Evidências sobre o uso de técnicas de geração
automática de dados de teste em programas
concorrentes

Ricardo Ferreira Vilela

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Ricardo Ferreira Vilela

Evidências sobre o uso de técnicas de geração automática de dados de teste em programas concorrentes

Dissertação apresentada ao Instituto de Ciências
Matemáticas e de Computação – ICMC-USP,
como parte dos requisitos para obtenção do título
de Mestre 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

USP – São Carlos
Setembro de 2016

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

V695e Vilela, Ricardo Ferreira
Evidências sobre o uso de técnicas de geração automática de dados de teste em programas concorrentes / Ricardo Ferreira Vilela; orientadora Simone do Rocio Senger de Souza. - São Carlos - SP, 2016.
124 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. Técnicas de Geração Automática de Dados de Teste. 2. Teste de Software. 3. Programação Concorrente. 4. Engenharia de Software Experimental. 5. Avaliação Experimental. I. Souza, Simone do Rocio Senger de, orient. II. Título.

Ricardo Ferreira Vilela

**Evidences about the use of automatic test data generation
techniques in context of concurrent programs**

Master dissertation submitted to the Instituto de
Ciências Matemáticas e de Computação – ICMC-
USP, in partial fulfillment of the requirements for the
degree of the Master 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

USP – São Carlos
September 2016

AGRADECIMENTOS

A Deus pela sua presença constante na minha vida, pelo auxílio nas minhas escolhas, por estar sempre comigo nos momentos mais difíceis e por fazer com que mais esse sonho se realize. Agradeço ainda por ter colocado em meu caminho pessoas tão especiais, que me ajudaram durante a realização deste mestrado. A estas pessoas estorno aqui meus sinceros agradecimentos.

A Simone e Paulo pela orientação, amizade, apoio, profissionalismo e sábias reflexões e contribuições para esta dissertação, por investir seus conhecimentos, tempo, paciência e me oferecerem a oportunidade de aprender. Sem vocês o desenvolvimento desta dissertação não seria possível.

Aos meus Pais Cairo e Maria Aparecida por serem a base de toda a minha formação. Agradeço por me ajudarem a realizar cada um dos meus sonhos e por se alegrarem comigo a cada conquista. Sem o apoio de vocês esse dia com certeza não se realizaria!

Um agradecimento especial aos meus irmãos Marcos e Laíze por acreditarem na minha capacidade, pelo apoio imprescindível, pelos conselhos e por serem exemplos para mim, vocês são essenciais na minha vida.

A minha querida namorada Letícia Dorneles, por me proporcionar tantos momentos de felicidade, por sua dedicação e carinho, e por sempre estar presente e minha vida.

Aos meus amigos do Labes, laSDPC e Biocom agradeço pela força, apoio, por estarem sempre presentes, sem vocês o caminho teria sido muito mais difícil.

Aos professores do Instituto de Ciências Matemáticas e de Computação, meu agradecimento por todas as oportunidades de crescimento e aprendizagem e pelos ensinamentos sempre valiosos.

Ao Instituto de Ciências Matemáticas e de Computação pela oportunidade do acesso ao conhecimento e a pesquisa.

A Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES) pelo apoio financeiro para a realização desta pesquisa.

Aos membros da banca pelas valiosas contribuições com este trabalho, e pela disponibilidade em aceitar participar da minha banca de defesa.

A todos aqueles que, de uma forma ou de outra, contribuíram para a realização deste trabalho.

A todos, muito obrigado!

*“E se seus sonhos estiverem nas nuvens, não se
preocupe... eles estão no lugar certo! Construa
agora os alicerces!”
(William Shakespeare)*

RESUMO

VILELA, R. F.. **Evidências sobre o uso de técnicas de geração automática de dados de teste em programas concorrentes**. 2016. 124 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Diversas pesquisas apoiam e investigam o teste de programas concorrentes, as quais objetivam, principalmente, a proposição de critérios de teste e mecanismos para execução das diferentes sincronizações entre processos ou *threads*. As características específicas dessas aplicações podem ocasionar diferentes tipos de defeitos, os quais, em sua maioria, não são facilmente identificados. Nesse contexto, a geração automática de dados de teste pode apoiar a atividade de teste atuando na seleção de entradas mais representativas, ou seja, aquelas com maior probabilidade de revelar defeitos. Apesar disso, poucas pesquisas abordam este tema no contexto de programas concorrentes, e as existentes não consideram aspectos importantes desse tipo de aplicação. A geração de dados de teste para programas sequenciais dispõe de uma variedade de técnicas que apoiam a seleção dos dados de teste. Essas técnicas têm sido estendidas para o contexto de programas concorrentes partindo da premissa que esses programas necessitam de abordagens mais complexas para seleção de entradas, em decorrência disso um maior custo é imposto ao teste. Considerando esse contexto, uma lacuna ainda em aberto é a avaliação das técnicas para o cenário de programas concorrentes. Neste trabalho a avaliação das técnicas foi explorada por meio da realização de estudos experimentais, os quais avaliaram diferentes técnicas de geração de dados de teste para o contexto de programas concorrentes, considerando a eficácia em revelar defeitos, cobertura de critérios e custo para atividade de teste. Os resultados obtidos demonstraram que as técnicas empregadas para programas sequenciais não atingem o mínimo esperado para este tipo aplicação. Apesar disso, as técnicas investigadas apresentaram características importantes que podem auxiliar a atividade de teste para programas concorrentes e a proposição de abordagens efetivas de geração de dados para esse contexto.

Palavras-chave: Técnicas de Geração Automática de Dados de Teste, Teste de Software, Programação Concorrente, Engenharia de Software Experimental, Avaliação Experimental.

ABSTRACT

VILELA, R. F.. **Evidências sobre o uso de técnicas de geração automática de dados de teste em programas concorrentes**. 2016. 124 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

The concurrent program testing has been largely investigated with propositions of testing criteria and mechanisms, which aim mainly to testing criteria proposition and mechanisms for execution of different synchronizations. The specific characteristics of these applications can lead to the different types of faults, which, in most of cases, are not easily identified. In this context, the automatic test data generation can support the testing activity acting in selecting the most representative data tests, i.e. those most likely to reveal faults. Nevertheless, few studies address this issue in the context of concurrent programs, and the these studies do not consider important aspects of this type of application. In contrast, we can find several techniques proposed to support the test data generation for sequential programs. These techniques have been extended to the context of concurrent programs on the premise that these programs require more complex approaches for selection of test data. As a result, a greater cost for testing activity is enforced. Considering this context, a gap still open is the evaluation of techniques for the scenario of concurrent programs. In this work the evaluation of techniques was explored through experimental studies, which different techniques of test data generatoon were evaluated, considering effectiveness, testing coverage and application cost. The results showed that the generation techniques used for sequential programs used in the experimental study do not reach the minimum expected in terms of effectiveness and cost for concurrent programs. Nevertheless, the techniques investigated showed significant features that can help the proposition of effective approaches for test data generation applied to concurrent programs.

Key-words: Automatic Data Test Generation Techniques, Software Testing, Concurrent Programming, Experimental Software Engineering, Experimental Evaluation.

LISTA DE ILUSTRAÇÕES

Figura 1 – Cenário típico da atividade de teste (Extraído de Delamaro, Maldonado e Jino (2007)).	30
Figura 2 – Exemplo de um grafo de fluxo de controle (Adaptado de Gao <i>et al.</i> (2003)).	33
Figura 3 – Modelo de memória em arquiteturas paralelas (Adaptado de Barney (2010)).	38
Figura 4 – Modelo de memória compartilhada de computação paralela (Adaptado de Quinn (2003)).	38
Figura 5 – O modelo de passagem de mensagens onde cada processador possui seu próprio espaço de endereçamento e uma rede de interconexão de apoio entre os processadores (Adaptado de Quinn (2003)).	40
Figura 6 – Arquitetura ferramenta ValiPar.	48
Figura 7 – Exemplo da técnica de execução simbólica (Extraído de Anand <i>et al.</i> (2013))	53
Figura 8 – Arquitetura ferramenta Randoop (Adaptado de (PACHECO; ERNST, 2007)).	63
Figura 9 – Visão geral dos experimentos realizados. Retângulos com símbolos $\vdash$ representam a existência de um subprocesso de atividade.	82
Figura 10 – Ambiente de geração de dados de teste empregado para as técnicas.	83
Figura 11 – Cenário de teste aplicado para cada conjunto de teste obtido.	83
Figura 12 – Cenário de execução ferramenta ValiPar.	84
Figura 13 – Organização dos arquivos gerados durante o experimento.	84
Figura 14 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo (GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC), Aleatória(RAND) em diferentes critérios de teste para programas de passagem de mensagem. <i>All Nodes</i> (AN), <i>All Edges</i> (AE), <i>All Uses</i> (AU), <i>All intEr-Uses</i> (AEU), <i>All Sync-Edges</i> (ASE), <i>All M-Uses</i> (AMU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração.	89
Figura 15 – Porcentagem de cobertura dos critérios <i>All Nodes</i> (AN), <i>All Edges</i> (AE), <i>All Uses</i> (AU), <i>All intEr-Uses</i> (AEU), <i>All Sync-Edges</i> (ASE), <i>All M-Uses</i> (AMU) em diferentes técnicas de geração de dados de teste para programas de passagem de mensagem. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.	90

Figura 16 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória(RAND) em diferentes critérios de teste para programas de memória compartilhada. *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All Sync-Edges* (ASE), *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração. 92

Figura 17 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de memória compartilhada. *All Uses* (AU) e *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste. 93

Figura 18 – Porcentagem de cobertura dos critérios *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All intEr-Uses* (AEU), *All Sync-Edges* (ASE), *All M-Uses* (AMU) em diferentes técnicas de geração de dados de teste para programas de memória compartilhada. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste. 95

Figura 19 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de passagem de mensagem e memória compartilhada. *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All Sync-Edges* (ASE), *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração. 96

Figura 20 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de passagem de mensagem e memória compartilhada. *All Uses* (AU) e *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração. 97

Figura 21 – Porcentagem de cobertura dos critérios <i>All Nodes</i> (AN), <i>All Edges</i> (AE), <i>All Uses</i> (AU), <i>All intEr-Uses</i> (AEU), <i>All Sync-Edges</i> (ASE), <i>All M-Uses</i> (AMU) em diferentes técnicas de geração de dados de teste para programas de passagem de mensagem e memória compartilhada. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.	98
Figura 22 – Porcentagem de eficácia das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC) e Aleatória (RAND) aplicadas para detecção de diferentes defeitos. Outros sinais ausentes ou inexistentes (F1), bloqueio de região crítica (F2), processo de destino ou origem incorreto (F3), defeito em um laço que antecede uma primitiva <i>send</i> ou <i>receive</i> (F4), atribuição de forma incorreta usando variáveis (F5) e operação não-atômica assumida como atômica (F6). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de eficácia entre as técnicas de geração.	100
Figura 23 – Porcentagem de eficácia em revelar defeitos, Outros sinais ausentes ou inexistentes (F1), bloqueio de região crítica (F2), processo de destino ou origem incorreto (F3), defeito em um laço que antecede uma primitiva <i>send</i> ou <i>receive</i> (F4), atribuição de forma incorreta usando variáveis (F5) e operação não-atômica assumida como atômica (F6). Considerando diferentes técnicas de geração, Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC) e Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de eficácia entre as técnicas de geração.	102
Figura 24 – Custo obtido pela média dos tamanhos dos conjuntos gerados pelas técnicas. Algoritmo Genético (GA), Algoritmo Genético Multi-Objetivo (GAMO), <i>Concolic Testing</i> (CONC), Aleatória Guiada (GRAND) e Aleatória (RAND). Letras diferentes sobre a linha e as barras simbolizam a existência de diferença significativa de custo entre as técnicas de geração.	103
Figura 25 – Relação de cobertura, eficácia e custo nas técnicas de geração de dados de teste. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), <i>Concolic Testing</i> (CONC), Aleatória (RAND). Letras minúsculas diferentes simbolizam diferenças significativas de cobertura entre as técnicas, letras maiúsculas diferentes simbolizam diferenças significativas de eficácia entre as técnicas (<i>Kruskal-Wallis</i> , $p<0,05$).	104

LISTA DE QUADROS

Quadro 1 – Defeitos típicos de programas concorrentes.	42
Quadro 2 – Relação da cobertura de critérios dos programas obtida pela técnica GA. . .	120
Quadro 3 – Relação da cobertura de critérios dos programas obtida pela técnica GRAND.	121
Quadro 4 – Relação da cobertura de critérios dos programas obtida pela técnica CONC.	122
Quadro 5 – Relação da cobertura de critérios dos programas obtida pela técnica GAMO.	123
Quadro 6 – Relação da cobertura de critérios dos programas obtida pela técnica GRAND.	124

LISTA DE TABELAS

Tabela 1 – Predicados de Ramificação	55
Tabela 2 – Técnicas de geração avaliadas no estudo.	72
Tabela 3 – Programas sob teste considerados.	74
Tabela 4 – Programas sob teste com defeitos.	75
Tabela 5 – Critérios de teste considerados.	75
Tabela 6 – Defeitos considerados no estudo.	76
Tabela 7 – Configuração de Hardware e Software	76
Tabela 8 – Ferramentas de geração de dados de teste consideradas no experimento. . .	77
Tabela 9 – Score de classificação dos fatores cobertura, eficácia e custo.	105
Tabela 10 – Classificação das técnicas em relação aos fatores cobertura, eficácia e custo obtida pela Equação 5.3.	105

SUMÁRIO

1	INTRODUÇÃO	23
1.1	Contextualização e Motivação	23
1.2	Objetivos	27
1.3	Organização	28
2	TESTE DE SOFTWARE	29
2.1	Considerações Iniciais	29
2.2	Conceitos Básicos	30
2.3	Teste estrutural	32
2.3.1	<i>Critérios baseados em fluxo de controle</i>	34
2.3.2	<i>Critérios baseados em fluxo de dados</i>	34
2.4	Programação Concorrente	35
2.4.1	<i>Conceitos Básicos sobre Programação Concorrente</i>	36
2.4.2	<i>Teste de Programas Concorrentes</i>	40
2.4.3	<i>A Ferramenta de Teste ValiPar</i>	47
2.5	Considerações Finais	49
3	GERAÇÃO AUTOMÁTICA E DADOS DE TESTE	51
3.1	Considerações Iniciais	51
3.2	Conceitos	51
3.3	Geração Aleatória	52
3.4	Execução Simbólica	53
3.5	Execução Dinâmica	55
3.6	<i>Concolic Testing</i>	56
3.7	Meta-heurísticas	56
3.7.1	<i>Hill Climbing</i>	57
3.7.2	<i>Simulated Annealing</i>	57
3.7.3	<i>Algoritmo Genético</i>	59
3.7.4	<i>Otimização Multi-Objetivo</i>	61
3.8	Discussão sobre técnicas de geração	61
3.9	Ferramentas de Geração de Dados de Teste	63
3.9.1	<i>Feedback-Directed Random Testing for Java - Randoop</i>	63
3.9.2	<i>Guided Random Testing - GRT</i>	64

3.9.3	<i>CodePro AnalytiX - JUnit Test Generation</i>	65
3.9.4	<i>Automatic Test Suite Generation for Java - EvoSuite</i>	66
3.9.5	<i>Many-Objective Sorting Algorithm - EvoSuite-MOSA</i>	66
3.10	Considerações Finais	67
4	DEFINIÇÃO E PLANEJAMENTO DO ESTUDO	69
4.1	Considerações Iniciais	69
4.2	Definição do Estudo	69
4.2.1	<i>Definição de Metas</i>	70
4.3	Planejamento	71
4.3.1	<i>Seleção do contexto</i>	71
4.3.2	<i>Formulação das hipóteses</i>	72
4.3.3	<i>Seleção de variáveis</i>	73
4.3.4	<i>Seleção dos Sujeitos de Teste</i>	74
4.3.5	<i>Métricas de Avaliação</i>	75
4.3.6	<i>Instrumentação</i>	76
4.3.7	<i>Avaliação de Validade</i>	77
4.4	Operação	79
4.4.1	<i>Preparação</i>	79
4.4.2	<i>Condução</i>	80
4.5	Considerações Finais	85
5	ANÁLISE DOS RESULTADOS	87
5.1	Considerações Iniciais	87
5.2	QP1 - Análise de Cobertura de Critérios	87
5.3	QP2 - Análise de Eficácia em Revelar Defeitos	99
5.4	QP3 - Análise de Custo	103
5.5	Discussão	105
5.6	Considerações Finais	107
6	CONCLUSÕES E TRABALHOS FUTUROS	109
6.1	Caracterização da Contribuição	109
6.2	Contribuições de Pesquisa	109
6.3	Trabalhos Futuros	110
	REFERÊNCIAS	111
APÊNDICE A	ANÁLISE DE COBERTURA DAS TÉCNICAS DE GE- RAÇÃO	119

INTRODUÇÃO

1.1 Contextualização e Motivação

A atividade de teste é fundamental para o desenvolvimento de software com qualidade, dado defeitos podem ser inseridos no software durante o seu desenvolvimento. Nesse sentido, o teste de software é uma das atividades de verificação e validação (V&V) utilizado para aumentar a garantia de qualidade dos software com o objetivo de revelar defeitos ainda não identificados. É importante ressaltar que a atividade de teste não garante que um software está livre de defeitos, apesar disso, pode contribuir para uma maior confiabilidade do software. Assim, é essencial que as aplicações sejam testadas antes de ser entregues aos usuários ([MYERS; SANDLER; BADGETT, 2011](#)).

A atividade de teste é uma atividade dinâmica, ou seja, para testar uma aplicação é necessária a execução do programa. Dessa forma, a principal atribuição do testador é fornecer entradas (dados de teste) e analisar se o comportamento do programa está de acordo com sua especificação. Para que um programa seja devidamente testado é necessário que o testador (ou ferramenta de teste) forneça entradas específicas que possuem maior probabilidade de revelar defeitos, uma vez que executar todas as suas possíveis entradas pode ser inviável devido à cardinalidade do domínio de entrada das aplicações ([AMMANN; OFFUTT, 2008](#)).

Para auxiliar a seleção de entradas do programa foram estabelecidos os critérios de teste. Os critérios podem ser considerados como regras que tem como principal objetivo definir elementos que devem ser exercitados (elementos requeridos) durante o teste de software. Assim, os critérios podem guiar a atividade de teste oferecendo informações mínimas que precisam ser exercitadas e, com isso, auxiliando no processo de seleção das entradas de teste. Existem diferentes critérios de teste os quais são classificados em diferentes técnicas, sendo as mais conhecidas: técnica funcional, estrutural e baseada em erros ([DELAMARO; MALDONADO; JINO, 2007](#)). A diferença entre essas técnicas está na informação de origem usada para gerar os

elementos requeridos a serem executados durante os testes. Neste trabalho são considerados os critérios estruturais de teste.

Os critérios estruturais utilizam aspectos estruturais do código-fonte para orientar a seleção de casos de teste exigindo a execução de partes ou de componentes elementares do programa. Dessa forma, é possível: i) garantir que todos os caminhos independentes sejam executados pelo menos uma vez; ii) executar todas as decisões lógicas em seus respectivos estados (verdadeiro e falso); e iii) exercitar estruturas de dados internas para assegurar a sua validade (PRESSMAN, 2010). A ideia central no teste estrutural não é testar todas as possibilidades de entrada de um software, e sim as mais representativas. Sendo assim, é possível exercitar todos os comandos e todas as condições pelo menos uma vez para cada um de seus valores possíveis (verdadeiro ou falso).

Mesmo assim, na maioria dos casos, a determinação de quais entradas selecionar é feita manualmente, com base no conhecimento do software e experiência do testador. Embora a automatização dessa atividade (seleção de dados de teste) seja muito desejada, não existe um algoritmo de propósito geral e eficiente que seja capaz de gerar automaticamente e a baixo custo um conjunto de teste efetivo. Nem mesmo é possível determinar automaticamente se esse conjunto existe para um determinado critério de teste (VERGILIO; MALDONADO; JINO, 2007).

Apesar de existirem limitações, algumas técnicas têm sido propostas para geração automática de dados de teste, as quais, em geral, são guiadas por critérios de teste (VERGILIO; MALDONADO; JINO, 2007; EDVARDSSON, 1999; MCMINN, 2004; KOIRALA; SHEIKH, 2009; PATTON, 2005; GRAHAM *et al.*, 2006). Uma das técnicas mais simples e baratas é a geração aleatória de dados de teste, a qual tem como objetivo selecionar pontos específicos do domínio de entrada de forma aleatória até que o critério de teste seja satisfeito (ou coberto) (DURAN; NTAFOU, 1984; CHEN; LEUNG; MAK, 2005). O problema principal dessa técnica é que ela não garante que trechos importantes do código serão executados.

A geração de dados de teste com execução simbólica geralmente é usada em conjunto com um critério de teste baseado em caminhos. A ideia é que são estabelecidos caminhos possíveis de execução do código e essa técnica gera dados de teste automáticos que são capazes de executar um dado caminho ou um grupo de caminhos. Para isso, expressões algébricas, que representam a execução de um caminho, são derivadas e o resultado é uma expressão simbólica, chamada computação do caminho que expressa as variáveis de saída em termos das variáveis de entrada, e uma outra expressão chamada condição do caminhos que representa condições para o caminho a ser executado (BOYER; ELSPAS; LEVITT, 1975; OFFUTT; SEAMAN, 1990). A aplicação dessa técnica é bastante dispendiosa, devido às derivações sistemáticas e a manipulação das expressões simbólicas. A medida que a complexidade do software aumenta, o uso dessa técnica pode se tornar proibitiva.

A geração de dados de teste com execução dinâmica se baseia na execução real do

programa em teste, utilizando métodos de minimização de funções e análise de fluxo de dados. Partes do programa são tratadas como funções que são avaliadas durante a execução, e os dados são avaliados e melhorados à medida que são gerados. No caso do programa apresentar um erro, a análise de fluxo de dados dinâmica é empregada buscando determinar quais variáveis de entrada são responsáveis pelo comportamento incorreto do programa. O método de minimização de função é usado no processo de busca do dado mais adequado (DARA *et al.*, 2009). A ideia central dessa técnica é coletar dados, em tempo de execução, que se aproximam da cobertura do critério. Depois de coletados, os melhores dados são alterados até que um deles satisfaça o critério.

Uma técnica que tem sido extensamente explorada para a geração de dados de teste é meta-heurística. Essa técnica busca resolver de forma genérica problemas que não possuem um algoritmo eficiente para a sua solução e possuem um espaço de soluções possíveis bastante extenso. De uma maneira geral, essa abordagem utiliza combinação de escolhas aleatórias e conhecimento histórico dos resultados anteriores adquiridos pela técnica para guiar e realizar novas buscas dentro do espaço de pesquisa. Algumas técnicas de meta-heurística conhecidas são: algoritmos genéticos, *simulated annealing*, busca tabu, colônia de formigas e colônia de abelhas (MCMINN, 2004). A geração de dados de teste com algoritmos genéticos utiliza conceitos da computação evolutiva, que baseia-se na evolução natural das espécies. Com essa técnica pode-se ter várias soluções que se encaixam em problemas com dificuldade na obtenção de resultados ótimos. Quando se implementa um algoritmo genético é preciso representar o espaço de busca do problema e suas possíveis soluções de forma computacional e encontrar a função de avaliação que é usada para avaliar quão boa é uma dada solução (RUSSELL; NORVIG, 2009). Algoritmo genético parte da ideia de que dada duas soluções com resultados aproximados para um dado problema, essas duas soluções combinadas podem conduzir uma melhor solução. A geração de dados com algoritmos genéticos geralmente se encontra classificada em duas categorias: função de avaliação baseada na cobertura, onde a avaliação é realizada com base na cobertura de cada dado de teste; e função de avaliação orientada a um objetivo, onde a função é especificada de acordo com um objetivo que é, geralmente, a cobertura de um elemento particular.

Neste contexto, o trabalho proposto por Panchapakesan, Abielmona e Petriu (2013) apresenta uma proposta híbrida de meta-heurística, a qual considera algoritmo genético e evolutivo para geração de dados de teste. O objetivo do trabalho teve foco na geração de dados de teste que pudessem obter maior cobertura no fluxo de dados dos sistemas em teste evitando dados redundantes. Para este fim, a abordagem híbrida verifica a semelhança entre os dados de teste para evitar que dados redundantes sejam selecionados, ou seja, em vez de utilizar uma população inicial aleatória assim como os algoritmos genéticos, o algoritmo proposto classifica os dados de entrada antes que os mesmos sejam inseridos no algoritmo genético e no algoritmo evolutivo. Outra característica importante apresentada é a otimização de um problema multiobjetivo, dessa forma, o algoritmo proposto gera dados de acordo com determinados caminhos existentes no

programa em teste, diferentemente de outros trabalhos que utilizam um dado de teste para cobertura de todos os caminhos. Os resultados obtidos por [Panchapakesan, Abielmona e Petriu \(2013\)](#) são baseados no teste de três sistemas em teste, e comparados com uma abordagem mono objetiva (geração de dados para cobertura de todos os caminhos de um sistema em teste). Em relação à cobertura de caminhos dos três programas utilizados, a abordagem híbrida não apresentou resultados superiores a abordagem comparada. No entanto, a quantidade de avaliação de *fitness* (um dos principais fatores que aumentam o custo de um algoritmo de meta-heurística) obtida pelo algoritmo híbrido é menor que a abordagem mono objetiva. Sendo assim, o tempo para atividade de teste pode ser menor utilizando a abordagem híbrida. Além disso, como a otimização do problema ocorre individualmente para cada caminho do sistema em teste, essa abordagem pode ser paralelizada, diminuindo ainda mais o tempo para o teste de programas.

A grande vantagem do uso de algoritmos evolutivos é que eles possuem grande flexibilidade e adaptabilidade aos problemas reais e, dado isso, são encontrados vários outros estudos que exploram o uso de algoritmos genéticos para geração de dados de teste ([ZHANG; GONG, 2010](#); [GHIDUK; HARROLD; GIRGIS, 2007](#); [XIONG; SHI, 2010](#); [PINTO; VERGILIO, 2010](#); [CAO; HU; LI, 2009](#); [BOUCHACHIA, 2007](#); [MAHAJAN; KUMAR; PORWAL, 2012](#); [GHIDUK, 2014](#); [KHOR; GROGONO, 2004](#); [PANCHAPAKESAN; ABIELMONA; PETRIU, 2013](#)), os quais são propostos essencialmente para programas com características sequenciais.

Diferentemente dos programas sequenciais, a computação distribuída envolve processos concorrentes que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, sendo que esses processos podem ou não concorrer pelos mesmos recursos computacionais. Esse tipo de computação vem sendo cada vez mais empregada e necessária, haja visto as tecnologias atuais, com processadores com múltiplos núcleos e com o uso crescente de *clusters* de computadores. Soluções presentes em telefones móveis, controle de veículos ou sistemas empresariais, onde o acesso remoto a sistemas web corporativos e integrados sob o paradigma SOA (*Service-Oriented Architecture*) e *cloud computing* fazem uso da concorrência para o seu desenvolvimento ou para permitir a distribuição e replicação de processos e dados. Além disso, é comum o uso de programação concorrente para reduzir o tempo computacional em vários domínios, tais como: previsão de tempo, dinâmica de fluídos, processamento de imagens, química quântica, dentre outros.

A construção de aplicações concorrentes requer o uso de algumas primitivas fundamentais, as quais permitem: 1) a definição de quais processos serão executados em paralelo; 2) a iniciação finalização dos processos concorrentes e; 3) a coordenação entre os processos concorrentes enquanto estes estiverem executando ([ALMASI; GOTTLIEB, 1994](#); [WONG; LEI; MA, 2005](#)). Desses três tipos de primitivas, destacam-se as primitivas necessárias à interação (comunicação e sincronização) entre os processos, devido à sua frequência de utilização bem como impacto no desempenho final do programa concorrente. A interação entre processos ocorre segundo dois paradigmas distintos, normalmente, vinculados à organização de memória

disponível (ALMASI; GOTTLIEB, 1994; HWANG; BRIGGS, 1993). No paradigma de memória compartilhada os processos concorrentes têm acesso ao mesmo espaço de endereçamento e, portanto, podem compartilhar variáveis de memória. Nesse caso, a sincronização pode ocorrer explicitamente através de construção como semáforos ou monitores (TANENBAUM, 2007). No segundo paradigma, passagem de mensagens, são usadas memórias distribuídas e os processos concorrentes têm acesso apenas ao seu espaço de endereçamento, ou seja, não há o compartilhamento de variáveis. A sincronização com memória distribuída ocorre com troca de mensagens, as quais podem ser representadas por exemplo por primitivas do tipo *send/receive*. A execução de aplicações concorrentes pode utilizar ambos os paradigmas de passagem de mensagens e memória compartilhada na mesma aplicação. Isso pode acontecer com *threads* iniciadas no mesmo processador ou com o uso de plataformas heterogêneas que incluam máquinas com memória compartilhada e também distribuída.

