executáveis e analisar o comportamento de tais requisitos com base na perspectiva humana;
- Replicação dos estudos experimentais considerando programas de maior complexidade;
- Evolução da abordagem Nonexec com a utilização de técnicas de priorização de requisitos para refinar o conjunto de requisitos possivelmente não executáveis e diminuir ainda mais o trabalho do testador;
- Utilizar a abordagem Nonexec para apoiar o rastreamento de *code smells* por meio da identificação de regiões de código que possuem características relacionadas as propriedades de identificação e, como consequência possibilitar a refatoração dessas regiões.

## 6.4 Limitações

Foram encontradas algumas limitações que de alguma forma limitaram ou atrasaram o desenvolvimento da proposta:

- A identificação de uma propriedade dependente do fluxo de dados não garante a identificação completa de um requisito não executável, sendo necessário uma nova classificação para os requisitos que apresentavam o comportamento descrito nas propriedades;
- Não foram identificadas ferramentas que auxiliassem na avaliação do critério Todos-caminhos sendo necessário a aplicação de critérios menos sensíveis as propriedades definidas no catálogo;
- Não haviam ferramentas que oferecessem o suporte identificação das propriedades catalogadas nem no contexto de programas sequenciais e quanto no contexto de programas concorrentes. Este fato possibilitou a criação da ferramenta Fi-paths e a extensão da ferramenta Valipar.

# REFERÊNCIAS

610.12-1990, I. S. IEEE standard glossary of software engineering terminology - std 610.12-1990. 1990. Citado na página 31.

Afzal, A.; Goues, C. L.; Hilton, M.; Timperley, C. S. A study on challenges of testing robotic systems. In: **2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)**. [S.l.: s.n.], 2020. p. 96–107. Citado na página 91.

AHMED, A. H.; TAHA, D. B. Generation of optimal testing paths using anti–ant colony algorithm. **International Journal of Computer Applications**, v. 975, p. 8887. Citado na página 59.

AISSAT, R.; GAUDEL, M.-C.; VOISIN, F.; WOLFF, B. A method for pruning infeasible paths via graph transformations and symbolic execution. In: IEEE. **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. [S.l.], 2016. p. 144–151. Citado na página 59.

AISSAT, R.; GAUDEL, M.-C.; VOISIN, F.; WOLFF, B. **Pruning infeasible paths via graph transformations and symbolic execution: a method and a tool**. Tese (Doutorado) — Laboratoire de Recherche en Informatique [LRI], UMR 8623, Bâtiments 650-660 . . . , 2016. Citado na página 59.

AISSAT, R.; VOISIN, F.; WOLFF, B. Infeasible paths elimination by symbolic execution techniques. In: SPRINGER. **International Conference on Interactive Theorem Proving**. [S.l.], 2016. p. 36–51. Citado na página 59.

ALMASI, G. S.; GOTTLIEB, A. Highly parallel computing. The Benjamin Cummings Publishing Company, 2 ed., 1994. Citado nas páginas 40 e 41.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016. Citado nas páginas 23, 31, 32 e 33.

ANDALAM, S.; ROOP, P. S.; GIRAULT, A. Pruning infeasible paths for tight wcrt analysis of synchronous programs. In: IEEE. **2011 Design, Automation & Test in Europe**. [S.l.], 2011. p. 1–6. Citado na página 58.

ANSARI, G. A. Detection of infeasible paths in software testing using uml application to gold vending machine. **Int. J. Edu. Manage. Eng.**, v. 7, n. 4, p. 21–28, 2017. Citado na página 59.

ARLT, S.; SCHÄF, M. Joogie: Infeasible code detection for java. In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 2012. p. 767–773. Citado na página 58.

AVGERINOS, T.; REBERT, A.; CHA, S. K.; BRUMLEY, D. Enhancing symbolic execution with veritesting. In: **Proceedings of the 36th International Conference on Software Engineering**. [S.l.: s.n.], 2014. p. 1083–1094. Citado nas páginas 39 e 40.

BALDONI, R.; COPPA, E.; D'ELIA, D. C.; DEMETRESCU, C.; FINOCCHI, I. A survey of symbolic execution techniques. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 3, p. 1–39, 2018. Citado nas páginas 39 e 40.

Baral, K.; Offutt, J. An empirical analysis of blind tests. In: **2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)**. [S.l.: s.n.], 2020. p. 254–262. Citado nas páginas 91, 92, 98, 99, 100 e 113.

BARHOUSH, B.; ALSMADI, I. Infeasible paths detection using static analysis. **Ijj.Acm.Org**, II, n. Iii, 2013. Disponível em: <http://ijj.acm.org/volumes/volume2/issue3/ijjvol2no34.pdf>. Citado nas páginas 24, 25, 35, 38, 58, 65, 70 e 76.

BARR, E. T.; HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. The oracle problem in software testing: A survey. **IEEE transactions on software engineering**, IEEE, v. 41, n. 5, p. 507–525, 2015. Citado na página 35.

BATISTA, R. N. **Otimizando o teste estrutural de programas concorrentes : uma abordagem determinística e paralela.** Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2015. Citado na página 87.

BERTOLINI, C.; SCHÄF, M.; SCHWEITZER, P. Infeasible code detection. In: SPRINGER. **International Conference on Verified Software: Tools, Theories, Experiments**. [S.l.], 2012. p. 310–325. Citado na página 58.

