**Um framework para avaliação sistemática de técnicas de teste
no contexto de programação concorrente**

Silvana Morita Melo

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

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Silvana Morita Melo

Um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutora 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
Maio de 2018

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

M528f Melo, Silvana Morita
Um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente / Silvana Morita Melo; orientadora Simone do Rocio Senger de Souza. -- São Carlos, 2018.
204 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2018.

1. Teste de Software. 2. Programação Concorrente. 3. Engenharia de Software Experimental. 4. Seleção de Técnicas de Teste. I. do Rocio Senger de Souza, Simone, orient. II. Título.

Silvana Morita Melo

**A Framework for systematic testing techniques evaluation
applied to concurrent programming**

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
May 2018

Este trabalho é dedicado à minha família e amigos.

AGRADECIMENTOS

Os agradecimentos principais são direcionados a Deus que sempre serviu de guia no direcionamento aos caminhos corretos e superação dos desafios impostos pela vida.

Aos meus pais Estela e Antônio que sempre acreditaram no meu potencial e sempre estiveram ao meu lado apoiando orgulhosos as escolhas que tenho tomado e uma delas levou a conclusão deste trabalho.

As minhas irmãs Elis e Ilze com as quais divido os sonhos de conquistar um futuro um pouco menos difícil que o enfrentado por nossos pais.

Um agradecimento mais que especial à professora Simone e ao professor Paulo Sérgio que atuaram como verdadeiros pais durante a minha jornada na pós-graduação. Eles que orientaram não apenas o doutorado mas também diversas questões de vida, de vocês eu vou levar muito mais que o título obtido. Obrigado também pelo acolhimento e boas risadas, meus irmãos adotivos Rafa e Felipe, que dividiram com carinho seus pais durante esse período.

I would like to thank Professor Jeffrey Carver, who received me in his research group and assist me during the stage abroad, contributing considerably with this research.

Ao professor José Carlos Maldonado, que apoiou a iniciativa do estágio no exterior e auxiliou em outras possibilidades de colaboração para evolução desta pesquisa.

Aos amigos que conquistei no ICMC e fizeram essa trajetória mais agradável. Um muito obrigada a vocês: Valéria, Rodolfo, Joice, Vânia, Thiago Gottardi, Maria Adelina, Draylson, Stevão, Camilo, Pedro, Danillo, Francielle, Leo, Valdemar, Juliana, Dario, Jadson, Lina, Brauner, Nilton, Faimison, Daniel, Misael, Ricardo, Armando, Livia, Ana e Bruno. Espero que a vida ainda cruze nossos caminhos profissionais no futuro.

Aos meus amigos de Tuscaloosa Jana Coffman e Brian Lackey que me mostraram o significado de “Sweet home Alabama”. Obrigada gente!

Ao ICMC/USP pela excelente infra-estrutura oferecida para o desenvolvimento do trabalho e ao pessoal envolvido na administração e secretaria do programa que estavam sempre dispostos a ajudar.

À FAPESP pelo apoio financeiro, sob processo nº 2013/05046-9, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

*“Por vezes sentimos que aquilo que fazemos não é senão uma gota de água no mar.
Mas o mar seria menor se lhe faltasse uma gota.”
(Madre Teresa de Calcutá)*

RESUMO

MELO, S. M. **Um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente**. 2018. 204 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Contexto: Embora diversas técnicas de teste de software tenham sido propostas para o contexto da programação concorrente, as informações sobre elas encontram-se de dispersas na literatura, não oferecendo uma caracterização apropriada e dados relevantes que possam auxiliar a compreensão e consequente aplicação efetiva dessas técnicas, dificultando o processo de transferência de conhecimento entre a academia e a comunidade interessada. **Objetivo:** Nesse contexto, o principal objetivo deste trabalho é oferecer subsídios, na forma de um framework, que seja capaz de apoiar a caracterização e seleção sistemática de técnicas de teste de software concorrente. **Metodologia:** Para atender esse objetivo, foi construído um corpo de conhecimento que reúne de maneira integrada informações relevantes ao processo de tomada de decisão sobre qual técnica de teste aplicar a um determinado projeto de software. Um design de experimentos é definido, funcionando como guia para condução de estudos empíricos que podem ser usados para a realimentação, atualização e evolução do corpo de conhecimento. Buscando sistematizar o processo de seleção de técnicas de teste, é definido um esquema de caracterização que considera as principais características da programação concorrente que influenciam a atividade de teste de software e calcula a adequação desses atributos aos atributos do projeto em desenvolvimento. **Resultados e Conclusões:** A fim de permitir que a comunidade interaja com o framework proposto, foi disponibilizada uma infraestrutura computacional que permite o acesso ao corpo de conhecimento e automatiza o processo de seleção de técnicas de teste de software concorrente. O estudo experimental conduzido para avaliação da proposta, mostrou que a abordagem contribui de maneira efetiva para caracterizar, comparar e quantificar a adequabilidade baseada em atributos, melhorando consideravelmente o processo de seleção de técnicas de teste para software concorrente segundo as expectativas dos usuários.

Palavras-chave: Teste de Software, Programação Concorrente, Engenharia de Software Experimental, Seleção de Técnicas de Teste.

ABSTRACT

MELO, S. M. **A Framework for systematic testing techniques evaluation applied to concurrent programming**. 2018. 204 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Background: Although a variety of concurrent software testing techniques have been proposed for the concurrent programming context, the information about them are scattered in the literature, not offering an appropriate characterization and relevant data that can aid the understanding and consequently the effective application of these techniques, hindering the process of knowledge transfer between the academia and the interested community. **Objective:** In this context, the main objective of this work is to provide subsidies in form of a framework which will be able to support the characterization and systematic selection of concurrent software testing techniques. **Methodology:** In order to meet this objective, a body of knowledge has been built that brings together, in an integrated way, information relevant for the decision-making process about what testing technique should be applied in a specific software project. A design of experiments is defined as a guide for conducting empirical studies that can be used for feedback, updating, and evolution of the body of knowledge. With the objective of systematizing the process of testing techniques selection is defined a characterization scheme that considers the main characteristics of the concurrent programming that influence the testing activity and calculates the adequacy for these attributes in comparison with the software project in development. **Results and Conclusions:** In order to allow the community interaction with the proposed framework was provided a computational infrastructure that allows access to the body of knowledge and the automation of the selection process. The empirical study conducted to evaluate the proposal showed that the approach effectively contributes to characterize, compare and quantify the adequacy based on the attributes, improving the selection process of concurrent software testing techniques according to the user's expectations.

Keywords: Software Testing, Concurrent Programming, Empirical Software Engineering, Testing Techniques Selection.

LISTA DE ILUSTRAÇÕES

Figura 1 – Etapas de evolução da pesquisa e artefatos gerados.	31
Figura 2 – Taxonomia de Flynn (QUINN, 2003).	46
Figura 3 – Arquitetura MIMD - (a) Multiprocessadores, (b) Multicomputadores (FLYNN; RUDD, 1996).	46
Figura 4 – Exemplo de não determinismo entre processos.	50
Figura 5 – Processo de teste para software concorrente.	52
Figura 6 – Erros típicos em software concorrente.	54
Figura 7 – Arquitetura do <i>framework</i>	63
Figura 8 – Processo de derivação para geração do corpo de conhecimento.	65
Figura 9 – Interface do corpo de conhecimento online.	66
Figura 10 – Evidências em teste de software concorrente por ano.	67
Figura 11 – Distribuição dos estudos por tipo de técnica de teste.	67
Figura 12 – Distribuição dos estudos por linguagem de programação.	68
Figura 13 – Estado-da-arte da área de teste de software concorrente.	68
Figura 14 – Porcentagem de adequação dos atributos de caracterização.	87
Figura 15 – Arquitetura da ferramenta SeleCTT (MOURA <i>et al.</i> , 2017).	90
Figura 16 – Módulo de inserção (MOURA <i>et al.</i> , 2017).	92
Figura 17 – Módulo de administração.	93
Figura 18 – Módulo de controle.	94
Figura 19 – Processo de operação da ferramenta SeleCTT.	96
Figura 20 – Módulo de apresentação dos resultados.	96
Figura 21 – Armazenamento de informações no corpo de conhecimento (MOURA <i>et al.</i> , 2017).	97
Figura 22 – Interface da página inicial da ferramenta.	98
Figura 23 – Interface do formulário de inserção de técnicas de teste.	99
Figura 24 – Estatísticas sobre o uso da ferramenta.	100
Figura 25 – Interface do formulário de inserção de atributos do projeto de software.	101
Figura 26 – Interface de apresentação da recomendação de técnicas de teste.	102
Figura 27 – <i>Box plots</i> para a variável completude dos atributos usados durante a seleção, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	117

Figura 28 – Análise da média (porcentagem de informação usada na seleção), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	117
Figura 29 – <i>Box plots</i> para a variável completude em relação aos atributos faltantes, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	118
Figura 30 – Análise da média (porcentagem de informação faltantes na seleção), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	119
Figura 31 – <i>Box plots</i> para a variável efetividade, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	120
Figura 32 – Análise da média (porcentagem de escolhas corretas), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	121
Figura 33 – <i>Box plots</i> para a variável eficiência relacionada ao tempo de resposta, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	122
Figura 34 – Análise da média (tempo de resposta em minutos), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	122
Figura 35 – <i>Box plots</i> para a porcentagem de informações usadas/porcentagem de escolhas corretas, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	123
Figura 36 – Análise da eficiência (porcentagem de informações usadas/porcentagem de escolhas corretas).	124
Figura 37 – <i>Box plots</i> para a porcentagem de escolhas corretas/tempo de seleção, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	125
Figura 38 – Análise da média (porcentagem de escolhas corretas/tempo de resposta), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	125
Figura 39 – <i>Box plots</i> para a variável usabilidade, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	126
Figura 40 – Análise da média (quantidade de problemas reportados), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).	127
Figura 41 – Satisfação do usuário - simplicidade da abordagem e possível uso na indústria.	132

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Código fonte do algoritmo de seleção (MOURA <i>et al.</i> , 2017).	95
---	----

LISTA DE TABELAS

Tabela 1 – Estudos primários sobre técnicas para o teste de software concorrente.	69
Tabela 2 – Modelo GQM para teste de software concorrente.	72
Tabela 3 – Classificação de técnicas de teste de software concorrente.	74
Tabela 4 – Variáveis dependentes pra avaliação de técnicas de teste de software concorrente.	75
Tabela 5 – Tipos de padrões de <i>design</i> (WOHLIN <i>et al.</i> , 2012).	76
Tabela 6 – Ameaças à validade no contexto de teste de software concorrente.	79
Tabela 7 – <i>design</i> de experimentos para teste de software concorrente.	80
Tabela 8 – Atributos para a caracterização de técnicas de teste de software concorrente.	82
Tabela 9 – Questões de pesquisa do estudo.	84
Tabela 10 – Caracterização dos sujeitos do estudo.	85
Tabela 11 – Avaliação dos atributos de caracterização.	86
Tabela 12 – Pesos dos atributos de caracterização.	88
Tabela 13 – Design do experimento.	106
Tabela 14 – Hipóteses por aspecto estudado.	108
Tabela 15 – Variáveis dependentes.	109
Tabela 16 – Atribuição dos sujeitos por grupos e subgrupos.	112
Tabela 17 – Atribuição dos tratamentos aos grupos.	112
Tabela 18 – Distribuição dos grupos entre os fatores para análise de influência.	112
Tabela 19 – Procedimento de execução do experimento.	113
Tabela 20 – Caracterização dos participantes do experimento.	115
Tabela 21 – Anova (Compleitude - atributos usados).	118
Tabela 22 – Teste-t (Compleitude - atributos usados).	118
Tabela 23 – Anova (Compleitude - atributos faltantes).	119
Tabela 24 – Anova (Efetividade - porcentagem de escolhas corretas).	121
Tabela 25 – Teste-t (Efetividade - porcentagem de escolhas corretas).	121
Tabela 26 – Anova (Eficiência - tempo de resposta).	123
Tabela 27 – Anova (Eficiência - porcentagem de informações usadas/porcentagem de escolhas corretas).	124
Tabela 28 – Teste-t (Eficiência - porcentagem de informações usadas/porcentagem de escolhas corretas).	124
Tabela 29 – Anova (Eficiência - porcentagem de escolhas corretas/tempo de resposta).	125
Tabela 30 – Anova (Usabilidade - quantidade de problemas reportados).	126

Tabela 31 – Síntese dos resultados da análise quantitativa.	127
Tabela 32 – Porcentagem de uso dos atributos de caracterização para a seleção.	128
Tabela 33 – Informações faltantes para cada abordagem de seleção.	129
Tabela 34 – Problemas/dúvidas reportados pelos participantes durante a condução do estudo.	130
Tabela 35 – Satisfação do usuário - facilidade da seleção.	131
Tabela 36 – Satisfação do usuário - vantagens do uso da abordagem.	133
Tabela 37 – Satisfação do usuário - desvantagens da abordagem.	133
Tabela 38 – Satisfação do usuário - sugestões de melhoria.	135
Tabela 39 – Satisfação do usuário - lições aprendidas.	136
Tabela 40 – Projeto 1.	160
Tabela 41 – Projeto 2.	161
Tabela 42 – Projeto 3.	162
Tabela 43 – Projeto 4.	163
Tabela 44 – Técnica de teste 1.	164
Tabela 45 – Técnica de teste 2.	165
Tabela 46 – Técnica de teste 3.	166
Tabela 47 – Técnica de teste 4.	167
Tabela 48 – Porcentagem de adequação das técnicas ao projeto.	167
Tabela 49 – Formulário de consentimento (F0).	168
Tabela 50 – Formulário de caracterização do participante (F1).	169
Tabela 51 – Formulário de seleção de técnicas (F2).	170
Tabela 52 – Formulário de informações faltantes durante a seleção (F3).	171
Tabela 53 – Formulário de dúvidas geradas durante a seleção (F4).	172
Tabela 54 – Formulário de avaliação da abordagem de seleção (F5).	173

SUMÁRIO

1	INTRODUÇÃO	25
1.1	Contexto e Motivação	25
1.2	Trabalhos Relacionados	27
1.3	Objetivos	29
1.4	Metodologia	30
1.5	Contribuições	31
1.6	Organização do Trabalho	32
2	FUNDAMENTAÇÃO TEÓRICA	33
2.1	Considerações Iniciais	33
2.2	Teste de Software	33
2.3	Programação Concorrente	43
2.4	Teste de Programas Concorrentes	50
2.5	Engenharia de Software Experimental	55
2.5.1	<i>Estudos experimentais em programação concorrente</i>	57
2.6	Considerações Finais	59
3	FRAMEWORK PARA AVALIAÇÃO SISTEMÁTICA DE TÉCNICAS DE TESTE DE SOFTWARE CONCORRENTE	61
3.1	Considerações Iniciais	61
3.2	Arquitetura do <i>Framework</i> : Principais Componentes	62
3.3	Corpo de Conhecimento Sobre Teste de Software Concorrente	64
3.3.1	<i>Definição do corpo de conhecimento</i>	64
3.3.2	<i>Estrutura do corpo de conhecimento</i>	66
3.3.3	<i>Conteúdo</i>	66
3.4	Design de Experimentos na Área de Teste de Software Concorrente	70
3.4.1	<i>Definição do experimento</i>	71
3.4.2	<i>Planejamento do experimento</i>	72
3.4.2.1	<i>Seleção do contexto</i>	72
3.4.2.2	<i>Definição de hipóteses</i>	73
3.4.2.3	<i>Seleção de variáveis</i>	74
3.4.2.4	<i>Seleção dos sujeitos</i>	75
3.4.2.5	<i>Escolha do tipo de design</i>	76