A atividade de teste no contexto de programas concorrentes é considerada mais complexa. Além das dificuldades inerentes à atividade de teste, outras ocorrem devido, principalmente, ao comportamento não determinístico dessas aplicações. Múltiplas execuções de um programa concorrente com a mesma entrada podem executar diferentes sequências de sincronização e podem produzir diferentes resultados. Cabe a atividade de teste, nesse cenário, identificar se todas as sequências de sincronização possíveis foram executadas e se as saídas obtidas estão corretas. Essa característica difere os programas concorrentes dos programas sequenciais e precisa ser considerada durante a atividade de teste de software (SOUZA *et al.*, 2015).

A geração automática de dados de teste pode contribuir efetivamente para o teste de aplicações concorrentes diminuindo significativamente o esforço necessário para o teste e melhorando sua qualidade. No entanto, os desafios descritos anteriormente provenientes desse tipo de aplicação podem dificultar a realização dessa tarefa. O estudo sobre geração automática de dados de teste para programas concorrentes ainda é bastante recente. Os trabalhos existentes (NISTOR *et al.*, 2012; TIAN; GONG, 2014a; SILVA; SOUZA; SOUZA, 2014) tentam mapear as técnicas atuais para esse domínio de aplicações, porém sem resultados consolidados em termos de ferramentas disponíveis. Por outro lado, não há estudo que indique que as técnicas atuais de geração de dados de teste, propostas para programas sequenciais, não são adequadas para o contexto de programas concorrentes, ou pelo menos que indiquem as limitações de seu uso. Esses aspectos motivaram o desenvolvimento deste projeto de mestrado.

1.2 Objetivos

Considerando o contexto acima, o objetivo deste projeto de mestrado foi a definição e a execução de um estudo experimental visando avaliar diferentes técnicas de geração de dados de teste analisando sua aplicabilidade para o contexto de programas concorrentes. O intuito é encontrar evidências sobre o custo das técnicas, a eficácia em revelar defeitos e a cobertura dos

critérios de teste, avaliando esses aspectos para programas concorrentes.

1.3 Organização

O presente trabalho está disposto da seguinte forma: o Capítulo 2 apresenta os conceitos relacionados ao teste de software, os quais são fundamentais para o entendimento do trabalho. No Capítulo 3 são descritas as características sobre a geração automática de dados de teste. O Capítulo 4 apresenta o protocolo experimental concebido para este estudo, a análise dos resultados é descrita no Capítulo 5. Por último no Capítulo 6 são apresentadas as conclusões deste estudo.

TESTE DE SOFTWARE

2.1 Considerações Iniciais

O desenvolvimento de software apresenta desafios inerentes à complexidade e a qualidade dos produtos desenvolvidos. Produzir softwares por si só, não é uma tarefa trivial, processos, métodos e técnicas são propostos para auxiliar desenvolvedores e equipes de desenvolvimento na construção de aplicações de software (SOMMERVILLE, 2010). Nesse contexto, atividades de verificação e validação (V&V) são relevantes e primordiais, as quais devem ser empregadas ao longo do processo de desenvolvimento, de modo a garantir que o produto atenda a sua especificação e que o mesmo seja construído adequadamente (MYERS; SANDLER; BADGETT, 2011).

Dessa forma, o teste de software é uma das atividades de V&V que tem como objetivo revelar defeitos no produto em desenvolvimento ou no software em operação (DELAMARO; MALDONADO; JINO, 2007). Para isto, técnicas e critérios de teste são estabelecidos de forma a sistematizar essa atividade, apoiando na seleção e avaliação de conjuntos de teste. Este capítulo tem como objetivo apresentar a definição de conceitos importantes sobre o teste de software, detalhando melhor a técnica estrutural, a qual é utilizada neste projeto. Além disso, conceitos importantes sobre aplicações concorrentes e as características que dificultam o seu teste são consideradas neste capítulo.

O capítulo está disposto da seguinte forma. Na Seção 2.2 são apresentados os principais conceitos sobre o teste de software. A Seção 2.3 descreve a técnica estrutural para o teste de software destacando os principais critérios definidos. Na Seção 2.4 é descrito os conceitos de programas concorrentes, assim como os aspectos relacionados ao teste dos mesmos. Por último, são apresentadas as considerações finais do capítulo.

2.2 Conceitos Básicos

O teste de software tem como objetivo verificar se as funcionalidades de um sistema estão de acordo com suas especificações, sendo assim, ele atua na detecção de defeitos, anomalias e informações sobre atributos não funcionais do software, o qual consiste em uma atividade dinâmica, pois requer a execução do software ou modelo de forma controlada afim de observar o seu comportamento. Sendo assim, a principal meta do teste de software é revelar defeitos no programa em análise, ou seja, o teste bem sucedido é dado por entradas que resultem na falha do programa em teste (SOMMERVILLE, 2010).

Para compreensão da atividade de teste alguns conceitos básicos devem ser considerados, como exemplo, o **domínio** de um programa consiste no conjunto de todos os valores que um programa pode assumir como entrada. Por sua vez, cada elemento desse conjunto é chamado de **dado de teste**. Finalmente, o conjunto entre cada dado de teste e sua respectiva saída esperada é chamado de **caso de teste**. Um aspecto importante que deve ser analisado no teste de software são os tipos de problemas, erros ou defeitos que os softwares apresentam, ou seja, para cada termo existe uma definição diferente, as quais são descritas a seguir:

- Engano (*mistake*): ação incorreta tomada pelo programador;
- Defeito (*fault*): passo, processo ou definição de dados incorreta, incompleta, ausente ou extra que, ao ser executada, pode produzir um erro no programa;
- Erro (*error*): ocorre quando o valor obtido não coincide com o valor esperado;
- Falha (*failure*): evento manifestado em que o sistema viola suas especificações.

Em adição aos termos descritos, a atividade de teste compreende algumas etapas já definidas para sua execução, sendo elas: planejamento, projeto de casos de teste, execução e análise. Este cenário pode ser observado na Figura 1.

Figura 1 – Cenário típico da atividade de teste (Extraído de Delamaro, Maldonado e Jino (2007)).

Dado um programa **P**, tal que o seu domínio de entrada seja denotado por **D(P)**, são definidos elementos específicos desse domínio para execução de **P**. Em um cenário ideal todos os elementos desse domínio seriam selecionados, entretanto, esse tipo de teste, conhecido como teste exaustivo, na maioria das vezes, é impraticável devido a cardinalidade de **D(P)**. Sendo assim,

no planejamento do teste, o conjunto de casos de teste **T** normalmente é composto apenas por elementos com maior probabilidade de revelar defeitos. Dessa forma, o programa é executado com **T**, ao passo que seu resultado obtido é verificado. Quando o resultado produzido pela execução de **P** coincide com o resultado esperado, significa que nenhum erro foi identificado. Por outro lado, se para algum casos de teste, o resultado obtido for diferente do resultado esperado, então o defeito foi revelado. Em geral, essa atividade é realizada por um testador, como sugere a Figura 1, o qual baseado em uma especificação do programa **S(P)** verifica se o resultado obtido é o mesmo resultado esperado. Nesse caso, o testador desempenha o papel de um oráculo de teste, ou seja, aquele instrumento capaz de verificar se a saída obtida de uma determinada execução coincide com a saída esperada.

De acordo com [Ammann e Offutt \(2008\)](#), os testes podem ser derivados a partir de requisitos e especificações do software, artefatos de projeto ou até mesmo do código do programa. Diferentes tipos de teste podem ser aplicados, os quais testam o software em ocasiões diferentes durante o processo de desenvolvimento. Em geral, os seguintes tipos de teste são encontrados:

- **Teste de unidade:** tem como objetivo testar unidades menores de um programa, como funções, procedimentos, métodos e classes;
- **Teste de integração:** é realizado após o teste de unidade e tem como foco o teste na construção de estruturas do sistema. Esse tipo de teste é fundamental para verificar a consistência da interação entre as unidades do sistema;
- **Teste de sistemas:** ocorre após a conclusão do sistema, o objetivo é verificar se as funcionalidades do sistema foram implementadas corretamente e se estão de acordo com a especificação do programa;
- **Teste de regressão:** é aplicado após o desenvolvimento do software na fase de manutenção. Esse tipo de teste é utilizado devido aos erros que podem ser inseridos no programa a cada nova versão implementada.

Independentemente do tipo de teste, diferentes técnicas podem ser empregadas. De fato, o que difere uma técnica de outra é tipo de informação utilizada para estabelecer quais as entradas do domínio serão utilizadas para testar o programa. Como exemplo, a técnica funcional normalmente é utilizada para projetar casos de teste quando o código do programa é desconhecido (caixa preta) e, para testá-los é necessário fornecer entradas e avaliar as saídas geradas para estabelecer se estas estão em conformidade com a especificação do programa. Nesse contexto, um dos principais critérios da técnica funcional é o particionamento de equivalência. Esse critério é utilizado principalmente para reduzir o número de casos de teste, buscando garantir um bom conjunto de teste capaz de revelar defeitos no software sob teste. O particionamento de equivalência divide o domínio de entrada em classes de equivalência, as quais são definidas com base na especificação do programa. Uma classe de equivalência representa um conjunto de

estados válidos ou inválidos para as condições de entrada. Dessa forma, após definir as classes de equivalência são gerados casos de teste para os seus respectivos estados, tal que, para cada estado deve ser gerado pelo menos um caso de teste.

Na técnica baseada em defeitos são utilizados enganos típicos do processo de desenvolvimento do software para derivar requisitos de teste. Um requisito de teste é uma parte do software ou especificação que um caso de teste deve satisfazer ou cobrir (AMMANN; OFFUTT, 2008). O principal critério dessa técnica é a análise de mutantes (ou teste de mutação), a qual realiza diversas alterações no programa (injeção de defeitos) sob teste gerando um conjunto de programas alternativos (ou mutantes). Dessa forma, o desafio encontrado é definir um conjunto de casos de teste capaz de distinguir o comportamento do programa mutante em relação ao programa original.

Por sua vez, a técnica estrutural utiliza como informação aspectos internos do código para derivar os requisitos de teste (DELAMARO; MALDONADO; JINO, 2007). A Seção 2.3 apresenta os principais critérios utilizados na técnica estrutural, dado a sua importância para este projeto de mestrado.

2.3 Teste estrutural

A técnica estrutural é uma filosofia de projeto de casos de teste que utiliza a estrutura de código de programas para derivar casos de teste (PRESSMAN, 2010). Essa técnica estabelece os requisitos de teste com base em uma implementação específica, requisitando a execução de trechos ou componentes elementares do software.

Em geral, os critérios pertencentes a essa técnica utilizam uma representação do programa conhecido como grafo de fluxo de controle (GFC). Um GFC $G=(N,E,s)$ é um grafo dirigido com apenas um único nó de entrada s e um único nó de saída, onde N é o conjunto de nós e cada nó $n \in N$ representa um bloco de execução, tal que, um bloco de execução é composto por uma sequência máxima de comandos indivisíveis. Por sua vez, E representa o conjunto de arestas no grafo, as quais representam o controle de movimentação de um nó para o outro (GAO *et al.*, 2003; BARBOSA *et al.*, 2007).

Na Figura 2 é apresentado o GFC extraído de um pseudocódigo para determinar se um valor inteiro é um número composto. É possível perceber que alguns nós apresentam mais de uma aresta (arco) de saída, essas arestas representam fluxos de execuções que o programa pode assumir, comandos como *if* possuem mais de uma saída e portanto devem ser expressos no grafo.

O conceito de caminho no GFC é dado quando para uma sequência finita de nós $(n_1, n_2, n_3, \dots, n_k)$, $k \geq 2$, existe uma aresta de n_i para n_{i+1} para $i = 1, 2, \dots, k-1$. Em adição, existem diferentes tipos de caminhos que podem estar presentes em um GFC, são eles:

Figura 2 – Exemplo de um grafo de fluxo de controle (Adaptado de [Gao et al. \(2003\)](#)).

- Caminho simples: é um caminho em que todos os nós, exceto possivelmente o primeiro e o último, são distintos;
- Caminho livre de laço: é definido como um caminho em que todos os nós são distintos;
- Caminho completo: é um caminho em que o primeiro nó coincide com o primeiro nó do grafo e seu último nó é também o último nó do grafo.

Os critérios pertencentes a técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados dos programas em teste ([BARBOSA et al., 2007](#)). As seções seguintes descrevem os principais tipos de critérios estruturais.

2.3.1 Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam características de controle da execução, como comandos ou desvios, de um programa para derivar requisitos de teste.

Nesse contexto, executar todos os caminhos possíveis do fluxo de controle de um programa por meio de casos de teste, pode ser uma boa estratégia para o teste de programas. No entanto, apesar de desejável, existem desafios para realização dessa tarefa. A quantidade de caminhos lógicos que um programa possui pode ser demasiadamente grande, ocasionando em um longo período de tempo para o teste do mesmo. Dessa forma, o teste de caminhos exaustivos pode ser impraticável, se não impossível. Além disso, ainda que todos os caminhos de um programa fossem executados pelo menos uma vez, o programa ainda poderia apresentar erros de sensibilidade e também poderia não estar de acordo com sua especificação ([MYERS; SANDLER; BADGETT, 2011](#)).

Apesar disso, na literatura existem critérios definidos para análise do fluxo de controle de programas, sendo os mais conhecidos:

- Todos-Nós: esse critério requer que a execução do programa passe, ao menos uma vez, em cada vértice do GFC. Dessa forma, todos os comandos devem ser executados para satisfazer este critério;
- Todas-Arestas (Arcos): exige que cada aresta do grafo (ou fluxo de controle), seja exercitada pelo menos uma vez;
- Todos-Caminhos: esse critério requer que todos os possíveis caminhos do programa sejam exercitados.

É importante ressaltar, que a presença de laços em um programa pode implicar em um número de caminhos muito grande ou até mesmo infinito, o que pode inviabilizar a utilização desse critério. Desse modo, critérios de fluxo de controle que testam o programa considerando um número n de repetições foram definidos e podem ser empregados com uma solução para esses casos ([DELAMARO; MALDONADO; JINO, 2007](#)).

2.3.2 Critérios baseados em fluxo de dados

Os critérios de fluxo de dados baseiam-se no pressuposto de que para testar um programa de forma adequada, é necessário avaliar os fluxos de valores de dados. Dessa forma, é importante assegurar que os valores assumidos em um determinado ponto do código são criados e usados adequadamente, o que implica em um destaque para as definições e usos de variáveis. A definição (def) consiste em um local do programa onde um valor é atribuído a uma variável (ex: `var_a = 1`), já o uso de uma variável é dado por um local do código onde o valor da variável é acessado. Existem dois tipos de acesso a uma variável: uso predicativo (p-uso) e uso computacional

(c-uso). Um p-uso ocorre quando o acesso a um valor da variável ocorre em um comando predicativo, que determina um fluxo de execução, como um comando de laço ou de decisão (ex: $if(var_a > 2)$). Por sua vez, um c-uso é dado pelo uso do valor da variável em uma computação (ex: na $var_a = var_b - 1$; existe um uso computacional de var_b e uma definição de var_a) (AMMANN; OFFUTT, 2008).

Como base no contexto apresentado acima, Rapps e Weyuker (1985) definem uma família de critérios para o fluxo de dados, sendo os principais critérios:

- Todas-Definições: Exige que cada definição de variável seja exercitada por pelo menos uma vez, seja ela c-uso ou p-uso;
- Todos-Usos: Requer que todas as associações entre uma definição de uma variável e seus usos (c-uso ou p-uso) sejam exercitadas pelos casos de teste, por meio de pelo menos um caminho livre de definição;
- Todos-Du-Caminhos: este critério requer que toda associação entre uma definição de variável e seus subsequentes usos, seja exercitada por todos os caminhos livres de definição e de laços que contemplem esta associação.

A utilização de critérios estruturais como complexidade, fluxo de controle e fluxo de dados têm-se mostrado significativo para o teste de aplicações sequenciais. No entanto, no que tange a programas concorrentes, existem aspectos diferentes que devem ser considerados.

Os critérios baseados na complexidade utilizam dados sobre a complexidade do software para gerar requisitos de teste. Uma medida de complexidade muito utilizada é a complexidade ciclomática (MCCABE, 1976), que pode ser definida da seguinte forma: se n é o número de estruturas de seleção e repetição no programa, a complexidade ciclomática do programa é $n+1$ (WAZLAWICK, 2013).

Em geral, programas que possuem complexidade ciclomática menor ou igual a 10 são simples e fáceis de testar, já programas que possuem complexidade acima de 50 são demasiadamente complexos para a atividade de teste (WAZLAWICK, 2013).

Isso limita a aplicação desse critério a unidades de código menores. Uma vantagem desse critério é que ele garante a cobertura de todos os comandos e arestas do GFC de um programa, uma característica necessária para os critérios de teste estruturais.

2.4 Programação Concorrente

Tradicionalmente, programação paralela/concorrente é utilizada principalmente para aplicações que necessitam de computação de alto desempenho, tais como simulações numéricas e processamento de imagens. No entanto, com o surgimento de processadores multi-core em

computadores pessoais, dispositivos móveis e *clusters* de computadores, o crescimento do grau de paralelismo disponível forneceu novas perspectivas no desenvolvimento de software paralelo (BERKA; HAGENAUER; VAJTERSIC, 2011).

Diferentemente dos programas sequenciais, a computação paralela envolve processos concorrentes que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, onde os processos podem ou não concorrer pelos mesmos recursos computacionais. Essa característica impõe novos desafios à atividade de teste, os quais precisam ser apropriadamente tratados. Desse modo, nesta seção será apresentada uma visão geral sobre programação paralela e concorrente e sobre os trabalhos do grupo de pesquisa do ICMC/USP na área de teste de programas concorrentes.

2.4.1 Conceitos Básicos sobre Programação Concorrente

Processos concorrentes são dois ou mais processos que começaram a executar e ainda não finalizaram a sua execução, concorrendo por recursos do sistema, como processadores, memórias e E/S. Podem estar executando em um processador ou em diferentes processadores. Se estes processos executam em diferentes processadores, ao mesmo tempo, eles são ditos paralelos. Assim, observa-se que processos concorrentes podem estar executando em apenas um processador e, dessa forma, não são paralelos. De fato, processos concorrentes são mais abrangentes e processos paralelos são um tipo especial de processos concorrentes.

Os processos concorrentes são usados em diferentes contextos: computação paralela, sistemas distribuídos e sistemas operacionais. Quando os processos concorrentes são usados no contexto de um sistema distribuído e não no contexto da computação paralela, visando, por exemplo, ao compartilhamento de recursos, tais processos são ditos distribuídos e não paralelos.

O berço dos processos concorrentes como conhecidos atualmente é o surgimento dos sistemas operacionais multiprogramados (HANSEN, 2002). Posteriormente as mesmas técnicas usadas para a construção dos programas concorrentes em sistemas operacionais foram empregadas na computação paralela e nos sistemas distribuídos.

Os programas concorrentes representam o código fonte (algoritmos), em uma determinada linguagem de programação, com primitivas adicionais para determinar quando os processos concorrentes deverão iniciar, finalizar e interagir.

A Computação Paralela compreende o uso do computador paralelo para, principalmente, reduzir o tempo necessário para resolver um problema computacional. O computador paralelo é um sistema computacional com múltiplos processadores que oferece suporte à programação concorrente. O computador paralelo também pode ser referenciado como plataforma paralela, plataforma distribuída, entre outros. Duas importantes categorias de computadores paralelos são os multicomputadores e os multiprocessadores, ambos pertencentes à categoria MIMD (Flynn, 1966). Os multicomputadores têm memórias distribuídas e permitem aos processadores

apenas acesso local às suas memórias. Os multiprocessadores, por sua vez, compartilham um mesmo espaço de endereçamento, oferecendo aos processadores uma visão global da memória disponível. Estas duas categorias são consideradas neste trabalho.

A Programação Concorrente é o ato de programar em uma linguagem para determinar quando e como os programas concorrentes podem ser executados, na forma de processos concorrentes. A programação concorrente também determina como os processos interagem para solucionar um problema específico. Os termos programação concorrente e programação paralela são sinônimos.

Uma aplicação não centralizada, isto é, aquela que utiliza processos concorrentes, necessita ser decomposta em tarefas menores, estas executadas concorrentemente depois que forem atribuídas a processos ou *threads*¹, que são então escalonadas a unidades computacionais físicas para a execução. As tarefas de uma aplicação podem ser independentes, mas também podem depender uma da outra, resultando em dependências de dados ou controle de execução, ou seja, se uma tarefa necessita de dados ou da execução de uma outra tarefa, a execução desta tarefa só pode iniciar após a conclusão da segunda tarefa ou até que essa envie os dados necessários para execução (RÜNGER; RAUBER, 2013).

Para que os processos ou *threads* possam transmitir dados ou organizar a ordem de execução eles dependem de mecanismos de comunicação e sincronização respectivamente. A comunicação permite que a execução de um processo influencie a execução de outro. Dessa forma, a comunicação pode ocorrer com o uso de variáveis compartilhadas (variáveis que podem ser referenciados por mais de um processo) ou por passagem de mensagem. Por outro lado, a sincronização pode ser vista como um conjunto de restrições na ordenação de eventos. Dessa forma, o programador pode empregar mecanismos de sincronização para atrasar a execução de um processo seguindo determinadas restrições (ANDREWS; SCHNEIDER, 1983).

Programas concorrentes necessitam proteger regiões críticas, ou seja, endereços de memória que possam ser acessados e escritos por mais de um processo. Programas concorrentes que executam em múltiplas CPUs, cada qual com seu espaço de endereçamento, necessitam de troca de mensagens para comunicar e sincronizar. Quando a programação segue o paradigma de passagem de mensagem as aplicações normalmente são compostas por processos. Ambientes de memória compartilhada são compostos por *threads* (TANENBAUM, 2007).

Dois modelos de comunicação e sincronização mais populares para processadores paralelos são os modelos de passagem de mensagem e memória compartilhada. Cada modelo apresenta características que refletem suposições sobre a máquina subjacente. O modelo de memória compartilhada utiliza a memória para comunicação entre os processos, e normalmente é associada a multiprocessadores de memória compartilhada (ver Figura 3a). O modelo de passagem de mensagem utiliza mensagens para comunicação entre processos e é tipicamente associado a

¹ Thread é o fluxo de controle de um processo que permite que múltiplas execuções ocorram no mesmo ambiente do processo com um grau de independência uma da outra (TANENBAUM, 2007).

multicomputadores com memória distribuída (ver Figura 3b)(LEBLANC; MARKATOS, 1992).

Figura 3 – Modelo de memória em arquiteturas paralelas (Adaptado de Barney (2010)).

O modelo de memória compartilhada é uma abstração do processador genérico centralizado. O hardware subjacente é assumido como sendo um conjunto de processadores, cada um com acesso à mesma memória compartilhada (veja Figura 4). Como eles possuem acesso aos mesmos locais de memória, processadores podem interagir e sincronizar um com o outro por meio de variáveis compartilhadas.

Figura 4 – Modelo de memória compartilhada de computação paralela (Adaptado de Quinn (2003)).

Quando dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo o resultado depende de quem executa precisamente e quando, existe uma condição de corrida (*race condition*). Quando existem condições de corrida (ou de disputa) a proteção da região crítica (parte do código que há acesso à memória compartilhada) pode evitar que dois processos utilizem ao mesmo tempo um só recurso. O acesso exclusivo a espaços de memória compartilhada é fundamental para comunicação entre processos, principalmente quando ocorre a escrita de variáveis. Garantir a exclusividade ao acesso evita-se que dados sejam perdidos devido à sobreposição da escrita e/ou utilização de dados antigos. Para garantir a eficácia da sincronização foram desenvolvidos mecanismos como monitores e semáforos.

Um semáforo é um mecanismo de sincronização proposto por Dijkstra (1965). Ele é composto por uma variável inteira normalmente iniciada em 1 que é associada a uma lista de processos inicialmente vazia. Dijkstra propôs a existência de duas operações, *down* e *up*. A

operação *down* sobre um semáforo verifica se seu valor é maior que 0. Em caso afirmativo, o valor é decrementado e o processo continua sua execução. Se o valor for 0, o processo será colocado como inativo (ou dormindo) antes mesmo de realizar a operação de *down*. A operação *up* incrementa o valor de um semáforo. Se um ou mais processos estiverem inativos neste semáforo, um deles é escolhido pelo sistema de acordo com uma ordem de prioridade para continuar sua execução e realizar a operação de *down* (TANENBAUM, 2007).

O mecanismo de monitores consiste em uma coleção de rotinas, variáveis e estruturas de dados agrupados em um tipo especial de módulo ou pacote. Os processos podem fazer chamadas de rotinas em um monitor quando necessitarem porém, não podem ter acesso direto às estruturas internas de dados ao monitor a partir de rotinas declaradas fora dele.

Além disso, o modelo de memória compartilhada suporta paralelização incremental, um processo de transformação de um programa sequencial em um programa paralelo. A capacidade do modelo de memória compartilhada para apoiar paralelização incremental é uma de suas maiores vantagens sobre o modelo de passagem de mensagens. Ele permite com a execução de um programa sequencial, classificar os blocos do programa de acordo com o quanto eles consomem, paralelizar cada bloco passível de execução paralela, e finalizar quando o esforço necessário para atingir melhorias de desempenho não é garantido (QUINN, 2003).

O modelo ou paradigma de passagem de mensagem é composto por duas principais características que definem esse tipo de comunicação entre processos. A primeira delas é o espaço de endereçamento particionado, onde para cada processo ou *thread* é alocada uma partição lógica da memória (veja Figura 5). Em segundo, o modelo de passagem de mensagens fornece suporte apenas à paralelização explícita, ou seja, o paralelismo fica a cargo do programador responsável pelo desenvolvimento dessa aplicação. Essa última característica, provê menor abstração ao programador no entanto, quando programas são escritos adequadamente esse aspecto pode torna-se um grande potencial da aplicação pois, pode atingir um alto nível de desempenho e fornecer escalabilidade para um número muito grande de processos (GRAMA *et al.*, 2003).

Para realizar a comunicação e a sincronização de processos por meio de passagem de mensagem, os programas necessitam implementar primitivas que viabilizem a ordem de execução e a forma como as mensagens são trocadas. Existem diferentes definições na literatura para os conceitos de síncrono/assíncrono e bloqueante/não-bloqueante. Neste trabalho utilizaremos a notação apresentada por Souza, Vergilio e Souza (2007).

A troca de mensagens é realizada por meio de primitivas de *send/receive* ou por meio de chamadas de procedimentos remotos. A sintaxe das primitivas é: *send(msg, destino)* e *receive(msg/origem)*.

Em vista disso, a comunicação entre processos pode ser considerada síncrona quando o *send* (ou *receive*) aguarda que o *receive* (ou *send*) seja executado. Para garantir que as mensagens sejam enviadas esse modelo utiliza confirmações (*ack*) a cada troca de mensagens. Dessa

Figura 5 – O modelo de passagem de mensagens onde cada processador possui seu próprio espaço de endereçamento e uma rede de interconexão de apoio entre os processadores (Adaptado de [Quinn \(2003\)](#)).

forma, podemos considerar que toda comunicação síncrona é bloqueante. Por outro lado, na comunicação assíncrona o comando *send* (ou *receive*) não necessita aguardar o comando *receive* (ou *send*) para continuar a sua execução. Para implementar a comunicação assíncrona, algumas bibliotecas oferecem a utilização de *buffers*. Neste contexto, as mensagens são copiadas/lidas do *buffer*, que fornece aos processos uma falsa impressão de sincronismo.

No que tange aos conceitos de bloqueante/não-bloqueante, os mesmos podem ser similares aos de síncrono/assíncrono se não houver *buffer* nem no transmissor nem no receptor da mensagem. Ainda assim, um *send* pode ser considerado bloqueante se garantir que o *buffer* com a mensagem possa ser reutilizado logo após a liberação do *send* sem que a mensagem seja perdida. Para que um *receive* seja bloqueante ele deve esperar até que a mensagem esteja disponível no *buffer*, e somente após o recebimento continuar sua execução. Neste dois casos, não é possível garantir a sincronização mas sim assegurar o envio e recebimento de mensagens.

2.4.2 Teste de Programas Concorrentes

Conforme apresentado nas seções anteriores, os programas concorrentes apresentam características adicionais em relação aos programas sequenciais. Essas características envolvem mecanismos como comunicação e sincronização, os quais são essenciais para realização das tarefas concorrentes. Apesar dos benefícios provenientes dos mecanismos empregados, existe um custo adicional no teste dessas aplicações, pois os defeitos originados das tarefas concorrentes demandam maior esforço para serem revelados. Diante disso, um dos principais objetivos no teste de aplicações concorrentes é identificar defeitos relacionados à comunicação, paralelismo e à sincronização ([SOUZA; SOUZA; ZALUSKA, 2014](#)).