BODÍK, R.; GUPTA, R.; SOFFA, M. L. Refining data flow information using infeasible paths. **ACM SIGSOFT Software Engineering Notes**, v. 22, n. 6, p. 361–377, 1997. ISSN 01635948. Disponível em: <http://portal.acm.org/citation.cfm?doid=267896.267921>. Citado nas páginas 36, 56, 62 e 74.

BROWN, J. Practical applications of automated software tools. In: **Proceedings of the Western Electronic Show and Convention (WESCON) Los Angeles California September**. [S.l.: s.n.], 1972. p. 19–22. Citado na página 53.

BRUNETON, E. Asm 3.0 a java bytecode engineering library. **URL: http://download. forge. objectweb. org/asm/asmguide. pdf**, 2007. Citado na página 84.

BUENO, P. M. S.; JINO, M. Identification of potentially infeasible program paths by monitoring the search for test data. **Proceedings ASE 2000: 15th IEEE International Conference on Automated Software Engineering**, p. 209–218, 2000. ISSN 1938-4300. Citado nas páginas 24, 25, 26, 35, 36, 38, 49, 57, 60, 66 e 70.

____. Automatic test data generation for program paths using genetic algorithms. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific, v. 12, n. 06, p. 691–709, 2002. Citado nas páginas 57 e 66.

CHAWLA, P.; CHANA, I.; RANA, A. A novel strategy for automatic test data generation using soft computing technique. **Frontiers of Computer Science**, v. 9, n. 3, p. 346–363, 2015. Cited By 0. Disponível em: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84938208965&partnerID=40&md5=6b7065f7903d0a046c17613f79b6ecd1>. Citado na página 26.

CHERAGHI, A.; HASHEMINEJAD, S. M. H. Automatic detection of infeasible all-du paths in the data flow test using an evolutionary approach. In: IEEE. **2020 6th Iranian Conference on Signal Processing and Intelligent Systems (ICSPIS)**. [S.l.], 2020. p. 1–6. Citado na página 59.

CHOMANETO, J. Automatic support for the identification of infeasible testing requirements. In: **Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2020. p. 587–591. Citado na página 59.

CHOMANETO, J.; MORI, A.; VILELA, R. F.; COLANZI, T. E.; SOUZA, S. R. How to identify the infeasible test requirements using static analyse? an exploratory study. In: **ICEIS (1)**. [S.l.: s.n.], 2021. p. 782–789. Citado nas páginas 59, 82 e 83.

CHRIST, J.; HOENICKE, J.; SCHÄF, M. Towards bounded infeasible code detection. **arXiv preprint arXiv:1205.6527**, 2012. Citado na página 58.

CLARKE, L. A. A system to generate test data and symbolically execute programs. **IEEE Transactions on software engineering**, IEEE, n. 3, p. 215–222, 1976. Citado nas páginas 24, 35, 36 e 69.

DAHIYA, S. S.; CHHABRA, J. K.; KUMAR, S. **PSO Based Pseudo Dynamic Method for Automated Test Case Generation Using Interpreter**. [S.l.: s.n.], 2011. 147 p. ISSN 0302-9743. ISBN 9783642215148. Citado nas páginas 39, 58 e 66.

DELAHAYE, M. Ipeg: Utilizing infeasibility. In: IEEE. **2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.], 2011. p. 318–319. Citado na página 58.

DELAHAYE, M.; BOTELLA, B.; GOTLIEB, A. Explanation-based generalization of infeasible path. **ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation**, p. 215–224, 2010. Citado na página 58.

____. Infeasible path generalization in dynamic symbolic execution. **Information and Software Technology**, v. 58, p. 403–418, 2015. ISSN 09505849. Citado nas páginas 24, 26, 59 e 69.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017. Citado nas páginas 32, 33, 35, 36, 40, 41, 42 e 44.

DIAZ, E.; TUYA, J.; BLANCO, R. Automated software testing using a metaheuristic technique based on tabu search. **18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.**, p. 310–313, 2003. ISSN 1938-4300. Disponível em: <http://ieeexplore.ieee.org/document/1240327/>. Citado nas páginas 57 e 66.

DING, S.; TAN, H. B. K. **Detection of infeasible paths: Approaches and challenges**. [S.l.], 2012. 64–78 p. Citado nas páginas 27, 54, 55 e 57.

DING, S.; TAN, H. B. K.; SHAR, L. K. Mining patterns of unsatisfiable constraints to detect infeasible paths. **Proceedings - 10th International Workshop on Automation of Software Test, AST 2015**, p. 65–69, 2015. Citado na página 59.

DING, S.; ZHANG, H.; TAN, H. B. K. Detecting infeasible branches based on code patterns. **2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings**, IEEE, p. 74–83, 2014. Citado nas páginas 58 e 62.

DIONNE, C.; FEELEY, M.; DESBIEN, J. A taxonomy of distributed debuggers based on execution replay. In: **PDPTA**. [S.l.: s.n.], 1996. p. 203–214. Citado nas páginas 41 e 44.

DOURADO, G. G. M. Contribuindo para a avaliação do teste de programas concorrentes: uma abordagem usando benchmarks. In: . [S.l.: s.n.], 2015. Citado na página 102.

DU, Q.; DONG, X. An improved algorithm for basis path testing. **BMEI 2011 - Proceedings 2011 International Conference on Business Management and Electronic Information**, IEEE, v. 3, p. 175–178, 2011. Citado nas páginas 25, 58 e 60.