3.4.2.6	<i>Instrumentação</i>	77
3.4.2.7	<i>Avaliação da validade</i>	77
3.5	Esquema de Caracterização e Seleção de Técnicas de Teste de Software Concorrente	81
3.5.1	<i>Atributos de caracterização</i>	81
3.5.2	<i>Definição dos critérios de adequação e avaliação do esquema</i>	83
3.5.2.1	<i>Planejamento e objetivo</i>	84
3.5.2.2	<i>Questões de pesquisa</i>	84
3.5.2.3	<i>Instrumentação e execução</i>	84
3.5.2.4	<i>Análise dos resultados</i>	84
3.5.2.5	<i>Ameaças à validade do estudo</i>	88
3.5.2.6	<i>Discussão</i>	89
3.6	SeleCTT: Uma Infraestrutura computacional de apoio à seleção de técnicas teste de software concorrente	89
3.6.1	<i>Definição da ferramenta</i>	89
3.6.2	<i>Arquitetura da ferramenta</i>	90
3.6.2.1	<i>Classes de usuários</i>	91
3.6.2.2	<i>Módulos</i>	91
3.6.2.3	<i>Base de dados</i>	97
3.6.3	<i>Funcionalidades</i>	98
3.6.3.1	<i>Busca por uma técnica específica</i>	98
3.6.3.2	<i>Inserção de uma técnica na base de dados</i>	99
3.6.3.3	<i>Seleção de técnicas de teste</i>	99
3.6.3.4	<i>Apresentação de dados estatísticos</i>	100
3.6.4	<i>Exemplo de aplicação</i>	101
3.6.4.1	<i>Caracterização do projeto de software</i>	101
3.6.4.2	<i>Cálculo da adequação</i>	102
3.6.4.3	<i>Recomendação</i>	102
3.7	Considerações finais	103
4	AVALIAÇÃO EXPERIMENTAL	105
4.1	Considerações Iniciais	105
4.2	Design do Experimento	105
4.2.1	<i>Objetivo</i>	106
4.2.2	<i>Seleção do Contexto</i>	106
4.2.3	<i>Questões de Pesquisa</i>	107
4.2.4	<i>Definição das Hipóteses</i>	107
4.2.5	<i>Definição das Variáveis</i>	109
4.2.5.1	<i>Variáveis dependentes</i>	109
4.2.5.2	<i>Variáveis independentes</i>	109

4.2.6	Fatores e alternativas	110
4.2.6.1	Abordagens de seleção	110
4.2.6.2	Projeto de software	111
4.2.7	Seleção dos sujeitos	111
4.2.8	Design do experimento	112
4.2.9	Projeto do experimento	113
4.2.10	Procedimento experimental	113
4.2.11	Instrumentação	114
4.3	Execução e Coleta de Dados	114
4.3.1	Avaliação estatística dos dados	114
4.4	Análise dos Resultados	116
4.4.1	Análise quantitativa dos resultados	116
4.4.1.1	Compleitude - Porcentagem de atributos usados	116
4.4.1.2	Compleitude - Quantidade de atributos faltantes	118
4.4.1.3	Efetividade - Porcentagem de escolhas corretas	120
4.4.1.4	Eficiência - Tempo de resposta	121
4.4.1.5	Eficiência - Porcentagem de informações usadas/porcentagem de escolhas corretas	123
4.4.1.6	Eficiência - Porcentagem de escolhas corretas/tempo de seleção	124
4.4.1.7	Usabilidade - Quantidade de problemas reportados	126
4.4.2	Análise qualitativa dos dados	128
4.4.2.1	Atributos de caracterização usados durante a seleção	128
4.4.2.2	Informações faltantes para a caracterização das técnicas	128
4.4.2.3	Acertos por abordagem de seleção	129
4.4.2.4	Análise do tempo gasto com a seleção	129
4.4.2.5	Problemas reportados pelos participantes	130
4.4.2.6	Satisfação do usuário	131
4.4.3	Avaliação da validade	136
4.4.4	Discussão e Conclusão	137
4.5	Considerações Finais	138
5	CONCLUSÕES	139
5.1	Considerações Iniciais	139
5.2	Resultados Obtidos	139
5.3	Contribuições do Trabalho	140
5.4	Limitações	141
5.5	Trabalhos Futuros	142
5.6	Produção Científica	142

REFERÊNCIAS	145
APÊNDICE A	INSTRUMENTAÇÃO DO EXPERIMENTO 159
APÊNDICE B	MAPEAMENTO SISTEMÁTICO SOBRE EVIDÊNCIAS EM TESTE DE SOFTWARE CONCORRENTE 175

INTRODUÇÃO

1.1 Contexto e Motivação

A atividade de teste de software é uma das mais onerosas do processo de desenvolvimento de software, podendo consumir grande parte do custo total do projeto, porém não pode ser desprezada quando se almeja construir um produto de qualidade e livre de defeitos (AMMANN; OFFUTT, 2008). Sabe-se que para comprovar que um produto está livre de defeitos é necessário sua execução utilizando todo o domínio de entrada, uma atividade impraticável quando é observado o tamanho do domínio de entrada que, até mesmo para programas triviais, pode ser imenso (DELAMARO *et al.*, 2016). Uma alternativa é criar subconjuntos do domínio de entrada formados por uma quantia reduzida de elementos, mas que representem o domínio como um todo. Para seleção desses elementos são utilizadas técnicas e critérios de teste.

CrITÉRIOS de teste são definidos para apoiar a seleção de dados de teste, sistematizando a escolha desses dados, guiados por informações oriundas do produto em teste. De acordo com a informação utilizada, os critérios podem ser classificados em diferentes técnicas de teste: funcional, que usa informações sobre a especificação do programa; estrutural, que usa informações sobre a estrutura interna do programa e a técnica baseada em erros, que usa informações sobre os erros mais comuns cometidos no processo de desenvolvimento de software (MYERS; SANDLER, 2004; DELAMARO *et al.*, 2016).

No contexto de programas sequenciais, inúmeras pesquisas foram desenvolvidas com o intuito de definir técnicas e critérios para validação e teste de programas, bem como estudos para avaliar as propostas (HERMAN, 1976; HOWDEN, 1976; MCCABE, 1976; DEMILLO *et al.*, 1978; RAPPS; WEYUKER, 1982; HALEY; ZWEBEN, 1984; NTAPOS, 1984; WEYUKER *et al.*, 1991; MALDONADO *et al.*, 1992; DELAMARO *et al.*, 2001; RICCA; TONELLA, 2001; FERRARI *et al.*, 2008; LEMOS *et al.*, 2009; FAROOQ; QUADRI, 2014). Esses trabalhos oferecem fundamentos básicos e norteiam a criação de novas técnicas e critérios de teste para

outros domínios de desenvolvimento.

Com os avanços computacionais principalmente na busca por maior desempenho computacional, surgem máquinas paralelas e, inerentemente, a necessidade por software paralelo. Computadores paralelos dispõem de múltiplos elementos de processamento, que trabalham simultaneamente para resolver um mesmo problema em menor tempo. Aplicações desenvolvidas para essas arquiteturas são por natureza mais complexas que programas sequenciais, pois envolvem a execução simultânea de processos (ou *threads*) que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, de maneira que esses programas (ou processos) podem ou não concorrer pelos mesmos recursos computacionais. A construção de processos concorrentes requer o uso de algumas primitivas fundamentais, necessárias para permitir: 1) a definição de quais processos serão executados em paralelo; 2) a inicialização e finalização dos processos concorrentes e; 3) a coordenação entre os processos concorrentes enquanto estes estiverem executando (ALMASI; GOTTLIEB, 1989). 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 software concorrente.

A definição de técnicas e/ou critérios de teste para aplicações concorrentes é mais complicada, pois além das dificuldades inerentes à atividade de teste, novos desafios são impostos devido principalmente ao comportamento não determinístico. Dentre esses desafios, estão: o elevado número de sincronizações a serem exercitadas durante o teste; a necessidade de detectar e corrigir erros típicos desse tipo de software, como: *deadlock*, violação de atomicidade e condições de disputa; e a necessidade de métodos eficientes de geração automática de dados de teste que facilitem e diminuam os custos da atividade de teste.

Embora o teste de software concorrente tenha sido intensivamente investigado nos últimos anos, apresentando uma grande variedade de técnicas e ferramentas publicadas na academia, conforme resultados descritos nas revisões de literatura em Mamun e Khanam (2009), Souza *et al.* (2011), Arora *et al.* (2015) e Melo *et al.* (2015), essas técnicas ainda permanecem pouco usadas na prática. Esse fator pode estar relacionado, dentre outros, ao fato de não existir uma metodologia para auxiliar o processo de seleção da técnica de teste mais apropriada para este contexto específico. Atualmente a escolha da técnica de teste a ser adotada em um projeto de teste de software concorrente permanece baseada no conhecimento do projetista de teste e muitas vezes não considera todas as técnicas de teste disponíveis na indústria ou academia. Isso é enfatizado pela ausência de evidência sobre o uso dessas técnicas em projetos de software, e o fato de que não existe um sólido corpo de conhecimento com evidências experimentais sobre a aplicabilidade dessas técnicas, que é um problema recorrente do teste de software para aplicações tradicionais (sequenciais) (VEGAS; BASILI, 2005).

Neste sentido, a proposta deste doutorado é definir um *framework* de suporte à caracterização e seleção de técnicas e ferramentas de teste para programas concorrentes. Fundamentos da

Engenharia de Software Experimental (ESE) são usados para possibilitar a extração de informação relevante ao processo de seleção de técnicas, permitindo que desenvolvedores consigam respostas para questões como: qual técnica é mais efetiva, menos custosa e que pode revelar mais defeitos em um determinado contexto de projeto. Essas respostas são difíceis de se obter somente com estudos de prova de conceito e sem a comparação de dados experimentais. Dados proveniente de estudos experimentais facilitam a comparação da efetividade entre técnicas de teste e tornam o processo de seleção mais sistemático e menos randômico (PERRY *et al.*, 2000).

1.2 Trabalhos Relacionados

Diversos trabalhos relevantes têm sido propostos na área de teste de software concorrente e podem ser classificados em diferentes abordagens, considerando por exemplo: o tipo de análise (estática ou dinâmica), o paradigma de comunicação entre os processos/threads (passagem de mensagem ou memória compartilhada) e a linguagem e/ou o padrão utilizados.

As contribuições para o teste de software concorrente, consideram as principais técnicas de teste utilizadas na programação sequencial e novas abordagens específicas para este contexto.

Para o teste baseado na especificação, são propostas abordagens, como: verificação formal (YANG; SAKALLAH, 2010; RAKAMARIC, 2010; WOOD *et al.*, 2010; SUGETA *et al.*, 2004; SOUZA *et al.*, 1999) e teste baseado em modelos (AICHERNIG *et al.*, 2009; LI *et al.*, 2009; CORDEIRO, 2010).

Abordagens são propostas também para a técnica baseada em defeitos, como o teste de mutação (GLIGORIC *et al.*, 2010; SILVA *et al.*, 2012; GLIGORIC *et al.*, 2013).

A técnica de teste estrutural também tem sido utilizada no teste de software concorrente, propondo novos critérios de teste e instanciando critérios utilizados no teste de software sequencial (YANG; CHUNG, 1992; KRAWCZYK; WISZNIEWSKI, 1995; SOUZA *et al.*, 2013; LU *et al.*, 2007; YANG *et al.*, 1998).

Técnicas para geração automática de casos de teste são propostas na tentativa minimizar o problema da explosão do tamanho do domínio de entrada na presença do não determinismo e facilitar o teste de aplicações concorrentes (DING *et al.*, 2008; TAN *et al.*, 2009; XIAOAN *et al.*, 2009; SILVA *et al.*, 2014).

Algumas abordagens buscam a execução determinística do software concorrente durante os testes, dentre elas estão: a execução controlada (BALL *et al.*, 2009; KAMIL; YELICK, 2010; RUNGTA; MERCER, 2009b) e o monitoramento de execução (HAELUND; PRESSBURGER, 2000; MAHESWARA *et al.*, 2010).

Entretanto, a proposição de novas técnicas e critérios pela academia, por si só, não é suficiente para a melhoria da qualidade do processo de software. A utilização da Engenharia de Software Experimental (SHULL *et al.*, 2001) visa a caracterizar, avaliar, prever, controlar ou

melhorar tanto os produtos, como também os processos, recursos, modelos ou teorias. Segundo Wohlin *et al.* (2012), a experimentação pode proporcionar uma base de conhecimento para reduzir incertezas sobre quais teorias, ferramentas e metodologias são adequadas e promissoras. Para avaliar as técnicas, métodos ou critérios é fundamental a realização de estudos experimentais.

No contexto de teste de programas concorrentes, alguns estudos experimentais têm sido conduzidos a fim de comprovar a validade de propostas (SEO *et al.*, 2006; BRITO, 2011; MAMUN *et al.*, 2010; SAINI, 2005; CORDEIRO; FISCHER, 2011; MELO *et al.*, 2012).

O desenvolvimento de estudos experimentais tem o intuito de analisar e avaliar o objeto de estudo (técnica, método ou ferramenta) por meio da repetição ou replicação desse experimento em diferentes contextos e por diferentes pessoas, caracterizando de modo mais significativo os resultados obtidos. Há de se considerar evidências sobre potenciais benefícios, limitações, custo e riscos associados a sua implantação na academia ou na indústria (KITCHENHAM *et al.*, 2004).

Embora estudos secundários em teste de software concorrente tenham sido conduzidos pela academia (ARORA *et al.*, 2015; QAWY *et al.*, 2012; SOUZA *et al.*, 2011; BRITO *et al.*, 2010; MAMUN; KHANAM, 2009), eles têm focado apenas na categorização de técnicas de teste, metodologias e ferramentas e não têm proporcionado informação para auxiliar na decisão sobre qual técnica usar em um determinado projeto de software, ou seja, considerando características de um projeto específico.

A seleção de técnicas, métodos, processos, e ferramentas que auxiliam no processo de desenvolvimento de software é uma atividade complexa (BASILI *et al.*, 1999) e este desafio está relacionado ao processo de decisão que pode envolver particularidades do projeto, como custo, recursos humanos disponíveis, tempo, problemas do domínio, aspectos políticos, entre outros aspectos (VEGAS; BASILI, 2005).

Algumas contribuições para a avaliação de técnicas de teste têm sido encontradas no contexto da programação sequencial, a saber:

Vegas e Basili (2005) propuseram um esquema de caracterização que torna possível a geração de um catálogo de técnicas de teste que auxilia o processo de tomada de decisão em projetos de teste. Neste trabalho, propomos um esquema de caracterização similar, mas com foco especificamente no teste de software para ambiente concorrente.