O principal desafio existente no teste de programas concorrentes é o não-determinismo, o qual ocorre quando em duas ou mais execuções com a mesma entrada, saídas diferentes são

obtidas. Isso pode ocorrer devido à ordem de sincronização que pode ocorrer entre os processos concorrentes. Nesse caso, isso não caracteriza uma situação de erro, porém é necessário garantir que o programa funciona corretamente para todas as possíveis sincronizações (TIAN; GONG, 2014a; SOUZA; VERGILIO; SOUZA, 2007).

Nesse contexto, as rotinas de programas concorrentes podem apresentar diferentes tipos de defeitos, os quais ocasionam diferentes falhas. Desses, dois são mais comuns entre as aplicações, sendo eles, erro de observabilidade e travamento. O erro de observabilidade está associado ao ambiente de teste e pode ocorrer quando o testador ou ferramenta de teste não possui o controle do comportamento do programa executado em paralelo. Como exemplo esse tipo de erro pode ocorrer quando um processo depende da sincronização com outro processo para prosseguir a sua execução. Assim, se o modelo utilizado não fornecer mecanismos que permitam o bloqueio do processo até que o sincronismo seja realizado, esse programa pode não garantir um resultado correto em todas as execuções, pois neste caso o tempo dispendido por cada processo determinará se a sincronização será executada corretamente ou não. Por sua vez, o erro de travamento pode ocorrer quando um processo não se comunica com o processo correto, consequentemente um processo ficará a espera de uma sincronização infinitamente ou até que um tempo limite seja excedido, pois a mensagem esperada não será recebida.

Além desses, outros tipos de defeitos também podem ser encontrados em taxonomias de defeitos, os quais são empregados, principalmente, em *benchmarks* de defeitos e na proposição de operadores de mutação. Wu e Kaiser (2011) descrevem um modelo geral de defeitos para programas concorrentes (Quadro 1).

Os defeitos citados estão presentes em diversas linguagens de programação, porém, existem investigações específicas para defeitos decorrentes de aplicações concorrentes desenvolvidas na linguagem Java. Farchi, Nir e Ur (2003) descrevem uma taxonomia para categorização de padrões de defeitos, os quais consistem na forma literária que descreve um defeito que ocorre comumente na implementação de projetos de software. Esses padrões são definidos como: (Note que na descrição dos padrões de defeitos, as rotinas descritas no ambiente de *threads* também podem ser abstraídas para o contexto de processos.)

1. **Código assumido como protegido:** Um código é dito protegido para um programa concorrente **P**, se para qualquer *interleaving*² e qualquer execução de um segmento de código de uma *thread* presentes em **P**, não existe interferência por qualquer outra *thread*. Isto é, nenhuma outra *thread* executa o mesmo segmento de código simultaneamente, de forma que a primeira *thread* seja capaz de executar todos os eventos concorrentes presentes nesse código. Este padrão de defeito pode ocorrer quando um segmento de código é assumido como protegido, porém, devido a alguns fatores, o código não está protegido. Neste contexto, destacam-se os fatores:

² *Interleaving* é dado pela sequência de eventos concorrentes que podem ocorrer em uma execução.

Quadro 1 – Defeitos típicos de programas concorrentes.

Defeito	Descrição
<i>Data Race</i>	Ocorre quando múltiplas <i>threads</i> realizam operações de escrita e leitura em um mesmo arquivo, e o resultado da execução depende exclusivamente da ordem em que o arquivo é acessado.
Memory Inconsistency	Ocorre quando diferentes <i>threads</i> possuem visões inconsistentes de uma mesma variável, em outras palavras, dada uma variável " <i>var</i> " e duas <i>threads</i> " <i>t1</i> " e " <i>t2</i> ", o valor armazenado em <i>var</i> é considerado diferente entre <i>t1</i> e <i>t2</i> .
Atomicity Violation	Originada da execução concorrente de múltiplas <i>threads</i> que transgridem a atomicidade de uma determinada região do código, ou seja, dado um trecho do código que deve ser executado sem interrupção, por algum motivo o mesmo é interrompido, assim, alguma computação pode ser perdida durante essa interferência.
DeadLock	Ocorre quando múltiplas <i>threads</i> são bloqueadas para sempre, isto é, dadas três <i>threads</i> " <i>t1</i> ", " <i>t2</i> " e " <i>t3</i> ", em um dado momento <i>t1</i> pode ficar bloqueada na espera de uma sincronização com <i>t2</i> , posteriormente <i>t2</i> pode ser bloqueada antes de sincronizar com <i>t1</i> , na espera de uma sincronização com <i>t3</i> , nesse momento se <i>t3</i> for bloqueada, esperando por exemplo por uma sincronização com <i>t1</i> , ambas as <i>threads</i> ficarão bloqueadas para sempre.
Livelock	A ação de uma <i>thread</i> pode ser realizada em resposta à ação de outra <i>thread</i> , assim, se a ação de uma <i>thread</i> <i>t2</i> ocorrer em resposta de <i>t1</i> , ao passo que um ciclo de ações e reações seja iniciado, ambas as <i>threads</i> ficarão executando infinitamente ocasionando um <i>Livelock</i> , apesar disso, diferentemente do <i>deadlock</i> , as <i>threads</i> não serão bloqueadas.
Starvation	Ocorre quando uma <i>thread</i> está bloqueada, impedida de acessar recursos compartilhados, e nunca é escalonada para acessar regiões compartilhadas, consequentemente não consegue progredir em suas atividades.
Suspension	Ocorre quando uma <i>thread</i> suspende suas atividades ou espera indefinidamente.

- **Operações não atômicas assumidas como atômicas:** Este defeito pode ocorrer devido ao nível de abstração de uma linguagem de programação. Como exemplo, na linguagem Java, a operação " $x++$ " pode ser considerada como protegida, pois consiste em apenas uma operação. Porém, a nível de *bytecode* essa operação é dividida em três eventos (leitura de x , incremento de x e escrita de x), as quais podem admitir a troca de contextos. Sendo assim, essa operação pode ser dada como atômica se considerada de um alto nível de abstração.
- **Acesso em dois estágios:** Existem casos em que a sequência das operações necessitam ser protegidas, assim, esse defeito pode ocorrer quando um programador assume que proteger cada operação separadamente é o suficiente.
- **Bloqueio incorreto ou ausência de bloqueio:** Um segmento de código é protegido por um bloqueio desde que outras *threads* não obtenham a mesma instância do bloqueio durante a execução. Contudo, a interferência entre *threads* em um segmento de código a ser protegido pode ocorrer. Como exemplo, considere uma variável global x com valor inicial 0, para a qual uma *thread* executa a operação $\text{synchronized}(o)\{x++\}$, em adição, uma segunda *thread* realiza a operação $x++$. Como resultado das operações, o valor de x pode ser 1, pois a operação $\text{synchronized}(o)\{x++\}$ não está protegida. Consequentemente x é atualizada duas vezes com o mesmo valor.

- **Bloqueio de dupla checagem:** Está relacionado a inicialização de objetos em programas concorrentes. Quando um objeto é inicializado, a *thread* copia os campos desse objeto para sua área local, porém nem todos os campos são copiados para a pilha, em decorrência disso, o objeto é parcialmente inicializado. Dessa forma, assim como no primeiro caso, a inicialização de um objeto é influenciada pelo nível de abstração da linguagem de programação.
2. **Interleavings previstos para nunca ocorrer:** Este padrão de defeito parte do pressuposto que um programador assume, erroneamente, que um determinado *interleaving* nunca irá ocorrer. Este padrão pode ocorrer por alguns fatores, sendo eles:
- **O *sleep()*:** Considere uma *thread* (pai) que deve aguardar o resultado de outra *thread* (filha), neste caso, o programador pode erroneamente utilizar uma primitiva *sleep* no pai, acreditando que a filha terá tempo suficiente para executar antes que o *sleep* seja encerrado. Contudo, o defeito pode ocorrer quando, ainda com o *sleep*, a *thread* pai for mais rápida que a *thread* filha. Apesar disso, esse defeito pode ser facilmente mitigado utilizando a primitiva *join()*.
 - **Perda de um *notify()*:** Se uma primitiva *notify()* é executada antes de sua operação correspondente *wait()*, o *notify()* será perdido e não terá efeito. Consequentemente, o código executando o *wait()* pode não ser acordado.
3. **Thread bloqueada ou morta:** Ocorre quando algum *interleaving* bloqueia indefinidamente uma aplicação, o qual pode ocorrer nos seguintes casos:
- **Bloqueio de uma região crítica:** Pode ocorrer quando uma *thread* acessa uma região crítica e, por algum motivo, nunca deixa a região crítica. Em outras palavras, uma *thread* pode entrar na região crítica e executar uma operação bloqueante de entrada e saída, que não pode ser atendida. Em decorrência disso, outras *threads* não poderão acessar a região crítica.
 - **Thread órfã:** Em um código de múltiplas *threads*, uma das *threads* pode ser definida como mestre. Neste caso, o gerenciamento é realizado por meio da comunicação entre as *threads*, assim, se a *thread* mestre finalizar a sua execução irregularmente, as demais *threads* continuarão executando a espera de mensagens da *thread* mestre, consequentemente o sistema apresentará uma falha.

Assim como nos programas sequenciais, o teste de programas concorrentes utiliza um modelo para abstrair os aspectos estruturais dos programas. Para isso, existem modelos baseados no fluxo de controle e no fluxo de dados para representação dos programas concorrentes.

O modelo proposto por Souza *et al.* (2008) considera um número n fixo e conhecido de processos paralelos criados durante a inicialização da aplicação paralela. Nesse modelo, a

comunicação entre processos ocorre de duas formas: ponto a ponto, onde um processo pode enviar uma mensagem para outro usando primitivas como *send* e *receive*, e comunicação coletiva, na qual um processo pode enviar uma mensagem para todos os processos da aplicação (ou para um grupo de processos) (SOUZA *et al.*, 2008).

Dessa forma, para um determinado programa paralelo *ProgPar* composto pelos processos $ProgPar = P^0, P^1, \dots, P^{n-1}$, cada processo P_i possui seu GFC, construído da mesma forma que em programas sequenciais. Sendo assim, um nó n pode ou não estar associado a uma função de comunicação do tipo *send* ou *receive*. Com base nisso, um Grafo de Fluxo de Controle Paralelo (GFCP) é construído para *ProgPar*, sendo composto pelos GFC de todos os processos presentes em *ProgPar* e pelas arestas de comunicação dos processos paralelos. Vale ressaltar que, neste modelo o conjunto de nós e arestas são representados por N e E , respectivamente. Um nó i em um processo p é representado pela notação n_i^p . Além disso, dois subconjuntos de nós são definidos: N_s , o qual é formado por nós que contém funções de envio de mensagens e N_r que é formado por nós que contém primitivas de recebimento de mensagens.

Para cada $n_i^p \in N_s$, um conjunto R_i^p é associado, o qual apresenta os possíveis nós que recebem a mensagem enviada pelo nó n_i^p . Já as arestas do GFCP podem ocorrer de duas formas:

- Arestas intraprocessos (E_i): são arestas que estão internas a um processo p ;
- Arestas interprocessos (E_s): são arestas que representam a comunicação entre processos distintos.

Assim como no teste de programas sequenciais, o teste de programas concorrentes verifica o uso de variáveis para revelar defeitos nas aplicações. Sendo assim, são definidos três tipos de usos de variáveis (SOUZA; VERGILIO; SOUZA, 2007):

1. Uso computacional (c-uso): ocorre em uma computação, relacionado a um nó n_i^p do GFCP;
2. Uso predicativo (p-uso): ocorre em uma condição (predicado), associado a comandos de fluxo de controle, em arestas intraprocessos (n_i^p, n_j^p) do GFCP;
3. Uso comunicacional (s-uso): ocorre em comando de sincronização (*send* e *receive*), em arestas interprocessos $(n_i^{p1}, n_j^{p2}) \in E_s$.

Com base nas informações acima, três tipos de associações são requeridas pelos critérios estruturais no contexto de programas concorrentes:

- associação c-uso: dada pela tripla (n_i^p, n_j^p, x) onde $x \in def(n_i^p)$, e n_j^p possui um c-uso de x e existe um caminho livre de definição com relação a x de n_i^p ;

- associação p-uso: dada pela tripla $(n_i^p, (n_j^p, n_k^p), x)$ onde $x \in \text{def}(n_i^p)$, e (n_j^p, n_k^p) possui um p-uso de x e existe um caminho livre de definição em relação a x de n_i^p para (n_j^p, n_k^p) ;
- associação s-uso: dada pela tripla $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x)$ onde $x \in \text{def}(n_i^{p1})$, e (n_j^{p1}, n_k^{p2}) possui um s-uso de x e existe um caminho livre de definição com relação a x de n_i^{p1} para (n_j^{p1}, n_k^{p2}) .

Em adição aos tradicionais usos, uso computacional (c-uso) e uso predicativo (p-uso), [Souza et al. \(2013\)](#) definem novos possíveis usos relacionados a memória compartilhada e passagem de mensagem:

- **uso em mensagem (m-use)**: ocorre quando uma mensagem com os dados é enviada de uma *thread* a outra, podendo pertencer a um mesmo processo ou a um processo distinto;
- **uso computacional compartilhado (s-c-uso)**: difere do c-uso por considerar a comunicação implícita entre *threads* em que o par definição e uso pode ocorrer em *threads* distintas.
- **uso predicativo compartilhado (s-p-uso)**: ocorre quando uma *thread* utiliza dados compartilhados em uma condição.

No contexto de programas concorrentes a definição de critérios considera, principalmente, os aspectos relacionados a interação entre processos (ou *threads*), apesar disso, assim como nos programas sequenciais, o fluxo de dados e o fluxo de controle também são analisados durante o teste.

A associação c-uso, associação p-uso e associação m-uso, presentes no fluxo de dados sequencial, apresentam caminhos intra-*threads*, ou seja, em uma única *thread*. Na comunicação por passagem de mensagem, em que diferentes *threads* estão presentes, são empregadas as seguintes associações:

- **associação intra-m-c-uso**: definida pela tupla $\langle n_1^{p,tk}, (n_2^{p,tk}, n_3^{p,tl}), n_4^{p,tl}, d1, d2 \rangle$ em que $\langle n_1^{p,tk}, n_2^{p,tk}, d1 \rangle$ é uma associação m-uso e $\langle n_3^{p,tl}, n_4^{p,tl}, d2 \rangle$ é uma associação c-uso;
- **associação intra-m-p-uso**: definida pela tupla $\langle n_1^{p,tk}, (n_2^{p,tk}, n_3^{p,tl}), (n_4^{p,tl}, n_5^{p,tl}), d1, d2 \rangle$ em que $\langle n_1^{p,tk}, n_2^{p,tk}, d1 \rangle$ é uma associação m-uso e $\langle n_3^{p,tl}, (n_4^{p,tl}, n_5^{p,tl}), d2 \rangle$ é uma associação p-uso;
- **associação inter-m-c-uso**: definida pela tupla $\langle n_1^{p_i,tk}, (n_2^{p_i,tk}, n_3^{p_j,tl}), n_4^{p_j,tl}, d1, d2 \rangle$ em que $\langle n_1^{p_i,tk}, n_2^{p_i,tk}, d1 \rangle$ é uma associação m-uso e $\langle n_3^{p_j,tl}, n_4^{p_j,tl}, d2 \rangle$ é uma associação c-uso;
- **associação inter-m-p-uso**: definida pela tupla $\langle n_1^{p_i,tk}, (n_2^{p_i,tk}, n_3^{p_j,tl}), (n_4^{p_i,tl}, n_5^{p_j,tl}), d1, d2 \rangle$ em que $\langle n_1^{p_i,tk}, n_2^{p_i,tk}, d1 \rangle$ é uma associação m-uso e $\langle n_3^{p_j,tl}, (n_4^{p_i,tl}, n_5^{p_j,tl}), d2 \rangle$ é uma associação p-uso.

Em vista dos aspectos observados, [Souza et al. \(2008\)](#) definem duas famílias de critérios de testes estruturais para programas concorrentes, sendo elas: critérios baseados em fluxo de controle e comunicação e critérios baseados em fluxo de dados e em passagem de mensagens.

Os critérios de fluxo e controle são baseados no teste de comunicações presentes no GFCP. Os principais critérios desta classe são:

- **Todos-Nós-s:** Este critério requer que todos os nós do conjunto N_s sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- **Todos-Nós-r:** Requer que todos os nós do conjunto N_r sejam exercitados pelo menos uma vez pelo conjunto de teste.
- **Todos-Nós:** Este critério requer que todos os nós do conjunto N sejam exercitados pelo menos uma vez pelo conjunto de teste.
- **Todas-Arestas-s:** Requer que todas as arestas do conjunto E_s sejam exercitadas pelo menos uma vez pelo conjunto de teste.
- **Todas-Arestas:** Requer que todas as arestas do conjunto E sejam exercitadas pelo menos uma vez pelo conjunto de teste.

Os critérios baseados no fluxo de dados utilizam a análise desse fluxo como fonte de informação para derivar os requisitos de teste, sendo eles:

- **Todas-Defs:** Requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação definição-uso (c-uso ou p-uso) com relação a x seja exercitada pelo conjunto de casos de teste;
- **Todas-Defs/s:** Requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação s-c-uso ou s-p-uso com relação a x seja exercitada pelo conjunto de casos de teste. Caso não exista associação s-c-uso ou s-p-uso com relação a x , é requerido qualquer outro tipo de associação interprocessos em relação a x ;
- **Todos-c-Usos:** Requer que todas as associações c-usos sejam exercitadas pelo conjunto de casos de teste;
- **Todos-p-Usos:** Requer que todas as associações p-usos sejam exercitadas pelo conjunto de casos de teste;
- **Todos-s-Usos:** Requer que todas as associações s-usos sejam exercitadas pelo conjunto de casos de teste;
- **Todos-s-c-Usos:** Requer que todas as associações s-c-usos sejam exercitadas pelo conjunto de casos de teste;

- **Todos-s-p-Usos:** Requer que todas as associações s-p-usos sejam exercitadas pelo conjunto de casos de teste.

Por fim [Souza et al. \(2013\)](#) introduziram novos critérios para o teste de programas concorrentes que realizam comunicação por meio dos paradigmas de passagem de mensagem e memória compartilhada.

- **Todos Usos (*All uses*):** requer que o conjunto de teste execute caminhos que cubram todas as associações de c-uso, p-uso, m-uso, s-c-uso e s-p-uso.
- **Todos Usos Compartilhados (*All Shared Uses*):** requer que o conjunto de teste execute caminhos que cubram todas as associações s-c-uso e s-p-uso, em relação a todas as variáveis compartilhadas do programa. O intuito deste critério é revelar defeitos na comunicação no paradigma de memória compartilhada.
- **Todos Usos Intra Mensagem (*All Intra Message Uses*):** requer que o conjunto de teste execute caminhos que cubram todas as associações intra-m-c-uso e intra-m-p-uso, em relação a todas as variáveis de comunicação de uma thread. O objetivo deste critério é revelar defeitos provenientes da comunicação de diferentes *threads* de um mesmo processo com o paradigma de passagem de mensagem.
- **Todos Usos Inter Mensagem (*All Inter Message Uses*):** requer que o conjunto de teste execute caminhos que cubram todas as associações inter-m-c-uso e inter-m-p-uso, em relação a todas as variáveis de comunicação entre dois processos do programa. O objetivo desse critério é revelar defeitos que podem ocorrer na comunicação de diferentes *threads* que pertençam a processos distintos com o paradigma de passagem de mensagem.

2.4.3 A Ferramenta de Teste ValiPar

A atividade de gerar GFCs, executar os dados de teste sobre um programa, verificar as saídas e quais elementos requeridos foram cobertos é uma tarefa dispendiosa. Dessa forma, a automatização dessa função pode diminuir os esforços gastos na atividade de teste e ainda agilizar esse processo. Com este propósito, a ferramenta Valipar tem como objetivo apoiar a aplicação de modelos e critérios de teste em programas concorrentes. A ValiPar utiliza a técnica estrutural para testar programas concorrentes que utilizam mecanismos para comunicação baseados nos paradigmas de passagem de mensagem e memória compartilhada. A ValiPar possui diferentes versões desenvolvidas para apoiar o teste de programas concorrentes, as quais distinguem-se, principalmente, pela linguagem de programação utilizada. Neste projeto, a versão utilizada considera o teste de aplicações desenvolvidas na linguagem Java. Diferentemente das versões anteriores, essa versão considera a unificação dos modelos de passagem de mensagem e memória compartilhada ([SOUZA et al., 2013](#); [PRADO et al., 2015](#)).

A ferramenta ValiPar é composta por cinco módulos, os quais recebem entradas específicas. Uma arquitetura da ValiPar é apresentada na Figura 6, onde os módulos são representados pelos retângulos enquanto as entradas são representadas pelas elipses.

Figura 6 – Arquitetura ferramenta ValiPar.

O módulo **ValiInst** recebe como entrada o programa concorrente a ser testado. Este módulo realiza a análise estática do programa e extrai as informações de fluxo de controle, fluxo de dados e fluxo de sincronização, os quais serão utilizados pelos demais módulos durante a atividade de teste. Além disso, o módulo ValiInst realiza a instrumentação do código fonte, tarefa necessária para permitir que posteriormente um rastro da execução seja gerado a partir da execução de um dado de teste.

Por sua vez, o módulo **ValiElem** recebe como entrada as informações de fluxo e o PCFG gerado pelo módulo ValiInst. Este módulo tem como objetivo gerar os elementos requeridos para os critérios de teste. Em adição, é gerado um descritor responsável por estabelecer possíveis caminhos que podem cobrir cada elemento requerido.

O módulo **ValiExec** 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 sucedida. Este módulo também permite 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 durante a execução, já com a execução determinística o usuário pode repetir uma sequência de sincronizações executada

previamente, considerando os mesmos dados de teste.

O módulo **ValiEval** utiliza os dados de rastro e os elementos requeridos gerados pelo módulo ValiElem para calcular a cobertura obtida e, então apresentar quais elementos foram cobertos pelo conjunto de teste utilizado. Além disso, este módulo fornece a alcançabilidade dos critérios obtida pelos dados fornecidos.

Finalmente, o módulo **ValiSync** tem como objetivo realizar a geração automática de variantes, isto é, impor a execução de diferentes pares de sincronização com um mesmo dado de teste, aumentando, se possível, a cobertura das arestas de sincronização (BATISTA, 2015).

2.5 Considerações Finais

O teste de software é fundamental para verificar se as aplicações realizam as funções de acordo com suas especificações. Além disso, o teste de software é considerado uma das principais atividades de garantia de qualidade, sendo responsável por uma grande parcela no custo de desenvolvimento.

Nesse contexto, a utilização de técnicas, critérios e demais métodos que auxiliam o teste de aplicações, podem contribuir para a qualidade do software e garantir que o mesmo atende a sua especificação. Neste capítulo foram apresentados os principais critérios da técnica estrutural, os quais são utilizados para verificar estruturas de código com o objetivo de revelar defeitos. Além disso, foram apresentados os conceitos sobre os programas concorrentes bem como os aspectos relacionado ao teste dessas aplicações.

GERAÇÃO AUTOMÁTICA E DADOS DE TESTE

3.1 Considerações Iniciais

O objetivo deste capítulo é apresentar uma visão das principais técnicas existentes na literatura para geração automática de dados de teste, apresentando também algumas ferramentas disponíveis que automatizam essas técnicas. O capítulo está organizado da seguinte forma: na Seção 3.2 são apresentados alguns conceitos sobre a geração automática de dados de teste. Posteriormente são apresentadas as principais técnicas de geração. Na Seção 3.8 é apresentada uma discussão sobre as técnicas. A Seção 8 é composta por algumas ferramentas da literatura propostas para geração de dados de teste. Por último, na Seção 3.10, são apresentadas as considerações finais deste capítulo.

3.2 Conceitos

O teste de software é considerado uma das fases mais dispendiosas do processo de desenvolvimento de software, sendo assim, a automação das atividades de teste busca diminuir os esforços necessários, melhorando a qualidade no teste das aplicações. Como exemplo, a geração automática de dados de teste têm sido empregada para apoiar a atividade de teste (BERTOLINO, 2007; MOHI-ALDEEN; DERIS; MOHAMAD, 2014). A geração automática de dados atua na seleção automática de entradas com maior probabilidade de revelar defeitos. Essa seleção geralmente é guiada por critérios de teste, assim os dados de teste são selecionados para atingir ou cobrir elementos definidos pelos critérios.

Embora a automatização dessa atividade (seleção de dados de teste) seja muito desejada, não existe um algoritmo de propósito geral e eficiente que seja capaz de gerar automaticamente e a baixo custo um conjunto de teste efetivo que seja adequado a um determinado critério de

teste. Nem mesmo é possível determinar automaticamente se esse conjunto existe (VERGILIO; MALDONADO; JINO, 2007). Além disso, existem restrições inerentes à atividade de teste que dificultam essa automatização, destacando:

- **Correção coincidente:** Ocorre quando o programa em teste apresenta um resultado correto para um dado de teste particular, não revelando a presença de um erro presente, ou seja, o erro existe mas devido ao valor do dado de teste coincidentemente apresentou um valor correto;
- **Caminho ausente:** corresponde a uma determinada funcionalidade requerida para o programa, mas que por algum motivo não foi implementada, ou seja, o caminho desejado não existe no programa;
- **Caminhos não executáveis:** ocorre quando não existe dado de teste que seja capaz de exercitar esse caminho. Isso ocorre com frequência devido principalmente a informações sobre o fluxo de dados de um programa;
- **Equivalência de programas:** não existe algoritmo de propósito geral capaz de determinar se dois programas apresentam, para qualquer dado possível do domínio de entrada, o mesmo comportamento.

Apesar das limitações, várias técnicas de geração de dados de teste são encontradas na literatura (AHMED; HERMADI, 2008; BAGNARA *et al.*, 2013; GHIDUK, 2014; SILVA; SOUZA; SOUZA, 2014). Em geral, essas técnicas utilizam informações sobre a cobertura em relação a critérios de teste. As mais utilizadas são: i) geração aleatória; ii) geração com execução simbólica; iii) geração com execução dinâmica; iv) geração de dados sensíveis a defeitos; e v) geração de dados por meio de técnicas evolutivas (VERGILIO; MALDONADO; JINO, 2007). As seções seguintes descrevem as principais técnicas identificadas na literatura.

3.3 Geração Aleatória

A geração aleatória é a técnica mais simples para geração de dados de teste, na qual o principal objetivo é selecionar randomicamente entradas para um programa sob teste até que dados úteis (dados capazes de revelar defeitos ou de cumprir algum objetivo de teste) sejam alcançados. No entanto, essa técnica pode não encontrar dados de teste capazes de satisfazer os requisitos de teste, pois as informações sobre os requisitos de teste ou informações sobre os programas não são incorporadas. Assim, por tratar-se de uma técnica simples e de baixo custo essa técnica é amplamente utilizada na literatura para efeito de comparação com novas abordagens (EDVARDSSON, 1999; DEVASENA; VALARMATHI, 2012).

Figura 7 – Exemplo da técnica de execução simbólica (Extraído de [Anand et al. \(2013\)](#))

3.4 Execução Simbólica

A geração de dados de teste por meio da execução simbólica é feita utilizando valores simbólicos ao invés de valores reais para as variáveis de entrada de um programa. Essa técnica representa os valores das variáveis do programa como expressões simbólicas de suas entradas. A execução simbólica é composta por três aspectos essenciais, os valores simbólicos das variáveis, a condição de caminho e um contador de programa.

A condição de caminho (*Path Constraint* - PC) consiste em uma fórmula booleana sobre as entradas simbólicas, ou seja, um conjunto de condições que as entradas devem satisfazer para que a execução simbólica possa executar um determinado caminho do programa. Para cada ramificação existente no programa, o PC possui as condições de caminho de forma que para alcançar um caminho deve ser possível satisfazer uma condição do PC. Quando uma condição não pode ser satisfeita o caminho indicado é chamado de não executável, pois não existe uma entrada capaz de exercitar esse caminho. Caso contrário, quando a condição de caminho pode ser alcançada, qualquer solução do PC pode ser utilizada para executar esse caminho. Por sua vez, o contador de programa tem como função identificar a próxima instrução a ser executada (ZHANG; CHEN; WANG, 2012).

Para exemplificar o funcionamento da execução simbólica, considere o código apresentado na Figura 7(a). No código existem duas variáveis do tipo inteiro representadas por x e y . O programa tem como função realizar a troca de valores entre as variáveis, e para isso o valor inicial de x deve ser maior que o valor inicial de y . Para referenciar as declarações presentes no código são utilizados os seus números de linha correspondentes.