DU, Q.; LU, Q. Generation model of basis-paths based on variable-dependence and interval-arithmetic. **Proceedings of 2013 3rd International Conference on Computer Science and Network Technology, ICCSNT 2013**, IEEE, p. 270–274, 2014. Citado nas páginas 58 e 60.

DU, X.; LIU, J.; HE, H. Generating feasible paths under c-zot coverage from efsm. In: **International Conference on Frontiers of Electronics, Information and Computation Technologies**. [S.l.: s.n.], 2021. p. 1–8. Citado na página 59.

ELER, M. M.; ENDO, A. T.; DURELLI, V.; PROCÓPIO-PR, C. Covering user-defined data-flow test requirements using symbolic execution. **Proceedings of the Thirteenth Brazilian Symposium On Software Quality (SBQS)**, p. 16–30, 2014. Citado na página 58.

FAZLI, E.; AFSHARCHI, M. A time and space-efficient compositional method for prime and test paths generation. **IEEE Access**, IEEE, v. 7, p. 134399–134410, 2019. Citado na página 59.

FORGÁCS, I.; BERTOLINO, A. Feasible test path selection by principal slicing. **Contract**, p. 378–394, 1997. ISSN 0163-5948. Disponível em: <http://www.springerlink.com/index/KU82283L26J6R137.pdf>. Citado nas páginas 56 e 61.

FRANKL, P. G. **The Use of Data Flow Information for the Selection and Evaluation of Software Test Data**. Tese (Doutorado), New York, NY, USA, 1987. AAI8801533. Citado nas páginas 24, 31, 35, 36, 56, 69 e 70.

FRASER, G.; ARCURI, A.; MCMINN, P. A memetic algorithm for whole test suite generation. **Journal of Systems and Software**, Elsevier, v. 103, p. 311–327, 2015. Citado na página 26.

GHIDUK, A. S. Automatic generation of basis test paths using variable length genetic algorithm. **Information Processing Letters**, Elsevier B.V., v. 114, n. 6, p. 304–316, 2014. ISSN 00200190. Disponível em: <http://dx.doi.org/10.1016/j.ipl.2014.01.009>. Citado nas páginas 26, 59, 61 e 67.

GIRGIS, M. R.; EL-NASHAR, A. I.; ELSIFY, A. M. Automatic classification of program paths feasibility using active learning. **Transactions on Machine Learning and Artificial Intelligence**, v. 6, n. 6, p. 35, 2019. Citado na página 59.

GOLDBERG, A.; WANG, T. C.; ZIMMERMAN, D. Applications of feasible path analysis to program testing. **Proceedings of the 1994 international symposium on Software testing and analysis - ISSTA '94**, p. 80–94, 1994. ISSN 09277757. Disponível em: <http://portal.acm.org/citation.cfm?doid=186258.186523>. Citado na página 56.

GONG, D.; TIAN, T.; YAO, X. Grouping target paths for evolutionary generation of test data in parallel. **Journal of Systems and Software**, Elsevier Inc., v. 85, n. 11, p. 2531–2540, 2012. ISSN 01641212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2012.05.071>. Citado nas páginas 58 e 66.

GONG, D.; YAO, X. Automatic detection of infeasible paths in software testing. **IET Software**, v. 4, n. 5, p. 361, 2010. ISSN 17518806. Disponível em: <http://digital-library.theiet.org/content/journals/10.1049/iet-sen.2009.0092>. Citado nas páginas 26, 58 e 61.

GONG, H.; ZHANG, Y.; XING, Y.; JIA, W. Detecting interprocedural infeasible paths via symbolic propagation and dataflow analysis. In: IEEE. **2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)**. [S.l.], 2019. p. 282–285. Citado na página 59.

GRAMA, A.; GUPTA, A.; KARYPIS, G.; KUMAR, V. Principles of parallel algorithm design. **Introduction to Parallel Computing, 2nd ed. Addison Wesley, Harlow**, 2003. Citado na página 78.

HARMAN, M.; MANSOURI, S.; ZHANG, Y. Search based software engineering: Trends, techniques and applications. **Journal ACM Computing Surveys (CSUR)**, v. 45, n. 1, p. 434–442, 2009. ISSN 0360-0300. Citado na página 23.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. **Department of Computer Science, King's College London, Tech. Rep. TR-09-03**, 2009. Citado na página 54.

HEDLEY, D.; HENNELL, M. A. The causes and effects of infeasible paths in computer programs. p. 259–266, 1985. ISSN 02705257. Disponível em: <http://dl.acm.org/citation.cfm?id=319568.319648>. Citado nas páginas 39, 52, 56, 62 e 71.

HERMADI, I.; AHMED, M. A. Genetic algorithm based test data generator. **The 2003 Congress on Evolutionary Computation, 2003. CEC '03.**, n. May, p. 184, 2003. Citado nas páginas 57 e 66.

HERMADI, I.; LOKAN, C.; SARKER, R. Dynamic stopping criteria for search-based test data generation for path testing. **Information and Software Technology**, Elsevier B.V., v. 56, n. 4, p. 395–407, 2014. ISSN 09505849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2014.01.001>. Citado nas páginas 26, 34, 59, 60 e 66.

HWANG, G.-H.; TAI, K.-C.; HUANG, T.-L. Reachability testing: An approach to testing concurrent software. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific, v. 5, n. 04, p. 493–510, 1995. Citado na página 45.