Wojcicki e Strooper (2007) desenvolveram uma abordagem que suporta a seleção de técnicas de verificação e validação por meio do cálculo de uma matriz de decisão que avalia a melhor combinação entre técnicas de teste e um projeto de software específico. Essa abordagem é relevante para o desenvolvimento de um processo de seleção de técnicas de teste para software concorrente considerando atributos específicos deste contexto. Um mecanismo de ordenação de técnicas também é proposto em nosso trabalho, baseado no cálculo da distância euclidiana (DOKMANIC *et al.*, 2015).

Vos *et al.* (2012) oferecem um *framework* metodológico para simplificar o design de estudos de caso para técnicas de teste de software e auxiliar a comparação entre diferentes técnicas. Esse *framework* provê atributos para avaliar técnicas de teste para um projeto de software no contexto da programação sequencial. Com base nesses resultados, neste trabalho de doutorado foram identificados atributos adicionais e alguns instanciados a partir dessa técnica que são aplicáveis à programação concorrente.

Pilar *et al.* (2014) simplificam a seleção de ferramentas de teste pela formalização de um processo de seleção de ferramentas como um conjunto de critérios de tomada de decisão e provê um catálogo de ferramentas e uma taxonomia de classificação. Em uma abordagem similar, neste trabalho de doutorado foram definidos atributos relevantes para caracterizar técnicas de teste específicas para software concorrente.

Neto e Travassos (2009) propõem uma abordagem que considera características de um projeto de software para determinar a adequação de técnicas de teste baseado em modelos para o projeto e o seu impacto na combinação dessas técnicas. Neste trabalho, além da técnica de teste baseado em modelos, são considerados o teste baseado em métodos formais, o teste baseado em erros e o teste estrutural. Assim como o trabalho anterior, a metodologia de seleção é baseada no trabalho de Vegas e Basili (2005) e adaptada à natureza da aplicação a ser testada, no nosso caso aplicações concorrentes.

Todos esses estudos são uma base importante para a definição de estratégias que classificam e selecionam tecnologias para outros contextos, como a programação concorrente. Entretanto, o desenvolvimento de uma nova metodologia é requerida, devido a uma série de características encontradas no desenvolvimento de aplicações concorrentes que influenciam a atividade de teste e que não são abordadas no contexto da programação sequencial. O processo de teste de software concorrente e suas particularidades são descritos em detalhes na Seção 2.4.

1.3 Objetivos

Dado o panorama atual das pesquisas desenvolvidas para o teste de programas concorrentes exposto em diferentes estudos secundários (BIANCHI *et al.*, 2017; ARORA *et al.*, 2015; QAWY *et al.*, 2012; BRITO, 2011; MAMUN; KHANAM, 2009), pode-se notar a relevância e diversidade das pesquisas que visam a solucionar o problema do teste em ambiente paralelo ou concorrente. No entanto, esses estudos secundários apenas apresentam uma visão geral da área e algumas classificações das técnicas propostas, não oferecendo uma caracterização apropriada das técnicas que possa apoiar o processo de tomada de decisão sobre qual técnica aplicar em um determinado projeto de software.

Diante desse contexto e considerando que ainda não existia uma base de conhecimento integrada em teste de software concorrente que reunisse informações relevantes ao processo de seleção de técnicas a serem usadas durante a atividade de teste, o objetivo deste trabalho foi

oferecer à comunidade um recurso que contemplasse os seguintes requisitos:

1. Ter uma base de dados que possibilitasse a centralização de informações sobre as técnicas de teste, de maneira organizada, para facilitar a compreensão e aquisição do conhecimento em teste de software concorrente pela comunidade interessada;
2. Ter um módulo que permitisse sistematizar a atividade de seleção de teste, oferecendo uma maneira padronizada de guiar a comunidade na escolha da técnica de teste adequada a um projeto de software definido;
3. Ter um mecanismo que viabilizasse a atualização e evolução da base de dados sobre evidências em teste de software concorrente de forma que a evolução fosse fundamentada na realimentação da comunidade, com base na aplicação prática das técnicas recomendadas em 2, evoluindo o corpo de conhecimento definido em 1.

Sendo assim, este projeto visa a contribuir de maneira geral com a definição de um *framework* de apoio à transferência de tecnologia da academia para a indústria, em relação à atividade de teste de programas concorrentes, oferecendo subsídios para o desenvolvimento de aplicações concorrentes com maior qualidade.

1.4 Metodologia

Considerando os objetivos estabelecidos, a metodologia para a condução deste trabalho segue o processo descrito na Figura 1, no qual cinco etapas são executadas e um artefato é gerado ao final de cada etapa, visando, ao final, a construção de um *framework* de suporte à avaliação de técnicas de teste de software concorrente que melhore o processo de seleção de técnicas de teste nesse contexto. Como o *framework* deve suprir os três objetivos definidos na Seção 1.3, a metodologia de trabalho segue as seguintes etapas:

Para atender ao objetivo 1 foi conduzido um mapeamento sistemático na área de teste de software concorrente a fim de reunir evidências sobre as técnicas de teste definidas nesse contexto. Os resultados desse estudo são usados para construção do corpo de conhecimento que reúne informações úteis ao processo de seleção de técnicas de teste e pode ser usado pela comunidade como fonte de conhecimento e evidência da efetividade das tecnologias propostas na academia.

Para atender ao objetivo 2, com base nas informações presentes no corpo de conhecimento, é definido um esquema para a caracterização das técnicas de teste e uma abordagem para o cálculo de adequação por meio da comparação dos atributos das técnicas de teste e do projeto de software. Pesos são fornecidos aos atributos definidos a fim de fornecer um mecanismo de recomendação que auxilie o processo de tomada de decisão sobre a escolha de técnicas a serem aplicadas no processo de desenvolvimento de software.

Para atender ao objetivo 3 de atualização do corpo de conhecimento, foi definido um design de experimentos para esse contexto, que funciona como um guia para condução de estudos experimentais que podem ser usados para realimentar o corpo de conhecimento e mantê-lo atualizado, com as informações sobre a aplicação prática das técnicas recomendadas pelo mecanismo definido conforme o objetivo 2.

Em adição a esses objetivos, foi disponibilizada uma infraestrutura computacional a fim de permitir que a comunidade interessada consiga interagir com a base de conhecimento, tendo acesso às informações e usando essas informações para evoluir seu processo de teste. Assim, o módulo de sistematização do processo de seleção de técnicas de teste foi automatizado pela implementação da ferramenta SeleCTT (*An Infrastructure for Selection of Concurrent Software Testing Techniques*).

Ao fim dessas etapas, a proposta foi avaliada por meio de um estudo experimental e os resultados foram analisados e discutidos por meio da estatística descritiva. Todas essas etapas e os artefatos gerados durante seu desenvolvimento fazem parte do *framework* de suporte a avaliação e seleção de técnicas de teste de software concorrente que é o objeto de estudo deste projeto de doutorado.

Figura 1 – Etapas de evolução da pesquisa e artefatos gerados.

1.5 Contribuições

Considerando os objetivos e a metodologia de pesquisa definidos, em suma, as contribuições deste trabalho envolvem:

- A definição de um corpo de conhecimento sobre técnicas de teste de software concorrente, com informações relevantes ao processo de teste;
- A proposição de um esquema de caracterização e uma abordagem de seleção com a definição de atributos e pesos para o cálculo de adequação e recomendação de técnicas a um projeto de software;
- A disponibilização de um design padrão para guiar a condução de estudos empíricos na área e possibilitar a realimentação do corpo de conhecimento definido;

- A disponibilização de uma infraestrutura web que permita o acesso da comunidade ao corpo de conhecimento e a aplicação automática da abordagem de seleção estabelecida, guiando os interessados no processo de tomada de decisão sobre qual técnica de teste aplicar considerando as restrições de seu projeto de software.

1.6 Organização do Trabalho

No Capítulo 2, são apresentados os conceitos fundamentais relacionados à atividade de teste de software, com ênfase no teste de software concorrente. No Capítulo 3, é apresentado o *framework* de avaliação sistemática de técnicas de teste de software concorrente proposto neste projeto. São descritos em detalhes: o corpo de conhecimento, o design de experimentos, o esquema de caracterização e a ferramenta de apoio que integra esses componentes e juntos compõem a estrutura do *framework*. No Capítulo 4, é descrito o estudo experimental para validação da proposta. No Capítulo 5, são discutidas conclusões, bem como alternativas a trabalhos futuros. Por fim, nos apêndices são fornecidas informações auxiliares sobre a condução do trabalho. No Apêndice A, são apresentados os formulários e demais instrumentos utilizados durante a execução do experimento. No Apêndice B, é descrito o mapeamento sistemático conduzido para a obtenção de evidências na área de teste de software concorrente.

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Este capítulo aborda as principais áreas envolvidas nesta pesquisa, o teste de software, a programação concorrente e também a intersecção dessas áreas, o teste de software concorrente. Além disso, é discutido o uso da Engenharia de Software Empírica nesse contexto.

O capítulo está organizado da seguinte forma: na Seção 2.2 são apresentadas algumas terminologias presentes no contexto da atividade de teste de software, uma definição para a atividade de teste, as técnicas usadas durante essa atividade, assim como sua importância na garantia da qualidade dos produtos de software e do processo de desenvolvimento.

Na Seção 2.3 são apresentados os principais conceitos relacionados à programação concorrente e ao trabalho de pesquisa proposto, descrevendo-se as principais arquiteturas, linguagens, paradigmas de suporte à programação concorrente.

Na Seção 2.4 são abordados o teste de aplicações concorrentes, os principais desafios e características que devem ser consideradas para a definição de abordagens de teste para essas aplicações.

Na Seção 2.5 são abordados a importância da utilização da engenharia de software empírica para validação das técnicas de teste propostas e disponibilização de informação relevante para o processo de seleção de metodologias de teste.

E por fim, na Seção 2.6 são apresentadas algumas considerações finais.

2.2 Teste de Software

Assim como em muitas outras áreas, na Engenharia de Software vários termos constantemente usados pela comunidade podem gerar dúvidas ou dualidades. Nessa direção, existe um

esforço na definição de um padrão para as terminologias nesta área.

O IEEE (*Institute of Electrical and Eletronic Engineers*) mantém um projeto de definição de padrões de termos utilizados, entre outros, no campo da informática, várias são as áreas que dispõem desse padrão de terminologias, Arquitetura de Computadores, Teoria da Computação, Inteligência Artificial, Computação Gráfica, entre outras (IEEE, 1990).

Um glossário de terminologias relacionado à Engenharia de Software o IEEE 729-1983 foi criado em 1983 e atualizado em 1990 IEEE 610.12 utilizando definições presentes no dicionário anterior, porém de maneira revisada e atualizada. Para o contexto de teste de software podem ser diferenciados termos essenciais, como (IEEE, 1990):

- **Engano** (*mistake*): ação humana que leva a um resultado incorreto.
- **Defeito** (*fault*): passo incorreto, definição incorreta de um dado, ou uso de uma instrução ou comando incorreto durante a codificação.
- **Erro** (*error*): circunstância incoerente ou inesperada durante a execução do programa, que pode ou não culminar em uma saída diferente da esperada e é resultado muitas vezes de um processo, passo, instrução ou definição de dados incorretos.
- **Falha** (*failure*): a existência de um erro se propaga até a saída do programa produzindo uma saída diferente da esperada.

Com objetivo de identificar e eliminar defeitos presentes no produto de software, atividades de VV&T (Verificação, Validação e Teste) visam garantir a qualidade do software produzido. Dentre essas, o teste de software é uma atividade que visa assegurar que o software faça somente aquilo que está descrito na sua especificação.

Segundo Myers e Sandler (2004), a atividade de teste de software tem como principal objetivo revelar a presença de erros no produto. Sendo assim, o teste dito bem sucedido é aquele que consegue selecionar casos de teste para os quais o programa em teste falhe. A atividade de projeto de casos de teste deve ser efetiva em revelar a presença de defeitos de software.

Defeitos são identificados em todos os níveis do processo de desenvolvimento do software. Portanto, a atividade de teste deve ser realizada durante todo o ciclo de desenvolvimento do software. Assim, o processo de teste é conduzido em três fases distintas (BINDER, 1999; MALDONADO *et al.*, 1992; PRESSMAN, 2001; DELAMARO *et al.*, 2016):

- **Teste de unidade**, nessa fase cada módulo do sistema é testado separadamente à procura de defeitos de lógica e de implementação.
- **Teste de integração**, visa identificar defeitos de comunicação entre os módulos testados na fase anterior. Durante a integração entre as partes do sistema, são testadas as várias interfaces do sistema, ou seja, a estrutura formada pelos módulos.

- **Teste de sistema** é realizado ao final da integração de todos os módulos, verificando se os elementos foram devidamente integrados e realizam as funções a eles atribuídas, além de verificar a existência de defeitos e resultados não condizentes com a especificação, colocando a prova o sistema como um todo.

Em cada uma dessas fases, a atividade de teste é sempre realizada em quatro etapas distintas (PRESSMAN, 2001; MYERS; SANDLER, 2004; BEIZER, 1990):

1. **Planejamento**, nessa etapa é desenvolvido o plano do projeto de teste, que inclui atividades como: definição do que será testado; estabelecimento do tempo a ser gasto com a atividade de teste e o limite de tempo estimado para seu fim; preparação da estrutura do ambiente dos testes; seleção de pessoal e ferramentas de automação, dentre outros recursos necessários.
2. **Projeto de casos de teste**, nessa etapa os casos de teste são definidos. A atividade principal é a elaboração e revisão dos cenários e roteiros de testes.
3. **Execução**, os casos de teste planejados na etapa anterior são executados nessa fase e os resultados são registrados.
4. **Análise**, ao final da execução dos casos de teste, os resultados gerados são analisados, com objetivo de verificar se o software se comporta conforme definido em sua especificação.

Tendo em vista o alto custo da atividade de teste em comparação às demais atividades que constituem o processo de desenvolvimento de software, é de suma importância que o teste consuma o mínimo de tempo e esforço possível preservando a sua qualidade. Assim, dentre as etapas citadas acima, a que merece maior atenção é a de projeto de casos de teste, que consiste na seleção dos casos de teste que tenham maior chance de identificar defeitos, poupando esforço desnecessário com a execução de casos de teste ineficazes. Para auxiliar essa atividade são desenvolvidas técnicas e critérios de teste. Técnicas de teste provêm normas para o projeto sistemático de casos de teste que exercitem tanto a lógica interna dos componentes do software, como seus domínios de entrada e saída (DELAMARO *et al.*, 2016).

A qualidade dos casos de testes está associada ao critério de teste utilizado, pois é ele quem define quais as propriedades ou requisitos deverão ser considerados na avaliação dos testes. Os critérios de teste podem ser definidos de acordo com a função que desempenham, critérios de seleção de dados de teste são procedimentos utilizados para escolha de casos de teste, e critérios de adequação dos dados de testes são usados para avaliar dados de testes. Diz-se que dado um programa P a ser testado e um conjunto de caso de testes T , T é C -adequado para a realização do teste de P se T satisfaz os requisitos de teste estabelecidos pelo critério C (ZHU, 1996; FRANKL, 1987; AMMANN; OFFUTT, 2008).