Na Figura 7(b) é apresentada uma árvore de execução simbólica correspondente ao código apresentado na Figura 7(a). A árvore consiste em uma representação compacta dos caminhos de execuções que podem ocorrer durante a execução simbólica do programa. Na árvore de execução, os retângulos (ou nós) representam os estados do programa, enquanto as arestas correspondem as transições entre esses estados.

Para cada estado da árvore existe um valor definido pelo contador de programa, o mesmo pode ser observado no canto superior direito de cada retângulo. Antes mesmo da execução da declaração 1, o PC é inicializado como *true*, pois essa declaração ocorre para qualquer entrada inserida no programa. Para as variáveis x e y são fornecidos os valores simbólicos X e Y , respectivamente. Após a execução do comando **if** presente na linha 1 do código o PC é atualizado adequadamente para $X > Y$ ou $X \leq Y$. Dessa forma, para atingir o caminho (1,2,3,4,5) é necessário fornecer valores que estejam compreendidos no intervalo $X > Y$. Caso contrário, se o objetivo de teste for percorrer o caminho (1,8), devem ser gerados valores no intervalo $X \leq Y$. A tabela exposta na Figura 7(c) apresenta as soluções de PC que correspondem a três caminhos

distintos do programa obtidos por meio do código.

Na coluna Entrada Programa são apresentadas sugestões de valores que possam ser utilizadas como entradas no programa para atingir determinados caminhos. Por exemplo, o PC correspondente ao caminho (1,2,3,4,5,8) é $X > Y \ \& \ Y - X \leq 0$. Portanto, a entrada necessária para exercitar esse caminho é obtida resolvendo o PC. Neste caso, a solução sugerida foi $X = 2$ e $Y = 1$, entretanto, qualquer valor pertencente ao intervalo estabelecido pelo PC também pode ser utilizado. Em uma situação adversa o programa pode possuir um caminho que não seja executável. Para exemplificar essa condição considere o caminho (1,2,3,4,5,6) onde o PC apresenta a condição $X > Y \ \& \ Y - X > 0$. Observe que não há uma entrada capaz de exercitar este caminho no código, pois a primeira condição do código apresentado na Figura 7(a) requer que o valor da variável x seja maior que o valor da variável y , sendo assim, como os valores das variáveis são invertidos não há como a condição $(x-y>0)$, presente na linha 6 do código, ser verdadeira.

Apesar dos benefícios que essa técnica pode fornecer, existem algumas limitações impostas na sua implementação:

- **Explosão de Caminhos:** Executar simbolicamente uma quantidade significativa de caminhos de um programa não é uma tarefa trivial, pois programas reais possuem um número extremamente grande de caminhos e executar toda essa massa de caminhos pode implicar em uma sobrecarga computacional. Além disso, a atividade de identificar todos os caminhos executáveis em um programa é afetada devido à proporção alta do número de caminhos não executáveis em relação ao número de caminhos executáveis (NGO; TAN, 2007).
- **Divergência de Caminhos:** A diversidade de linguagens de programação e o formato disponível de aplicações reais (como o formato binário) dificultam a tarefa de identificar os caminhos executáveis em um programa. Gerar dados automaticamente para esses programas requer uma grande quantidade de esforço na implementação de ferramentas de teste, as quais podem necessitar de modelos grandes e complexos para calcular as restrições de caminhos. Dessa forma, a incapacidade de calcular precisamente as restrições de caminhos dos programas pode levar a uma divergência de caminhos, ou seja, o caminho que o programa percorre com os dados de teste inseridos, diverge do caminho para o qual os dados foram gerados (ANAND *et al.*, 2013).
- **Restrições Complexas:** Solucionar as condições de caminho nem sempre é possível para determinados tipos de aplicações, pois existem classes de condições que são em geral indetermináveis. Sendo assim, é possível que condições de caminhos sejam complexas, e, portanto não podem ser solucionadas pelas condições disponíveis. Como exemplo, considere programas que realizem cálculos não-lineares, como multiplicação e divisão e funções matemáticas como *seno* e logaritmo. Em decorrência disso, a dificuldade de

solucionar as condições de caminhos reduzem a quantidade de caminhos executáveis distintos que uma ferramenta de execução simbólica pode identificar.

3.5 Execução Dinâmica

A execução dinâmica para geração de dados de teste baseia-se na execução real do programa em teste analisando o seu comportamento diante as entradas estabelecidas. Diferentemente da execução simbólica, esta técnica utiliza valores reais de entrada para as variáveis do programa sendo inicialmente proposta por [Korel \(1990\)](#).

A geração de dados por meio de geração dinâmica tem como principal objetivo encontrar uma entrada para o programa $x \in D$ de forma que P possa ser executado, onde D corresponde ao domínio de entradas do programa e P ao caminho a ser alcançado pela entrada. Entretanto, esse objetivo pode ser reduzido a uma sequência de subobjetivos, onde cada subobjetivo pode ser melhor resolvido utilizando-se técnicas de minimização de funções ([VERGILIO; MALDONADO; JINO, 2007](#)).

Essa técnica, considera os predicados simples e lineares (desigualdade e igualdade), ou seja, todos os predicados das ramificações são representados da seguinte forma:

$$E_1 \text{ op } E_2$$

onde E_1 e E_2 são expressões aritméticas, e op pode assumir os símbolos $\{<, \leq, >, \geq, =, \neq\}$. Além disso, assume-se que os predicados não apresentem AND's, OR's ou outros operadores *booleanos*. Cada predicado pode ser transformado em um predicado equivalente, da seguinte forma: $F \text{ rel } 0$ onde F e rel são dados pela Tabela 1.

Tabela 1 – Predicados de Ramificação

Predicados	Função F	rel
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$\text{abs}(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$\text{abs}(E_1 - E_2)$	$\leq$

Na Tabela 1 F representa uma função de valores reais referida como uma função de ramificação, em que: 1) positivo(ou zero se rel é $<$) quando um predicado de ramificação é falso ou 2) negativo (ou zero se rel é $=$ ou $\leq$) quando o predicado de ramificação é verdadeiro.

Dito isso, considere x^0 o conjunto de entradas (as quais podem ser escolhidas aleatoriamente) em que um programa é executado, para o qual o caminho $P = (n_1, \dots, n_i, \dots, n_m)$ é executado, então x^0 é a solução para o problema da geração de dados de teste. Caso contrário, considere que se P' seja o caminho executado, $P_1 = (n_1^P, n_2^P, \dots, n_i^P)$ é o mais longo sub caminho de

P' , em que $(n_1^p = n_1, \dots, n_i^p = n_i)$ e que o caminho correto não foi executado porque o predicado $F_i(x) \text{ rel}_i 0$, associado a aresta (n_i, n_{i+1}) não foi avaliado como se esperava. Assim, a meta é encontrar entradas x , em que a condição $F_i(x) \text{ rel}_i 0$ seja satisfeita (KOREL, 1990; VERGILIO; MALDONADO; JINO, 2007).

3.6 Concolic Testing

A técnica *Concolic Testing* é estabelecida pela união entre a execução simbólica e execução dinâmica. Nesta técnica, é mantido um mapa de memória simbólica. Este mapa é preenchido durante o curso de execução do programa por endereços de memória concretos das expressões simbólicas juntamente com as restrições de caminho, os detalhes da construção de expressões simbólicas são padrão, conforme descrito anteriormente.

A execução do *Concolic Testing* para geração de um novo teste é realizada da seguinte forma: (1) Primeiramente é gerado um conjunto de valores aleatórios para o programa sob teste. Posteriormente, (2) o programa é executando com os valores concretos de modo que possam ser obtidas e armazenadas as condições de caminhos dessa execução. Nesse momento, (3) a execução simbólica é então empregada para solucionar cada condição identificada, a qual pode ser realizada de forma aleatória ou utilizando *Constraint Solvers* (CS), terminado essa etapa o fluxo de trabalho é redirecionado para segunda etapa gerando um ciclo que estende-se até que todas as condições sejam satisfeitas (MAJUMDAR; SEN, 2007).

3.7 Meta-heurísticas

A otimização do teste de software é uma área recente de pesquisa que utiliza técnicas de busca e otimização matemática para resolução de problemas complexos provenientes das atividade de teste. Normalmente, as técnicas de busca são empregadas para otimização de problemas em que o espaço de busca é muito grande, envolvendo a seleção de soluções ótimas ou quase ótimas para um determinado problema. Dessa forma, um novo campo de pesquisa denominado *Search-Based Software Testing* (SBST) vêm sendo amplamente investigado por diversos trabalhos na literatura (COLANZI *et al.*, 2013).

Em geral, os trabalhos de SBST utilizam técnicas meta-heurísticas para otimização de buscas. Uma meta-heurística é composta por formas genéricas de heurísticas que podem ser utilizadas para resolução de diferentes problemas. Sendo assim, as técnicas pertencentes a essa classe utilizam ideias de diversos domínios como inspiração para a busca de soluções para problemas de otimização.

A execução de uma meta-heurística realiza a busca por soluções a partir da visitação de regiões no espaço de busca, a qual é guida por uma função de *Fitness* também conhecida como função objetivo. A função *Fitness* define valores para as soluções candidatas e identifica qual

solução é mais indicada para o problema. O processo de busca de soluções é definido por cada tipo de meta-heurística e todas as técnicas buscam, de forma inteligente, encontrar boas soluções (GENDREAU; POTVIN, 2010).

No que tange a geração de dados de teste, o objetivo do teste é transformado em um problema de otimização, onde o espaço de busca é dado pelo domínio de entrada dos programas em teste. Os algoritmos de busca exploram o espaço de busca para encontrar dados de teste que satisfaçam o objetivo do teste. As principais técnicas de meta-heurística são apresentadas nas seções seguintes.

3.7.1 *Hill Climbing*

O algoritmo *Hill Climbing* (ou subida de encosta) consiste em um dos principais algoritmos para busca local. A execução do algoritmo ocorre da seguinte forma: inicialmente é criada uma solução aleatória no espaço de busca, e de forma iterativa o algoritmo procura melhorar a solução atual deslocando-se para soluções vizinhas no espaço. A subida de encosta caracteriza um algoritmo iterativo de melhoria, onde a encosta corresponde às soluções vizinhas da solução atual, melhorando-a passo a passo. Quando existe uma solução vizinha que apresenta um resultado melhor que a solução corrente, essa nova solução passa a ser a solução corrente no algoritmo de busca. Dessa forma, essa tarefa é repetida até que melhores soluções sejam encontradas ou que o número de vizinhos gerados exceda um limite (DOGANAY; BOHLIN; SELLIN, 2013).

O algoritmo *Hill Climbing* é conhecido por realizar buscas de soluções ótimas para problemas que possuem grandes espaços de soluções e que não necessitam de soluções exatas de tempo polinomial. A técnica *Hill Climbing* é apresentada no Algoritmo 1 (MARTINEZ; GONZALEZ; STOJMENOVIC, 2002).

Em adição à técnica, existem variações na estratégia de busca do *Hill Climbing* sendo elas: i) subida íngreme: todas as soluções vizinhas são analisadas e partir dessa atividade é possível eleger a melhor solução; e ii) subida aleatória: a vizinhança é explorada aleatoriamente substituindo a solução atual pela primeira que oferecer melhor resultado, repetindo os passos até alcançar uma meta estabelecida.

O sucesso desse algoritmo está estritamente ligado à forma em que o espaço está disposto, ou seja, se houver alguns máximos locais provavelmente a estratégia de busca aleatória vai encontrar uma boa solução rapidamente. Por outro lado, a maioria dos problemas reais possuem soluções ótimas dispersas nos locais de busca (RUSSELL; NORVIG, 2009).

3.7.2 *Simulated Annealing*

Um algoritmo como o *HillClimbing* nunca retrocede em direção de estados com valores menores (ou maiores) em seu processo de otimização, característica que o torna um algoritmo

Algoritmo 1: Algoritmo *Hill-Climbing* (adaptado) [Martinez, Gonzalez e Stojmenovic \(2002\)](#)

```

1 Algoritmo hill-climbing( $X$ ,  $best\pi$ , best-access-time);
2  $\pi :=$  initial permutation (generated at random);
3  $best\pi := \pi$ ;
4  $best - access - time := X(\pi)$ ;
5  $i := 0$ ;
6 repeat
7    $\pi := neighbour(\pi)$ ;
8    $access - time := X(\pi)$ ;
9   if  $access-time < best-access-time$  then
10      $best\pi := \pi$ ;
11      $i := 0$ ;
12      $best-access-time := access-time$  ;
13   else
14      $i := i + 1$ ;
15 until  $i = max$ ;

```

incompleto, pois o mesmo pode ficar preso em máximo local que o impede de encontrar a melhor solução no espaço de busca (máximo global). Dessa forma, é razoável combinar esse tipo de algoritmo com técnicas aleatórias, de forma que seja possível obter eficiência e completude na otimização de problemas.

Nesse contexto, o algoritmo *Simulated Annealing* (ou Têmpera Simulada) origina-se da analogia entre o processo de têmpera de sólidos e a solução de problemas de otimização combinatória. O processo de têmpera é dado pelo resfriamento de um sólido para chegar a um estado de energia mínima (estado fundamental). Em temperaturas iniciais elevadas, todas as moléculas em estado sólido arranjam-se aleatoriamente no estado líquido, como a temperatura decresce gradualmente a estrutura cristalina torna-se mais ordenada e atinge o estado congelado assim que a temperatura chega a zero. Dessa forma, se a temperatura diminuir rapidamente o cristal pode não atingir o equilíbrio térmico necessário para alcançar um estado de energia mínima. Portanto, uma temperatura inicial adequada e uma programação de arrefecimento adequada são cruciais para o processo de têmpera.

No *Simulated Annealing* a energia representa a qualidade da solução encontrada. Quanto menor é a energia, melhor é a solução identificada, sendo assim, as alterações que levam a uma energia mais baixa são automaticamente aceitas. Por outro lado, as mudanças que ocasionam a obtenção de uma energia superior são aceitas com base em uma probabilidade definida pela taxa obtida do fator de aceitação de Boltzmann. Esta probabilidade é definida pela expressão $(-\Delta E/kT)$. Onde ΔE é a mudança de energia, k é uma constante e T é a temperatura. Ao executar o *Simulated Annealing*, a temperatura T é inicialmente definida com um valor alto ([XIAO et al., 2007](#)). A temperatura então é repetidamente reduzida por meio de um processo denominado

“cooling schedule”. Estudos sobre “schedules” podem ser encontrados em [Nourani e Andresen \(1998\)](#) e [Chainate, Thapatsuwan e Pongcharoen \(2008\)](#).

A maior vantagem do *Simulated Annealing* em relação a outras técnicas, em especial o *Hill Climbing*, é a probabilidade de aceitar uma solução com menor qualidade, o que permite ao algoritmo alcançar resultados melhores (ou não) no espaço de busca, ou seja, essa característica permite ao algoritmo desviar-se de mínimos locais.

3.7.3 Algoritmo Genético

Um Algoritmo Genético (AG) utiliza uma coleção (população) de soluções, que por meio de estratégias de reprodução e recombinação seletivas, melhores soluções podem ser produzidas.

A geração de dados de teste com algoritmos genéticos utiliza conceitos da computação evolutiva, que baseia-se na evolução natural das espécies. Com essa técnica pode-se ter várias soluções que se encaixam em problemas com dificuldade na obtenção de resultados ótimos. Quando se implementa um algoritmo genético é preciso representar o espaço de busca do problema e suas possíveis soluções de forma computacional e encontrar a função de aptidão que é usada para avaliar quão boa é uma dada solução. O algoritmo genético parte da premissa de que dada duas soluções com resultados aproximados para um determinado problema, essas duas soluções combinadas podem conduzir uma melhor solução. A geração de dados com algoritmos genéticos geralmente se encontra classificada em duas categorias: função de avaliação baseada na cobertura, onde a avaliação é realizada com base na cobertura de cada dado de teste; e função de avaliação orientada a um objetivo, onde a função é especificada de acordo com um objetivo que é, geralmente, a cobertura de um elemento particular. É importante salientar que o AG é a técnica mais popular para geração de dados de teste ([MOHI-ALDEEN; DERIS; MOHAMAD, 2014](#)).

Operadores genéticos como *crossover* e mutação são utilizados para a construção de novas soluções a partir de soluções antigas, de tal maneira que, para muitos problemas a população melhora constantemente. O operador *crossover* pode ser entendido da seguinte forma: dadas duas cadeias **a** e **b**, compostas por 6 variáveis (genes),

$$(a1, a2, a3, a4, a5, a6) \text{ e } (b1, b2, b3, b4, b5, b6),$$

as mesmas representam duas soluções para um determinado problema. Um ponto de *crossover* é escolhido aleatoriamente para gerar novas soluções. Neste exemplo considera-se que o ponto escolhido seja 2. A nova geração será composta pelos seguintes indivíduos:

$$(a1, a2, \mathbf{b3}, \mathbf{b4}, \mathbf{b5}, \mathbf{b6}) \text{ e } (b1, b2, \mathbf{a3}, \mathbf{a4}, \mathbf{a5}, \mathbf{a6}),$$

A maioria das aplicações utilizam cadeias compostas por apenas 0s e 1s, por facilitar a tarefa e por vantagens impostas por esta abordagem. Outro operador utilizado para geração

de novos indivíduos é a mutação. O operador de mutação oferece a oportunidade de alcançar espaços de busca não alcançados pelo *crossover*. Cada gene da cadeia é examinado e possui uma pequena probabilidade de ocorrer uma alteração arbitrária em sua estrutura. Para ilustrar a situação, considere um indivíduo composto por 10 genes:

1001010100,

após aplicação desse operador, suponha que o terceiro e oitavo gene sofram uma mutação. A sequência obtida após a mutação resulta no novo indivíduo:

10**1**1010**0**00

Além da aplicação de operadores, os indivíduos também são escolhidos com base em uma função de aptidão. A função de aptidão avalia cada indivíduo com base no problema a ser otimizado. Dessa forma, a função de aptidão é considerada uma das partes mais importantes do AG. Após esta fase, existem abordagens utilizadas para seleção dos indivíduos, dentre elas podemos destacar: **i) Roleta**: As habilidades de todos os indivíduos são calculadas e os valores obtidos corresponderão, proporcionalmente, aos setores de uma roleta. Assim, quando realizado o sorteio os maiores setores possuirão maiores chances de serem selecionados; **ii) Aleatória**: Consiste em escolher indivíduos da população de forma aleatória; e **iii) Torneio**: Um número n de indivíduos são escolhidos aleatoriamente formando uma sub-população. Dessa forma, o indivíduo selecionado dependerá de uma probabilidade k definida previamente.

No entanto, a forma como os operadores são configurados em um AG para gerar bons indivíduos (dados de teste capazes de revelar defeitos ou atender determinados critérios de teste) não é uma tarefa trivial. Por exemplo, o tamanho elevado de uma população pode gerar inúmeros dados de teste dificultando a tarefa de selecionar esses dados. Por outro lado, como o AG é baseado na otimização em um espaço de busca, uma taxa equivocada de mutação pode não garantir a efetividade de um AG para selecionar os dados de teste fazendo com que este fique preso em um ótimos locais e não encontre os melhores indivíduos no espaço de busca. Além disso, uma taxa inapropriada de cruzamento, pode não gerar bons indivíduos para uma nova geração do algoritmo. Assim, diferentes configurações podem ser definidas e a questão em aberto é decidir qual delas escolher para obter melhores resultados. Nesse sentido, um mapeamento sistemático foi conduzido com o objetivo de investigar a forma como os AGs são configurados para a geração de dados de teste (VILELA *et al.*, 2016).

Para automação do teste de programas com a análise de cobertura de caminho, o primeiro trabalho em que algoritmos genéticos foram utilizados para gerar automaticamente os dados de teste, foi proposto por Xanthakis *et al.* (1992). Desde então, diversos trabalhos foram propostos na literatura (KHOR; GROGONO, 2004; GIRGIS, 2005; ANDREOU; ECONOMIDES; SOFOKLEOUS, 2007; AHMED; HERMADI, 2008; RIBEIRO; RELAS; VEGA, 2008; MAO; YU, 2013; GHIDUK, 2014) com o objetivo de minimizar os esforços gastos na atividade de teste.

3.7.4 Otimização Multi-Objetivo

Como o próprio nome sugere, um problema de otimização multi-objetivo é dado pela existência de mais de uma função objetivo. Essa técnica é empregada para problemas em que a tomada de decisões é baseada em diferente critérios, os quais devem ser balanceados. Originalmente esses problemas eram solucionados por técnicas tradicionais, como AG Mono-Objetivo, para encontrar uma soluções ótimas, entretanto, descobriu-se que as soluções encontradas poderiam não ser as melhores para os problemas investigados.

Um problema de otimização multi-objetivo possui um número de funções objetivo a serem otimizadas, as quais são empregadas para maximizar ou minimizar um problema. Em adição, existem restrições que devem ser satisfeitas por todas as soluções geradas. De forma geral, o problema de otimização multi-objetivo pode ser entendido da seguinte forma:

$$\left. \begin{array}{ll} \text{maximizar/minimizar } f_m(x), & m = 1, 2, \dots, M; \\ \text{restrita a } g_j(x) \geq 0, & j = 1, 2, \dots, J; \\ h_k(x) = 0, & k = 1, 2, \dots, K; \\ x_i^{(L)} \leq x_i \leq x_i^{(U)}, & i = 1, 2, \dots, n. \end{array} \right\}$$

onde x é dado pelo vetor de n variáveis de decisão $x = (x_1, x_2, \dots, x_n)^T$. Os Valores $x_i^{(L)}$ e $x_i^{(U)}$ representam o mínimo e máximo valor para variável x_i , respectivamente. Esse intervalo define o espaço de variáveis de decisão, ou espaço de decisão.

As J desigualdades (g_j) e as K igualdades (h_k) representam as funções de restrição, assim uma solução x factível será aquela que satisfaça as $J + K$ funções de restrição e os n limites, caso contrário essa solução é dada como não factível. O conjunto de todas as soluções factíveis formam o espaço de busca de um problema (DEB; KALYANMOY, 2001).

3.8 Discussão sobre técnicas de geração

Como pode ser observado, diversas técnicas são utilizadas para geração automática de dados de teste, sejam elas de meta-heurística ou demais tipos de técnicas. O que difere um trabalho de outro é a forma como são realizadas as avaliações dos indivíduos (dados de teste) que são gerados por meio dessas técnicas.

De acordo com Mohi-aldeen, Deris e Mohamad (2014), nenhuma técnica de geração de dados de teste é superior as demais técnicas. Desta forma, diversas técnicas devem ser comparadas quando surgem novos problemas. Assim, as técnicas mais eficazes podem ser apresentadas.

Nesse sentido, Ahmed e Zamli (2011) apresentam uma comparação de meta-heurísticas para geração de dados de teste baseado no critério de cobertura para interação de elementos. Os elementos de interação são criados por meio de todas a combinações de parâmetros do sistema,

em seguida, são atribuídos valores relacionados para cada combinação. Este critério tem como objetivo cobrir todos os elementos de interação pelo menos uma vez, para assegurar que todas as interações sejam testadas. A cobertura dos elementos de interação depende do caso de teste escolhido. Um caso de teste pode abranger um ou mais elementos de interação de acordo com o arranjo do próprio processo de teste.

Dessa forma, o objetivo do trabalho foi descobrir o quão bem os elementos de interação são cobertos pelos algoritmos meta-heurísticas existentes. Para tanto, a medida de cobertura dos elementos de interação foi baseada nos dados de teste gerados pelas meta-heurísticas. As técnicas analisadas foram: *Hill Climbing*, *Simulated Annealing*, Busca Tabu e *Particle Swarm Optimization*. Os resultados obtidos foram baseados em diferentes números de dados de teste, os quais variam de 25 até 350 dados de teste.

O resultado obtido apontou que nos primeiros 25 dados de teste, o algoritmo *Simulated Annealing* apresenta uma melhor cobertura dos elementos de interação. No entanto, o PSO produz uma melhor cobertura para as demais quantidades de dados de teste com uma menor quantidade de dados de teste.

Ainda que exista uma grande quantidade de trabalhos na literatura sobre a geração de dados de teste, pouco existe com o foco em programas concorrentes ou similares como programas paralelos e distribuídos.

Tian e Gong (2014a) apresentam uma proposta para geração de dados de teste para programas paralelos denominado EGMP - *Method of Evolutionary Generation of Test Data for Multiple Paths Coverage*. A abordagem utiliza um AG para geração de dados de teste com o foco na cobertura de múltiplos caminhos que ocorrem devido a troca de mensagens de sistemas em teste. O estudo experimental realizado pelos autores utilizou cinco *benchmarks* para avaliação de desempenho comparando os resultados com a geração aleatória e com um algoritmo genético com o foco em apenas um caminho por execução, o qual foi proposto pelos mesmos autores em um trabalho anterior (TIAN; GONG, 2013). Os resultados apresentados em (TIAN; GONG, 2014a) foram avaliados por meio do tempo gasto para execução dos algoritmos, número de iterações e cobertura alcançada por cada algoritmo. Em todos os fatores e níveis utilizados no experimento, o método EGMP mostrou-se superior em relação às demais abordagens.

No entanto, ainda que o método apresentado possa ser superior em relação às demais abordagens, não é possível garantir que o mesmo pode ocorrer em um número maior de *benchmarks*, haja visto que foram testados apenas cinco programas. Outro fator importante que deve ser observado, é que a comparação ocorreu apenas entre um tipo de meta-heurística (algoritmo genético), sendo assim, não é possível garantir que o algoritmo genético seja a técnica mais indicada para o teste de programas concorrentes.

Tendo em vista os aspectos observados, existe uma necessidade de estudos que avaliem a eficácia das abordagens para geração de dados de testes para programas concorrentes. Dessa

forma, um aspecto importante no contexto de teste de software é a avaliação experimental, analisando e comparando o custo de aplicação, aspecto complementar e eficácia em revelar defeitos. Esses estudos são fundamentais para identificar vantagens e limitações, auxiliando na proposição de novas abordagens.

3.9 Ferramentas de Geração de Dados de Teste

Como descrito anteriormente, gerar dados de forma automática pode diminuir os esforços gastos pelo testador e melhorar a qualidade dos dados selecionados. Para tanto, na literatura existem ferramentas que podem auxiliar o testador nessa tarefa. Esta seção apresenta algumas ferramentas de geração de dados de teste.

3.9.1 *Feedback-Directed Random Testing for Java - Randoop*

A Randoop foi proposta por [Pacheco e Ernst \(2007\)](#), e consiste em uma ferramenta de geração automática de testes de unidade para programas desenvolvidos na linguagem Java ou .Net. Para isto, a Randoop utiliza a técnica aleatória direcionada por *feedback* com o objetivo de gerar testes significativos. A Randoop trata-se de uma ferramenta acadêmica sendo utilizada por diversos trabalhos na literatura, principalmente, como baseline para avaliação de outras ferramentas (A implementação, documentação e os guias de usuário podem ser encontrados em: [Randoop](#)). Uma arquitetura dessa ferramenta é apresentada na Figura 8.

Figura 8 – Arquitetura ferramenta Randoop (Adaptado de [\(PACHECO; ERNST, 2007\)](#)).

Para a geração dos testes, a Randoop recebe como entrada as classes sob teste, um tempo limite de geração e, opcionalmente, um conjunto de verificadores de contrato (como *Object.equals()*, *Object.hashCode()*, *Object.clone()*, entre outros) que podem estender os já utilizados por *default* pela Randoop. Em posse desses atributos, a Randoop cria de forma

incremental sequências de métodos, selecionando aleatoriamente uma chamada de método para ser aplicada. Já os argumentos desses métodos são obtidos por meio das sequências previamente construídas. Posteriormente, a execução e a verificação dessas sequências são realizadas. Finalmente, a saída da Randoop é composta por dois tipos de casos de teste, sendo eles: 1) testes de violação de contratos, os quais podem expor cenários em que o código testado viola algum contrato de API; e 2) testes de regressão, os quais, diferentemente do primeiro, não buscam violar os contratos, mas sim capturar os aspectos da implementação. Assim, os testes de regressão poderão contribuir na descoberta de inconsistências entre diferentes versões do software.