INDUMATHI, C.; AJINA, A. Generating feasible path between path testing and data flow testing. In: **Evolutionary Computing and Mobile Sustainable Networks**. [S.l.]: Springer, 2021. p. 325–335. Citado na página 59.

JAFFAR, J.; NAVAS, J. A.; SANTOSA, A. E. A path-sensitive control flow graph. Citeseer. Citado na página 58.

JASPER, R.; BRENNAN, M.; WILLIAMSON, K.; CURRIER, B.; ZIMMERMAN, D. Test data generation and feasible path analysis. In: **Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis**. [S.l.: s.n.], 1994. p. 95–107. Citado na página 56.

JAYARAMAN, D.; TRAGOUDAS, S. Performance validation through implicit removal of infeasible paths of the behavioral description. **Proceedings - International Symposium on Quality Electronic Design, ISQED**, IEEE, p. 552–557, 2013. ISSN 19483287. Citado na página 58.

JIANG, S.; WANG, H.; ZHANG, Y.; XUE, M.; QIAN, J.; ZHANG, M. An approach for detecting infeasible paths based on a smt solver. **IEEE Access**, IEEE, v. 7, p. 69058–69069, 2019. Citado na página 59.

JIANG, S.; ZHANG, Y.; YI, D. Test data generation approach for basis path coverage. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 37, n. 3, p. 1–7, 2012. Citado na página 58.

KEMPF, K.; KOLLMANN, S.; POLLEX, V.; SLOMKA, F. Relaxing event densities by exploiting infeasible paths in control flow graphs. In: CITESEER. **RTNS**. [S.l.], 2011. p. 75–84. Citado na página 58.

KEMPF, K.; SLOMKA, F. Direct handling of infeasible paths in the event dependency analysis. In: IEEE. **2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.], 2014. p. 1–10. Citado na página 58.

Koc, U.; Wei, S.; Foster, J. S.; Carpuat, M.; Porter, A. A. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In: **2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)**. [S.l.: s.n.], 2019. p. 288–299. Citado nas páginas 91, 92, 98, 99, 100 e 113.

KOJIMA, H.; KAKUDA, Y.; TAKAHASHI, J.; OHTA, T. A model for concurrent states and its coverage criteria. In: **International Symposium on Autonomous Decentralized Systems (ISADS 2009)**. Athens, Greece: [s.n.], 2009. p. 1 –6. Citado na página 25.

KUMAR, T. B.; HARISH, N.; KUMAR, V. S. An catholic and enhanced study on basis path testing to avoid infeasible paths in cfg. In: SPRINGER. **International Conference on Computing and Communication Systems**. [S.l.], 2011. p. 386–395. Citado na página 58.

KUNDU, D.; SARMA, M.; SAMANTA, D. A uml model-based approach to detect infeasible paths. **Journal of Systems and Software**, Elsevier Ltd., v. 107, p. 71–92, 2015. ISSN 01641212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2015.05.007>. Citado nas páginas 25 e 38.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. In: **Concurrency: the Works of Leslie Lamport**. [S.l.: s.n.], 2019. p. 179–196. Citado na página 79.

LEE, W.; QIN, X. **Lecture Notes in Computer Science**. [S.l.: s.n.], 2003. 73–93 p. ISSN 0302-9743. ISBN 3540408789. Citado nas páginas 25, 38 e 57.

LEI, Y.; CARVER, R. Reachability testing of semaphore-based programs. In: IEEE. **Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.** [S.l.], 2004. p. 312–317. Citado na página 45.

____. A new algorithm for reachability testing of concurrent programs. In: IEEE. **Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on**. [S.l.], 2005. p. 10–pp. Citado na página 45.

LEI, Y.; CARVER, R. H. Reachability testing of concurrent programs. **IEEE Transactions on Software Engineering**, IEEE, v. 32, n. 6, p. 382–403, 2006. Citado nas páginas 25, 26 e 57.

LU, G.; MIAO, H. Feasibility analysis of the efsm transition path combining slicing with theorem proving. In: IEEE. **2013 International Symposium on Theoretical Aspects of Software Engineering**. [S.l.], 2013. p. 153–156. Citado na página 58.

MAHMOOD, S. A systematic review of automated test data generation techniques. **School of Engineering, Blekinge Institute of Technology Box**, v. 520, n. October, p. 65, 2007. ISSN 0721-9121. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.3335{&}rep=rep1{&}ty>. Citado na página 54.

MALDONADO, J. C. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. Biblioteca Digital da Unicamp, 1991. Citado na página 24.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. d. R. S. de; DELAMARO, M. E. **Aspectos teóricos e empíricos de teste de cobertura de software**. [S.l.]: ICMSC-USP, 1998. Citado nas páginas 24 e 31.

MALEVRIS, N. A path generation method for testing lcsajs that restrains infeasible paths. **Information and Software Technology**, v. 37, n. 8, p. 435–441, 1995. ISSN 09505849. Citado na página 56.

MALEVRIS, N.; YATES, D. F.; VEEVERS, A. Predictive metric for likely feasibility of program paths. **Information and Software Technology**, v. 32, n. 2, p. 115–118, 1990. ISSN 09505849. Citado na página 56.

MARASHDIH, A. W.; ZAABA, Z. F. Infeasible paths in static analysis: Problems and challenges. **AIP Conference Proceedings**, v. 2016, n. September, 2018. ISSN 15517616. Citado nas páginas 25, 38, 39 e 54.