As técnicas e os critérios de teste são fundamentais na seleção de casos de teste, pois por meio deles consegue-se minimizar o número de casos de teste, garantindo que partes

específicas do código sejam cobertas pelos testes. As técnicas de teste diferem entre si pela origem da informação utilizada para seleção de dados de teste. A técnica funcional baseia-se na especificação do produto de software, a técnica estrutural utiliza a estrutura interna do programa e a técnica baseada em defeitos utiliza os defeitos típicos cometidos durante o processo de desenvolvimento do software.

Existem ainda outras técnicas que vem ganhando a atenção dos pesquisadores na área de teste, como o teste baseado em modelos e o teste baseado em aspectos. O teste baseado em modelos utiliza um modelo formal para representar o sistema em teste e, a partir desse modelo, sequências de teste são geradas. Diferentes métodos são usados para geração de sequências de teste dentre eles, podem-se destacar: TT, W, UIO e DS (GILL, 1962). A principal vantagem desta técnica está na facilidade de automação, onde diferentes métodos podem ser usados para derivar sistematicamente casos de teste a partir do modelo. Além disso, essa técnica permite avaliar propriedades importantes do sistema (SIMÃO; PETRENKO, 2008; SIMÃO, 2016). O teste baseado em aspectos se relaciona com a Programação Orientada a Aspectos (POA). Aspectos são unidades do software não relacionadas a sua funcionalidade mas sim com as propriedades que afetam o desempenho ou a semântica do sistema de alguma maneira. Os aspectos atravessam e coexistem nas funcionalidades do sistema e a ideia é apoiar a separação dos mesmos de modo a facilitar principalmente o reuso do software. Assim, o teste baseado em aspectos foca em defeitos específicos relacionados com a separação dos aspectos no código. Nesse contexto, Lemos (2005) propõe o teste estrutural focado em pontos de junção, buscando avaliar todos os usos de um aspecto transversal no código, verificando também se o aspecto está implementado corretamente.

A seguir são apresentadas de maneira detalhada algumas das principais técnicas de teste de software.

Técnica de Teste Funcional

Técnica conhecida também como teste de caixa preta, na qual os requisitos de teste são derivados da especificação do produto de software, ou seja, não há conhecimento da estrutura interna do programa em teste (MYERS; SANDLER, 2004).

Nos testes funcionais casos de teste são executados utilizando entradas válidas e inválidas e os resultados produzidos pelo programa são analisados. Além dos resultados o comportamento e as mensagens de erro geradas durante a execução também são consideradas. O objetivo é garantir que o software age conforme definido na especificação e atende as regras de negócios do cliente (MOLINARI, 2008).

Segundo Myers e Sandler (2004) os principais critérios definidos para o teste funcional são:

- **Particionamento em classes de equivalência:** com base na especificação, os dados do domínio de entrada são divididos em conjuntos de valores (classes), os valores de entradas

pertencentes a uma mesma classe são considerados equivalentes. São definidos dois tipos de classes de equivalência, classes válidas e inválidas. O número de casos de teste é reduzido, pois é utilizado apenas um valor de entrada para cada classe, que é representativo para toda a classe.

- **Análise do valor limite:** complementa o critério de particionamento de classes de equivalência usando não apenas um valor qualquer para satisfazer cada classe, mas sim os valores situados nos limites de cada classe. A intenção é encontrar erros relacionados aos limites possíveis para cada entrada do programa.
- **Grafo de Causa-Efeito:** os critérios anteriores não trabalham com combinações de condições de entradas. Já no critério de grafo causa-efeito os requisitos de teste são selecionados com base nas possíveis combinações das condições de entrada. Os casos de teste são projetados a partir de uma representação do programa chamado grafo causa efeito, onde as possíveis combinações de entrada são as causas e as possíveis ações são os efeitos. As causas e efeitos são relacionados no grafo que é convertido em uma tabela de decisão que é usada para derivar os casos de teste.

[Linkman et al. \(2003\)](#) propõem outro critério para o teste funcional denominado teste funcional sistemático, que combina os critérios de particionamento em classes de equivalência e análise do valor limite para gerar os casos de teste com base na especificação do programa ([MALDONADO et al., 2004](#)).

Ainda que o teste funcional seja muito útil, nessa técnica a estrutura interna do produto em teste é desconhecida, conseqüentemente, não há garantia que segmentos essenciais ou críticos do código foram cobertos. Portanto, embora utilizem diferentes requisitos as técnicas são complementares, porém não excludentes visto que cada uma delas cobre uma classe específica de erros ([MYERS; SANDLER, 2004](#); [PRESSMAN, 2001](#)).

Técnica de Teste Baseada em Defeitos

A técnica baseada em defeitos utiliza o conhecimento sobre os defeitos típicos encontrados no processo de desenvolvimento para derivar dados de teste. O principal critério dessa técnica é a análise de mutantes.

O critério análise de mutantes se baseia na hipótese do programador competente, segundo a qual um programador competente desenvolve um programa correto ou muito próximo do correto ([DEMILLO et al., 1978](#)). Considerando que essa hipótese está correta é possível afirmar que erros são inseridos nos programas por meio de pequenos desvios sintáticos que alteram a semântica do programa e podem conduzir a um comportamento incorreto. A técnica de análise de mutantes visa identificar os desvios sintáticos mais comuns por meio da aplicação de pequenas alterações sobre o programa em teste para revelar tais erros. Outra hipótese também explorada

nesse contexto é o efeito de acoplamento, que afirma que erros comuns estão associados a erros simples. Portanto, espera-se que conjuntos de casos de teste capazes de revelar erros simples consequentemente sejam capazes de revelar erros mais complexos (DEMILLO *et al.*, 1978).

Com o objetivo de modelar os desvios sintáticos mais comuns, operadores de mutação são aplicados ao programa em teste, transformando-o em programas similares (mutantes). Um operador de mutação determina as regras que definem as alterações a serem inseridas no programa em teste. Operadores de mutação podem ser de vários tipos: de comando, de operadores, de constantes, de variáveis; dependendo do tipo de erros que se deseja revelar e da cobertura que se deseja garantir (BARBOSA *et al.*, 2016).

Após a geração, os mutantes devem ser executados utilizando o conjunto de casos de teste adequado ao programa original. Um caso de teste eficiente é aquele que identifica um mutante (mata o mutante), ou seja que revela que o mutante e o programa original são diferentes. O objetivo desse critério é aumentar a qualidade do teste. Se um programa mutante não é distinguido do programa original durante o teste, pode-se concluir que o conjunto de teste não é suficientemente completo, e, portanto fraco, exceto é claro se o mutante for equivalente ao programa em teste (DEMILLO *et al.*, 1978).

O principal problema associado ao teste de mutação é a quantidade de mutantes gerados. O custo da execução de todos os mutantes é alto, o que aumenta a complexidade e o esforço gasto com o teste. Existem atualmente diversas estratégias que propõem tornar a técnica de análise de mutantes menos custosa. Essas estratégias podem ser classificadas em dois grandes grupos: estratégias que tentam diminuir o tempo de execução dos mutantes (MATHUR; KRAUSER, 1988; CHOI *et al.*, 1989; ZHANG *et al.*, 2013) e estratégias que procuram diminuir o número de mutantes gerados (OFFUTT *et al.*, 1996; BARBOSA *et al.*, 2001; BIEMAN *et al.*, 2001; BELLI *et al.*, 2016; DENG *et al.*, 2017).

Técnica de Teste Estrutural

O teste estrutural ou teste de caixa branca, em oposição ao teste funcional, utiliza o conhecimento da estrutura interna do programa para derivação dos casos de teste.

Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados (FRANKL, 1987; PRESSMAN, 2001; WEYUKER *et al.*, 1991; SUGETA, 1999).

Critérios baseados na complexidade do software utilizam uma representação do fluxo de controle do programa para calcular a complexidade lógica do programa e a partir dela derivar requisitos de teste. A medida da complexidade lógica de um programa é denominada complexidade ciclomática. A complexidade ciclomática define o número de caminhos linearmente independentes, ou seja, caminhos que introduzam novos conjuntos de instruções de processamento ou novas condições. Esses caminhos definem o conjunto básico de casos de teste de um

programa e estabelece um limite máximo para o número de testes que deve ser realizado para garantir que todas as instruções sejam executadas pelo menos uma vez. Um critério baseado em complexidade bastante utilizado é o critério de McCabe ([MCCABE, 1976](#)) que utiliza a medida da complexidade ciclomática para derivação dos casos de teste ([PRESSMAN, 2001](#)).

Critérios baseados no fluxo de controle derivam seus requisitos de teste com base nas informações de controle existentes no programa, ou seja, informações relacionadas a desvios ou repetições. Para isso, uma representação gráfica do programa conhecida como Grafo de Fluxo de Controle (GFC) ou Grafo de Programa é utilizada. No GFC, um programa P é representado por um conjunto de nós que correspondem a blocos indivisíveis de comando e por arestas que representam o controle do fluxo de informações entre os nós. A partir desse grafo caminhos de execução no programa são estabelecidos os quais iniciam em um nó de entrada e terminam em um nó de saída do grafo. Essa representação serve de auxílio para a realização do teste, usado como base na aplicação e avaliação dos critérios e para derivar casos de teste. Os testes são realizados com base na implementação, por meio do controle da execução do programa.

Dentre os critérios baseados no fluxo de controle do programa, os mais utilizados são:

- **Todos-Nós:** requer que cada nó, ou seja, cada comando do programa seja executado pelo menos uma vez durante a execução do programa em teste.
- **Todos-Arcos:** requer que todos os desvios entre os blocos ou arcos do GFC sejam percorridos ao menos uma vez.
- **Todos-Caminhos:** requer que todos os caminhos sejam percorridos ao menos uma vez. Esse critério é subdividido de acordo com o tipo de caminho a ser coberto.
- **Todos-Caminhos-Completo:** requer a cobertura de todos os caminhos completos. Um caminho completo é aquele em que o primeiro nó é o nó de entrada e o último nó é o nó de saída.
- **Todos-Caminhos-Simples:** exige que sejam percorridos todos os caminhos simples. Caminho simples é aquele no qual todos os nós que o constituem são distintos exceto possivelmente o primeiro e o último.
- **Todos-Caminhos-Livre-de-Laços:** requer que todos os caminhos livres de laço sejam percorridos ao menos uma vez pelo conjunto de sequências de teste. Caminho livre de laço é um caminho simples onde todos os nós são distintos inclusive o primeiro e o último.

Os critérios baseados no fluxo de dados utilizam informações sobre as variáveis definidas e usadas pelo programa para os testes. Nesse caso, o GFC é estendido com informações sobre o fluxo de dados, sendo denominado Grafo Def-Use ([RAPPS; WEYUKER, 1985](#)). O Grafo Def-Use é similar ao GFC, porém contém informações adicionais sobre associações entre pontos

do programa onde ocorre definição de variáveis, ou seja, onde são atribuídos valores a uma variável, e pontos nos quais esse valor é utilizado, ou seja, onde é feito um uso ou referência à variável. Os requisitos de teste são determinados com base nessas associações.

O uso de variáveis pode ser classificado em dois tipos distintos: *c-uso* quando a utilização da variável afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado, e *p-uso* quando a variável é usada em uma expressão em comandos de desvios, afetando diretamente os fluxos de controle do programa. Geralmente no grafo *c-usos* são associados aos nós e *p-usos* às arestas (MALDONADO *et al.*, 1992).

Rapps e Weyuker (1985) propuseram um conjunto de critérios baseados em fluxo de dados. Os principais são:

- **Todas-Definições:** nesse critério todas as definições de variáveis devem ser executadas em algum instante seja por um *p-uso* ou *c-uso*.
- **Todos-Usos:** requer a execução de todas as associações (par definição e uso de uma variável) por meio de pelo menos um caminho livre de definição. O critério Todos-Usos pode ser especializado em Todos-p-Usos, Todos-p-Usos/Alguns-c-Usos e Todos-c-Usos/Alguns-p-Usos.
- **Todos-Du-Caminhos:** este critério exige que toda associação entre uma definição de variável e seus subsequentes *p-usos* ou *c-usos* sejam executados por todos os caminhos livres de definição e livres de laço que essa associação percorre.

Os critérios propostos por Rapps e Weyuker (1985) exigem a ocorrência explícita do uso de variáveis definidas. Como uma alternativa a esses critérios Maldonado *et al.* (1992) propôs um novo grupo de critérios, onde o uso da variável não precisa estar aparente, basta que seja possível sua ocorrência. Esses critérios são denominados Potenciais-Usos e subdividem-se em:

- **Todos-Potenciais-Usos:** requer que pelo menos um caminho livre de definição de uma variável x definida em um nó i para todo nó e toda aresta alcançável a partir de i seja exercitada.
- **Todos-Potenciais-Usos/Du:** requer que pelo menos um Potencial-Du-Caminho com relação à variável x definida em i para todo nó e para aresta alcançável a partir de i seja exercitada.
- **Todos-Potenciais-Du-Caminhos:** requer que todos os Potenciais-Du-Caminhos com relação a todas as variáveis x definidas e todos os nós e arestas alcançáveis a partir dessa definição sejam exercitados.

Embora uma diversidade de critérios estruturais sejam propostos na literatura, uma desvantagem associada a eles é a existência de elementos requeridos não executáveis, uma

sequência de comandos (um caminho), não executável de um programa. No entanto, tais critérios estabelecem de maneira sistemática os requisitos a serem exercitados, fornecendo métricas quantitativas sobre a adequação de um conjunto de teste usado para testar um programa, facilitando a automatização do teste (BARBOSA *et al.*, 2016).

Técnica baseada em modelos e métodos formais

Na literatura os termos teste baseado em modelos e teste baseado em métodos formais têm sido usados por alguns autores de maneira distinta. Devido ao fato da técnica de teste baseada em modelos poder utilizar modelos não formais, como diagramas UML, por exemplo. Em contrapartida, a técnica baseada em modelos formais exige o uso de um modelo formal ou uma linguagem formal para descrever a especificação do sistema. Embora exista essa distinção, em geral ambas as técnicas usam modelos formais como: Máquinas de Estados Finitos (MEFs) (SESHU, 1963), Máquinas de Estados Finitos Estendidas (MEFEs) (PETRENKO *et al.*, 2004), *Statecharts* (HAREL, 1987) e Redes de Petri (PETERSON, 1977).

Algumas técnicas possibilitam descrever em um grafo os passos básicos do funcionamento de um sistema. Essas técnicas são denominadas máquina de transições de estados. Dentre as técnicas baseadas em máquinas de transições de estados, um modelo comumente usado são as máquinas de estados finitos (MEFs). Uma MEF pode ser representada por diagramas ou tabelas de transição. Nos diagramas, vértices representam os estados e arcos representam transições entre estados (SIMÃO, 2016).