3.9.2 Guided Random Testing - GRT

A ferramenta GRT foi proposta por [Ma et al. \(2015\)](#) com o propósito de gerar, de forma automática, testes de unidade para programas desenvolvidos em Java. Para isto, a GRT faz a extração de informações estáticas e dinâmicas do software sob teste e utiliza essas informações para guiar a geração aleatória de testes. A GRT é composta por seis componentes colaborativos, esses componentes trabalham em conjunto para extrair as informações estáticas e dinâmicas dos programas. Assim, a efetividade da GRT não resulta dos componentes individuais, mas sim da orquestração entre os componentes. Os componentes compreendidos pela GRT são:

- **Constant mining**: enriquece o conjunto inicial de valores primitivos extraindo constantes dos programas por meio de uma análise estática, assim, a extração contribui para criação de novos objetos que poderão melhorar a cobertura do código. A mineração de constantes é realizada por meio do *framework* ASM ([BRUNETON; LENGLET; COUPAYE, 2002](#)), o qual analisa o bytecode Java de cada classe sob teste para propagação das constantes, dessa forma é possível realizar a análise estática mesmo na ausência do código fonte.
- **Impurity**: incrementa a mineração de constantes realizando a fuzzificação de entradas por meio de métodos que podem modificar os estados de um objeto. Ao mesmo tempo, a fuzzificação de entradas pode ser reforçada com as constantes extraídas previamente. Para os valores numéricos primitivos é adotada uma distribuição de Gauss, com o pressuposto de que as constantes extraídas estão próximas de satisfazer alguns ramos de condição. O desvio da distribuição de Gauss e a razão para realizar a fuzzificação são fornecidos como parâmetros que podem ser ajustados de acordo com as características específicas do sistema sob teste.
- **Elephant brain**: gerencia todos os objetos armazenados no *pool* de objetos utilizando o tipo exato de objetos. A informação de tipo é obtida analisando objetos retornados na execução de sequências de métodos por meio da operação de *reflection*. As sequências de métodos são armazenadas em um *pool* de objetos, para o qual é utilizado o tipo exato do objeto retornado como chave. Assim, quando um método requer uma entrada declarada

estaticamente com o tipo T , é realizada uma busca no *pool* de objetos por sequências que retornem um objeto compatível com T .

- **Detective**: constrói objetos necessários para testar os métodos sob teste que não podem ser encontrados no *pool* principal de objetos. Este componente executa a análise estática nos métodos sob teste para encontrar suas dependências de entrada e saída, construindo os objetos necessários em tempo de execução.
- **Orienteering**: estima o custo de execução de cada sequência de método por meio do tempo de execução e tamanho da sequência. Baseado nesses aspectos, um peso é atribuído para cada sequência, de modo que as sequências de métodos menos dispendiosas possuem maior probabilidade de serem selecionadas, por outro lado, as sequências de métodos dispendiosas, mas que geram novos objetos relevantes, também são incluídas. Este componente contribui para aceleração do processo de geração dos testes da GRT.
- **Bloodhound**: seleciona de maneira inteligente os métodos que não são bem cobertos. Deste modo, mais código é coberto e portanto mais estados do programa podem ser alcançados, o que potencialmente cria objetos para posteriormente melhorar a eficácia do teste.

A execução da GRT consiste em duas etapas: 1) Uma análise estática da classe sob teste extrai informações sobre o código, tais como possíveis constantes durante uma execução, efeitos secundários dos métodos existentes e possíveis dependências. Assim, baseada no resultado dessa análise, a GRT cria um conjunto de valores iniciais para as constantes identificadas e determina as propriedades dos métodos que formam a base para a geração de sequências de testes; 2) Em tempo de execução, a informação estática extraída é combinada, de forma inteligente, com o *feedback* dinâmico durante a execução.

3.9.3 CodePro AnalytiX - JUnit Test Generation

A ferramenta CodePro AnalytiX consiste em uma solução de software desenvolvida para indústria, utilizada para detectar e corrigir problemas relacionados à qualidade do código, eventualmente enfrentados no desenvolvimento de software. CodePro AnalytiX contém um conjunto de ferramentas de análise de software composto por uma coleção de *plugins* para o Eclipse. A ferramenta apresenta funções que permitem a auditoria de código, cobertura de código, recursos de colaboração entre equipes e geração automática de dados de teste.

O módulo de geração de dados de teste da CodePro AnalytiX projeta testes para o JUnit afim de apoiar os desenvolvedores na escolha de testes mais abrangentes. Para isso, essa ferramenta utiliza-se da combinação entre a análise estática do código e pela a execução dinâmica do programa sob teste com o intuito de observar o comportamento do código ([GOOGLE-INC.](#),

Accessed Feb 2016). Por tratar-se de uma ferramenta desenvolvida para indústria os aspectos relacionados ao funcionamento interno da ferramenta não estão disponíveis.

3.9.4 Automatic Test Suite Generation for Java - EvoSuite

A EvoSuite é uma ferramenta desenvolvida para gerar automaticamente casos de teste para classes desenvolvidas na linguagem Java. Para isso, a EvoSuite utiliza um algoritmo genético que gera e otimiza conjuntos de teste com objetivo de satisfazer critérios de cobertura.

As informações utilizadas para guiar a geração dos testes são extraídas do *byte-code* do programa sob teste. Essa característica além de dispensar o código fonte do programa, também permite a aplicabilidade da ferramenta para outras linguagens que compilam seus códigos em Java *byte-code*, tais como Scala ou Groovy. Em adição, esse procedimento evita problemas relacionados ao nível de abstração, descritos no Capítulo 2, ao considerar apenas operações atômicas para o tratamento de critérios (FRASER; ARCURI, 2013).

A EvoSuite realiza instrumentação do *byte-code* adicionando marcações que permitem coletar informações necessárias para calcular os valores de *fitness* do algoritmo genético. Durante a geração dos testes a EvoSuite considera uma classe de alto nível por vez, essa classe e todas as classes anônimas são instrumentadas de modo a permitir a chamada de métodos durante a execução. Para produzir testes de unidade executáveis a ferramenta acessa apenas as interfaces públicas para geração dos testes, apesar disso, as demais subclasses também são consideradas como parte do teste de unidade para permitir o teste de classes abstratas. Para executar os testes durante a busca a EvoSuite utiliza Java *Reflection*, e antes de apresentar o resultado ao usuário os testes são minimizados por meio de um algoritmo de minimização, o qual remove declarações de modo a minimizar a quantidade de testes bem como o tamanho dos testes gerados.

O processo realizado pela ferramenta pode ser lento, podendo ocorrer infinitas iterações, apesar disso, a ferramenta disponibiliza um tempo limite para execução dos testes e otimização dos conjuntos. Quando um tempo limite é atingido, o conjunto é formado pelos testes com melhor valor de *fitness*, o qual é composto pela soma entre cobertura de métodos e caminhos.

3.9.5 Many-Objective Sorting Algorithm - EvoSuite-MOSA

A EvoSuite-MOSA consiste em um protótipo de pesquisa implementada com o propósito de gerar automaticamente teste de unidade para classes desenvolvidas em linguagem Java. Desenvolvida sobre a ferramenta EvoSuite, descrita na seção anterior, a EvoSuite-MOSA é classificada como uma ferramenta de teste baseada em busca, a qual utiliza um algoritmo genético multi-objetivo, denominado *Many-Objective Sorting Algorithm* (MOSA), para geração dos testes (PANICHELLA; KIFETEW; TONELLA, 2015).

A principal característica do algoritmo MOSA é a reformulação do critério de cobertura de decisões para um problema multi-objetivo, na qual cada decisão diferente é considerada

como um objetivo diferente de otimização. Nessa formulação, uma solução candidata possui como *fitness* um vetor de proximidade dessa solução com todas as decisões não exercitadas no programa sob teste. Neste caso, um problema em potencial é o número de decisões existentes, uma vez que a quantidade de objetivos pode ser grande dificultando a classificação das soluções candidatas.

Para minimizar esse problema a EvoSuite-MOSA utiliza um algoritmo de ordenação de preferência que impõe uma ordem entre os testes baseando-se na cobertura de decisões. Esse processo ocorre da seguinte forma: Dada uma decisão d_i , um dado de teste x é preferível em relação a outro y (ou $x \prec_{d_i} y$) se, e somente se, o valor da função objetivo para d_i satisfaz a seguinte condição:

$$f_i(x) < f_i(y)$$

onde $f_i(x)$ denota o valor de *fitness* do dado de teste x para a decisão d_i . O melhor dado de teste para uma decisão d_i é preferível sobre todos os outros para esse ramo ($x_{best} \prec_{d_i} \forall y \in T$), onde T é dado pelo conjunto de soluções candidatas. O conjunto de melhores dados de teste em todas as decisões define um subconjunto de eficiência de *Pareto* que é dado prioridade sobre os demais dados de teste. Quando há vários dados de teste com o mesmo valor mínimo de aptidão para uma determinada decisão o comprimento do teste é utilizado como critério de preferência secundário.

3.10 Considerações Finais

Seja qual for a fase do teste de software, automatizar o seu processo apresenta grandes desafios. A automação do teste só pode ser considerada útil, quando os resultados podem ser verificados por uma máquina, sem o julgamento humano ou esperteza necessária, ou para dados onde os testes são repetitivos, do contrário, o teste manual ainda é a melhor abordagem (WHITTAKER; ARBON; CAROLLO, 2012).

Nesse contexto, a geração de dados de teste é fortemente indicada para automação, uma vez que, além de possuir diversas etapas repetitivas como gerar os dados, a geração automática possui inteligência necessária para avaliar a qualidade dos dados de teste utilizados para determinada aplicação. Além disso, é passível de adição de critérios que auxiliam a geração de requisitos de teste e a classificação de entradas candidatas ao teste. No entanto, ainda é necessário um oráculo de teste para verificar se a saída gerada é condizente com a especificação do programa. Na maioria dos trabalhos, esta tarefa é realizada manualmente.

Neste capítulo foram apresentadas as principais técnicas existentes para geração de dados de teste, bem como ferramentas propostas para auxiliar esse processo. O presente trabalho tem objetivo de demonstrar as características das técnicas de geração no contexto de programas concorrentes. Observa-se que nesse contexto, a geração automática de dados de teste ainda é

incipiente, poucas contribuições são encontradas. Além disso, não foram identificados estudos que avaliem a adequação das técnicas de geração de dados de teste para esse contexto. A descrição da proposta deste trabalho é descrita no capítulo seguinte.

DEFINIÇÃO E PLANEJAMENTO DO ESTUDO

4.1 Considerações Iniciais

Um processo experimental fornece a base necessária para o avanço no entendimento e conhecimento sobre o tema investigado (SHULL *et al.*, 2004). A experimentação na engenharia de software é empregada para apoiar a avaliação, predição, entendimento e controle do processo de desenvolvimento e do produto. Neste contexto, a primeira fase de um processo experimental é a definição do estudo, na qual são definidos os objetos de estudo, os tratamentos a serem aplicados e o modo como será conduzido o estudo experimental.

Neste trabalho a avaliação das técnicas de geração de dados de teste será explorada por meio da condução de experimentos, os quais avaliarão diferentes técnicas para o contexto de programas concorrentes. Para definição experimental deste estudo foram consideradas as diretrizes presentes no *framework* de experimentação proposto por Wohlin *et al.* (2000).

Este capítulo está organizado da seguinte forma: na Seção 4.2 é apresentada a definição do estudo experimental. Na Seção 4.3 é descrito o planejamento do estudo experimental, o qual apresenta as atividades que serão realizadas durante o experimento. Por último, na Seção 4.4 é apresentada a etapa de operação do experimento, a qual descreve a preparação do ambiente e a forma como o mesmo será conduzido.

4.2 Definição do Estudo

A proposta deste estudo é investigar os benefícios e as limitações das técnicas de geração automática de dados de teste, propostas para programas sequenciais, quando aplicadas a programas concorrentes. Para isso, pretende-se aplica-las em programas concorrentes com o intuito de

obter evidências sobre a eficácia dessas técnicas em gerar conjuntos de testes capazes de revelar defeitos, a cobertura de critérios de teste para programas concorrentes e o custo de aplicação.

Deseja-se demonstrar quão boas podem ser as técnicas existentes para auxiliar o teste de programas concorrentes, de forma que seja possível descobrir os potenciais de cada técnica para gerar dados de teste, em relação a cobertura dos requisitos de teste dos paradigmas de comunicação (passagem de mensagem e memória compartilhada) e descoberta de defeitos.

Nesse sentido, a aptidão de uma técnica de geração para o contexto de programas concorrentes pode ser medida por diversos fatores, ainda assim, é possível definir alguns requisitos básicos que uma técnica deve possuir para ser considerada eficiente:

1. Gerar dados que sejam capazes de cobrir todos os elementos requeridos dos critérios considerados ou, pelo menos, alcançar uma cobertura superior a técnica de geração aleatória.
2. Gerar um conjunto pequeno de dados de teste para cobrir todos os elementos requeridos dos critérios considerados ou, pelo menos, uma quantidade inferior a técnica de geração aleatória.

Considerando os aspectos apresentados, este estudo foi definido para abordar três principais questões de pesquisa:

- **Questão de Pesquisa 1 (QP1): Os conjuntos de teste obtidos pelas técnicas de geração apresentam alta cobertura para os critérios de teste considerados?** Em outras palavras, pretende-se descobrir se os conjuntos gerados são capazes de exercitar todos os elementos requeridos ou pelo menos uma quantidade superior à técnica de geração aleatória.
- **Questão de Pesquisa 2 (QP2): Os conjuntos de teste obtidos pelas técnicas de geração são eficazes para revelar defeitos específicos de programas concorrentes?** Pretende-se investigar se os conjuntos são capazes de revelar defeitos triviais, ou seja, aqueles que sempre são revelados pela técnica aleatória, e também aqueles que exigem maior esforço para serem revelados.
- **Questão de Pesquisa 3 (QP3): Qual o custo das técnicas de geração em termos do tamanho dos conjuntos de teste?** Pretende-se avaliar se o custo de cada técnica é baixo para o teste de programas concorrentes, considerando a técnica de geração aleatória como base para essa avaliação.

4.2.1 Definição de Metas

- **Objeto de estudo:** técnicas de geração automática de dados de teste definidas para programas sequenciais.

- **Propósito:** caracterizar e avaliar as técnicas existentes para geração de dados de teste no contexto de programas concorrentes.
- **Foco de Qualidade:** consiste na avaliação do custo de geração, eficácia em revelar defeitos e efetividade dos conjuntos de teste.
- **Perspectiva:** do ponto de vista do pesquisador.
- **Contexto:** da condução de experimentos, para os quais serão utilizados programas concorrentes a fim de avaliar a aplicabilidade de cinco técnicas de geração automática de dados de teste.

4.3 Planejamento

O planejamento do estudo tem como objetivo estabelecer as hipóteses e variáveis afim de guiar a condução e análise do tópico investigado (WOHLIN *et al.*, 2000). As seções seguintes descrevem o planejamento deste estudo.

4.3.1 Seleção do contexto

Evidências sobre técnicas de geração de dados de teste podem ser encontradas em diversos trabalhos que investigam a automatização da seleção de dados de teste, como exemplo os trabalhos propostos por Mohi-aldeen, Deris e Mohamad (2014) e Quintana e Solari (2012) definem estratégias de busca para descobrir quais são as técnicas e como têm sido empregadas no contexto de teste de software. Em complemento, estudos comparativos (AHMED; ZAMLI, 2011; MOSHIZI; BARDSIRI, 2015; BRAR; GARG, 2014; HOODA; CHHILLAR, 2014; SHAMSHIRI *et al.*, 2015) analisam as técnicas por meio de estudos experimentais que caracterizam o comportamento dessas técnicas, os quais, fornecem constatações que apoiam a proposição de novas abordagens ou adaptações de abordagens existentes. Porém, ainda que desejado, até então não é possível encontrar evidências sobre avaliações de técnicas de geração que demonstrem experiências no contexto de programas concorrentes.

Considerando essa lacuna, este estudo avalia quatro diferentes técnicas de geração dados de teste (Tabela 2) para o teste de programas concorrentes. As técnicas selecionadas são amplamente utilizadas para o teste de programas sequenciais, além disso, novas abordagens as utilizam no contexto de programas concorrentes. Essas técnicas serão aplicadas na geração de dados de teste para programas concorrentes, buscando evidências sobre os potenciais e limitações das técnicas.

Nesse sentido, o contexto deste estudo é caracterizado como sendo *off-line*, conduzido por um aluno de mestrado, abordando um problema real (análise do custo de aplicação, eficácia em revelar defeitos e cobertura de critérios de teste das técnicas de geração de dados de teste), no contexto específico de programas concorrentes.

Tabela 2 – Técnicas de geração avaliadas no estudo.

#	Técnica
GA	Algoritmo genético
GRAND	Aleatória guiada por critérios
CONC	<i>Concolic Testing</i>
GAMO	Algoritmo Genético Multi-Objetivo

4.3.2 Formulação das hipóteses

Com o intuito de auxiliar nas respostas para as questões de pesquisa, hipóteses foram definidas para o estudo, as quais devem ser aceitas ou refutadas por meio dos resultados obtidos dos experimentos. Assim, para cada questão de pesquisa são definidas hipóteses nula e alternativas.

QP1 - *Os conjuntos de teste obtidos pelas técnicas de geração apresentam alta cobertura para os critérios de teste considerados?*

Hipótese Nula (H_0): Não existe diferença, em razão da cobertura de critérios, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_0 : T_N = T_A$$

Hipótese Alternativa (H_1): Existe diferença, em razão da cobertura de critérios, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_1 : T_N \neq T_A$$

QP2 - *Os conjuntos de teste obtidos pelas técnicas de geração são eficazes para revelar defeitos específicos de programas concorrentes?*

Hipótese Nula (H_0): Não existe diferença, em razão da eficácia em revelar defeitos, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_0 : T_N = T_A$$

Hipótese Alternativa (H_1): Existe diferença, em razão da eficácia em revelar defeitos, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_1 : T_N \neq T_A$$

QP3 - *Qual o custo das técnicas de geração em termos dos tamanhos dos conjuntos de teste?*

Hipótese Nula (H_0): Não existe diferença, em razão do custo do teste, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_0 : T_N = T_A$$

Hipótese Alternativa (H_1): Existe diferença, em razão do custo do teste, entre os conjuntos de teste da técnica T_N e a técnica aleatória T_A .

$$H_1 : T_N \neq T_A$$

4.3.3 Seleção de variáveis

Com base no contexto e nas hipóteses estabelecidas para o estudo, as variáveis independentes e dependentes foram caracterizadas e são descritas a seguir.

Variáveis Independentes

Uma variável independente pode ser manipulada ou controlada durante o processo de experimentação. Neste estudo são manipuladas quatro variáveis independentes:

- **VI-1:** Técnica de geração automática de dados de teste / Ferramenta de Geração;
- **VI-2:** Programas concorrentes sob teste / complexidade / paradigmas de comunicação / defeitos semeados;
- **VI-3:** Critérios de teste empregados;
- **VI-4:** Ferramenta de apoio ao teste;

Um experimento estuda o efeito da troca entre uma ou mais variáveis independentes, as quais também são conhecidas como **fatores**. Neste estudo pretende-se analisar, principalmente, os efeitos relacionados as técnicas de geração automática de dados de teste. Para isto, as demais variáveis são fixadas no experimento de modo que não interfiram nos efeitos obtidos. Um **tratamento** é dado por um dos valores que um fator pode assumir, neste caso cada técnica aplicada é considerada um tratamento.

Variáveis Dependentes

As variáveis dependentes de um experimento são obtidas por meio dos resultados obtidos na manipulação das variáveis independentes. Neste estudo são manipuladas três variáveis dependentes, as quais devem caracterizar as técnicas estudadas.

- **VD-1:** Cobertura de critérios: Medida pela quantidade de elementos exercitados / elementos existentes;

- **VD-2:** Eficácia em revelar defeitos: Medida pela quantidade de defeitos revelados / defeitos semeados;
- **VD-3:** Custo de teste: Medida pela quantidade de dados de teste gerados.

4.3.4 Seleção dos Sujeitos de Teste

Assim como descrito na definição, o objetivo deste estudo é avaliar os conjuntos de teste fornecidos pelas técnicas de geração automática de dados de teste. Para isso, programas com características concorrentes, extraídos do *benchmark* TestPar definido por [Dourado \(2015\)](#), foram empregados a fim de avaliar as técnicas investigadas.

Esses programas foram implementados na linguagem Java com o objetivo de apoiar a validação e avaliação de teste estrutural de programas concorrentes com passagem de mensagem e memória compartilhada. A amostra considerada neste estudo, para a análise de cobertura, é composta por 15 programas (Tabela 3), os quais apresentam diferentes características relacionadas à concorrência, como: padrão de comunicação, número de processos e *threads* e quantidade de comunicações. Oito desses programas comunicam-se por meio de passagem de mensagem, quatro por memória compartilhada e três por ambos os paradigmas.

Tabela 3 – Programas sob teste considerados.

#	Programa	LOC	Padrão de Comunicação	Paradigma	Nº Processos	Nº Threads
#1	<i>Greatest Common Divisor</i>	180	Mestre/Escravo	PM	4	4
#2	<i>Greatest Common Divisor - Two Slaves</i>	201	Mestre/Escravo	PM	3	3
#3	<i>Greatest Common Divisor/LeastCommon Multiple</i>	259	Mestre/Escravo	PM	4	4
#4	<i>Token Ring Iterations</i>	175	Produtor-Consumidor	PM	4	4
#5	<i>Token Ring Both Directions Same Primitives</i>	221	Produtor-Consumidor	PM	4	4
#6	<i>Token Ring Both Directions Different Primitives</i>	224	Produtor-Consumidor	PM	4	4
#7	<i>Parallel Sieve of Eratosthenes</i>	274	Árvore	PM	4	4
#8	<i>Roller Coaster</i>	374	Produtor-Consumidor	PM	6	6
#9	<i>Producer Consumer Semaphore Lock Conditions Iterations</i>	241	Todos-para-todos	MC	1	5
#10	<i>Cigarette Smokers</i>	253	Todos-para-todos	MC	1	5
#11	<i>Matrix</i>	228	Todos-para-todos	MC	1	13
#12	<i>Jacobi</i>	411	Todos-para-todos	MC	1	5
#13	<i>Token Ring File</i>	246	Entre vizinhos	PM/MC	4	13
#14	<i>Token Ring Broadcast</i>	294	Entre vizinhos	PM/MC	3	7
#15	<i>Greatest Common Divisor/LeastCommon Multiple Both</i>	462	Mestre/Escravo	PM/MC	4	10

Por sua vez, para avaliação de eficácia em revelar defeitos foram considerados seis

programas do mesmo *benchmark*, porém, sendo estas versões com defeitos (Tabela 4) típicos de programas concorrentes semeados propositalmente. É importante ressaltar que existe apenas um único defeito por programa para se evitar a interferência de um defeito no outro. Assim, o mascaramento (quando um defeito anula o outro) pode ser evitado.

Tabela 4 – Programas sob teste com defeitos.

Programa	Defeito	Paradigma
<i>Producer-Consumer with Semaphore and Locks/Condition</i>	Outros sinais ausentes ou inexistentes	MC
<i>Producer-Consumer with Semaphore and Locks/Condition</i>	Bloqueio de uma região crítica	MC
<i>Greatest Common Divisor</i>	Processo de destino ou origem incorreto em um <i>send</i> ou <i>receive</i>	PM
<i>Greatest Common Divisor</i>	Defeito em um laço que antecede uma primitiva <i>send</i> ou <i>receive</i> impedindo que ocorra a mensagem	PM
<i>Greatest Common Divisor - Two Slaves</i>	Atribuição de forma incorreta usando variáveis	PM
<i>Matrix</i>	Operação não-atômica assumida como atômica	MC

4.3.5 Métricas de Avaliação

Conforme descrito anteriormente, este projeto pretende analisar três fatores distintos: a cobertura de critérios estruturais, detecção de defeitos e custo de teste. O fator de cobertura de critérios pode assumir dois tratamentos, critérios sequenciais e critérios concorrentes. Para este fator foram selecionados oito critérios de teste (Tabela 5), os quais compreendem aspectos do fluxo de dados e de controle dos programas sob teste. Além disso, o conjunto de critérios considerados para este estudo aborda as principais características dos paradigmas de comunicação (passagem de mensagem e memória compartilhada) contemplando também aspectos de programas sequenciais. Os critérios selecionados são utilizados por diversos trabalhos, nos quais é comprovada a aplicabilidade dos critérios por meio de estudos empíricos (BRITO; SOUZA; SOUZA, 2013; HONG *et al.*, 2015).

Tabela 5 – Critérios de teste considerados.

#	Critério	Tipo	Análise
AN	All nodes	Sequencial	Fluxo de controle
AE	All edges	Sequencial	Fluxo de controle
AU	All uses	Concorrente	Fluxo de dados
ASU	All s-uses	Concorrente	Fluxo de dados
AEU	All inter-uses	Concorrente	Fluxo de dados
ASE	All sync-edges	Concorrente	Fluxo de controle
AMU	All m-uses	Concorrente	Fluxo de dados

O fator de detecção de defeitos pode assumir dois tratamentos, nos quais são considerados os erros de observabilidade e de travamento, sendo esses os principais tipos de erros dos programas concorrentes. Os defeitos considerados para este estudo são apresentados na Tabela 6.

Tabela 6 – Defeitos considerados no estudo.

Defeito	Tipo de Erro
Outros sinais ausentes ou inexistentes	travamento
Bloqueio de uma região crítica	travamento
Processo de destino ou origem incorreto em um <i>send</i> ou <i>receive</i>	travamento
Defeito em um laço que antecede uma primitiva <i>send</i> ou <i>receive</i> impedindo que ocorra a mensagem	travamento
Atribuição de forma incorreta usando variáveis	observabilidade
Operação não-atômica assumida como atômica	observabilidade

Finalmente, para o fator do custo de teste é considerado o tratamento de tamanho do conjunto de teste. Esse tratamento visa estabelecer o grau de dificuldade para testar os programas concorrentes por meio de uma técnica de geração automática de dados de teste. O tempo para gerar o conjunto de teste adequado poderia também ser uma métrica para avaliar o custo da técnica. Neste estudo foi decidido não utilizar essa métrica pois o tempo de geração pode ser influenciado em diferentes plataformas das ferramentas de geração.

4.3.6 Instrumentação

A instrumentação tem como objetivo proporcionar meios para a realização do experimento e como monitora-lo, de forma que o controle do experimento não seja afetado, ou seja, os resultados do experimento devem ser os mesmos independentemente de como o mesmo é instrumentado.

Este experimento será conduzido na Universidade de São Paulo (USP) Campus São Carlos. Para a avaliação das técnicas será empregada a ferramenta de apoio ao teste de programas concorrentes ValiPar. Essa ferramenta é utilizada na prova de conceitos e implementa todos os critérios de teste estabelecidos neste estudo. Além disso, permite o teste de aplicações que utilizam os paradigmas de comunicação: passagem de mensagem e memória compartilhada fornecendo um *feedback* sobre a cobertura alcançada pelos conjuntos de teste e defeitos revelados nos programas. As atividades do experimento serão executadas no *Cluster* Halley do laboratório de pesquisa LaSDPC (*Distributed Systems and Concurrent Programming Laboratory*). As configurações do *cluster* são apresentadas na Tabela 7.

Tabela 7 – Configuração de Hardware e Software

Fator	Configuração
Processador Físico	Intel(R) Core(TM) i7-4790 3.60GHz
Memória Física	32 GB RAM
Processador Máquina Virtual	Quad-Core
Memória Máquina Virtual	8 GB RAM
Sistema Operacional	Ubuntu 14.04.1 LTS
Rede	Gigabit Ethernet
Edição Java Runtime	build 1.7.0 71b14

Tendo em vista que o objetivo do experimento foi a avaliação de técnicas de geração foi necessária a inclusão de instrumentos que permitiram a condução do experimento. Neste contexto,

cinco ferramentas de geração de dados de teste escritas na linguagem Java foram selecionadas (ver Tabela 8), as quais implementam as técnicas definidas para este estudo. Em virtude disso, a avaliação de cada técnica será baseada nos resultados obtidos por essas ferramentas.

Tabela 8 – Ferramentas de geração de dados de teste consideradas no experimento.

Ferramenta	Técnica	Execução	Ref.
Evosuite	Algoritmo genético	<i>Plugin</i> /Linha de comando	(FRASER; ARCURI, 2013)
Evosuite-MOSA	Algoritmo Genético Multi-Objetivo	Linha de comando	(PANICHELLA; KIFETEW; TONELLA, 2015)
Randooop	Aleatória (baseline)	<i>Plugin</i> /Linha de comando	(PACHECO; ERNST, 2007)
GRT	Aleatória guiada por critérios	Linha de comando	(MA <i>et al.</i> , 2015)
CodePro	<i>Concolic Testing</i>	<i>Plugin</i>	(GOOGLE-INC., Accessed Feb 2016)

A ferramenta Randooop foi considerada para este estudo com o intuito de comparar os resultados das demais ferramentas, atuando como baseline na execução dos experimentos. Dessa forma, os resultados da Randooop serão utilizados como um nível de aceitação para as demais técnicas, em outras palavras, para que as demais técnicas sejam consideradas apropriadas ou aceitáveis para o teste de programas concorrentes elas devem ser superiores a técnica aleatória com relação às métricas avaliadas. A Randooop foi adicionada neste estudo por se tratar de uma ferramenta aleatória com um baixo custo de geração, além disso, também é utilizada por diversos trabalhos como baseline para comparações de técnicas, ferramentas, proposição de critérios, entre outros.