MARASHDIH, A. W.; ZAABA, Z. F.; ALMUFTI, S. M. The problems and challenges of infeasible paths in static analysis. **International Journal of Engineering & Technology**, v. 7, n. 4.19, p. 412–417, 2018. Citado na página 54.

MARASHDIH, A. W.; ZAABA, Z. F.; SUWAIS, K. An approach for detecting feasible paths based on minimal ssa representation and symbolic execution. **Applied Sciences**, MDPI, v. 11, n. 12, p. 5384, 2021. Citado na página 59.

MELO, S. M.; SOUZA, S. d. R. S. de; SARMANHO, F. S.; SOUZA, P. S. L. de. Contributions for the structural testing of multithreaded programs: coverage criteria, testing tool, and experimental evaluation. **Software Quality Journal**, Springer, v. 26, n. 3, p. 921–959, 2018. Citado nas páginas 25, 40 e 54.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011. Citado nas páginas 23, 24, 26, 31, 32, 33 e 34.

NAOI, K.; TAKAHASHI, N. Detection of infeasible paths using presburger arithmetic. **Systems and Computers in Japan**, Wiley Online Library, v. 30, n. 9, p. 74–87, 1999. Citado na página 56.

NAVAS, J. J. J.; SANTOSA, A. E. Demand-driven path-sensitive program slicing. Citeseer. Citado na página 57.

NGO, M. N.; TAN, H. B. K. Detecting large number of infeasible paths through recognizing their patterns. **Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07**, p. 215, 2007. Disponível em: <http://portal.acm.org/citation.cfm?doid=1287624.1287655>. Citado nas páginas 25, 26, 34, 35, 39, 57, 63, 72 e 75.

____. Heuristics-based infeasible path detection for dynamic test data generation. **Information and Software Technology**, v. 50, n. 7-8, p. 641–655, 2008. ISSN 09505849. Citado nas páginas 24, 25, 26, 35, 36, 40, 57, 64, 65, 69 e 74.

PACHECO, P. **An introduction to parallel programming**. [S.l.]: Elsevier, 2011. Citado nas páginas 25, 40 e 41.

PANDA, M.; SARANGI, P. P. Performance analysis of test data generation for path coverage based testing using three meta-heuristic algorithms. **International Journal of Computer Science and Informatics**, v. 3, n. 2, p. 34–41, 2013. Citado na página 58.

PANIGRAHI, C. R.; MALL, R. Regression test size reduction using improved precision slices. **Innovations in Systems and Software Engineering**, Springer London, v. 12, n. 2, p. 153–159, 2016. ISSN 16145054. Citado na página 59.

PAPADAKIS, M.; MALEVRIS, N. A symbolic execution tool based on the elimination of infeasible paths. **Proceedings - 5th International Conference on Software Engineering Advances, ICSEA 2010**, IEEE, p. 435–440, 2010. Citado nas páginas 25 e 58.

PATHADE, K.; KHEDKER, U. P. Computing partially path-sensitive mfp solutions in data flow analyses. **Proceedings of the 27th International Conference on Compiler Construction**, p. 37–47, 2018. Citado nas páginas 25, 26 e 59.

____. Path sensitive mfp solutions in presence of intersecting infeasible control flow path segments. In: **Proceedings of the 28th International Conference on Compiler Construction**. [S.l.: s.n.], 2019. p. 159–169. Citado na página 59.

PERES, L. M.; VERGILIO, S. R.; JINO, M.; MALDONADO, J. C. Path selection in the structural testing: Proposition, implementation and application of strategies. In: IEEE. **SCCC 2001. 21st International Conference of the Chilean Computer Science Society**. [S.l.], 2001. p. 240–246. Citado na página 57.

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. **Journal of object-oriented programming**, SIGS Publications, v. 2, n. 5, p. 13–19, 1990. Citado nas páginas 24 e 33.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and Software Technology**, Elsevier, v. 64, p. 1–18, 2015. Citado nas páginas 47 e 48.

PIERCE, L.; TRAGOUDAS, S. Unreachable code identification for improved line coverage. In: IEEE. **Sixteenth International Symposium on Quality Electronic Design**. [S.l.], 2015. p. 345–351. Citado na página 59.

PINTO, G. H.; VERGILIO, S. R. A multi-objective genetic algorithm to test data generation. **Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI**, IEEE, v. 1, p. 129–134, 2010. ISSN 10823409. Citado nas páginas 26, 58 e 66.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [S.l.]: McGraw Hill Brasil, 2016. Citado na página 23.

QI, X.; LI, Y. Parallel reachability testing based on hadoop mapreduce. In: SPRINGER. **International Conference on Software Analysis, Testing, and Evolution**. [S.l.], 2018. p. 173–184. Citado na página 59.

QIAN, Z.; HONG, D.; ZHAO, C.; ZHU, J.; ZHU, Z. A strategy for multi-target paths coverage by improving individual information sharing. **KSII Transactions on Internet and Information Systems (TIIS)**, Korean Society for Internet Information, v. 13, n. 11, p. 5464–5488, 2019. Citado na página 59.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE transactions on software engineering**, IEEE, n. 4, p. 367–375, 1985. Citado nas páginas 40 e 92.

RAUBER, T.; RÜNGER, G. **Parallel programming**. [S.l.]: Springer, 2013. Citado nas páginas 77 e 78.