A partir das MEFs é possível derivar sequências de símbolos de entrada (sequências de teste) a fim de verificar a conformidade da saída gerada pela implementação com a saída especificada na sequência de teste. Dentre os métodos de geração de sequências de teste baseados em MEFs estão (PETERSON, 1977; SIMÃO, 2016):

- **Método TT (*Transition Tour*)**: esse método utiliza uma sequência de teste que começa no estado inicial, passando por todas as transições ao menos uma vez e retorna ao estado inicial, permitindo a detecção de erros de saída, porém não garante a detecção de erros de transferência (erros ocasionados pelas transições e que levam a MEF a um estado diferente do esperado).
- **Método UIO (*Unique Input/Output*)**: esse método produz uma sequência de identificação de estado. A sequência de entrada de entrada leva a uma única saída na especificação correta, mas isso não é garantido em implementações que apresentam erros.
- **Método DS (*Distinction Sequence*)**: esse método utiliza uma sequência de distinção (DS) que é uma sequência de símbolos de entrada na qual sua aplicação aos estados da MEF produz saídas distintas para cada um dos estados.

- **Método W:** esse método utiliza um conjunto de caracterização para verificar se todos os estados e transições estão corretamente implementadas. Essa sequência que permite distinguir todos os estados da MEF detectando erros estruturais, sempre que a MEF for completa, mínima e fortemente conexa.

Embora o uso de MEFs seja bastante efetivo, existem classes de programas em que o modelo resultante é excessivamente grande, impossibilitando a representação explícita da concorrência, resultando no problema da explosão combinatória do número de estados. Algumas tecnologias são propostas a fim de estender as MEFs e aumentar o poder da modelagem, com o uso de *Statecharts* e Redes de Petri (PETRENKO *et al.*, 2004).

Statecharts é uma técnica de modelagem que tem por objetivo representar os aspectos dinâmicos de um sistema, seus estados, e comportamento reativo aos eventos de entrada. É uma extensão das MEFs tradicionais que possibilita a especificação de sistemas reativos e dinâmicos. Dentre as principais vantagens dessa técnica estão as características de decomposição, ortogonalidade, *broadcasting* e história, assim como o uso de hierarquia de MEFs, o que traz mais clareza ao modelo, e permite visualizar aspectos de concorrência como a sincronização (MALDONADO *et al.*, 2004).

Redes de Petri são modelos baseados em grafos usados para representar formalmente sistemas complexos, não determinísticos ou estocásticos, como; sistemas paralelos, concorrentes, assíncronos, distribuídos, não-determinísticos e/ou estocásticos. Esse modelo permite a representação de propriedades dinâmicas do sistema por meio da simulação do grafo, o que não é possível de representar usando as técnicas anteriores (PETERSON, 1977).

Métodos formais podem ser usados para escrever especificações corretas e não ambíguas de um sistema de software. Linguagens de especificações formais baseadas em linguagens matemáticas podem ser usadas tanto para especificar os sistemas quanto para analisar sua funcionalidade, possibilitando verificar as principais características de interesse. Métodos formais por si só não garantem a que o sistema está correto, porém aumentam o entendimento do sistema revelando erros ou aspectos de incompletude do sistema que podem ser caros para corrigir em etapas posteriores (MALDONADO *et al.*, 2004; SIMÃO, 2016).

Os principais métodos formais utilizados para auxiliar o teste de software são (SOUZA *et al.*, 2000):

- **Linguagens de descrição formal:** linguagens formais são utilizadas para escrever especificações corretas e não ambíguas do sistema, em geral empregadas em sistemas críticos. Nessa categoria destacam-se: a notação Z, SDL (*Specification Description Language*, Lotos, entre outras linguagens formais.
- **Linguagens baseadas em estados finitos:** linguagens baseadas em estados finitos definem os estados de um sistema a partir de um conjunto finito de valores, os quais são

geralmente representados graficamente por meio de transições que representam mudanças no estado. Exemplos dessas linguagens incluem Máquinas de Estados Finitos (MEFs), SDL, *Statecharts*.

O teste baseado em modelo auxilia a eliminação da ambiguidade e aumento do rigor nos testes. A eficácia desse teste está associada primordialmente ao potencial de automação que ele oferece. Se o modelo pode ser processado formalmente, os casos de teste podem ser derivados mecanicamente por meio da derivação sistemática de casos de teste a partir do modelo, podendo ser adaptados e estendidos para outros domínios de aplicação como a programação concorrente (SIMÃO, 2016).

O uso de especificação formal traz benefícios ao teste, auxiliando a automação da geração de casos de teste permitindo ao testador pensar sobre a efetividade do teste (SINGH *et al.*, 2016). Além disso, o uso de métodos formais possibilita que o desenvolvedor analise os modelos do sistema e argumente sobre eles com precisão e rigor matemático. Ambos, métodos formais e teste de software podem ser usados para aumentar a qualidade do software.

Naturalmente, os métodos formais não garantem a correção, o seu uso pode aumentar a compreensão do sistema, revelando erros ou aspectos de incompletude que podem ser caros de corrigir no futuro. O ciclo de desenvolvimento ainda é iterativo, porque os requisitos de software geralmente mudam durante o desenvolvimento de um projeto e o uso de linguagens de especificação formal não garante a eliminação total dos erros (SIMÃO, 2016).

Critérios pertencentes a diferentes técnicas de teste são complementares e devem ser usados em conjunto, pois, cobrem classes distintas de defeitos. O estabelecimento de estratégias incrementais que explorem as diversas características dos critérios favorece a obtenção de casos de teste mais abrangentes, confiáveis e de qualidade (DELAMARO *et al.*, 2016).

2.3 Programação Concorrente

A vantagem no uso da programação concorrente em contrapartida à programação sequencial está principalmente ligada ao fato de que a programação concorrente possibilita ocupar de maneira mais inteligente os recursos que a aplicação dispõe, evitando-se a ociosidade de recursos presente no processamento sequencial de dados, aumentando a eficácia e diminuindo os custos (TANENBAUM; STEEN, 2006; GRAMA *et al.*, 2003).

A computação paralela apresenta-se como uma alternativa à necessidade por maior desempenho computacional, diminuindo o tempo gasto com problemas complexos. Ela foi empregada inicialmente na solução de problemas que exigiam grande quantidade de processamento, principalmente em áreas como: previsão climática, dinâmica molecular, projetos estruturais, cosmologia, biologia e dinâmica dos fluídos (QUINN, 2003). Atualmente, a computação paralela pode ser encontrada em outras áreas de atuação, como grandes sistemas de bancos de dados, e

na web considerando servidores e serviços (GRAMA *et al.*, 2003). Os princípios nestas áreas de atuação permanecem os mesmos: divisão do problema a ser resolvido e execução paralela de diferentes módulos, a fim de se obter redução no tempo de execução, maior tolerância a falhas e soluções mais elegantes para problemas intrinsecamente paralelos.

Os sistemas distribuídos, por outro lado, compreendem um conjunto de computadores autônomos ligados por uma rede de comunicação de dados e gerenciados por um sistema operacional distribuído. O software existente em tais sistemas coordena as atividades e permite o compartilhamento de recursos de hardware e software (aplicações e dados). A transparência é uma das características fortes em tais sistemas. De fato, seus usuários devem visualizar um sistema único e integrado, mesmo que este esteja sendo executado em muitos computadores e em diferentes locais. O software distribuído é responsável por diferenciar os sistemas distribuídos das redes de computadores. Alguns objetivos relacionados aos sistemas distribuídos são: permitir a interação de sistemas distantes geograficamente, melhorar o desempenho e a disponibilidade dos recursos, manter a autonomia dos sistemas, reduzir custos e permitir a interação entre usuários e aplicativos.

Essas duas grandes áreas, Computação Paralela e Sistemas Distribuídos apresentam uma grande intersecção em relação a plataformas, programação das aplicações e objetivos. Nesse sentido, a computação paralela pode ser vista como um sistema distribuído fortemente acoplado, onde as exigências por processamento e redes de comunicação são maiores. Dada essa intersecção, termos como “computação paralela distribuída” ou mesmo “computação distribuída” são empregados atualmente, principalmente depois da consolidação do uso de *clusters* de computadores como plataforma popular para o suporte à computação distribuída.

Em relação à programação, observa-se que a programação concorrente é um ponto central neste contexto, pois fornece as mesmas técnicas e paradigmas de programação para ambas as áreas. A programação concorrente é fortemente baseada em paradigmas de programação relacionados à arquitetura usada e, dependendo do paradigma usado, disponibiliza diferentes primitivas que viabilizam a criação, a comunicação e a sincronização de processos concorrentes, independentemente dos objetivos traçados para estes.

Neste ponto, cabe ressaltar que processos concorrentes e processos paralelos são abstrações que representam diferentes situações. Processos concorrentes são processos que iniciaram a sua execução e, em um determinado instante do tempo, não finalizaram. Isso independe de onde estão executando. Se eles estiverem em execução no mesmo processador (assumindo que processador com um núcleo é capaz de executar um processo por vez), apenas um deles estará executando em um determinado instante do tempo. Isso é normalmente chamado de pseudo-parallelismo. Processos concorrentes concorrem a recursos como processador, memórias e dispositivos de E/S. Processos paralelos, por sua vez, são um tipo especial de processos concorrentes, pois representam processos concorrentes que iniciaram as suas execuções em processadores diferentes. Apresentam um parallelismo real, pois estão em execução no mesmo

instante de tempo em processadores distintos (GRAMA *et al.*, 2003).

A computação distribuída considera que as funcionalidades de uma aplicação podem ser divididas em tarefas que executam concorrentemente. O tamanho dessas tarefas em relação à quantidade de interação entre os processos é denominado granularidade ou nível de paralelismo. A granularidade pode ser de três tipos: fina, média e grossa. Na granularidade fina, o problema é decomposto em um grande número de tarefas com pouca computação cada uma. Na granularidade grossa, o problema é decomposto em um pequeno número de tarefas, as quais englobam várias instruções chegando normalmente ao nível de processos, usada geralmente quando a quantidade de processadores é reduzida. A granularidade média encontra-se entre os dois níveis anteriores.

Diversos são os tipos de hardware que suportam a computação distribuída, várias classificações foram propostas, porém uma das mais usadas e aceitas até o momento é a classificação proposta por Flynn (1966), que classifica o hardware pelo fluxo de dados e instruções que ele executa. Sendo assim arquiteturas podem ser do tipo:

- **SISD** (*Single Instrution, Single Data stream*) esse tipo de arquitetura é composta por apenas uma unidade de controle (UC) que suporta apenas um fluxo de instrução e um fluxo de dados de cada vez. Compreende a *Arquitetura de Von Neumann* (NEUMANN, 1988).
- **SIMD** (*Single Instrution, Multiple Data stream*) é uma arquitetura composta por um computador paralelo que trabalha com uma única UC enviando a mesma instrução para ser executada em diversos elementos de processamento com dados diferentes.
- **MISD** (*Multiple Instrution, Single Data stream*) arquitetura composta por um computador paralelo, que distribui um único fluxo de dados entre diversas unidades de processamento, controladas por diferentes UCs (há vários fluxos de instrução) cada. Não é muito utilizada na prática.
- **MIMD** (*Multiple Instrution, Multiple Data stream*) esse tipo de arquitetura é a mais utilizada atualmente, onde cada processador pode executar fluxos de instruções e fluxos de dados diferentes, *Clusters* e *Grids* se enquadram nessa categoria.

A taxonomia de Flynn abrange então quatro classes de arquiteturas de computadores: SISD, SIMD, MISD e MIMD, cada uma delas é classificada seguindo duas dimensões independentes: instruções e dados, cada dimensão com dois valores distintos: simples ou múltiplo, conforme ilustrado na Figura 2.

Computadores do tipo MIMD podem ainda ser classificados em dois grupos distintos, de acordo com a maneira pela qual os processadores realizam o acesso à memória (TANENBAUM; STEEN, 2006):

Figura 2 – Taxonomia de Flynn (QUINN, 2003).

- **Multiprocessadores** (memória compartilhada): sistemas compostos por duas ou mais *CPUs* que compartilham uma memória em comum. A comunicação entre *CPUs* é implícita e realizada pelo uso de variáveis compartilhadas.
- **Multicomputadores** (memória privada): possuem vários pares de *CPU-memória* conectados geralmente por uma placa de interface de rede. Cada *CPU* possui sua própria memória local e a comunicação é estabelecida por meio da troca de mensagens.

A Figura 3 ilustra a arquitetura MIMD: (a) Multiprocessadores com memória compartilhada e (b) Multicomputadores com memória distribuída (FLYNN; RUDD, 1996).

Figura 3 – Arquitetura MIMD - (a) Multiprocessadores, (b) Multicomputadores (FLYNN; RUDD, 1996).

Na computação distribuída, a execução de programas pode ser realizada através da criação de processos ou *threads*. Um processo nada mais é que o conjunto formado pelo software

em execução e as informações necessárias à sua execução como os valores do contador de programa (*PC* - *program counter*), dos registradores e das variáveis. Cada processo possui um espaço próprio de endereçamento que conterá o código e os dados do programa além de outras informações utilizadas na execução e controle de sua execução. Diferentemente dos processos, as *threads* (ou *lightweight processes* - processos leves) compartilham um mesmo espaço de endereçamento, que é o espaço de endereçamento do processo pai, porém com seu próprio *PC*, pilha e registradores. Elas possuem as mesmas variáveis globais ou seja compartilham variáveis, o que permite a cooperação entre as *threads* na realização de uma tarefa. Para um aumento de desempenho o ideal seria a execução de múltiplas *threads* (*multithreads*) de maneira paralela (multiprocessamento) ou de maneira concorrente (monoprocessamento). *Threads* podem ser consideradas uma maneira do processo paralelizar partes do código que serão escalonadas para execução sobre a *CPU* (TANENBAUM; STEEN, 2006).

Para o desenvolvimento de processos concorrentes é necessário o uso de primitivas fundamentais que não estão presentes no desenvolvimento sequencial, tais como: definir quais processos serão executados concorrentemente; como inicializar e finalizar processos concorrentes; e como coordenar e determinar a interação entre os processos concorrentes durante a execução (ALMASI; GOTTLIEB, 1989).

Métodos para inicialização de processos são definidos de acordo com o objetivo do código a ser paralelizado e com a linguagem ou biblioteca de programação usada. No caso da ativação de processos paralelos o *fork/join* pode ser usado quando se necessita de um mecanismo mais flexível e não estruturado, já o *cobegin/coend* e *doall* são mais estruturados, porém menos flexíveis.

Além de iniciar e finalizar um processo é necessário que haja interação entre os mesmos e que ao final da execução o resultado global seja atingido. Essa interação é obtida através da comunicação e sincronização.

A interação entre processos é realizada por meio de variáveis compartilhadas ou troca de mensagens. A troca de mensagens ocorre quando há interação entre processos que não compartilham o mesmo espaço de endereçamento. Neste caso, tanto a comunicação quanto a sincronização são feitas através das primitivas básicas *send* e *recv*, usadas, respectivamente, para enviar e receber dados entre pares ou grupos de processos. Quando estas primitivas são síncronas, elas também permitem a sincronização entre os processos. Há diferentes modos de operação das primitivas *send* e *recv*, sendo que elas podem ser classificadas como (GRAMA *et al.*, 2003; COULOURIS *et al.*, 2007):

- **Bloqueantes e não bloqueantes.** Em operações bloqueantes, a função chamada não poderá retornar até que o conteúdo da mensagem esteja seguro, p. ex., tenha sido enviado ou então copiado para um *buffer* de comunicação, e tal conteúdo possa ser alterado. Nas operações não bloqueantes uma função pode retornar antes do conteúdo estar seguro

para a transmissão, sendo necessário o uso de outra função para verificar se a operação foi completada. Esta função complementar garante que o conteúdo da mensagem não bloqueante não seja perdido.