Vale ressaltar que, todas as ferramentas selecionadas são frequentemente utilizadas em estudos empíricos e contribuem ativamente para demonstrar o estado da arte na geração de dados de teste. Além disso, a maioria dessas ferramentas participa periodicamente de eventos como *Unit Testing Tool Competition* (RUEDA; VOS; PRASETYA, 2015), um dos principais eventos relacionados à área, com o objetivo de avaliar constantemente o cenário atual das ferramentas de geração. Nesse sentido, a escolha das ferramentas foi baseada na representatividade e estabilidade apresentada, uma vez que, apesar de existir um grande número de ferramentas, muitas dessas não estão aptas para o uso, mesmo que em programas sequenciais. Além disso, a maioria não são disponibilizadas pelos autores. A descrição das ferramentas da Tabela 8 é apresentada no Capítulo 3

4.3.7 Avaliação de Validade

Uma questão fundamental sobre um experimento é quão válido é o seu resultado, considerando a possibilidade de generalização dos resultados. Por isso, é importante considerar a questão da validade dos resultados já na fase de planejamento, pois dessa forma é possível planejar a validade adequada dos resultados. A adequação de validade refere-se ao quão válido podem ser os resultados para uma população de interesse.

• Validade de conclusão

A validade de conclusão diz respeito às características que afetam a habilidade de inferir conclusões corretas sobre a relação entre os tratamentos e saídas do experimento. Neste estudo, as conclusões obtidas como resultado podem ser influenciadas pelo teste estatístico escolhido. O teste estatístico é escolhido com base na normalidade dos dados e nas características dos dados coletados. Além disso, uma relação entre as amostras deve ser realizada. Posteriormente, para as variáveis de estudo, serão aplicados testes baseados nos fundamentos da estatística experimental. No que se refere ao risco *Fishing*, isto é, o favorecimento de uma determinada hipótese durante a análise dos dados, o mesmo não é considerado uma ameaça neste estudo, pois não há interesse em concluir em nenhuma das hipóteses, mas sim descobrir evidências do problema investigado.

• Validade interna

Ameaças à validade interna são influências que podem afetar as variáveis independentes no que diz respeito à causalidade, as quais o pesquisador não possui conhecimento ou controle. Considerando esse grupo de ameaças, o presente estudo apresenta alguns aspectos que podem influenciar os resultados obtidos. Como exemplo, as ferramentas de geração são consideradas uma ameaça, pois não foram desenvolvidas pelo pesquisador e consequentemente existem aspectos desconhecidos na implementação das mesmas, os quais podem influenciar os resultados. Para tratar essa ameaça foram utilizados os mesmos cenários de avaliação, ou seja, todas as ferramentas foram aplicadas no mesmo conjunto de programas e avaliadas na mesma ferramenta de apoio ao teste, de forma que nenhuma ferramenta seja privilegiada. Outro fator a ser observado é a ferramenta de apoio ao teste (Valipar), pois a instrumentação, a geração de requisitos e a avaliação dos testes são obtidas por meio dessa ferramenta. No entanto, para este caso acredita-se que esta não é considerada uma ameaça à validade, pois essa ferramenta já foi utilizada, satisfatoriamente, por outros estudos que investigam a aplicabilidade de critérios e demais características da ferramenta. Diante disto, entende-se que as ameaças levantadas foram mitigadas para a validade interna do experimento.

• Validade de Construção

A validade de construção refere-se à relação entre a teoria e a observação de um experimento. Em outras palavras, a validade de construção está relacionada ao fato do experimento refletir de maneira realística a influência dos tratamentos nos resultados observados. Para este estudo considera-se o conhecimento do pesquisador sobre os programas sob teste uma possível ameaça na aplicação das técnicas de geração de dados de teste. Essa ameaça poderia ocorrer em caso de preferência por uma determinada hipótese, para a qual o pesquisador pudesse favorecer uma determinada técnica em virtude do conhecimento sobre os programas empregados e então planejar e executar o experimento de modo a favorecer essa técnica. Entretanto, considerando que nenhuma das ferramentas foi desenvolvida neste projeto e que ambas foram executadas no mesmo ambiente, essa ameaça não apresenta complicações que possam influenciar os resultados no que diz respeito ao favorecimento de alguma técnica analisada.

- **Validade externa**

A validade externa refere-se a generalização dos resultados do experimento para outros contextos, como por exemplo programas de software industriais. Devido a restrições do conjunto de ferramentas utilizado e demais características no escopo deste estudo essa aplicação não foi realizada. Apesar disso, os sujeitos de teste selecionados para este estudo apresentam os principais aspectos intrínsecos de programas concorrentes. Assim como descrito anteriormente, o conjunto de programas selecionados retratam diferentes características concorrentes, considerando diferentes paradigmas e padrões de comunicação, complexidades distintas e defeitos típicos de estratégias concorrentes.

4.4 Operação

A etapa de operação do experimento consiste na preparação dos artefatos e ferramentas, na execução das atividades definidas no planejamento do estudo e na validação dos resultados obtidos durante a execução. A seguir são descritas em detalhes cada uma dessas fases.

4.4.1 Preparação

A fase de preparação do experimento estabelece o ambiente onde o mesmo será executado. Neste estudo foi necessária a preparação de um ambiente de testes para que os dados fornecidos pelas técnicas de geração pudessem ser executados e posteriormente avaliados. O ambiente de teste considerado neste trabalho consiste na ferramenta de apoio ao teste de programas concorrentes ValiPar. A ferramenta requer uma versão do *Java Runtime* e *Java Development Kit* versão 7 ou superior e um sistema operacional que ofereça suporte para Java, porém a ferramenta só foi testada nos ambientes Ubuntu, Mac e Windows. A instalação da ferramenta requer o arquivo de instalação (ainda não disponível publicamente) e a configuração de uma variável de ambiente necessária para execução dos comandos da ferramenta. É importante ressaltar que para uma execução confiável da ferramenta, principalmente para um grande número de testes, é necessário executá-la em um diretório com um grande espaço de armazenamento, pois a ferramenta gera um grande conjunto de arquivos e sub-diretórios durante cada execução de cada teste.

Conforme descrito anteriormente foram utilizadas outras ferramentas para geração dos dados de testes, as quais representam as técnicas avaliadas no estudo. Dessa forma foi necessário a configuração do ambiente de cada ferramenta. A seguir são descritas em detalhes as configurações de cada ferramenta empregada. Vale ressaltar que todas as ferramentas foram empregadas em sistema operacional Ubuntu 14.04 LTS com uma JVM (*Java Virtual Machine*) versão 7.

- **EvoSuite**

A EvoSuite é uma ferramenta desenvolvida em Java e pode ser executada com Java

runtime 8 ou superior. A ferramenta oferece execução por meio de linha de comando, plugin para *Eclipse IDE* e plugin para *IntelliJ IDE*. Neste estudo a ferramenta foi utilizada como plugin do Eclipse no sistema operacional Ubuntu. A instalação do plugin foi realizada automaticamente inserindo o link da EvoSuite (<http://www.evosuite.org/update/>) no Eclipse.

- **EvoSuite-MOSA**

Conforme descrito anteriormente a EvoSuite-MOSA é uma extensão da EvoSuite, entretanto, essa ferramenta pode ser executada apenas em linha de comando. Dessa forma, é necessário um arquivo JAR da ferramenta (disponível em <http://goo.gl/tzOF09>) e um sistema operacional que suporte Java runtime 7 ou superior.

- **Randoop**

A ferramenta Randoop pode ser executada em linha de comando ou como plugin para o Eclipse IDE. Neste estudo foi utilizado o plugin para o eclipse, o qual foi instalado automaticamente por meio do link da Randoop (<http://randoop.googlecode.com/hg/plugin.updateSite/>).

- **GRT**

A GRT é uma ferramenta de geração de testes de unidade desenvolvida em Java, a qual pode ser utilizada apenas por linha de comando. Para isso é necessário um arquivo Jar da ferramenta (disponível em [GRT](#)) e um sistema operacional que suporte Java runtime 7 ou superior.

- **CodePro AnalytiX**

A ferramenta CodePro AnalytiX é uma ferramenta de teste que possui um componente de geração de testes de unidade. A ferramenta pode ser utilizada apenas como plugin e requer uma versão do Eclipse 3.4 ou superior. A CodePro pode ser instalada automaticamente no Eclipse IDE por meio de uma busca como um termo chave (codePro) no repositório de plugins do Eclipse.

Além das ferramentas empregadas foi necessária a preparação do *benchmark* utilizado. Para todos os 21 programas utilizados neste estudo foram necessárias alterações que possibilitaram a geração de dados de teste por meio das técnicas avaliadas e para a execução dos mesmos na ferramenta ValiPar. Essas alterações foram necessárias devido às restrições das ferramentas e algumas limitações no *benchmark*.

4.4.2 Condução

A avaliação foi dividida em quatro experimentos, cada experimento compreende uma técnica diferente a ser analisada. Apesar disso, todas as técnicas foram avaliadas considerando os mesmos tratamentos e os mesmos programas sob teste, afim de obter resultados que não sejam influenciados por nenhuma configuração específica ou que haja o favorecimento de uma técnica em particular. Cada experimento é composto por 7 etapas, sendo elas:

1. **Geração dos dados de testes para o programa sob teste:** A geração é realizada por meio da ferramenta estabelecida para cada técnica avaliada. Neste experimento cada técnica gerou dez conjuntos de teste para cada programa.
2. **Transformação dos dados em entradas para ferramenta ValiPar:** Os dados fornecidos pelas ferramentas são inicialmente armazenados em classes do JUnit. Porém, a ferramenta ValiPar necessita de entradas específicas que requerem o dado gerado e informações sobre identificação de processos e *threads*. Nesta fase os dados gerados são transformados, computacionalmente, em um arquivo *shell script* de entradas válidas para ferramenta ValiPar. Esse arquivo, quando chamado, realiza as operações de adição de testes na ferramenta ValiPar.
3. **Instrumentação do programa sob teste:** Para avaliar os testes inseridos é necessário armazenar o caminho percorrido por esse teste durante a sua execução. A instrumentação refere-se a marcações no código que permitem identificar quais trechos de código foram executados. Nesta etapa é gerado um novo código instrumentado no qual serão realizados os testes.
4. **Geração dos elementos requeridos:** Conforme descrito anteriormente, os critérios estabelecem elementos que devem ser exercitados durante o teste. Nesta etapa a ferramenta ValiPar faz uma busca no código para identificar esses elementos, considerando os critérios implementados para essa busca. Assim, o artefato gerado por essa atividade é a lista de elementos requeridos do programa sob teste.
5. **Adição dos testes:** Os testes fornecidos pelas técnicas devem ser carregados na ValiPar antes da execução. Para este objetivo o *script* de execução do experimento realiza uma chamada ao arquivo contendo os dados de teste para ValiPar. Quando os testes são adicionados, a ValiPar cria um diretório para cada teste, pois dessa forma é possível rastrear individualmente o caminho percorrido por um teste. Além disso, cada execução desse teste também é armazenada em um diretório diferente.
6. **Execução dos testes:** Após carregar os dados de teste para a ferramenta é possível executá-los por meio de um identificador único criado para cada teste durante a sua inserção. Neste experimento, os dez conjuntos gerados pelas técnicas devem ser repetidos dez vezes. Essa repetição é utilizada para minimizar o problema do não-determinismo, assim, considera-se que as dez repetições são aceitáveis para executar a maior parte, senão todas, as possíveis sincronizações executáveis de um programa sob teste.
7. **Análise de cobertura/defeitos revelados:** A avaliação de cobertura de elementos requeridos e defeitos revelados pode ser realizada após a execução dos testes. Para cada um dos dez conjuntos executados é gerado um arquivo de avaliação diferente, esses arquivos serão utilizados para definir uma média final que representa, com maior nível de significância,

os resultados obtidos pelas técnicas. É importante considerar que, conforme descrito anteriormente, a ValiPar fornece um módulo para testar diferentes sequências de sincronização com o objetivo de melhorar a cobertura do código, considerando os mesmos testes executados. Entretanto, apesar de desejável, esse módulo não foi considerado neste trabalho devido ao custo de sua utilização para o número de testes fornecidos pelas técnicas. Pois, para cada técnica foram considerados quinze programas, para os quais foram gerados dez conjuntos de teste cada, e todos foram executados por dez vezes neste experimento.

8. **Tabulação dos resultados:** Os resultados obtidos por cada conjunto de teste são armazenados em um arquivo texto. Esta etapa consiste em tabular, computacionalmente, esses dados para extrair a cobertura alcançada por cada conjunto. Além disso, é realizada a quantificação dos defeitos revelados por uma técnica para cada conjunto de teste gerado. A tabulação dos resultados permite a manipulação dos dados e a aplicação dos testes estatísticos, os quais são fundamentais para os testes de hipóteses.
9. **Testes de hipóteses:** Os resultados obtidos dos testes de hipóteses deverão auxiliar as respostas das questões de pesquisa. Assim, para aceitar ou rejeitar as hipóteses de interesse será considerado um nível de significância nos testes estatísticos.

Exceto o teste de hipóteses e a geração dos dados para algumas ferramentas, todas as demais etapas são executadas computacionalmente, por meio de *scripts* que permitiram as execuções e repetições dos experimentos no Cluster. Em síntese, o experimento pode ser compreendido em três etapas principais, conforme apresentado cronologicamente na Figura 9.

Figura 9 – Visão geral dos experimentos realizados. Retângulos com símbolos |+| representam a existência de um subprocesso de atividade.

Na primeira fase do experimento, para cada técnica estudada, são gerados os conjuntos de teste. Todas as técnicas foram executadas, por meio das ferramentas de geração, utilizando-se os mesmos sujeitos de teste sob as mesmas condições. O cenário de execução que cada técnica foi submetida é ilustrado na Figura 10.

O subprocesso **Geração dos conjuntos de teste** inicia-se com a atividade de adequação dos sujeitos de teste. Essa adequação é necessária devido as diferentes restrições impostas pelas

Figura 10 – Ambiente de geração de dados de teste empregado para as técnicas.

ferramentas de geração automática de dados de teste. Posteriormente as fases de geração dos conjuntos de teste foram executadas, as quais foram repetidas até que dez conjuntos de teste para cada sujeito (ou programa) fossem obtidos. A repetição dessa atividade teve como intuito aumentar a confiança dos resultados obtidos.

Todos os conjuntos fornecidos pelas ferramentas foram executados nos seus respectivos programas e então avaliados. O cenário de execução de testes para cada conjunto obtido é apresentados na Figura 11.

Figura 11 – Cenário de teste aplicado para cada conjunto de teste obtido.

Assim como para as ferramenta de geração, para a execução da ferramenta ValiPar foram necessárias adaptações que possibilitaram a condução do experimento. Note que existem dois códigos diferentes para um mesmo programa sob teste, um para as ferramentas de geração e outro para o teste na ValiPar. Os códigos para geração e teste possuem pequenas diferenças que são necessárias para a execução de ambas as tarefas, apesar disso, essas modificações não alteram o programa do ponto de vista do teste de software.

Após a adequação do sujeito de teste, inicia-se a execução da ferramenta ValiPar, contendo o código modificado recebido da etapa anterior. Essa etapa consiste no teste dos sujeitos aplicando-se os conjuntos obtidos pelas técnicas de geração automática de dados de teste, a qual fornece os resultados que serão tabulados na etapa seguinte. A execução da ValiPar contempla subetapas importantes para a execução dos conjuntos de teste. O cenário de execução da ferramenta ValiPar é apresentado na Figura 12.

Figura 12 – Cenário de execução ferramenta ValiPar.

A execução inicia-se com a instrumentação do código do sujeito teste, a qual permite verificar o rastro de uma execução do código. Posteriormente, é gerado um conjunto de elementos requeridos para o programa sob teste, os quais são definidos de acordo com os critérios de teste definidos pela ferramenta. Após a geração dos elementos requeridos adiciona-se o conjunto de teste fornecido pela técnica de geração na ferramenta ValiPar. Em seguida, é realizada a execução dos dados de teste contidos no conjunto. Essa etapa foi executada dez vezes para cada teste, afim de aumentar a confiança dos resultados e diminuir o impacto do não-determinismo sobre os resultados. Por último, é realizada avaliação de critérios, a qual fornece um arquivo de avaliação contendo a cobertura atingida pelos testes em relação aos elementos requeridos exercitados. Na Figura 13 é ilustrado os diretórios e arquivos necessários para cada programa durante o teste na ValiPar.

Figura 13 – Organização dos arquivos gerados durante o experimento.

Cada programa sob teste possui dois diretórios principais, um para o código e outro para

os arquivos necessários para ValiPar. O diretório de código é composto pelos principais arquivos necessários para geração de dados de testes e pelos artefatos de saída da geração, enquanto o diretório ValiPar consiste nos arquivos necessários para a execução do teste e os arquivos de saída da ValiPar. Para cada teste executado no programa é possível identificar os caminhos percorridos e os elementos alcançados por cada execução desse teste. O sub-diretório "Projeto ValiPar" é replicado para cada um dos dez conjuntos gerados, assim as informações de avaliação de todos os conjuntos são armazenadas separadamente.

4.5 Considerações Finais

Neste capítulo foi abordado a definição e planejamento do experimento seguindo as diretrizes definidas por Wohlin *et al.* (2000), e também a descrição das possíveis ameaças à validade deste estudo. Posteriormente, foram descritas a preparação do ambiente experimental e forma como os dados foram coletados a partir do experimento. A análise dos dados coletados é apresentada no próximo capítulo.

ANÁLISE DOS RESULTADOS

5.1 Considerações Iniciais

Neste capítulo são apresentados os dados coletados durante as execuções dos experimentos juntamente com as análises realizadas. Para a análise dos resultados os dados obtidos por meio dos experimentos foram avaliados baseando-se nos princípios da estatística descritiva. Esses princípios contribuem para organização e sumarização dos resultados contribuindo para melhor compreensão dos mesmos. Além disso, também foi utilizado o teste de hipótese com o objetivo de validar a pesquisa e obter conclusões claras sobre os aspectos investigados.

Este capítulo está organizado da seguinte forma: a Seção 5.2 apresenta a análise de cobertura de critérios em diferentes paradigmas de comunicação. Na Seção 5.3 é apresentada a análise de eficácia em revelar defeitos considerando diferentes tipos de defeitos. A análise de custo em relação ao teste de programas concorrentes é apresentada na Seção 5.4, em seguida é realizada a discussão dos resultados na Seção 5.5. Por último são apresentadas as considerações finais do capítulo.

5.2 QP1 - Análise de Cobertura de Critérios

A análise de cobertura de critérios considera sete critérios de teste, os quais apresentam características referentes a aspectos sequenciais e concorrentes. A taxa de cobertura é dada pela seguinte equação:

$$Cobertura = \left(\frac{ElementosCobertos}{ElementosRequeridos} \right) * 100 \quad (5.1)$$

Para melhor compreensão desta análise os resultados foram divididos em relação aos paradigmas de comunicação.

Passagem de Mensagem

Para análise de cobertura de critérios foram aplicados testes de normalidade nas amostras com o intuito de auxiliar a escolha do teste de média adequado para essa análise. Os testes *Anderson-Darling* e *Shapiro-Wilk* foram empregados, porém, ambos indicaram que as amostras não seguem a normalidade. Por esse motivo optou-se por um teste de média não-paramétrico que pode ser utilizado para análises que apresentam mais de duas amostras, *Kruskal-Wallis*. Em adição aplicou-se o teste de comparações múltiplas de *Bonferroni*. Na Figura 14 são apresentados gráficos dos critérios de teste em relação à cobertura atingida por cada técnica de geração.

Os resultados apresentados na Figura 14 demonstram que em todos os critérios a técnica aleatória foi superior as demais técnicas. Sendo assim, a Hipótese Nula(H_0) foi rejeitada com 95% de significância, ou seja, há evidências na análise de cobertura critérios, em programas de passagem de mensagem, que demonstram diferenças entre as técnicas para os critérios investigados.

Conforme esperado, os critérios utilizados em programas sequenciais (*all nodes* (AN) e *all edges* (AE)), ou critérios mínimos, apresentaram maior cobertura em relação aos demais critérios. Apesar disso, um resultado importante obtido nessa análise foi cobertura do critério específico para programas de passagem de mensagem *all m-uses* (AMU), o qual apresentou cobertura entre as técnicas de forma similar aos critérios sequenciais. Em três técnicas investigadas (Aleatória (RAND), Concolic Testing (CONC) e Algoritmo Genético Multi-objetivo (GAMO)) a cobertura do critério AMU foi superior a 81%, enquanto a menor cobertura avaliada foi de 24%, obtida pela técnica Aleatória Guiada (GRAND).

A técnica *Concolic Testing* foi a segunda técnica melhor avaliada pelos critérios, sendo inferior a terceira melhor (GAMO) apenas para o critério AMU. Este resultado apresenta uma questão importante sobre a relação de cobertura entre os critérios sequenciais e concorrentes. Pois neste caso, uma melhor cobertura dos critérios sequenciais não foi um bom parâmetro para alcançar critérios concorrentes, uma vez que, existem diferenças significativas que demonstram a superioridade da técnica GAMO sobre a técnica CONC em relação ao critério AMU, apesar dessa mesma técnica ter sido inferior a técnica CONC nos dois critérios sequenciais.

Nesta análise também foi possível identificar uma relação em comum entre as técnicas GA e GRAND, pois somente para o critério *all synchronization-edges* (ASE) foi possível encontrar resultados que demonstram diferenças significantes de cobertura em relação as duas técnicas. Ainda assim, ambas as técnicas apresentaram resultados muito inferiores em relação as demais técnicas.

Outra questão importante a ser considerada nesta análise é a relação de cobertura de critérios nas diferentes técnicas investigadas, pois dessa forma é possível identificar a relação entre critérios sob uma mesma técnica. Na Figura 15 são apresentados os resultados obtidos por cada técnica em relação a todos os critérios em análise.

Figura 14 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo (GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória(RAND) em diferentes critérios de teste para programas de passagem de mensagem. *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All intEr-Uses* (AEU), *All Sync-Edges* (ASE), *All M-Uses* (AMU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração.

Figura 15 – Porcentagem de cobertura dos critérios *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All intEr-Uses* (AEU), *All Sync-Edges* (ASE), *All M-Uses* (AMU) em diferentes técnicas de geração de dados de teste para programas de passagem de mensagem. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.

O critério de maior cobertura na maioria das técnicas foi o AN seguido pelos critérios AE e AMU. Em todas as técnicas é possível identificar uma forte relação entre esses três critérios, uma vez que entre as técnicas esses podem ser considerados os critérios de menor complexidade. Neste mesmo cenário, o critério AEU apresenta a menor cobertura entre os critérios para todas as técnicas investigadas. Em virtude disto, existem evidências significativas que demonstram ser o AEU o critério de maior complexidade, entre os investigados, para as técnicas em análise. Dessa forma, percebe-se o quão distante estão as técnicas investigadas em relação a este critério, uma vez que a técnica aleatória, sendo esta uma técnica simples e de baixo custo computacional, foi a melhor técnica em relação a este critério, conforme apresentado na Figura 14. Ainda assim, apenas 13% dos elementos requeridos para este critério foram cobertos pela técnica RAND.

Dentre as técnicas apresentadas o GA possui a menor variação entre os critérios, uma vez que apresenta apenas três grupos significativamente diferentes, enquanto as demais técnicas apresentam pelo menos seis grupos. Este resultado demonstra que essa técnica apresenta maior amplitude durante a otimização, pois os diferentes critérios possuem maior relação em comparação com as demais técnicas. Apesar disso, esses resultados não garantem que em um nível mais alto de cobertura essa relação seria semelhante.

Retomando a questão sobre a relação entre os critérios sequenciais e o critério AMU, conforme apresentado na Figura 15 percebe-se que a cobertura do critério AMU é significativamente superior ao critério AE apenas nas técnicas GAMO e CONC. Sendo assim, não é possível inferir que estes critérios requerem testes distintos, uma vez que essa relação só é identificada em duas das técnicas avaliadas. Além disso, a DMS (Diferença Mínima Significativa) analisada é pequena em relação a estes critérios.

Memória Compartilhada

Assim como nos programas de passagem de mensagem, a análise de memória compartilhada utilizou o teste de média *Kruskal-Wallis* e o procedimento de comparações múltiplas de *Bonferroni*. Na Figura 16 são apresentados os gráficos dos critérios de teste em relação as técnicas investigadas.

A análise de cobertura dos programas de comunicação por memória compartilhada apresenta diferenças significativas entre todos os critérios avaliados. Sendo assim, rejeita-se a Hipótese Nula (H_0) em relação à diferença entre as técnicas, ou seja, há evidências que demonstram diferenças entre as técnicas analisadas.

Em oposto a análise de cobertura dos programas de passagem de mensagem, a técnica de melhor cobertura em todos os critérios foi a técnica GRAND, apesar disso, não existe diferença significativa nos critérios AU e ASU que demonstre a superioridade em relação a técnica baseline (técnica aleatória). Apesar disso, considera-se este um bom resultado, uma vez que pode ser considerado superior ou igual ao mínimo esperado para os programas concorrentes.

Figura 16 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória(RAND) em diferentes critérios de teste para programas de memória compartilhada. *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All Sync-Edges* (ASE), *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração.

Outra questão inversa são os resultados obtidos pela técnica GAMO, dado que na análise anterior a mesma foi superior as técnicas GRAND e GA, sendo a terceira melhor técnica para todos os critérios. Nesta análise a mesma obteve os piores resultados para os critérios avaliados.

No entanto, a diferença entre as técnicas para todos os critérios analisados é menor se comparada aos programas de passagem de mensagem. Como exemplo, os critérios sequenciais obtiveram coberturas superiores a 74% nos programas de memória compartilhada, ao passo que nos programas de passagem de mensagem a mínima identificada foi de 30%. É importante ressaltar, que nessa análise existem diferenças entre os critérios concorrentes em relação à análise anterior, pois esta análise foi acrescida com o critério o critério ASU (*all shared-uses*), ao passo que os critérios AMU e AEU não foram analisados devido as suas características específicas de programas de passagem de mensagem.

Os critérios ASU e AU apresentam uma grande quantidade de elementos requeridos para os programas analisados, conforme apresentado no Apêndice A. Visualmente os gráficos correspondentes a esses critérios não apresentam diferenças, pois a cobertura obtida é muito inferior a escala apresentada nos gráficos. Dessa forma, para uma melhor discussão sobre estes critérios na Figura 17 os critérios AU e ASU são apresentados em uma escala menor.

Figura 17 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de memória compartilhada. *All Uses* (AU) e *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.

Como pode ser observado nos gráficos da Figura 17, ambos os critérios possuem baixa cobertura em todas as técnicas investigadas, sendo mais agravante para o critério ASU. A baixa cobertura do critério AU deve-se ao fato de que a grande maioria dos seus requisitos de teste é composta pelos elementos requeridos do critério ASU.

Grande parte dos elementos requeridos pelo critério ASU são não-executáveis, assim,

uma análise sobre a porcentagem de elementos executáveis poderia apresentar melhor os resultados das técnicas, no entanto, devido às restrições impostas pela ferramenta de teste e a inviabilidade de calcular manualmente esse grupo requisitos, essa análise não foi realizada. Apesar disso, esses resultados demonstram existência de igualdade de cobertura entre GRAND e CONC em comparação a técnica baseline (aleatória), ou seja, o mínimo esperado é atingido.

Para identificar as diferenças entre critérios sob cada técnica investigada, uma relação de cobertura dos critérios para cada técnica é apresentada na Figura 18.

Na análise de programas de memória compartilhada os grupos de diferenças significativas entre os critérios foi a mesmo para todas as técnicas investigadas. Os critérios de maior cobertura foram os critérios sequenciais AN e AE, respectivamente nessa ordem de magnitude. Enquanto os critérios ASU e AU foram os que obtiveram menor cobertura dentre todas as técnicas de geração.

Passagem de Mensagem e Memória Compartilhada

A análise de ambos os paradigmas de comunicação teve como objetivo identificar o comportamento das técnicas em relação aos critérios, quando as características de ambos os paradigmas estão presentes. A importância dessa análise em conjunto é comprovada devido aos diferentes resultados entre as análises dos paradigmas de forma isolada. Para isso, os mesmos testes de média e de comparações múltiplas das análises anteriores foram empregados. Na Figura 19 são apresentados os resultados obtidos por meio da cobertura dos critérios em relação às técnicas de geração.

Os resultados obtidos dos programas com ambos os paradigmas de comunicação apresentaram diferenças significativas em todos os sete critérios analisados. Assim, rejeita-se a hipótese nula (H_0) em relação à igualdade de cobertura de critérios entre as técnicas investigadas.