RAYCHEV, V.; MUSUVATHI, M.; MYTKOWICZ, T. Parallelizing user-defined aggregations using symbolic execution. In: **Proceedings of the 25th Symposium on Operating Systems Principles**. [S.l.: s.n.], 2015. p. 153–167. Citado na página 59.

ROBSCHINK, T.; SNELTING, G. Efficient path conditions in dependence graphs. In: IEEE. **Proceedings of the 24th International Conference on Software Engineering. ICSE 2002**. [S.l.], 2002. p. 478–488. Citado na página 57.

RODRIGUES, L. A. L.; BRANCHER, J. D. Playing an educational game featuring procedural content generation: which attributes impact players' curiosity? In: . [S.l.: s.n.], 2019. Citado nas páginas 91, 92, 98, 99, 100 e 113.

RUIZ, J.; CASSÉ, H. Using smt solving for the lookup of infeasible paths in binary programs. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. **15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)**. [S.l.], 2015. Citado na página 59.

SALES, C. P.; JúNIOR, V. A. de S. Investigating multi and many-objective metaheuristics to support software integration testing. In: . New York, NY, USA: Association for Computing Machinery, 2020. (SAST 20), p. 1–10. ISBN 9781450387552. Disponível em: <https://doi.org/10.1145/3425174.3425175>. Citado na página 100.

SANTOS, S. a. H. N.; SILVEIRA, B. N. C. da; ANDRADE, S. a. A.; DELAMARO, M.; SOUZA, S. R. S. An experimental study on applying metamorphic testing in machine learning applications. In: **Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: Association for Computing Machinery, 2020. (SAST 20), p. 98–106. ISBN 9781450387552. Disponível em: <https://doi.org/10.1145/3425174.3425226>. Citado na página 100.

SARMANHO, F. S.; SOUZA, P. S. L.; SOUZA, R. S. S.; aO, A. S. S. Structural testing for semaphore-based multithread programs. In: **8th International Conference on Computational Science (ICCS)**. Krakow, Polonia: [s.n.], 2008. p. 337–346. Citado na página 25.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). **Biometrika**, JSTOR, v. 52, n. 3/4, p. 591–611, 1965. Citado na página 101.

SHU, T.; DING, Z.; CHEN, M.; XIA, J. A heuristic transition executability analysis method for generating efsm-specified protocol test sequences. **Information Sciences**, Elsevier, v. 370, p. 63–78, 2016. Citado na página 59.

SHU, T.; HUANG, Y.; DING, Z.; XIA, J.; JIANG, M. Generating feasible protocol test sequences from efsm models using monte carlo tree search. **Information and Software Technology**, Elsevier, v. 135, p. 106557, 2021. Citado na página 59.

SHULL, F.; CARVER, J.; BLDG, A.; TRAVASSOS, G. An empirical methodology for introducing software processes. **ACM SIGSOFT Software Engineering Notes**, v. 26, 07 2001. Citado nas páginas 91, 92, 99, 100 e 113.

SILVA, E. J. D.; PREDTOOL. Predtool: Uma ferramenta para apoiar o teste baseado em predicados. 2003. Citado nas páginas 57 e 61.

SNELTING, G. Combining slicing and constraint solving for validation of measurement software. In: SPRINGER. **International Static Analysis Symposium**. [S.l.], 1996. p. 332–348. Citado na página 56.

SNELTING, G.; ROBSCHINK, T.; KRINKE, J. Efficient path conditions in dependence graphs for software safety analysis. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 15, n. 4, p. 410–457, 2006. Citado na página 57.

SOMMERVILLE, I. Engenharia de software, 8 edição. **Pearson, Addison Wesley**, v. 8, n. 9, 2007. Citado nas páginas 23 e 31.

SONG, Y.; ZHANG, X.; GONG, Y.-Z. Infeasible path detection based on code pattern and backward symbolic execution. **Mathematical Problems in Engineering**, Hindawi, v. 2020, 2020. Citado na página 59.

SOUTER, A. L.; POLLOCK, L. L. Characterization and automatic identification of type infeasible call chains. **Information and Software Technology**, Elsevier, v. 44, n. 13, p. 721–732, 2002. Citado na página 57.

SOUZA, M.; VILLANES, I. K.; DIAS-NETO, A. C.; ENDO, A. T. On the exploratory testing of mobile apps. In: **Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: Association for Computing Machinery, 2019. (SAST 2019), p. 42–51. ISBN 9781450376488. Disponível em: <https://doi.org/10.1145/3356317.3356322>. Citado na página 100.

SOUZA, P. L.; SAWABE, E. T.; aO, A. S. S.; VERGILIO, S. R.; SOUZA, S. R. S. ValiPVM - a graphical tool for structural testing of PVM programs. In: **Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 257–264. ISBN 978-3-540-87474-4. Citado na página 25.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 26, n. 1, p. 21–50, 2014. Citado na página 25.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, Elsevier, v. 18, p. 149–158, 2013. Citado nas páginas 41 e 100.

SOUZA, S.; SOUZA, P.; MACHADO, M.; CAMILLO, M.; SIMÃO, A.; ZALUSKA, E. Using coverage and reachability testing to improve concurrent program testing quality. 2011. Citado na página 58.

SOUZA, S.; VERGILIO, S. R.; SOUZA, P.; SIMAO, A.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, John Wiley and Sons Ltd., Chichester, UK, v. 20, p. 1893–1916, November 2008. ISSN 1532-0626. Citado nas páginas 25, 27, 34, 40, 43, 44 e 67.