- **Síncrona e assíncrona.** Na comunicação síncrona, o processo que envia a mensagem fica esperando até que o processo receptor pelo menos inicie o recebimento da mensagem para prosseguir a execução. No modo assíncrono, o processo transmissor não aguarda a confirmação do receptor para prosseguir.
- **Com *buffer* e sem *buffer*.** No modo de comunicação sem *buffer*, caso o envio da mensagem ocorra antes que a função de recepção tenha sido iniciada, a função de envio bloqueia o processo transmissor até que a mensagem seja confirmada pelo processo receptor. No modo de envio com *buffer*, o conteúdo da mensagem é copiado para o *buffer* do sistema.
- **Ponto-a-ponto e coletivas.** A comunicação ponto-a-ponto é realizada apenas entre um par de processos, que utilizam operações para enviar e receber mensagens entre si. Já a comunicação coletiva envolve um grupo de processos.

Quando a interação ocorre entre *threads* que compartilham o mesmo espaço de endereçamento, há comunicação através do uso de variáveis e a sincronização é feita com o auxílio de primitivas, como semáforos e monitores (TANENBAUM; STEEN, 2006). A sincronização por meio de variáveis compartilhadas pode ser classificada em: exclusão mútua e sincronização condicional (ANDREWS, 1991). A sincronização condicional garante o bloqueio de uma *thread* até que uma condição se torne verdadeira e ela possa prosseguir sua execução. O objetivo é coordenar a computação das *threads*. A exclusão mútua garante que regiões críticas do programa sejam acessadas de maneira síncrona, ou seja, uma *thread* por vez. Existem vários mecanismos de sincronização que garantem exclusão mútua e a sincronização condicional entre processos. Dentre eles podemos citar:

- **Semáforos (DIJKSTRA, 1965):** um semáforo pode ser definido como uma variável inteira não negativa para contar o número de sinais já emitidos e que permitem a continuidade da execução de uma *thread*. Se o recurso compartilhado está disponível o valor do semáforo é maior que zero, caso contrário seu valor é zero. Duas operações atômicas são implementadas como *system calls* (chamadas de sistema), são elas o *up* e *down*. Quando o processo executa um *down* em um semáforo com valor maior que zero ele decrementa o valor do semáforo e continua a execução, se o valor for igual a zero ele é colocado em estado de espera (*sleeping*) e é automaticamente colocado na fila de pronto para execução quando o outro processo liberar o recurso usando o *up*. A atomicidade das instruções garante a sincronização e evita condições de disputa.
- **Mutex:** uma versão mais simplificada dos semáforos, cujo contador possui o valor 1 ou 0. Assim, o mutex é uma variável utilizada para gerenciar exclusão mútua que pode conter

dois estados: bloqueado (*lock*) e desbloqueado (*unlock*). Quando um processo ou *thread* tenta acessar a região crítica ele usa o procedimento *mutex_lock*, se a região crítica estiver livre, a *thread* que chamou o procedimento continua sua execução acessando a região crítica. Caso a região crítica estiver ocupada, a *thread* que chamou o *mutex_lock* ficará bloqueada até que a *thread* que está na região crítica termine sua execução e chame o procedimento *mutex_unlock*, liberando o recurso e permitindo a execução da *thread* que estava bloqueada.

- **Monitores** (HOARE, 1974; HANSEN, 1973): são mecanismos de alto nível para garantir sincronização e exclusão mútua, implementados pelo próprio compilador. Processos não têm acesso à estrutura interna do monitor apenas utilizam chamadas a seus procedimentos. A exclusão mútua é garantida pelo compilador e o programador só precisa especificar quais as regiões críticas. Em um monitor apenas um processo pode estar ativo num dado momento e se outro processo tentar acessá-lo, ele precisa ser bloqueado. Uma maneira de realizar o bloqueio é por meio da utilização de variáveis de condição, que permitem duas operações (*wait* e *signal*). Quando um procedimento não pode prosseguir, ele emite um *wait* sobre uma variável condicional e isso bloqueia o processo que está chamando, permitindo que o outro processo proibido de entrar no monitor entre. Esse outro processo pode acordar o parceiro adormecido a partir da emissão de um *signal* para a variável condicional do parceiro que está esperando. A utilização de variáveis de condição proporciona uma sincronização condicional (TANENBAUM; STEEN, 2006).

O desenvolvimento de aplicações paralelas pode ser realizado por meio de (SOUZA *et al.*, 2016): 1. ambientes automáticos que geram código paralelo a partir de códigos sequenciais, tais como algumas versões de compiladores para Fortran e C; 2. linguagens de programação concorrente, como OCCAM e ADA; e 3. extensões para linguagens sequenciais como C e Fortran, implementadas por meio de ambientes de passagem de mensagens e memória compartilhada.

O suporte às primitivas necessárias ao desenvolvimento de programas concorrentes é obtido com a utilização de uma linguagem de programação que garanta a divisão do problema de maneira justa entre os processadores e a execução concorrente dos processos (leves ou não). Além de linguagens de programação propriamente ditas, o paralelismo pode ser implementado através de bibliotecas que associam uma nova *API* (conjunto de rotinas para programação de aplicações que não envolvem detalhes de implementação) às linguagens sequenciais, estendendo suas funcionalidades para atender aos requisitos da programação concorrente. As principais ferramentas de suporte à programação concorrente são definidas de acordo com a arquitetura de memória utilizada, que pode ser compartilhada ou distribuída. No contexto de memória distribuída o principal ambiente de programação usado é o *MPI* (*Message Passing Interface*) (LASTOVETSKY, 2003). No contexto de memória compartilhada os principais modelos usados são o *OpenMP* e o *Padrão Posix para Threads* (GRAMA *et al.*, 2003; QUINN, 2003; ANDREWS, 1991).

Assim como na programação sequencial, os programas concorrentes não estão livres de erros, e por isso também devem ser testados, visando a garantir a qualidade do software produzido. Porém, o teste de software concorrente é mais complexo, considerando que apresentam características não encontradas nos programas sequenciais como: não determinismo, sincronização e comunicação, que tornam a atividade de teste ainda mais custosa e requer que novas técnicas de teste sejam criadas ou que técnicas de teste sequenciais sejam adaptadas para este contexto.

2.4 Teste de Programas Concorrentes

A utilização da programação concorrente traz inúmeras vantagens para a aplicação desenvolvida, porém aliado a ela existe o comportamento não determinístico que torna a atividade de teste mais complexa. Na Figura 4 é apresentado um exemplo de não determinismo, no qual pode-se observar um programa concorrente composto por quatro processos paralelos. Nesse exemplo, a execução do programa é dividida em um conjunto de processos (P1, P2, P3 e P4), que se comunicam por meio da troca de mensagens utilizando as primitivas *send* (S1, S2, S3 e S4) e *receive* (R1, R2, R3 e R4). Uma condição de corrida pode ser observada entre S1 e S2 relacionado com R1 e R2. O mesmo acontece com S3 e S4 relacionado com R3 e R4. Cada execução (Exec1, Exec2, Exec3 e Exec4) representa uma sequência de sincronização que pode ocorrer nesse programa. A atividade de teste é responsável por identificar todas as possíveis sequências de sincronização e analisar as saídas geradas.

Figura 4 – Exemplo de não determinismo entre processos.

O teste de programas concorrentes é uma atividade desafiadora, devido ao não determinismo e a uma série de fatores que precisam ser considerados nesse contexto (YANG; POLLOCK, 1997):

- Desenvolver técnicas de análise estática apropriadas para programas concorrentes;
- Detectar situações não desejadas como: erros de sincronização, comunicação, erros de fluxo de dados e *deadlock*;

- Reproduzir uma execução com a mesma entrada de teste e forçar a execução de um caminho na presença de não determinismo;
- Gerar uma representação do programa concorrente que capture informações pertinentes ao teste;
- Investigar a aplicação de critérios de teste sequenciais para programas concorrentes;
- Projetar critérios de fluxo de dados para programas concorrentes, considerando troca de mensagens e variáveis compartilhadas.

Observa-se também a existência de novos desafios ao teste de software concorrentes, dentre eles (SOUZA *et al.*, 2011):

- Reduzir o custo do teste de programas concorrentes;
- Gerar automaticamente dados de teste;
- Explorar eficientemente o espaço de *interleaving* de eventos.

A Figura 5 ilustra o processo de teste de software para programas concorrentes. Em que elipses representam as atividades que compõem a atividade de teste, setas representam o fluxo de execução entre atividades e diamantes representam atividades de decisão, onde (V) e (F) significam respostas verdadeiras e falsas, respectivamente. Círculos representam elementos gerados pela execução de uma atividade. O processo de teste para programas concorrentes pode ser definido como uma interação entre as seguintes atividades:

1. Geração de elementos requeridos para o programa em teste de acordo com o critério usado;
2. Execução do programa com os dados de teste e informações sobre os elementos requeridos a serem cobertos;
3. Atualização dos elementos requeridos cobertos durante a execução do programa;
4. Coleta dos traces produzidos pela execução do programa e uso dessa informação para geração da lista de *interleavings* para o programa;
5. Re-execução do programa guiado pela lista de possíveis sincronizações;
6. Geração da lista de *interleavings* a serem executados;
7. Análise dos elementos requeridos cobertos pelo novo *interleaving* selecionado e decisão sobre a necessidade de seleção de um novo dado de teste para continuar a atividade de teste;

Figura 5 – Processo de teste para software concorrente.

8. Decisão de parada da atividade de teste quando não houver mais elementos requeridos remanescentes a serem cobertos.

Diversas abordagens de teste têm sido propostas para sanar esses desafios ao teste impostos pelo software concorrente. Alguns trabalhos têm proposto técnicas de análise estática específicas para programas concorrentes (CHEN *et al.*, 2009; CHRISTAKIS; SAGONAS, 2011; SHIRES *et al.*, 1999; ABADI *et al.*, 2006). Outras iniciativas têm apresentado soluções para detecção de erros de sincronização, comunicação, fluxo de dados e *deadlock* (KIM *et al.*, 2011; HONG *et al.*, 2012; SALEH *et al.*, 2001; AMTOFT *et al.*, 1998; CHAKI *et al.*, 2005; ZHAI *et al.*, 2012; SHENG *et al.*, 2011; CHEW; LIE, 2010; CAI *et al.*, 2013). Tentando tratar o problema do não determinismo, diversas pesquisas apresentam soluções para reprodução da execução de programas concorrentes a fim de obter uma mesma saída para uma determinada entrada de teste (CHOI; ZELLER, 2002; BECHINI *et al.*, 1998; BALL *et al.*, 2009; DANTAS, 2008; DANTAS *et al.*, 2007; EMMI *et al.*, 2011; KAMIL; YELICK, 2010; RUNGTA; MERCER, 2009b). A geração de representações do programa concorrente, com objetivo de capturar informações pertinentes ao teste, também tem sido estudada (KATAYAMA *et al.*, 1995; ZHAO, 1999; SHIRES *et al.*, 1999;

ITYENGAR; DEBNATH, 1993). Resultados anteriores indicam que é promissor aplicar critérios de teste definidos para programas tradicionais no contexto de outros domínios de aplicações, como é o caso da programação concorrente (VERGILIO *et al.*, 2005; SOUZA *et al.*, 2005; SOUZA *et al.*, 2008; YANG *et al.*, 1998; YANG; POLLOCK, 2003). Nesse sentido, técnicas e critérios de teste específicos para programas concorrentes são propostos, complementando critérios de teste já existentes para programas sequenciais, considerando aspectos específicos da programação concorrente como: troca de mensagens, variáveis compartilhadas, sincronização e não determinismo (SOUZA *et al.*, 2008; KOJIMA *et al.*, 2009; LU *et al.*, 2007; TAKAHASHI *et al.*, 2008; SARMANHO *et al.*, 2008).

Além dessas dificuldades, devem ser considerados os erros comuns a esse tipo de domínio, não existentes nas aplicações sequencias. A fim de identificar esses erros, são definidas taxonomias ou classificações de erros.

Krawczyk *et al.* (1994) definem tipos de erros presentes em software concorrente. Essa definição é uma extensão da classificação de erros definida por Howden (1976) para programas sequenciais. A nova classificação considera características específicas dos programas concorrentes como comunicação e sincronização. Nela são definidas duas classes de erros:

- **Erro de observabilidade** ocorre num programa P se para algum caminho $p_i \in P$ existe outro caminho $p_j \in P$, tal que, $D(p_i) = D(p_j)$ e $C(p_i) \neq C(p_j)$.
- **Erro de travamento** ocorre num programa P se para algum caminho $p \in P$, formado pelo conjunto de nós $p = q_0, q_1, \dots, q_j$, existe um nó q_i em p tal que no subcaminho $p_s = q_0, q_1, \dots, q_i$ contendo um domínio $D(p_s)$, existe pelo menos um elemento de $D(p_s)$ em que todos os predicados avaliados a partir de q_i são falsos.

O erro de observabilidade ocorre quando o testador não pode controlar o comportamento do programa que está sendo executado em paralelo e diferentes sequências de comandos são executadas podendo produzir ao final um resultado esperado ou não. Nesse caso, uma entrada pode produzir uma saída avaliada como correta, o que não necessariamente está correta devido as sequências de sincronização que foram executadas. Um outro valor de entrada que exercitasse as mesmas sequências de sincronização poderia revelar o erro. O erro de travamento ocorre quando um processo não se comunica com o processo correto, ocasionando travamento a partir de determinado ponto da execução do programa (SOUZA *et al.*, 2016).

Outras situações que podem levar a ocorrência de erros em programas concorrentes são apresentadas por Lewis e Berg (1996). Alguns autores propõem taxonomias de erros específicas para determinadas linguagens de programação concorrente, listando os principais tipos de erros para um dado contexto (FARCHI *et al.*, 2003; BRADBURY; JALBERT, 2009). Arora *et al.* (2015) ilustram uma taxonomia geral de erros encontrados em aplicações concorrentes, nós categorizamos esses erros nas duas categoria citadas anteriormente, resultando na taxonomia

apresentada na Figura 6. A definição de taxonomias é importante nesse contexto, pois facilita a atividade de identificação de erros, tornando possível o desenvolvimento de técnicas e critérios de teste mais eficazes.

Figura 6 – Erros típicos em software concorrente.

Técnicas e critérios de teste são definidos para o tratamento de tipos específicos desses erros. Dentre os principais tipos de erros encontrados no contexto de programação concorrente os mais comuns são:

- **Erro de comunicação:** esse erro ocorre quando uma mensagem errada é enviada ou quando o envio da mensagem é feito para um destinatário errado;
- **Erro de sincronização:** esse erro ocorre quando uma sincronização não esperada entre uma *thread*/processo ocorre;
- **Condição de disputa:** esse erro ocorre quando uma região crítica do código fica desprotegida, permitindo que várias *threads*/processos alterem a variável compartilhada.
- **Violação de atomicidade:** é causado pela execução concorrente de várias *threads*/processos violando a atomicidade de uma região do código.
- **Deadlock:** esse erro ocorre quando duas *threads*/processos se bloqueiam mutuamente;
- **Livelock:** esse tipo de erro ocorre quando uma *thread*/processo não consegue terminar a execução ou entrar em uma região crítica por excesso de trabalho ou falta de velocidade;
- **Starvation:** esse erro ocorre quando uma *thread*/processo nunca é escalonada para execução.