Conforme demonstram os gráficos apresentados na Figura 19, a técnica RAND apresenta superioridade significativa nos critérios AE, ASE, AEU e AMU em relação as outras técnicas. Por sua vez, a técnica GAMO é superior as demais técnicas no que diz respeito ao critério AN. Nos critérios AU e ASU as técnicas RAND e GAMO são iguais, porém superiores as demais técnicas avaliadas.

As menores coberturas atingidas nesta análise pertencem a técnica GRAND, para a qual existem diferenças significativas que demonstre sua inferioridade em relação às demais técnicas nos critérios AE, AEU, AN, ASU e AU. Os demais critérios estão em um mesmo nível de igualdade a técnica GA.

Assim como na análise de memória compartilhada os critérios AU e ASU obtiveram resultados que não podem ser visivelmente comparados na mesma escala dos demais critérios. Assim, os mesmos são apresentados na Figura 20 para melhor discussão dos resultados.

Figura 18 – Porcentagem de cobertura dos critérios *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All intEr-Uses* (AEU), *All Sync-Edges* (ASE), *All M-Uses* (AMU) em diferentes técnicas de geração de dados de teste para programas de memória compartilhada. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.

Figura 19 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de passagem de mensagem e memória compartilhada. *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All Sync-Edges* (ASE), *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração.

Figura 20 – Porcentagem de cobertura das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND) em diferentes critérios de teste para programas de passagem de mensagem e memória compartilhada. *All Uses* (AU) e *All S-Uses* (ASU). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre as técnicas de geração.

Assim como na avaliação dos programas de memória compartilhada uma grande parte dos requisitos do critério AU é composta pelos requisitos do critério ASU, entretanto, como estes programas apresentam características dos dois paradigmas, o critério AU neste caso é composto tanto pelos elementos requeridos do critério ASU como os do critério AMU. Ainda assim, a quantidade de elementos requeridos pelo critério ASU é bem superior, atingindo uma média aproximada de 175 mil elementos ao passo que o critério AMU apresenta uma média de 17 elementos requeridos.

Vale ressaltar que os resultados obtidos pelas técnicas são obtidos de três programas diferentes durante esta análise, para os quais aplicaram-se cálculos de médias que determinaram os resultados aqui demonstrados. Em razão disto, os resultados da técnica CONC obtiveram uma média inferior devido à incapacidade de gerar dados de teste para um dos programas, conforme apresentado na Tabela 4 no Apêndice A. O programa em questão apresenta o maior número de elementos sequenciais dentre todos os demais analisados, assim, acredita-se que a complexidade do programa inviabilizou a utilização da técnica CONC.

Os resultados desta análise em relação aos critérios por técnica investigada são apresentados na Figura 21.

Os critérios sequenciais, assim como nas análises anteriores, apresentaram a maior cobertura, já os demais critérios apresentaram diferentes classificações de cobertura entre as técnicas de geração, exceto os critérios AU e ASU que possuem o mesmo grau de cobertura em todas as técnicas, sempre nesta ordem AU superior a ASU.

Em vista dos resultados observados, em relação a cobertura dos critérios, percebe-se

Figura 21 – Porcentagem de cobertura dos critérios *All Nodes* (AN), *All Edges* (AE), *All Uses* (AU), *All intEr-Uses* (AEU), *All Sync-Edges* (ASE), *All M-Uses* (AMU) em diferentes técnicas de geração de dados de teste para programas de passagem de mensagem e memória compartilhada. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de cobertura entre os critérios de teste.

que em grande maioria a técnica RAND (ou técnica aleatória) possui maior cobertura em relação as demais. Este resultado demonstra que as técnicas não realizam o mínimo esperado para cobertura de critérios de programas concorrentes. A técnica aleatória possui um baixo nível de complexidade e realiza a geração dos dados de teste sem nenhuma informação sobre características dos programas. Sendo assim, o custo imposto durante a geração pelas demais técnicas investigadas não apresentam melhorias na cobertura dos programas concorrentes aqui explorados, ou pelo menos não o suficiente para viabilizar a sua utilização.

5.3 QP2 - Análise de Eficácia em Revelar Defeitos

A análise de eficácia em revelar defeitos considera seis defeitos distintos, os quais estão presentes em seis programas diferentes, sendo um defeito por programa. Esses programas possuem tanto características de passagem de mensagem quanto memória compartilhada. O teste de média realizado considerou dez repetições, as quais forneceram uma taxa de detecção de defeitos. O teste *Kruskal-Wallis* foi empregado no teste de médias e o procedimento de *Bonferroni* aplicado no teste de análises múltiplas. O cálculo de eficácia é dado pela seguinte equação:

$$Eficácia = \left(\frac{DefeitosRevelados}{DefeitosInseridos} \right) * 100 \quad (5.2)$$

Os resultados obtidos da análise de eficácia são apresentados na Figura 22.

O primeiro fator avaliado durante a análise de eficácia foi o teste de hipótese, o qual demonstrou diferenças significativas entre as técnicas em relação a eficácia em revelar defeitos, rejeitando assim a Hipótese Nula (H_0). No entanto, considerando amostras independentes não é possível rejeitar a hipótese nula para o defeito F1, uma vez que o mesmo foi revelado por todas as técnicas em todas as repetições realizadas. Por outro lado, todos os demais defeitos apresentam diferenças entre as técnicas.

Conforme apresentado na Figura 22 a técnica RAND pertence ao grupo de melhor eficácia em todos os defeitos analisados, além disso, foi a única técnica capaz de revelar o defeito F5. Assim como a técnica RAND, a técnica CONC apresentou 100% de eficácia para todos os demais defeitos. Os gráficos correspondem a eficácia de cada técnica para cada tipo de defeito, enumerados de F1 a F6.

Por outro lado, as técnicas GA, GAMO e GRAND foram inferior as demais técnicas na maioria dos defeitos analisados. Como exemplo o defeito F6 (operação não-atômica assumida como atômica) foi coberto por todas as técnicas de geração, no entanto a técnica GA só foi capaz de revelar esse defeito em 80% das repetições. Considerando que em dez repetições esse defeito foi revelado oito vezes esse resultado não pode ser considerado crítico. Em contraste, nos defeitos F3 e F4 existe uma grande diferença entre as três piores técnicas e as duas melhores,

Figura 22 – Porcentagem de eficácia das técnicas de geração Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC) e Aleatória (RAND) aplicadas para detecção de diferentes defeitos. Outros sinais ausentes ou inexistentes (F1), bloqueio de região crítica (F2), processo de destino ou origem incorreto (F3), defeito em um laço que antecede uma primitiva *send* ou *receive* (F4), atribuição de forma incorreta usando variáveis (F5) e operação não-atômica assumida como atômica (F6). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de eficácia entre as técnicas de geração.

pois enquanto as técnicas CONC e RAND revelaram estes defeitos em todas as repetições, as técnicas GA, GAMO e GRAND revelaram os defeitos em apenas duas, uma e nenhuma repetição, respectivamente.

Outra questão importante em relação à análise de eficácia é verificar o grau de dificuldade que cada técnica possui em relação aos defeitos considerados. Essas informações são ilustradas na Figura 23, na qual são apresentadas diferenças significativas ou não em relação aos defeitos sob uma mesma técnica.

Conforme ilustrado nos gráficos da Figura 23 o defeito de maior dificuldade de detecção entre as técnicas é o defeito F5, sendo revelado apenas pela técnicas RAND. Todos os demais defeitos foram revelados por pelo menos uma técnica com um grau de 100% de eficácia, ou seja, revelado em todas as repetições.

Em relação as técnicas GA, GRAND e GAMO a maior complexidade encontrada, com exceção ao critério F5, foi ocasionada pelos defeitos F3 e F4. Esses defeitos foram inseridos em programas de comunicação por passagem de mensagem. Sendo assim, a dificuldade em revelar estes defeitos, no que diz respeito as técnicas GA e GRAND, pode estar ligada a baixa cobertura dos critérios específicos desse tipo de programa, pois, conforme ilustrado na Figura 14, essas técnicas obtiveram as piores taxas de cobertura em relação as demais técnicas. Em contraste, a técnica GAMO apresentou um nível de cobertura próximo das técnicas RAND e CONC e ainda assim não foi capaz de revelar estes defeitos de forma eficaz.

Ainda sobre a relação cobertura/eficácia vale ressaltar que os critérios específicos de programas de comunicação por memória compartilhada não apresentaram cobertura satisfatória entre as técnicas analisadas, conforme ilustrados na Figura 16. Por outro lado, os defeitos inseridos nos programas de memória compartilhada (F1 e F2) foram revelados com 100% de eficácia por pelo menos três técnicas de geração. Nesse sentido, pode-se inferir que, neste caso, os defeitos presentes no *Benchmark* utilizado não estão intrinsecamente ligados aos critérios correspondentes ao tipo de programa investigado.

Uma questão importante revelada nestes resultados é que a dificuldade em revelar os defeitos de memória compartilhada (F1,F2,F6) é menor em relação aos defeitos inseridos nos programas de passagem de mensagem (F3,F4,F5). Considerando que os critérios de passagem de mensagem apresentaram maior cobertura dentre técnicas em relação aos critérios de memória compartilhada, conforme ilustrado nas Figuras 14 e 16, é possível identificar que a detecção dos defeitos analisados, nos programas de memória compartilhada, não é fortemente afetada pelas características de fluxo de dados desses programas. Em outras palavras, mesmo com a baixa cobertura dos critérios AU e ASU os defeitos desses programas foram revelados com alta eficácia pelas técnicas, podendo indicar que estes critérios não podem ser considerados bons parâmetros para revelar esses tipos de defeitos.

Figura 23 – Porcentagem de eficácia em revelar defeitos, Outros sinais ausentes ou inexistentes (F1), bloqueio de região crítica (F2), processo de destino ou origem incorreto (F3), defeito em um laço que antecede uma primitiva *send* ou *receive* (F4), atribuição de forma incorreta usando variáveis (F5) e operação não-atômica assumida como atômica (F6). Considerando diferentes técnicas de geração, Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC) e Aleatória (RAND). Letras diferentes sobre as barras simbolizam a existência de diferença significativa de eficácia entre as técnicas de geração.

5.4 QP3 - Análise de Custo

A análise de custo baseou-se no tamanho do conjunto de teste gerado pelas técnicas investigadas. A importância dessa análise deve-se ao custo computacional necessário para testar programas concorrentes, pois conjuntos de tamanhos exorbitantes podem ser inviáveis para realização da atividade de teste. Baseando-se nessas informações a técnica aleatória foi definida como baseline para esta avaliação.

Para efeito de comparação entre as técnicas foi utilizado o teste de média *Kruskal-Wallis* e o procedimento de comparações múltiplas de *Bonferroni*. Os resultados obtidos durante essa avaliação estão sumarizados na Figura 24.

Figura 24 – Custo obtido pela média dos tamanhos dos conjuntos gerados pelas técnicas. Algoritmo Genético (GA), Algoritmo Genético Multi-Objetivo (GAMO), *Concolic Testing* (CONC), Aleatória Guiada (GRAND) e Aleatória (RAND). Letras diferentes sobre a linha e as barras simbolizam a existência de diferença significativa de custo entre as técnicas de geração.

O teste de média realizado apresentou diferenças entre as técnicas, ou seja, há evidências que demonstram custos diferentes entre as técnicas ao nível de 95% de significância. Desse modo, rejeita-se a Hipótese Nula (H_0) de que todas as técnicas são iguais em relação ao custo dos conjuntos de teste.

Com uma ampla diferença de custo em relação as demais técnicas, a técnica GRAND foi a técnica de maior custo entre as técnicas. Considerando a técnica baseline (RAND) como controle para esta avaliação apenas a técnica GRAND apresentou custo superior a técnica de controle, gerando conjunto de teste com média de 279 dados de teste por conjunto. É importante

ressaltar que este custo poderia ser ainda maior, pois durante a geração foi necessário limitar em 500 dados de teste por conjunto, dado que na ausência desse limite a técnica forneceu conjuntos com quantidade superior a 30 mil dados de teste para alguns dos programas.

Os melhores resultados obtidos, em relação ao custo, foram fornecidos pelas técnicas GA e GAMO, as quais apresentaram em média 10 e 5,8 dados de teste por conjunto gerado, respectivamente. Esses resultados demonstram que essas técnicas estão aptas, em relação ao custo, para o teste de programas concorrentes, uma vez que apresentaram custo inferior a técnica aleatória. Por sua vez, a técnica CONC também fornece resultados que viabilizam a sua utilização, considerando que o custo obtido não apresenta diferenças significativas em relação ao custo da técnica aleatória. Apesar disso, esses resultados não demonstram a aplicabilidade dessas técnicas em um contexto geral, ou seja, em relação à eficácia, cobertura e custo. Assim, optou-se por realizar uma avaliação considerando todos os fatores analisados, de forma que seja possível ranquear as técnicas considerando o custo de seus conjuntos gerados, a eficácia em revelar defeitos de programas concorrentes e a cobertura de critérios dos programas avaliados. Na Figura 25 é apresentado a relação entre cobertura, eficácia e custo para todas as técnicas investigadas.

Figura 25 – Relação de cobertura, eficácia e custo nas técnicas de geração de dados de teste. Algoritmo Genético(GA), Algoritmo Genético Multi-objetivo(GAMO), Aleatória Guiada(GRAND), *Concolic Testing*(CONC), Aleatória (RAND). Letras minúsculas diferentes simbolizam diferenças significativas de cobertura entre as técnicas, letras maiúsculas diferentes simbolizam diferenças significativas de eficácia entre as técnicas (*Kruskal-Wallis*, $p < 0,05$).

Com o objetivo de ranquear as técnicas foi elaborado um score para a classificações dos fatores, os valores para cada classificação é apresentado na Tabela 9.

Tabela 9 – Score de classificação dos fatores cobertura, eficácia e custo.

Eficácia		Custo		Cobertura	
Símbolo	Score	Símbolo	Score	Símbolo	Score
A	100	a	100	a	100
B	50	b	50	b	50
C	25			c	25
D	12,5	c	25	d	12,5
E	6,25			e	6,25

Com base nos scores definidos na Tabela 9 foi realizada uma classificação das técnicas em relação aos três fatores aqui investigados. O cálculo realizado para atribuição do score final de cada técnica é dado pela seguinte equação:

$$ScoreFinal = \left(\frac{Custo}{100} * \frac{100}{3} \right) + \left(\frac{Cobertura}{100} * \frac{100}{3} \right) + \left(\frac{Eficácia}{100} * \frac{100}{3} \right) \quad (5.3)$$

Os resultados parciais e finais de cada técnica para a classificação são apresentados na Tabela 10. A primeira coluna apresenta classificação das técnicas, já a segunda coluna são apresentadas as identificações das técnicas. As colunas seguintes apresentam o score de cobertura, score de eficácia e score de custo, respectivamente. Por último, na sexta coluna, é apresentado o score final das técnicas.

Tabela 10 – Classificação das técnicas em relação aos fatores cobertura, eficácia e custo obtida pela Equação 5.3.

Class.	Técnica	Score Cobertura	Score Eficácia	Score Custo	Score Final
1º	RAND	100	100	50	83,3
2º	GAMO	6,25	50	100	52,1
3º	GA	12,5	12,5	100	41,7
3º	CONC	50	25	50	41,7
5º	GRAND	25	6,25	25	18,8

Na análise geral dos fatores considerados foi possível identificar que o melhor resultado pertence a técnica baseline (RAND), ou seja, as técnicas investigadas não apresentaram o mínimo esperado para o teste de programas concorrentes. As técnicas GA e CONC obtiveram resultados iguais, entretanto, a técnica CONC foi melhor em relação à cobertura e eficácia. Ainda nesta análise, os piores resultados obtidos foram obtidos pela técnica GRAND.

Com o objetivo de sumarizar as análises realizadas e levantar algumas questões sobre as mesmas, foi realizada uma discussão geral sobre os resultados obtidos, a qual é descrita na seção seguinte.

5.5 Discussão

Até onde se sabe, este estudo é o primeiro relato sobre a avaliação de técnicas de geração automática de dados de teste em programas concorrentes. A maioria dos estudos relacionados a

geração de dados teste para programas concorrentes estão ligados a proposição de novos métodos ou adaptações de métodos já existentes. Além disso, este é primeiro estudo que considera a geração de dados de teste para ambos os paradigmas de comunicação, passagem de mensagem e memória compartilhada. Neste estudo foi possível identificar que as técnicas empregadas para programas sequenciais não apresentam bons resultados em relação à cobertura de critérios concorrentes, eficácia em revelar defeitos específicos dessa classe de programas e também em relação ao custo empregado para o teste.

A maioria dos estudos na literatura para geração de dados de teste em programas concorrentes utilizam a técnica de Algoritmo Genético durante a geração (STEENBUCK; FRASER, 2013; SILVA; SOUZA; SOUZA, 2014; TIAN; GONG, 2014a; TIAN; GONG, 2014b), os quais consideram apenas um paradigma de comunicação por trabalho. Além desses, também existem estudos que exploram a técnica aleatória por meio de estratégias que guiam a geração dos dados de teste (NISTOR *et al.*, 2012). Em geral, os estudos reportam sobre defeitos, antes não revelados, detectado por essas técnicas ou cobertura superior em relação a outro método de geração, como exemplo a técnica aleatória. No entanto, muitos dos trabalhos não consideram todas as possíveis sincronizações, conforme requerido pelo critério *all synchronization-edges* (ASE), ou não abordam o fluxo de dados entre diferentes processos ou *threads*, como requerido pelos critérios *all message-uses* (AMU), *all shared-uses* (ASU) e *all uses* (AU).

Em contraste a esses trabalhos, os resultados descritos neste estudo demonstram que nenhuma das técnicas é superior a técnica aleatória em nenhum dos fatores analisados. Além disso, as técnicas GRAND e GA, empregadas por trabalhos da literatura em programas concorrentes, apresentaram os piores resultados em relação à eficácia em revelar defeitos para quatro dos seis defeitos considerados (Figura 22). A técnica GRAND apresenta o maior custo entre todas as técnicas investigadas (Figura 24) sendo necessária a limitação para o tamanho dos conjuntos de teste, dado a inviabilidade de executar a quantidade de testes gerados. No que diz respeito a cobertura de critérios essas técnicas obtiveram os piores resultados em todos critérios analisados para os programas de comunicação por passagem de mensagem (Figura 14). Apesar disso, na análise dos programas com memória compartilhada houveram resultados significativos em relação aos critérios obtidos por essas técnicas, principalmente para os critérios sequencias (Figura 16). Ainda assim, o custo despedindo para técnica GRAND chega aos 500 dados de teste por conjunto de teste gerado, o máximo estipulado nos experimentos para essa técnica. Já na análise dos programas de ambos os paradigmas os piores resultados se repetem para essas técnicas na maioria dos critérios considerados (Figura 19).

A avaliação neste estudo considerou apenas cinco técnicas de geração, sendo uma delas utilizada como controle. Neste contexto, os resultados não garantem que todas as técnicas de geração utilizadas em programas sequenciais são inviáveis para o teste de programas concorrentes, uma vez que existe uma diversidade de técnicas que não foram avaliadas. Apesar disso, as técnicas avaliadas foram extraídas de três principais grupos distintos de técnicas, aleatórias, meta-

heurísticas e execução simbólica em conjunto com execução dinâmica. Assim acredita-se que estes resultados podem ser replicados de forma semelhante para as demais técnicas existentes.

Em relação aos benefícios deste estudo é possível citar três características mais importantes:

- Cobertura dos critérios sequenciais: Os resultados relacionados à cobertura de critérios sequenciais dos programas demonstram uma cobertura aceitável para os programas concorrentes. Sendo assim, as características presentes nas técnicas podem contribuir de forma efetiva na cobertura desses critérios contribuindo para melhoria de cobertura dos demais critérios;
- Cobertura dos critérios concorrentes: Em todas as técnicas analisadas, inclusive a técnica aleatória, existem critérios que não possuem cobertura superior a 1,5% dos elementos requeridos (Figura 17 e 20), assim espera-se que trabalhos futuros dediquem esforços significativos para esses critérios. O mesmo não ocorre para o critério *all m-uses* (AMU), o qual apresenta uma cobertura próxima, em alguns casos superior ou igual (Técnicas CONC, RAND, GA e GAMO Figura 14), aos critérios sequenciais;
- Eficácia em revelar defeitos: Em relação à técnica aleatória apenas para um dos seis defeitos (Defeito F1 Figura 22) considerados houve diferença significativa que demonstra a técnica aleatória superior as demais técnica, ainda assim esse defeito só foi coberto em 20% de suas execuções. Os resultados demonstram que, em relação à eficácia em revelar defeitos, as técnicas analisadas não estão aptas na detecção defeitos, apesar disso, os resultados obtidos pela técnica aleatória não estão distante para algumas das técnicas analisadas.

Ainda que a técnica RAND seja mais eficiente para geração de dados de teste que as demais técnicas os resultados obtidos não são suficiente para o teste dessas aplicações, uma vez que nem todos os tipos de defeitos são efetivamente revelados por essa técnica. Estes resultados indicam a necessidade de técnicas específicas para programas concorrentes.

5.6 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos na condução do estudo experimental. O capítulo apresentou as análises referentes a cobertura de critérios, considerando os diferentes paradigmas de comunicação. Além disso, foi explorada a análise de eficácia em revelar defeitos específicos de programas concorrentes e a análise custo empregado ao teste. Por último foi descrita a discussão do trabalho.

CONCLUSÕES E TRABALHOS FUTUROS

6.1 Caracterização da Contribuição

Este projeto de mestrado teve como objetivo o estabelecimento e a condução de um estudo experimental para avaliar as técnicas de geração automática de dados de teste em programas concorrentes considerando a eficácia em revelar defeitos, cobertura de critérios e custo empregado para a atividade de teste. Para alcançar esse objetivo foram selecionadas técnicas de geração, *benchmark* de programas concorrentes e critérios de teste com características concorrentes. Os resultados obtidos por meio de análises estatísticas forneceram subsídios que permitiram classificar as técnicas investigadas em relação aos fatores considerados durante a avaliação. Estes resultados indicaram que as técnicas utilizadas em programas sequenciais não podem ser empregadas de forma eficaz em programas concorrentes, uma vez que a técnica de controle (aleatória) apresentou melhores resultados em todos os fatores observados. Além disso, foi possível observar que os critérios de programas com memória compartilhada apresentam maiores desafios em comparação aos de passagem de mensagem, uma vez que demandam uma quantidade superior de elementos requeridos. Outra questão observada foi que os defeitos analisados foram, em grande parte, cobertos pela maiorias das técnicas, mesmo com um baixo nível de cobertura de critérios concorrentes dos programas com defeitos.

6.2 Contribuições de Pesquisa

Dentre as contribuições identificadas podem-se destacar:

- Os resultados obtidos em relação a cobertura do critério *all m-uses* (AMU): O critério AMU obteve cobertura próxima aos critérios sequenciais, o que pode indicar que as técnicas existentes podem contribuir para satisfazer esse critério;

- Os resultados a respeito da cobertura de critérios específicos de programas com memória compartilhada: Os critérios de memória compartilhada apresentaram um baixo grau de cobertura entre as técnicas, inclusive a aleatória, assim percebe-se que esse critério necessita de maior esforço na geração dos dados de teste;
- Os resultados identificados em relação à eficácia em revelar defeitos de todas as técnicas observadas: Em sua maioria, os defeitos foram revelados por todas as técnicas investigadas, o que pode indicar que esses defeitos sejam revelados utilizando-se de características sequenciais dos programas sob teste;
- A classificação das técnicas em relação a todos os fatores avaliados: Esta avaliação permitiu ranquear as técnicas considerando todos os fatores investigados. Assim foi possível indicar quais os potenciais de cada técnica em relação às demais e também seus pontos negativos, contribuindo para melhorias futuras de ambas as técnicas.

6.3 Trabalhos Futuros

Como trabalhos futuros pretende-se realizar a replicação deste estudo considerando abordagens híbridas entre essas e outras técnicas de geração, analisando o impacto que a junção das técnicas fornece ao teste de programas concorrentes. Além disso, espera-se definir quais elementos são não-executáveis e determinar se algum tipo de técnica é capaz de cobrir elementos não exercitado pelas demais. Para isso, pretende-se utilizar uma amostra mais significativa considerando diferentes *benchmarks* e sistemas empregados na indústria. Após a realização dessas atividades pretende-se desenvolver uma abordagem de geração de dados de teste para programas concorrentes considerando os potenciais das técnicas anteriormente investigadas e elaborar estratégias que atendam as demais características não alcançadas pelas técnicas existentes.

REFERÊNCIAS

AHMED, B. S.; ZAMLI, K. Z. Comparison of metaheuristic test generation strategies based on interaction elements coverage criterion. In: **IEEE Symposium on Industrial Electronics and Applications**. [S.l.: s.n.], 2011. Citado 2 vezes nas páginas 61 e 71.

AHMED, M. A.; HERMADI, I. Ga-based multiple paths test data generator. **Computers and Operations Research**, Elsevier Science, v. 35, n. 10, p. 3107–3124, out. 2008. ISSN 0305-0548. Citado 2 vezes nas páginas 52 e 60.

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing**. 2nd. ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 0805304436 9780805304435. Citado 2 vezes nas páginas 26 e 27.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381. Citado 4 vezes nas páginas 23, 31, 32 e 35.

ANAND, S.; BURKE, E.; CHEN, T. Y.; CLARK, J.; COHEN, M. B.; GRIESKAMP, W.; HARMAN, M.; HARROLD, M. J.; MCMINN, P. An orchestrated survey on automated software test case generation. **Journal of Systems and Software**, v. 86, n. 8, p. 1978–2001, August 2013. Citado 3 vezes nas páginas 13, 53 e 54.

ANDREOU, A.; ECONOMIDES, K.; SOFOKLEOUS, A. An automatic software test-data generation scheme based on data flow criteria and genetic algorithms. In: **7th IEEE International Conference on Computer and Information Technology**. [S.l.: s.n.], 2007. p. 867–872. Citado na página 60.

ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and notations for concurrent programming. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 15, n. 1, p. 3–43, mar. 1983. ISSN 0360-0300. Citado na página 37.

BAGNARA, R.; CARLIER, M.; GORI, R.; GOTTLIEB, A. Symbolic path-oriented test data generation for floating-point programs. In: **Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on**. [S.l.: s.n.], 2013. p. 1–10. Citado na página 52.

BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. In: _____. [S.l.: Elsevier, 2007. cap. Teste Estrutural, p. 46–76. Citado 2 vezes nas páginas 32 e 33.

BARNEY, B. **Introduction to Parallel Computing**. 2010. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Citado 2 vezes nas páginas 13 e 38.

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 49.

BERKA, T.; HAGENAUER, H.; VAJTERSIC, M. A middleware for concurrent programming in mpi applications. In: **40th International Conference on Parallel Processing Workshops (ICPPW)**. [S.l.: s.n.], 2011. p. 269–278. ISSN 1530-2016. Citado na página 36.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: **Future of Software Engineering**. [S.l.: s.n.], 2007. p. 85–103. Citado na página 51.

BOUCHACHIA, A. An immune genetic algorithm for software test data generation. In: **7th International Conference on Hybrid Intelligent Systems**. [S.l.: s.n.], 2007. p. 84–89. Citado na página 26.

BOYER, R. S.; ELSPAS, B.; LEVITT, K. N. A formal system for testing and debugging programs by symbolic execution. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 10, n. 6, p. 234–245, abr. 1975. ISSN 0362-1340. Citado na página 24.

BRAR, M. K.; GARG, S. Article: Survey on automated test data generation. **International Journal of Computer Applications**, v. 108, n. 15, p. 1–4, December 2014. Citado na página 71.

BRITO, M. A.; SOUZA, S. R.; SOUZA, P. S. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. **Procedia Computer Science**, v. 18, p. 250–259, 2013. Citado na página 75.

BRUNETON, E.; LENGLET, R.; COUPAYE, T. Asm: a code manipulation tool to implement adaptable systems. **Adaptable and extensible component systems**, v. 30, p. 19, 2002. Citado na página 64.

CAO, Y.; HU, C.; LI, L. An approach to generate software test data for a specific path automatically with genetic algorithm. In: **8th International Conference on Reliability, Maintainability and Safety**. [S.l.: s.n.], 2009. p. 888–892. Citado na página 26.

CHAINATE, W.; THAPATSUWAN, P.; PONGCHAROEN, P. Investigation on cooling schemes and parameters of simulated annealing for timetabling university courses. In: **Advanced Computer Theory and Engineering, 2008. ICACTE '08. International Conference on**. [S.l.: s.n.], 2008. p. 200–204. Citado na página 59.

CHEN, T.; LEUNG, H.; MAK, I. Adaptive random testing. In: MAHER, M. (Ed.). **Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making**. [S.l.]: Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3321). p. 320–329. ISBN 978-3-540-24087-7. Citado na página 24.