SOUZA, S. d. R. S. de; VERGILIO, S. R.; SOUZA, P. S. L. de; SIMÃO, A. da S.; GONCALVES, T. B.; LIMA, A. de M.; HAUSEN, A. C. Valipar: A testing tool for message-passing parallel programs. In: **SEKE**. [S.l.: s.n.], 2005. v. 2005, n. November 2015, p. 386–391. Citado nas páginas 11, 85, 86 e 87.

SOUZA, S. R.; SOUZA, P. S.; BRITO, M. A.; SIMAO, A.; ZALUSKA, E. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 25, n. 3, p. 310–332, 2015. Citado na página 59.

SOUZA, S. R. S.; SOUZA, P. S. L.; BRITO, M. A. S.; SIMAO, A. S.; ZALUSKA, E. J. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 25, n. 3, p. 310–332, maio 2015. ISSN 0960-0833. Disponível em: <http://dx.doi.org/10.1002/stvr.1568>. Citado na página 25.

STUDENT. The probable error of a mean. **Biometrika**, Oxford University Press, v. 6, n. 1, p. 1–25, 1908. Citado nas páginas 101 e 111.

SUN, B.; WANG, J.; GONG, D.; TIAN, T. Scheduling sequence selection for generating test data to cover paths of mpi programs. **Information and Software Technology**, Elsevier, v. 114, p. 190–203, 2019. Citado na página 59.

TAI, K.-C. Reachability testing of asynchronous message-passing programs. In: IEEE. **Proceedings of PDSE'97: 2nd International Workshop on Software Engineering for Parallel and Distributed Systems**. [S.l.], 1997. p. 50–61. Citado na página 45.

TAMRAWI, A.; KOTHARI, S. Event-flow graphs for efficient path-sensitive analyses. **arXiv preprint arXiv:1404.1279**, 2014. Citado na página 59.

TANENBAUM, A. S. Modern operating systems, 2001. **Prentice-Hall, Inc., second edition, pages**, v. 200, p. 205–206, 2009. Citado na página 40.

UMESH, N.; SRIVASTAVA, S. Path prioritization using meta-heuristic approach. **International Journal of Computer Applications**, Citeseer, v. 77, n. 11, 2013. Citado na página 58.

VENGADESWARAN, S.; GEETHA, K. Symbolic execution—an efficient approach for test case generation. In: IEEE. **2013 International Conference on Recent Trends in Information Technology (ICRTIT)**. [S.l.], 2013. p. 575–581. Citado na página 58.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Caminhos não executáveis na automação das atividades de teste. **VI Simpósio Brasileiro de Engenharia de Software**, p. 343 – 356, 1992. Citado na página 56.

____. **Caminhos Não executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas**. 1992. Citado nas páginas 27, 56, 65 e 71.

____. Infeasible paths within the context of data flow based criteria. In: **VI International Conference on Software Quality**. [S.l.: s.n.], 1996. p. 310–321. Citado na página 49.

____. Constraint based criteria: An approach for test case selection in the structural testing. **Journal of Electronic Testing: Theory and Applications (JETTA)**, v. 17, n. 2, p. 175–183, 2001. ISSN 09238174. Citado nas páginas 49, 57 e 60.

____. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. **Journal of the Brazilian Computer Society**, v. 12, n. 1, p. 73–88, 2006. ISSN 16784804. Citado nas páginas 24, 27, 49, 57, 62, 69 e 72.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M.; SOARES, I. W. Constraint based structural testing criteria. **Journal of Systems and Software**, v. 79, n. 6, p. 756–771, 2006. ISSN 01641212. Citado nas páginas 27, 57 e 61.

VIJAYKUMAR, N. L.; SENNE, E. L. F. Geração de testes caixa branca para aplicações multithreads : Geração abordagem por statecharts. 2011. Citado na página 23.

VILELA, R. F.; NETO, J. C.; PINTO, V. H. S. C.; SOUZA, P. S. Lopes de; SOUZA, S. do Rocio Senger de. Bio-inspired optimization to support the test data generation of concurrent software. **Concurrency and Computation: Practice and Experience**, v. 35, n. 2, p. e7489, 2023. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7489>. Citado na página 101.

WANG, X.; JIANG, Y.; TIAN, W. An efficient method for automatic generation of linearly independent paths in white-box testing. **International Journal of Engineering and Technology Innovation**, Taiwan Association of Engineering and Technology Innovation, v. 5, n. 2, p. 108–120, 2015. Citado na página 59.

WANG, Y.-w.; XING, Y.; ZHANG, X.-Z. A method of path feasibility judgment based on symbolic execution and range analysis. **International Journal of Future Generation Communication and Networking**, v. 7, n. 3, p. 205–212, 2014. Citado nas páginas 25 e 58.

WANG, Z.; WEN, L.; ZHU, X.; LIU, Y.; WANG, J. Detecting infeasible traces in process models. In: **ICEIS (3)**. [S.l.: s.n.], 2012. p. 212–217. Citado na página 58.

WATSON, A. H.; WALLACE, D. R.; MCCABE, T. J. **Structured testing: A testing methodology using the cyclomatic complexity metric**. [S.l.]: US Department of Commerce, Technology Administration, National Institute of . . . , 1996. v. 500. Citado nas páginas 33 e 34.

WILLIAMS, N. Abstract path testing with pathcrawler. **AST '10 Proceedings of the 5th Workshop on Automation of Software Test Pages 35-42**, p. 35–42, 2010. Citado na página 58.