2.5 Engenharia de Software Experimental

A engenharia de software tem adotado métodos de pesquisas de outras áreas (como a medicina) a fim de validar suas pesquisas. Nesse contexto, métodos empíricos como experimentos controlados, estudos de caso, *surveys* e análises *pós-mortem* são usados na avaliação da validade dos resultados obtidos nas pesquisas.

Na pesquisa experimental, os fenômenos a nossa volta são investigados por meio de observação. Em Engenharia de Software os fenômenos de interesse estão relacionados ao desenvolvimento de software. De modo geral, a pesquisa na área envolve caracterizar e avaliar as tecnologias utilizadas durante o desenvolvimento de software (métodos, técnicas, ferramentas) (SJOBERG *et al.*, 2007).

A aquisição de evidências é feita sempre por um conjunto de passos distintos: definição da questão de pesquisa, planejamento do estudo, coletar os dados, analisar e interpretar os resultados. Dependendo do tipo de dado coletado no estudo a análise pode ser: análise quantitativa, quando os dados coletados são numéricos e a análise é baseada em métodos estatísticos, análise qualitativa, quando os dados coletados estão no formato de texto (ou outro formato não numérico).

Estudos experimentais em engenharia de software podem ser de diferentes maneiras e sua escolha é baseada em diversos fatores, como relacionamento do método com a questão de pesquisa, a disponibilidade de recursos, a maneira como os pesquisadores desejam analisar os dados coletados. Projeto da pesquisa é o processo de selecionar um método para um problema particular de pesquisa levando em conta sua força e mitigando suas fraquezas. A confiabilidade dos resultados está relacionada ao quão bom é o design adotado compensado pelas fraquezas do método usado. Os principais métodos de estudos empíricos usados em pesquisas na área de Engenharia de Software (WOHLIN *et al.*, 2012; SHULL *et al.*, 2001; EASTERBROOK, 2007):

- **Experimento Controlado:** é um método de pesquisa que investiga um conjunto de hipóteses formais por meio da manipulação de variáveis independentes a fim de medir seus efeitos nas variáveis dependentes. Esse método é usado para observar fenômenos, formular teorias, confirmar o conhecimento convencional, explorar os relacionamentos, avaliar a predição dos modelos e validar medidas. Geralmente, experimentos controlados são executados em ambientes controlados, como laboratórios de pesquisa. Estudos experimentais são usados pelos pesquisadores para analisar como as variáveis se relacionam a fim de entender a relação entre a causa (tratamentos de variáveis) e os efeitos (resultados).
- **Estudo de Caso:** é um método experimental que busca investigar um fenômeno em seu contexto real, permitindo ter uma compreensão de como e porque tal fenômeno ocorre. Uma questão de pesquisa clara e que busca entender como e porque um fenômeno ocorre é o pré-requisito pra realização de um estudo de caso. É preciso destacar que há divergências na literatura quanto ao que constitui um estudo de caso na ES, uma vez que podem variar

de estudos ambiciosos e bem organizados em um ambiente de vida real (*in vivo*), para exemplos pequenos em laboratório universitário (*in vitro*). Frequentemente o termo é usado simplesmente para designar um exemplo de caso real, porém, como método experimental, autores têm conceituado estudo de caso como um método sistemático e rigoroso através do qual se apóiam pesquisas tipicamente exploratórias e qualitativas.

- **Survey:** é um método experimental que usa questionários para identificar características de uma população. Esse método é usado quando outras técnicas ou ferramentas foram anteriormente usadas, tendo como objetivo descrever, explicar e explorar as informações preliminares, tanto quantitativas quanto qualitativas por meio da coleta de dados usando questionário, além de levantar as variáveis do estudo a serem avaliadas.
- **Etnografia:** é um método experimental que tem como foco o estudo de uma comunidade de pessoas através da observação, com o objetivo de obter um entendimento aprofundado das práticas de uma determinada população, que não é possível conseguir a partir de outras técnicas, por exemplo, entrevista ou questionário. A etnografia visa a estudar uma comunidade de pessoas para entender como os membros interagem socialmente, na busca pela resposta de um problema de pesquisa. Na ES, a etnografia pode ajudar a compreender a cultura de práticas de software, tanto em pequenos contextos, por exemplo, os desenvolvedores de uma empresa, quanto em grandes comunidades, por exemplo, os programadores da linguagem Java.
- **Pesquisa-ação:** é um método de pesquisa no qual o pesquisador deseja resolver problemas do mundo real enquanto, simultaneamente, estuda a experiência de resolver o problema. Na pesquisa-ação os pesquisadores tentam estudar (pesquisa) e resolver (ação) o problema ao mesmo tempo, ou seja, intervir nas situações pesquisadas com o objetivo de melhorá-las. Esse método distingue-se dos anteriores em virtude do pesquisador deixar de ser um observador neutro para atuar, modificar e aprender a partir da ação que realiza durante a execução do estudo. Neste sentido, as principais características dessa abordagem são: expandir o conhecimento científico, desenvolver competências dos envolvidos, estimular um trabalho colaborativo em determinada situação e utilizar *feedback* dentro de um processo crítico.

A utilização de uma combinação de mais de um método experimental também tem sido adotada pelos pesquisadores a fim de que um método complemente as limitações do outro. Além dos métodos apresentados outros métodos de pesquisa aplicáveis no âmbito da investigação em ESE são discutidos na literatura (HÖFER; TICHY, 2007; GLASS *et al.*, 2002; ZELKOWITZ *et al.*, 2003): estudo de campo, simulação, teoria fundamentada nos dados, estudo correlacional e meta-análise. Assim como Easterbrook (2007), Wohlin *et al.* (2012) também fornecem uma classificação de estratégias empíricas utilizadas na área. Tais classificações são parecidas, no

entanto, [Wohlin et al. \(2012\)](#) não incluem o método etnografia. No entanto, consideram revisões sistemática da literatura e mapeamentos sistemáticos como estratégias de pesquisa experimentais.

2.5.1 Estudos experimentais em programação concorrente

Embora diversas técnicas de teste têm sido propostas para o contexto da programação concorrente, poucas têm apresentado avaliação experimental. Estudos experimentais sobre as vantagens e desvantagens das técnicas oferecem informações que podem ser úteis ao profissional de teste na escolha da técnica mais adequada ao seu projeto de software. Nós conduzimos um mapeamento sistemático (Apêndice B) buscando técnicas de teste para programas concorrentes dos quais foram incluídos 393 estudos que propõem técnicas para o teste de aplicações concorrentes das quais 109 apresentaram avaliação experimental de sua validade.

As informações requeridas para selecionar a técnica mais apropriada para um dado contexto pode ser obtida a partir métodos de engenharia de software experimental (ESE), que proporcionam um entendimento sobre os benefícios e limitações de cada técnica. Métodos de engenharia de software empírica usam análise de dados empíricos, quantitativa e qualitativa, para comparar técnicas em uma maneira cientificamente rigorosa. Os resultados empíricos podem ajudar profissionais de teste a escolher a técnica mais apropriada para seu contexto.

No contexto de programação concorrente, existem poucos estudos que usam conceitos da ESE ([BASILI et al., 1999](#)). A grande maioria dos estudos focam em oferecer uma prova de conceito, como um exemplo executável da técnica/ferramenta. Tipicamente esses estudos sofrem pela falta de comparações concretas entre técnicas diferentes. Isso dificulta para os pesquisadores e profissionais de teste entender fatores específicos que precisam ser considerados para determinar qual técnica apresenta melhor custo-benefício ao seu domínio de aplicação.

Embora estudos secundários em teste de software concorrente existam ([VEGAS; BASILI, 2005](#); [NETO; TRAVASSOS, 2008](#); [VOS et al., 2012](#)), eles focam somente em técnicas de teste, metodologias, e ferramentas usadas. Esses estudos primários não focam em como as técnicas tem sido validadas em relação ao tipo de evidência empírica apresentada. O nível de maturidade do conhecimento sobre as técnicas de teste podem ser avaliadas via estudos empíricos que validem essas propostas ([JURISTO et al., 2004](#)).

As revisões anteriores ([ARORA et al., 2015](#); [QAWY et al., 2012](#); [SOUZA et al., 2011](#); [BRITO et al., 2010](#); [MAMUN; KHANAM, 2009](#)) focam em descrever o estado da arte do teste de software concorrente. Categorizando técnicas, abordagens, métodos, tipos de erros revelados e ferramentas de suporte. Essas revisões pecam em não discutir sobre como as técnicas são avaliadas (como por exemplo usando estudos experimentais). Essas informações são importantes para analisar o nível de maturidade do conhecimento sobre as técnicas de teste, incluindo suas vantagens e desvantagens ([JURISTO et al., 2004](#)). Nosso estudo é o primeiro que foca em coletar evidências sobre estudos experimentais usados para validar técnicas de teste de software

concorrente. Além disso, embora alguns trabalhos tenham abordado o uso de evidência em teste de software (JURISTO *et al.*, 2004; JURISTO; MORENO, 2010; BRIAND; LABICHE, 2004; TYAGI; MALHOTRA, 2014) eles não abrangem o software concorrente.

Mamun e Khanam (2009) apresentam uma revisão sistemática sobre pesquisas em teste de software concorrente que utilizam o paradigma de memória compartilhada. A revisão abrange: características do software concorrente, tipos de erros, técnicas/ferramentas de teste, técnicas e ferramentas de geração de casos de teste e *benchmarks* de programas. Os autores classificam as técnicas de teste em estática ou dinâmica, além de identificar a linguagem de programação suportada pela ferramenta de teste e os *benchmarks* de programas. As contribuições desse trabalho incluem: a identificação de 51 técnicas de teste, 38 ferramentas de teste, e 87 padrões de erros para a linguagem Java.

Brito *et al.* (2010), Souza *et al.* (2011) apresentam uma revisão sistemática sobre estudos relacionados com abordagens de teste de software concorrente, classificação de erros, e ferramentas de teste. Essa revisão reusa algumas categorias similares as apresentadas em Mamun e Khanam (2009) e são adicionadas novas categorias relacionadas ao paradigma de programação concorrente (passagem de mensagem e memória compartilhada). O estudo apresenta um mapeamento do estado da arte do teste de software concorrente e apresenta as seguintes categorias: 1) 166 artigos que apresentam uma taxonomia, classificação ou discussão sobre teste de software concorrente; 2) 6 estudos que apresentam uma taxonomia, classificação ou discussão sobre erros concorrentes; e 3) 50 estudos que apresentam uma ferramenta que automatiza uma metodologia para auxiliar o teste de software concorrente. O total de estudos excede o total de 175 estudos que os autores dizem ter incluído na revisão pois alguns estudos apresentam contribuição para mais de uma categoria.

Assim como os estudos anteriores, a revisão apresentada por Arora *et al.* (2015) sumariza a pesquisa sobre teste de software concorrente. Mas esse é o primeiro estudo que classifica as técnicas de teste em uma abordagem em particular. Nesse estudo, os autores classificam os estudos selecionados em oito categorias: 1) teste de alcançabilidade; 2) teste estrutural; 3) teste baseado em modelo; 4) teste de mutação; 5) teste de particionamento; 6) métodos formais; 7) teste randômico; e 8) teste baseado em busca. Esses estudos focam em técnicas de teste e classificam elas em: 8 abordagens de teste, 80 algoritmos, 11 representações gráficas usadas em diferentes técnicas de teste e 83 ferramentas disponíveis que avaliam e implementam metodologias específicas.

O mapeamento sistemático apresentado neste trabalho (descrito no Apêndice B) é similar às revisões anteriores, levando em conta que busca estudos sobre técnicas de teste de software concorrente. Entretanto tem um escopo diferente, focando em buscar evidências sobre estudos experimentais existentes em teste de software concorrente. Uma visão geral dos aspectos das técnicas de teste, que têm sido estudados experimentalmente, proporciona aos pesquisadores mais facilmente identificar lacunas que podem ser cobertas em pesquisas futuras. Os resultados

construídos baseados na ESE podem ser úteis para criar modelos preditivos que podem ser utilizados em aplicações similares na indústria.

O objetivo foi obter juntamente evidência experimental sobre técnicas de teste de software concorrente e possibilitar que desenvolvedores respondam questões sobre as melhores técnicas, em relação ao custo dessas técnicas, os tipos de falhas que cada uma revela, e qual é mais apropriada para um dado contexto. Esses tipos de conclusões podem ser mais difíceis de se obter baseadas somente em estudos de prova de conceito sem dados experimentais comparativos. Dados extraídos de estudos experimentais ajudam desenvolvedores a compararem técnicas e tornam o processo de seleção mais sistemático e menos randômico (PERRY *et al.*, 2000).

2.6 Considerações Finais

Neste capítulo foram apresentados os principais conceitos e terminologias relacionados as principais áreas que este projeto envolve: teste de software que é o campo da engenharia de software no qual o projeto está inserido, a programação concorrente que é o objeto de estudo para a condução dos testes, e a intersecção dessas áreas, o teste de software concorrente que envolve a definição de técnicas e critérios de teste específicos para este domínio. Além disso é dada ênfase à importância da condução de estudos experimentais para a avaliação das técnicas propostas, bem como a evolução da engenharia de software experimental na área de teste de software concorrente. O desenvolvimento deste capítulo objetivou contextualizar as áreas de pesquisa englobadas por esse projeto de doutorado evidenciando os elementos de estudo que motivaram a proposição do trabalho.

FRAMEWORK PARA AVALIAÇÃO SISTEMÁTICA DE TÉCNICAS DE TESTE DE SOFTWARE CONCORRENTE

3.1 Considerações Iniciais

Neste trabalho de doutorado foi desenvolvido um *framework* que apoia o processo de avaliação para seleção de técnicas de teste de software no contexto da programação concorrente. Um mapeamento sistemático foi conduzido a fim de obter uma visão geral do estado da arte da área de teste de software concorrente e um corpo de evidências foi construído baseado no conhecimento levantado. O *framework* é constituído por: 1) um corpo de evidências sobre técnicas de teste de software concorrente, com informações relevantes ao processo de seleção de técnicas de teste; 2) um esquema de caracterização que compõe uma abordagem de auxílio à seleção de técnicas de teste; 3) um design padrão de experimentos a fim de prover a condução de novos estudos experimentais e realimentar o corpo de evidências e 4) uma infraestrutura web que disponibiliza o acesso ao corpo de conhecimento e ao processo sistematizado de seleção de técnicas de teste (baseado na comparação de atributos de projeto e técnicas de teste e na análise de critérios de adequação). Por meio do *framework* é possível obter informações sobre técnicas de teste que têm sido propostas no contexto da programação concorrente, bem como buscar técnicas que atendam a uma característica específica. Além disso, é possível fornecer informações a respeito de um projeto de software e a partir daí obter recomendações sobre as técnicas que melhor se adequam ao projeto. Outra contribuição disponível é um guia para condução de estudos experimentais nesse contexto a fim de ampliar as evidências na área, repovoando o corpo de conhecimento proposto.