COLANZI, T. E.; VERGILIO, S. R.; ASSUNÇÃO, W. K. G.; POZO, A. Search based software engineering: Review and analysis of the field in brazil. **Journal of Systems and Software**, v. 86, n. 4, p. 970–984, 2013. Citado na página 56.

DARA, R.; LI, S.; LIU, W.; SMITH-GHORBANI, A.; TAHVILDARI, L. Using dynamic execution data to generate test cases. In: **IEEE International Conference on Software Maintenance**. [S.l.: s.n.], 2009. p. 433–436. ISSN 1063-6773. Citado na página 25.

DEB, K.; KALYANMOY, D. **Multi-Objective Optimization Using Evolutionary Algorithms**. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN 047187339X. Citado na página 61.

DELAMARO, M.; MALDONADO, J.; JINO, M. **Introdução ao teste de software**. [S.l.]: Elsevier, 2007. ISBN 9788535226348. Citado 6 vezes nas páginas 13, 23, 29, 30, 32 e 34.

DEVASENA, M. S. G.; VALARMATHI, M. L. Article: Search based software testing technique for structural test case generation. **International Journal of Applied Information Systems**, v. 1, n. 6, p. 20–25, February 2012. Published by Foundation of Computer Science, New York, USA. Citado na página 52.

DIJKSTRA, E. W. **Cooperating Sequential Processes**, Technical Report EWD-123. [S.l.], 1965. Citado na página 38.

DOGANAY, K.; BOHLIN, M.; SELLIN, O. Search based testing of embedded systems implemented in iec 61131-3: An industrial case study. In: **Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on**. [S.l.: s.n.], 2013. p. 425–432. Citado na página 57.

DOURADO, G. G. M. **Contribuindo para a avaliação do teste de programas concorrentes: uma abordagem usando benchmarks**. Dissertação (Mestrado) — Universidade de São Paulo, 2015. Citado na página 74.

DURAN, J. W.; NTAFOSS, S. An evaluation of random testing. **IEEE Transactions on Software Engineering**, SE-10, n. 4, p. 438–444, July 1984. ISSN 0098-5589. Citado na página 24.

EDVARDSSON, J. **A Survey on Automatic Test Data Generation**. 1999. Citado 2 vezes nas páginas 24 e 52.

FARCHI, E.; NIR, Y.; UR, S. Concurrent bug patterns and how to test them. In: **Proceedings of International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2003. ISSN 1530-2075. Citado na página 41.

FRASER, G.; ARCURI, A. Whole test suite generation. **IEEE Transactions on Software Engineering**, v. 39, n. 2, p. 276–291, 2013. Citado 2 vezes nas páginas 66 e 77.

GAO, J. Z.; TSAO, J.; WU, Y.; JACOB, T. H.-S. **Testing and Quality Assurance for Component-Based Software**. Norwood, MA, USA: Artech House, Inc., 2003. ISBN 1580534805. Citado 3 vezes nas páginas 13, 32 e 33.

GENDREAU, M.; POTVIN, J.-Y. **Handbook of Metaheuristics**. 2nd. ed. [S.l.]: Springer Publishing Company, 2010. Citado na página 57.

GHIDUK, A.; HARROLD, M.; GIRGIS, M. Using genetic algorithms to aid test-data generation for data-flow coverage. In: **14th Asia-Pacific Software Engineering Conference**. [S.l.: s.n.], 2007. p. 41–48. ISSN 1530-1362. Citado na página 26.

GHIDUK, A. S. Automatic generation of basis test paths using variable length genetic algorithm. **Information Processing Letters**, v. 114, n. 6, p. 304–316, 2014. ISSN 0020-0190. Citado 3 vezes nas páginas 26, 52 e 60.

GIRGIS, M. Automatic test data generation for data flow testing using a genetic algorithm. **Journal of Universal Computer Science**, v. 11, n. 6, p. 898–915, 2005. Citado na página 60.

GOOGLE-INC. **Codepro analytix user guide**. Accessed Feb 2016. Disponível em: <https://developers.google.com/java-dev-tools/codepro/doc/features/junit/test_case_generation>. Citado 2 vezes nas páginas 66 e 77.

GRAHAM, D.; BLACK, R.; VEENENDAAL, E. V.; EVANS, I. **Foundations Of Software Testing: ISTQB Certification**. [S.l.]: Cengage Learning, 2006. ISBN 9781844803552. Citado na página 24.

GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. **Introduction to Parallel Computing**. 2. ed. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 9780201648652. Citado na página 39.

HANSEN, P. B. **The Origin of Concurrent Programming**. New York, NY, USA: Springer-Verlag New York, Inc., 2002. 3-61 p. ISBN 0-387-95401-5. Citado na página 36.

HONG, S.; STAATS, M.; AHN, J.; KIM, M.; ROTHERMEL, G. Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. **Software Testing, Verification and Reliability**, v. 25, n. 4, p. 334–370, 2015. Citado na página 75.

HOODA, I.; CHHILLAR, R. A review: Study of test case generation techniques. **International Journal of Computer Applications**, v. 107, n. 16, p. 33–37, December 2014. Citado na página 71.

HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. [S.l.]: McGraw-Hill Inc, 1993. Citado na página 27.

KHOR, S.; GROGONO, P. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In: **Proceedings of the 19th IEEE International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2004. (ASE '04), p. 346–349. ISBN 0-7695-2131-2. Citado 2 vezes nas páginas 26 e 60.

KOIRALA, S.; SHEIKH, S. **Software Testing Interview Questions**. [S.l.]: Ebsco Publishing, 2009. (Computer science series). ISBN 9780763782979. Citado na página 24.

KOREL, B. Automated software test data generation. **Software Engineering, IEEE Transactions on**, v. 16, n. 8, p. 870–879, Aug 1990. ISSN 0098-5589. Citado 2 vezes nas páginas 55 e 56.

LEBLANC, T.; MARKATOS, E. Shared memory vs. message passing in shared-memory multiprocessors. In: **Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing**. [S.l.: s.n.], 1992. p. 254–263. Citado na página 38.

MA, L.; ARTHO, C.; ZHANG, C.; SATO, H.; GMEINER, J.; RAMLER, R. Grt: Program-analysis-guided random testing (t). In: **30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 212–223. Citado 2 vezes nas páginas 64 e 77.

MAHAJAN, M.; KUMAR, S.; PORWAL, R. Applying genetic algorithm to increase the efficiency of a data flow-based test data generation approach. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 37, n. 5, p. 1–5, set. 2012. ISSN 0163-5948. Citado na página 26.

MAJUMDAR, R.; SEN, K. Hybrid concolic testing. In: **International Conference on Software Engineering**. [S.l.: s.n.], 2007. ISSN 0270-5257. Citado na página 56.

MAO, C.; YU, X. Test data generation for software testing based on quantum-inspired genetic algorithm. **International Journal of Computational Intelligence and Applications**, v. 12, n. 01, 2013. Citado na página 60.

- MARTINEZ, F.; GONZALEZ, J.; STOJMENOVIC, I. A parallel hill climbing algorithm for pushing dependent data in clients-providers-servers systems. In: **Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on**. [S.l.: s.n.], 2002. p. 611–616. ISSN 1530-1346. Citado 2 vezes nas páginas 57 e 58.
- MCCABE, T. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, Dec 1976. ISSN 0098-5589. Citado na página 35.
- MCMINN, P. Search-based software test data generation: A survey: Research articles. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Citado 2 vezes nas páginas 24 e 25.
- MOHI-ALDEEN, S. M.; DERIS, S.; MOHAMAD, R. Systematic mapping study in automatic test case generation. In: **New Trends in Software Methodologies, Tools and Techniques**. [S.l.]: IOS Press Ebooks, 2014, (Frontiers in Artificial Intelligence and Applications). p. 703–720. Citado 4 vezes nas páginas 51, 59, 61 e 71.
- MOSHIZI, M. M.; BARDSIRI, A. K. The application of meta-heuristic algorithms in automatic software test case generation. **International Journal of Mathematical Sciences and Computing**, v. 1, n. 3, p. 1–8, 2015. Citado na página 71.
- MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962, 9781118031964. Citado 3 vezes nas páginas 23, 29 e 34.
- NGO, M. N.; TAN, H. B. K. Detecting large number of infeasible paths through recognizing their patterns. In: **Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2007. p. 215–224. ISBN 978-1-59593-811-4. Citado na página 54.
- NISTOR, A.; LUO, Q.; PRADEL, M.; GROSS, T. R.; MARINOV, D. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In: **34th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2012. ISSN 0270-5257. Citado 2 vezes nas páginas 27 e 106.
- NOURANI, Y.; ANDRESEN, B. A comparison of simulated annealing cooling strategies. **Journal of Physics A: Mathematical and General**, v. 31, n. 41, p. 8373, 1998. Citado na página 59.
- OFFUTT, A.; SEAMAN, E. Using symbolic execution to aid automatic test data generation. In: **Computer Assurance, 1990. COMPASS '90, Systems Integrity, Software Safety and Process Security., Proceedings of the Fifth Annual Conference on**. [S.l.: s.n.], 1990. p. 12–21. Citado na página 24.
- PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed random testing for java. In: **Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications**. [S.l.: s.n.], 2007. p. 815–816. Citado 3 vezes nas páginas 13, 63 e 77.
- PANCHAPAKESAN, A.; ABIELMONA, R.; PETRIU, E. Dynamic white-box software testing using a recursive hybrid evolutionary strategy/genetic algorithm. In: **IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2013. p. 2525–2532. Citado 2 vezes nas páginas 25 e 26.

- PANICHELLA, A.; KIFETEW, F.; TONELLA, P. Reformulating branch coverage as a many-objective optimization problem. In: **IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2015. p. 1–10. Citado 2 vezes nas páginas 66 e 77.
- PATTON, R. **Software Testing (2nd Edition)**. Indianapolis, IN, USA: Sams, 2005. ISBN 0672327988. Citado na página 24.
- PINTO, G.; VERGILIO, S. A multi-objective genetic algorithm to test data generation. In: **22nd IEEE International Conference on Tools with Artificial Intelligence**. [S.l.: s.n.], 2010. v. 1, p. 129–134. ISSN 1082-3409. Citado na página 26.
- PRADO, R. R.; SOUZA, P. S. L.; DOURADO, G. G. M.; SOUZA, S. R. S.; ESTRELLA, J. C.; BRUSCHI, S. M.; LOURENCO, J. Extracting static and dynamic structural information from java concurrent programs for coverage testing. In: **Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2015. p. 1–8. Citado na página 47.
- PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 7. ed. New York, NY, USA: McGraw Hill, Inc., 2010. Citado 2 vezes nas páginas 24 e 32.
- QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656. Citado 4 vezes nas páginas 13, 38, 39 e 40.
- QUINTANA, G.; SOLARI, M. A systematic mapping study on experiments with automatic structural test case generation. In: **38th Latin America Conference on Informatics, CLEI 2012 - Conference Proceedings**. [S.l.: s.n.], 2012. Citado na página 71.
- RAPPS, S.; WEYUKER, E. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, SE-11, n. 4, p. 367–375, April 1985. ISSN 0098-5589. Citado na página 35.
- RIBEIRO, J. C. B.; RELA, M. Z.; VEGA, F. F. de. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software. In: **Proceedings of the 3rd International Workshop on Automation of Software Test**. New York, NY, USA: [s.n.], 2008. (AST '08), p. 85–92. ISBN 978-1-60558-030-2. Citado na página 60.
- RÜNGER, G.; RAUBER, T. **Parallel Programming - for Multicore and Cluster Systems; 2nd Edition**. [S.l.]: Springer, 2013. ISBN 978-3-642-37800-3. Citado na página 37.
- RUEDA, U.; VOS, T.; PRASETYA, I. Unit testing tool competition – round three. In: **IEEE/ACM 8th International Workshop on Search-Based Software Testing (SBST)**. [S.l.: s.n.], 2015. p. 19–24. Citado na página 77.
- RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN 0136042597, 9780136042594. Citado 2 vezes nas páginas 25 e 57.
- SHAMSHIRI, S.; JUST, R.; ROJAS, J. M.; FRASER, G.; MCMINN, P.; ARCURI, A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: **Proceedings of the International Conference on Automated Software Engineering (ASE)**. Lincoln, NE, USA: [s.n.], 2015. p. 201–211. Citado na página 71.

SHULL, F.; MENDONÇA, M. G.; BASILI, V.; CARVER, J.; MALDONADO, J. C.; FABBRI, S.; TRAVASSOS, G. H.; FERREIRA, M. C. Knowledge-sharing issues in experimental software engineering. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 9, n. 1-2, p. 111–137, 2004. ISSN 1382-3256. Citado na página 69.

SILVA, J. D. P.; SOUZA, S. R.; SOUZA, P. S. L. Geração automática de dados de teste para programas concorrentes com uso de meta-heurísticas. In: **8th Brazilian Workshop on Systematic and Automated Software Testing (SAST)**. [S.l.: s.n.], 2014. p. 71–80. Citado 3 vezes nas páginas 27, 52 e 106.

SOMMERVILLE, I. **Software Engineering**. 9. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1. Citado 2 vezes nas páginas 29 e 30.

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**, v. 26, n. 1, p. 21–50, 2014. ISSN 1532-0634. Citado na página 40.

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**, v. 18, p. 149–158, 2013. ISSN 1877-0509. Citado 2 vezes nas páginas 45 e 47.

SOUZA, S.; VERGILIO, S.; SOUZA, P. S. L. Introdução ao teste de software. In: _____. [S.l.]: Elsevier, 2007. v. 1, cap. Teste de Programas Concorrentes, p. 231–249. Citado 3 vezes nas páginas 39, 41 e 44.

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. **Software Testing, Verification and Reliability**, 2015. ISSN 1099-1689. Citado na página 27.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; aO, A. S. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., v. 20, n. 16, p. 1893–1916, 2008. ISSN 1532-0634. Citado 3 vezes nas páginas 43, 44 e 46.

STEENBUCK, S.; FRASER, G. Generating unit tests for concurrent classes. In: **IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2013. p. 144–153. Citado na página 106.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Citado 3 vezes nas páginas 27, 37 e 39.

TIAN, T.; GONG, D. Evolutionary generation of test data for path coverage of message-passing parallel programs. **China journal of computers**, v. 36, n. 11, p. 2212–2223, 2013. Citado na página 62.

_____. Evolutionary generation approach of test data for multiple paths coverage of message-passing parallel programs. **Chinese Journal of Electronics**, v. 23, p. 291, 2014. Citado 4 vezes nas páginas 27, 41, 62 e 106.

_____. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. **Automated Software Engineering**, Springer US, p. 1–32, 2014. ISSN 0928-8910. Citado na página 106.

- VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. In: _____. [S.l.]: Elsevier, 2007. v. 1, cap. Geração de Dados de Teste, p. 269–313. Citado 4 vezes nas páginas 24, 52, 55 e 56.
- VILELA, R. F.; SOUZA, P. S. L.; DELAMARO, M. E.; SOUZA, S. R. S. Evidências sobre configurações de algoritmos genéticos para geração automática de dados de teste. In: **Proceedings of XIX Ibero-American Conference on Software Engineering**. [S.l.: s.n.], 2016. Citado na página 60.
- WAZLAWICK, R. S. **Engenharia de software: conceitos e práticas**. 1. ed. Rio de Janeiro: Elsevier, 2013. Citado na página 35.
- WHITTAKER, J. A.; ARBON, J.; CAROLLO, J. **How Google tests software**. Upper Saddle River, NJ: Addison-Wesley, 2012. ISBN 978-0-321-80302-3. Citado na página 67.
- WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5. Citado 3 vezes nas páginas 69, 71 e 85.
- WONG, W.; LEI, Y.; MA, X. Effective generation of test sequences for structural testing of concurrent programs. In: **Proceedings 10th IEEE International Conference on Engineering of Complex Computer Systems**. [S.l.: s.n.], 2005. p. 539–548. Citado na página 26.
- WU, L. L.; KAISER, G. E. **Constructing Subtle Concurrency Bugs Using Synchronization-Centric Second-Order Mutation Operators**. [S.l.], 2011. Citado na página 41.
- XANTHAKIS, S.; ELLIS, C.; SKOURLAS, C.; GALL, A. L.; KATSIKAS, S.; KARAPOULIOS, K. Application of genetic algorithms to software testing. In: **Proceedings of the 5th International Conference on Software Engineering and Applications**. [S.l.: s.n.], 1992. p. 625–636. Citado na página 60.
- XIAO, M.; EL-ATTAR, M.; REFORMAT, M.; MILLER, J. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. **Empirical Software Engineering**, Kluwer Academic Publishers-Plenum Publishers, v. 12, n. 2, p. 183–239, 2007. ISSN 1382-3256. Citado na página 58.
- XIONG, L.; SHI, J. Automatic generating test paper system based on genetic algorithm. In: **Second International Workshop on Education Technology and Computer Science**. [S.l.: s.n.], 2010. v. 3, p. 272–275. Citado na página 26.
- ZHANG, Y.; CHEN, Z.; WANG, J. Speculative symbolic execution. In: **IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2012. p. 101–110. ISSN 1071-9458. Citado na página 53.
- ZHANG, Y.; GONG, D. Evolutionary generation of test data for multiple paths coverage with faults detection. In: **IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications**. [S.l.: s.n.], 2010. p. 406–410. Citado na página 26.

ANÁLISE DE COBERTURA DAS TÉCNICAS DE GERAÇÃO

Quadro 2 – Relação da cobertura de critérios dos programas obtida pela técnica GA.

Programas		Cobertura de Critérios						
		AN	AE	AU	ASU	AEU	ASE	AMU
Passagem de Mensagem	GCD	36,1% ¹	27,6%	14,6%	-	4,5%	12,1%	40%
		101 ²	131	250		195	33	10
	GCD - Two Slaves	59%	51%	15,4%	-	5,5%	27,5%	55%
		86	107	271		80	16	8
	GCD/LCM	20,5%	15,4%	6,8%	-	2,1%	10,9%	36,3
		232	308	637		384	57	16
	T. R. Iterations	60%	54,1%	27,82%	-	10%	20%	40%
		96	136	284		24	12	4
	T. R. Both Directions Same Primitives	29%	20,9%	9,1%	-	3,2%	13,3%	20%
		89	118	391		63	12	4
	T. R. Both Directions Different Primitives	28,5%	24,3%	10,3%	-	0,7%	5,9%	11,8%
		161	221	442		726	51	22
Memória Compartilhada	Sieve of Eratosthenes	28,5%	24,5%	7,5%	-	6,4%	18%	21,4%
		199	272	1279		42	15	7
	Roller Coaster	32%	29,5%	3,6%	-	0,2%	2,4%	12,3%
		118	141	1318		4372	206	31
	Producer Consumer	84,8%	82,5%	0,3%	0,1%	-	60%	-
		202	231	36392	36140		20	
	Cigarette Smokers	86,1%	74,1%	0,7%	0,1%	-	35,5%	-
		153	174	5377	5136		42	
	Matrix	100%	100%	0,2%	0,1%	-	11,3%	-
		206	200	127387	127152		192	
	Jacobi	63,8%	60,8%	0,2%	0%	-	7,1%	-
		556	633	278345	277092		532	
Ambos	T. R. File	39,8%	37,6%	0,8%	0,1%	8,8%	10%	22,5%
		209	236	6648	6456	68	96	8
	T. R. Broadcast	98%	97,7%	2,1%	0,6%	30,6%	17,4%	70%
		193	216	7496	7304	72	46	10
	GCD/LCM Both	19,1%	15,6%	0,1%	0%	0,7%	7,1%	27,1%
		649	794	213987	212058	2658	147	34

¹ Cobertura obtida por critério.² Total de elementos requeridos por critério.

Quadro 3 – Relação da cobertura de critérios dos programas obtida pela técnica GRAND.

Programas		Cobertura de Critérios						
		AN	AE	AU	ASU	AEU	ASE	AMU
Passagem de Mensagem	<i>GCD</i>	25,3% ¹	17,5%	9,8%	-	3,1%	9,1%	30%
		101 ²	131	250		195	33	10
	GCD - Two Slaves	31,9%	24,3%	4,8%	-	2,3%	11,3%	22,5%
		86	107	271		80	16	8
	GCD/LCM	73,8%	64,2%	36,3%	-	7,7%	26,8%	80,6%
		232	308	637		384	57	16
	T. R. Iterations	37,1%	36,5%	8,5%	-	0%	0%	0%
		96	136	284		24	12	4
	T. R. Both Directions Same Primitives	13,5%	6,8%	1%	-	0%	0%	0%
		89	118	391		63	12	4
	T. R. Both Directions Diferent Primitives	14,9%	12,7%	2,7%	-	0%	0%	0%
		161	221	442		726	51	22
Memória Compartilhada	Sieve of Eratosthenes	4%	1,5%	0,6%	-	0%	0%	0%
		199	272	1279		42	15	7
	Roller Coaster	99,2%	98,6%	14,4%	-	1%	12,1%	61,3%
		118	141	1318		4372	206	31
	Producer Consumer	100%	100%	0,4%	0,2%	-	99,5%	-
		202	231	36392	36140		20	
	Cigarette Smokers	86,9%	75,3%	0,7%	0,1%	-	38,1%	-
		153	174	5377	5136		42	
	Matrix	100%	100%	0,2%	0,1%	-	10,8%	-
		206	200	127387	127152		192	
	Jacobi	63,9%	60,8%	0,2%	0%	-	7,3%	-
		556	633	278345	277092		532	
Ambos	T. R. File	100%	99,6%	2,1%	0,5%	29,4%	37,3%	75%
		209	236	6648	6456	68	96	8
	T. R. Broadcast	9,3%	9,7%	0,1%	0%	0%	0%	0%
		193	216	7496	7304	72	46	10
	GCD/LCM Both	7,7%	6,4%	0%	0%	0,3%	5,2%	22,4%
		649	794	213987	212058	2658	147	34

¹ Cobertura obtida por critério.² Total de elementos requeridos por critério.

Quadro 4 – Relação da cobertura de critérios dos programas obtida pela técnica CONC.

Programas		Cobertura de Critérios						
		AN	AE	AU	ASU	AEU	ASE	AMU
Passagem de Mensagem	GCD	73,3% ¹	59,5%	28%	-	10,3%	24,2%	80%
		101 ²	131	250		195	33	10
	GCD - Two Slaves	90,7%	82,2%	28,8%	-	10%	50%	100%
		86	107	271		80	16	8
	GCD/LCM	52,6%	42,5%	22,6%	-	5,7%	21,1%	62,5%
		232	308	637		384	57	16
	T. R. Iterations	100%	94,1%	59,5%	-	33,3%	66,7%	100%
		96	136	284		24	12	4
	T. R. Both Directions Same Primitives	100%	94,1%	43%	-	15,9%	66,7%	100%
		89	118	391		63	12	4
	T. R. Both Directions Diferent Primitives	100%	100%	49,5%	-	4,7%	37,3%	77,3%
		161	221	442		726	51	22
	Sieve of Eratosthenes	80,9%	71,7%	23,5%	-	21,4%	60%	71,4%
		199	272	1279		42	15	7
	Roller Coaster	99,2%	97,9%	13,6%	-	1%	12,1%	61,3%
		118	141	1318		4372	206	31
Memória Compartilhada	Producer Consumer	99,9%	99,3%	0,3%	0,2%	-	83,5%	-
		202	231	36392	36140		20	
	Cigarette Smokers	72%	60,5%	0,6%	0,0%	-	24,8%	-
		153	174	5377	5136		42	
	Matrix	100%	100%	0,1%	0%	-	11,1%	-
		206	200	127387	127152		192	
	Jacobi	63,8%	60,8%	0,2%	0%	-	7,3%	-
		556	633	278345	277092		532	
Ambos	T. R. File	100%	97%	2%	0,4%	29,4%	32,1%	75%
		209	236	6648	6456	68	96	8
	T. R. Broadcast	79,8%	76,9%	1,8%	0,4%	30,6%	17,4%	70%
		193	216	7496	7304	72	46	10
	GCD/LCM Both	*	*	*	*	*	*	*
		649	794	213987	212058	2658	147	34

¹ Cobertura obtida por critério.² Total de elementos requeridos por critério.

Quadro 5 – Relação da cobertura de critérios dos programas obtida pela técnica GAMO.

Programas		Cobertura de Critérios						
		AN	AE	AU	ASU	AEU	ASE	AMU
Passagem de Mensagem	<i>GCD</i>	78,0% ¹	63,2%	32,9%	-	12,3%	26,7%	88,0%
		101 ²	131	250		195	33	10
	GCD - Two Slaves	89,3%	80,9%	27,5%	-	10,0%	50,0%	100%
		86	107	271		80	16	8
	GCD/LCM	43,1%	33,8%	17,3%	-	6,0%	23%	68,8%
		232	308	637		384	57	16
	T. R. Iterations	91,7%	82,4%	52,1%	-	16,7%	33,3%	100%
		96	136	284		24	12	4
	T. R. Both Directions Same Primitives	100%	94,1%	43,0%	-	16%	66,7%	100%
		89	118	391		63	12	4
	T. R. Both Directions Diferent Primitives	100%	100%	49,5%	-	4,7%	37,3%	77,3%
		161	221	442		726	51	22
Memória Compartilhada	Sieve of Eratosthenes	82,4%	73,2%	24,4%	-	21,4%	60,0%	71,4%
		199	272	1279		42	15	7
	Roller Coaster	99,2%	98,2%	14,0%	-	1,0%	12,1%	61,3%
		118	141	1318		4372	206	31
	Producer Consumer	78,5%	75,8%	0,3%	0,1%	-	45,0%	-
		202	231	36392	36140		20	
	Cigarette Smokers	74,4%	62,8%	0,7%	0,1%	-	25,0%	-
		153	174	5377	5136		42	
	Matrix	100%	100%	0,2%	0,1%	-	11,3%	-
		206	200	127387	127152		192	
	Jacobi	63,9%	60,8%	0,2%	0,0%	-	6,9%	-
		556	633	278345	277092		532	
Ambos	T. R. File	100%	99,6%	2,0%	0,4%	29,4%	29,7%	75,0%
		209	236	6648	6456	68	96	8
	T. R. Broadcast	100%	100%	2,2%	0,7%	30,6%	17,4%	70,0%
		193	216	7496	7304	72	46	10
	GCD/LCM Both	33,4%	28,8%	0,1%	0%	1,3%	11,8%	45,6%
		649	794	213987	212058	2658	147	34

¹ Cobertura obtida por critério.² Total de elementos requeridos por critério.

Quadro 6 – Relação da cobertura de critérios dos programas obtida pela técnica GRAND.

Programas		Cobertura de Critérios						
		AN	AE	AU	ASU	AEU	ASE	AMU
Passagem de Mensagem	GCD	91,49% ¹	78,85%	46,36%	-	13,49%	29,70%	98,00%
		101 ²	131	250		195	33	10
	GCD - Two Slaves	99,65%	95,51%	40,15%	-	10,00%	50,00%	100%
		86	107	271		80	16	8
	GCD/LCM	79,22%	68,93%	38,96%	-	8,72%	31,58%	95,63%
		232	308	637		384	57	16
	T. R. Iterations	100%	94,12%	61,97%	-	33,33%	66,67%	100%
		96	136	284		24	12	4
	T. R. Both Directions Same Primitives	100%	94,07%	42,97%	-	15,87%	66,67%	100%
		89	118	391		63	12	4
	T. R. Both Directions Different Primitives	100%	100%	49,55%	-	4,68%	37,25%	77,27%
		161	221	442		726	51	22
Memória Compartilhada	Sieve of Eratosthenes	80,90%	70,22%	23,54%	-	21,43%	60,00%	71,43%
		199	272	1279		42	15	7
	Roller Coaster	99,15%	98,58%	14,42%	-	0,96%	12,14%	61,29%
		118	141	1318		4372	206	31
	Producer Consumer	100%	100%	0,42%	0,20%	-	100%	-
		202	231	36392	36140		20	
	Cigarette Smokers	82,22%	70,23%	0,66%	0,05%	-	30,95%	-
		153	174	5377	5136		42	
	Matrix	100%	100%	0,15%	0,05%	-	11,30%	-
		206	200	127387	127152		192	
	Jacobi	63,85%	60,82%	0,20%	0,04%	-	7,37%	-
		556	633	278345	277092		532	
Ambos	T. R. File	100%	99,58%	2,05%	0,47%	29,41%	35,31%	75,00%
		209	236	6648	6456	68	96	8
	T. R. Broadcast	100%	100%	2,22%	0,72%	30,56%	17,39%	70,00%
		193	216	7496	7304	72	46	10
	GCD/LCM Both	32,19%	30,37%	0,13%	0%	1,40%	13,95%	51,47%
		649	794	213987	212058	2658	147	34

¹ Cobertura obtida por critério.² Total de elementos requeridos por critério.