WILLIAMS, N.; MARRE, B.; MOUY, P. On-the-fly generation of k-path tests for c functions: towards the automation of grey-box testing. Citeseer. Citado na página 57.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: **Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: ACM, 2014. (EASE '14), p. 38:1–38:10. ISBN 978-1-4503-2476-2. Disponível em: <http://doi.acm.org/10.1145/2601248.2601268>. Citado nas páginas 47 e 48.

WONG, W. E.; LEI, Y.; MA, X. Effective generation of test sequences for structural testing of concurrent programs. In: **10th IEEE International Conference on Engineering of Complex Systems (ICECCS 2005)**. Shanghai, China: [s.n.], 2005. p. 539– 548. Citado na página 25.

XIA, C.; WANG, X.; QIAO, L.; ZHANG, Y.; MA, B.; SHI, C. Feasible basic path generation based on genetic algorithm. In: IEEE. **2019 6th International Conference on Dependable Systems and Their Applications (DSA)**. [S.l.], 2019. p. 353–358. Citado na página 59.

XIAOTONG, Z.; TAO, Z.; PANDE, S. Using branch correlation to identify infeasible paths for anomaly detection. **Proceedings of the Annual International Symposium on Microarchitecture, MICRO**, IEEE, p. 113–122, 2006. ISSN 10724451. Citado na página 57.

XIBO, W.; NA, S. Automatic test data generation for path testing using genetic algorithms. **Proceedings - 3rd International Conference on Measuring Technology and Mechatronics Automation, ICMTMA 2011**, v. 1, p. 596–599, 2011. Citado nas páginas 25, 58 e 66.

YAN, J.; LI, Z.; YUAN, Y.; SUN, W.; ZHANG, J. Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In: IEEE. **2006 17th International Symposium on Software Reliability Engineering**. [S.l.], 2006. p. 75–84. Citado na página 57.

YAN, J.; ZHANG, J. An efficient method to generate feasible paths for basis path testing. **Information Processing Letters**, v. 107, n. 3-4, p. 87–92, 2008. ISSN 00200190. Citado nas páginas 57 e 65.

YANG, C.-S. D. **Program-based, structural testing of shared memory parallel programs**. [S.l.]: University of Delaware, 1999. Citado na página 56.

YANG, R.; CHEN, Z.; XU, B.; WONG, W. E.; ZHANG, J. Improve the effectiveness of test case generation on efsm via automatic path feasibility analysis. In: IEEE. **2011 ieee 13th international symposium on high-assurance systems engineering**. [S.l.], 2011. p. 17–24. Citado na página 58.

YANO, T.; MARTINS, E.; SOUSA, F. L. de. Generating feasible test paths from an executable model using a multi-objective approach. In: IEEE. **2010 third international conference on software testing, verification, and validation workshops**. [S.l.], 2010. p. 236–239. Citado na página 58.

YATES, D.; MALEVRIS, N. Reducing the effects of infeasible paths in branch testing. **ACM SIGSOFT Software Engineering Notes**, v. 14, n. 8, p. 48–54, 1989. ISSN 01635948. Disponível em: <http://portal.acm.org/citation.cfm?doid=75309.75315>. Citado nas páginas 24, 26, 35, 36, 40, 56 e 69.

YOUNG, M.; TAYLOR, R. N. Combining static concurrency analysis with symbolic execution. **IEEE Transactions on Software Engineering**, IEEE, v. 14, n. 10, p. 1499–1511, 1988. Citado nas páginas 39 e 40.

ZHANG, J.; CHEN, X.; WANG, X. Path-oriented test data generation using symbolic execution and constraint solving techniques. p. 242–250, 2004. Citado nas páginas 39, 57 e 61.

ZHANG, J.; WANG, X. A constraint solver and its application to path feasibility analysis. **International Journal of Software Engineering and Knowledge Engineering**, v. 11, n. 02, p. 139–156, 2001. ISSN 0218-1940. Disponível em: <http://www.worldscientific.com/doi/abs/10.1142/S0218194001000487>. Citado nas páginas 33, 57 e 61.

ZHANG, Y.; JIANG, S.; HAN, H. Research progress on infeasible path detecting problem. **Journal of Computational and Theoretical Nanoscience**, American Scientific Publishers, v. 12, n. 8, p. 1931–1935, 2015. Citado na página 54.

ZHOU, X.; LIN, F.; YANG, L.; NIE, J.; TAN, Q.; ZENG, W.; ZHANG, N. Aii-du-path coverage for parallel programs. **ISSTA '98 Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis**, v. 5, n. 1, p. 153–162, 1998. ISSN 21931801. Citado nas páginas 41, 56, 60 e 67.

ZHU, H.; JIN, D.; GONG, Y.; XING, Y.; ZHOU, M. Detecting interprocedural infeasible paths based on unsatisfiable path constraint patterns. **IEEE Access**, IEEE, v. 7, p. 15040–15055, 2019. Citado na página 59.

Zhu, Q.; Panichella, A.; Zaidman, A. An investigation of compression techniques to speed up mutation testing. In: **2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2018. p. 274–284. Citado nas páginas 91, 98, 99 e 113.

ZIVIANI, N. **Algorithms Project with Implementations in JAVA and C ++ (in Portuguese)**. [S.l.]: Cengage Learning Edições Ltda., 2010. Citado na página 92.


ICMC
USP
SÃO CARLOS