Neste capítulo é apresentado o *framework*, descrevendo na Seção 3.2 sua arquitetura, seus principais componentes e a interação entre eles. Na Seção 3.3 é descrito em detalhes o corpo de

conhecimento construído para a área de teste de software concorrente. Na Seção 3.4 é apresentado o design de experimentos como uma estratégia para a evolução desse corpo de conhecimento. Na Seção 3.5 é descrito o esquema de caracterização e os atributos que compõem a abordagem de seleção proposta. E por fim, na Seção 3.6 é apresentada a infraestrutura computacional que utiliza essas informações e a implementação desses componentes para automatizar o processo de seleção de técnicas de teste de software concorrente.

3.2 Arquitetura do *Framework*: Principais Componentes

Como pode ser observado em revisões de literatura sobre teste de software concorrente (SOUZA *et al.*, 2011; MAMUN; KHANAM, 2009; ARORA *et al.*, 2015), existe uma grande quantidade de dados sobre técnicas de teste de software propostas para o contexto da programação concorrente. Embora essas revisões tragam uma visão geral da área, e uma classificação das técnicas de maneira geral, elas não oferecem uma caracterização apropriada das técnicas a fim de apoiar o processo de tomada de decisão sobre qual técnica usar durante o desenvolvimento de um produto de software. Segundo Basili (1985) e Basili *et al.* (2007), dados dispersos (não reunidos de uma maneira lógica ou sistematizada) não possibilitam a aquisição de conhecimento útil sobre o contexto a que se referem e não trazem informações relevantes que possibilitem seu uso efetivo.

Como uma iniciativa para reversão desse quadro, tentando tornar mais visível o conhecimento desenvolvido na academia para a indústria, foi definido um mapeamento sistemático buscando evidências na área de teste de software concorrente (Apêndice B). Nesse contexto e devido ao fato de que não foi encontrada na literatura uma base de conhecimento integrada que reúna informações relevantes sobre o processo de teste para aplicações concorrentes de forma que essas informações estejam disponíveis à academia e ao mercado de trabalho, foi proposto em Melo *et al.* (2017) um corpo de conhecimento de apoio a caracterização e seleção de técnicas de teste de software concorrente (Seção 3.3).

O corpo de conhecimento é a base para a construção do *framework* de auxílio à sistematização do processo de seleção de técnicas de teste. Na Figura 7 é ilustrada a arquitetura do *framework* proposto neste projeto de doutorado, o qual é composto por: 1) uma base de conhecimento que reúne as evidências na área de teste de software concorrente, disponível à comunidade interessada; 2) uma abordagem de seleção, composta por um esquema de caracterização de técnicas de teste e critérios de adequação, que definem pesos aos critérios (informações sobre as técnicas) a fim de oferecer um mecanismo de comparação e definir qual técnica é mais adequada a um projeto especificado; 3) um design de experimentos que oferece diretrizes para condução de estudos experimentais nessa área, funcionando como um guia para evolução do conhecimento e 4) a implementação da abordagem proposta em uma infraestrutura computacional que automatiza o processo de seleção e disponibiliza o conhecimento do repositório online, proporcionando à

comunidade acadêmica e de profissionais de teste uma ferramenta para gestão de conhecimento e evolução do processo de teste de software.

Figura 7 – Arquitetura do *framework*.

Considerando a adoção da técnica recomendada ao projeto de software, é possível o desenvolvimento de novos estudos experimentais sobre a técnica e a reunião de novas evidências que justifiquem a eficiência e aplicabilidade da técnica de teste. Nesse sentido, um design de experimentos é proposto para funcionar como guia para a construção de evidência na área e também facilitar a comparação e replicação. Esse design atua no *framework* como um método para a atualização e evolução do corpo de conhecimento, permitindo que novos estudos experimentais sejam conduzidos e que esses estudos possam realimentar o repositório de dados. Mais detalhes sobre o design de experimentos são apresentados em [Melo et al. \(2016\)](#) e na Seção 3.4.

A fim de melhorar o processo de teste, mais especificamente a atividade de seleção da técnica de teste a ser aplicada no projeto de software desenvolvido, é proposta uma abordagem de seleção de técnicas. A ideia é reunir informações relevantes sobre o projeto de software e confrontar com informações já existentes sobre técnicas de teste (disponíveis no corpo de conhecimento). Com base em métricas pré-definidas, as técnicas podem ser comparadas e analisadas conforme sua adequabilidade ao projeto de software, fornecendo aos interessados alternativas ao processo de teste. As métricas e critérios definidos, são descritos em detalhes na Seção 3.5.

O processo de seleção foi automatizado por meio da implementação da ferramenta SeleCTT (Seção 3.6), uma infraestrutura web que utiliza o corpo de conhecimento e informações sobre requisitos do projeto para fornecer recomendação sobre técnicas possíveis de serem adotadas em um projeto específico de teste. O processo de seleção por meio da ferramenta é dividido em três etapas: 1) fornecimento das informações sobre os requisitos do projeto de software, conforme solicitado via formulário de dados pela ferramenta; 2) extração das informações sobre as técnicas de teste contidas no corpo de evidência; 3) comparação das informações sobre o projeto com as informações sobre as técnicas de teste e cálculo da técnica adequada via critérios de adequação definidos na abordagem de seleção e 4) disponibilização aos interessados (profissionais/pesquisadores) de uma lista ordenada das técnicas mais relevantes ao projeto analisado. Como métricas para avaliação das técnicas são definidos critérios de adequação com pesos sob a influência de cada atributo no processo de seleção. O cálculo da distância euclidiana (DOKMANIC *et al.*, 2015) é usado para ordenar as técnicas conforme seu nível de adequabilidade ao projeto e uma recomendação é fornecida à equipe de teste. Cabe à equipe de teste avaliar dentre as opções fornecidas, qual se encaixa melhor ao seu processo de desenvolvimento e restrições de projeto.

3.3 Corpo de Conhecimento Sobre Teste de Software Concorrente

3.3.1 Definição do corpo de conhecimento

Embora uma grande variedade de técnicas tenha sido proposta pela academia para o teste de aplicações concorrentes, a aplicação prática dessas técnicas na indústria ainda permanece como um desafio para a área. A transferência de conhecimento entre academia e indústria é, em grande parte, dificultada pela falta de um corpo de conhecimento que contenha informações que ajudem no processo de tomada de decisão e seleção das técnicas existentes (VEGAS; BASILI, 2005).

Apesar da disponibilidade de grande quantidade de dados sobre a área de teste de software concorrente, essas informações encontram-se de forma dispersa e muitas vezes sem conexão, o que aumenta ainda mais a dificuldade dos usuários em compreender os conceitos relacionados as tecnologias e consequentemente a escolha de onde e quando usá-las. Além disso, a falta de caracterização apropriada da tecnologia contribui para dificultar o processo de aquisição de conhecimento e na tomada de decisão sobre qual técnica usar durante o processo de desenvolvimento de software.

Alguns trabalhos têm tratado o problema da caracterização de técnicas de teste de software apenas para o escopo da programação sequencial, não agregando conhecimento suficiente para instanciar a área de teste de software concorrente. Como consequência, as equipes de teste

permanecem usando seu próprio conhecimento e hábitos para seleção de uma técnica de teste para um novo projeto de software. Esse fator pode levar ao reúso de uma mesma técnica, sem considerar a existência de outras. Além disso, a dificuldade de transferência de conhecimento da academia para indústria tem resultado na escassez de estudos experimentais relevantes na área. A fim de auxiliar a equipe de teste a tomar decisões, são necessárias informações que possam ser comparadas e levem à escolha correta da técnica.

Nesse contexto, foi definido neste trabalho um corpo de conhecimento com o objetivo de armazenar e disponibilizar informações úteis ao processo de seleção de tecnologias de teste, na forma de um repositório que permite aos pesquisadores investigar hipóteses gerais sobre técnicas de teste e estabelecer comparações entre diferentes estudos.

A Figura 8 ilustra o processo de derivação do corpo de conhecimento. O processo inicia-se com a análise das características das técnicas de teste que influenciam no processo de seleção (Passo 1). Esta informação é usada para o desenvolvimento de um catálogo de técnicas e ferramentas existentes (MELO; SOUZA, 2014), resultando em um corpo de conhecimento preliminar que pode ser atualizado com a proposição de novas técnicas. Os resultados da análise de estudos experimentais (Passo 2) são usados para classificar os estudos que irão compor o repositório. Informações para caracterização de projetos de software (Passo 3) são extraídas da literatura técnica, baseados no esquema de caracterização proposto em (VEGAS; BASILI, 2005), incluindo atributos relacionados ao teste de software concorrente, validados por meio de um survey com especialistas na área. Finalmente, todas as informações obtidas são usadas para construir o esquema de caracterização apresentado na Tabela 7 e o corpo de evidências (Passo 4) disponível online¹.

Figura 8 – Processo de derivação para geração do corpo de conhecimento.

¹ <<http://www.labes.icmc.usp.br/~selectt/content/techniques>>

3.3.2 Estrutura do corpo de conhecimento

O corpo de conhecimento preliminar contém informação relevante sobre técnicas de teste obtidas de 109 estudos primários. O gerente de teste pode comparar essas informações com os atributos do projeto de software e selecionar a técnica de teste que é mais adequada aos objetivos e restrições de seu projeto.

O corpo de conhecimento é implementado como um repositório de dados e está disponível online possibilitando a consulta às técnicas existentes por título do estudo científico, linguagem de programação que a técnica suporta e o tipo de técnica de teste. A estrutura do corpo de conhecimento segue o esquema de caracterização de técnicas de teste proposto na Seção 3.5. A Figura 9 ilustra o corpo de conhecimento dentro do ambiente SeleCTT (Seção 3.6).

Figura 9 – Interface do corpo de conhecimento online.

3.3.3 Conteúdo

Um total de 109 estudos foram identificados por meio do mapeamento sistemático (Apêndice B). Esses estudos propõem técnicas de teste de software para aplicações concorrentes e apresentam uma avaliação experimental da aplicabilidade da proposta.

A Figura 10 mostra o número de estudos experimentais publicados a cada ano na área. Pode-se observar um aumento no número de evidências a partir de 2006, que pode ser atribuída a importância do teste de software com o surgimento, nesse período, dos processadores com dois núcleos e múltiplas threads nos computadores pessoais. O mapeamento foi executado no primeiro semestre de 2016, ou seja não haviam resultados completos sobre esse período, por esse motivo, esse ano não é apresentado no gráfico.

Os estudos que compõem o corpo de conhecimento foram categorizados por tipo de técnica de teste proposta conforme apresentado na Figura 11. Os resultados mostram que a frequência de evidências na área varia de acordo com o tipo de técnica de teste de software. A técnica estrutural apresentou o maior número de avaliações experimentais, seguida pela

Figura 10 – Evidências em teste de software concorrente por ano.

técnica baseada em erros. Além disso, foram identificados estudos experimentais para todas as principais técnicas de teste de software concorrente. Essa observação é uma boa indicação aos pesquisadores que buscam aplicar métodos de avaliação experimental às técnicas de teste propostas, pois mostra que essa é uma área que está avançando.

Figura 11 – Distribuição dos estudos por tipo de técnica de teste.

Os estudos foram categorizados também por tipo de linguagem de programação usada para implementação da aplicação. A Figura 12 ilustra a distribuição dos estudos por tipo de linguagem de programação, os resultados mostram que as linguagens predominantes são Java e C/C++. No entanto, há técnicas de teste propostas para uma variedade de outras linguagens de programação, mas de fato, como constatado em outro estudo na área (MELO *et al.*, 2015), a maioria das contribuições em teste de software concorrente e ferramentas de teste estão relacionadas a essas linguagens de programação.

A Figura 13 sumariza o estado-da-arte da pesquisa em engenharia de software experimental no contexto de teste de software concorrente. Na figura, o tamanho da bolha representa o número de estudos incluídos para cada técnica de teste relacionada ao tipo da amostra e ao método empírico empregado. Algumas considerações podem ser obtidas por meio da análise do gráfico: 1) a técnica de teste baseada em erros usa amostras mais realísticas, isso se deve ao fato

Figura 12 – Distribuição dos estudos por linguagem de programação.

de apresentar um alto número de estudos de casos e 2) a técnica de teste estrutural usa em sua maioria estudos de caso, mais usa aplicações reais e programas exemplo também. Além disso, no gráfico é possível identificar algumas lacunas na área: 1) há poucos estudos empíricos sobre teste baseado em modelos, 2) existe uma deficiência de experimentos controlados sobre teste baseados em métodos formais e 3) a maioria dos estudos são estudos de caso que focam em uma prova de conceito para uma nova técnica ou ferramenta e não proporciona análises entre estudos.

Figura 13 – Estado-da-arte da área de teste de software concorrente.

Na Tabela 1 são apresentadas as 109 técnicas incluídas nesse estudo. Um catálogo auxiliar sobre ferramentas para o teste de aplicações concorrentes pode ser consultado em [Melo et al. \(2015\)](#). Essas técnicas foram analisadas e foram extraídas informações que compõem o esquema de caracterização e a abordagem de seleção descritos na Seção 3.5. Além de prover a base para a fundamentação do esquema de seleção, o corpo de conhecimento pode ser usado por qualquer interessado nessa área para identificar técnicas de teste compatíveis com os objetivos e restrições do projeto de software.

Tabela 1 – Estudos primários sobre técnicas para o teste de software concorrente.

Título do estudo	Autor	Ano
Types for safe locking: Static race detection for Java	Abadi et al.	1997
Incremental integration testing of concurrent programs	Koppol et al.	2002
Concurrent software verification with states, events, and deadlocks	Chaki et al.	2005
Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs	Wang et al.	2006
Enforcer - Efficient failure injection	Artho et al.	2006
Reachability testing of concurrent programs	Lei et al.	2006
Runtime analysis of atomicity for multithreaded programs	Wang et al.	2006
AVIO: Detecting atomicity violations via access interleaving invariants	Lu et al.	2006
Comparative assessment of testing and model checking using program mutation	Bradbury et al.	2007
Effective random testing of concurrent programs	Sen Koushik	2007
A combinatorial testing strategy for concurrent programs	Lei et al.	2007
A race-detection and flipping algorithm for automated testing of multi-threaded programs	Sen et al.	2007
Iterative context bounding for systematic testing of multithreaded programs	Musuvathi et al.	2007
Learning from mistakes: a comprehensive study on real world concurrency bug characteristics	Lu et al.	2008
Incremental testing of concurrent programs using value schedules	Chen et al.	2008
Race directed random testing of concurrent programs	Sen et al.	2008
CTrigger: Exposing atomicity violation bugs from their hiding places	Park et al.	2008
TM UNIT : A Transactional Memory Unit Testing and Workload Generation Tool	Harmanci et al.	2008
Fair stateless model checking	Musuvathi et al.	2008
Clash of the Titans: Tools and techniques for hunting bugs in concurrent programs	Rungta et al.	2009
Saturation-based testing of concurrent programs	Sherman et al.	2009
A meta heuristic for effectively detecting concurrency errors	Rungta et al.	2009
Kendo: efficient deterministic multithreading in software	Olsewski et al.	2009
How Good is Static Analysis at Finding Concurrency Bugs?	Kester et al.	2010
A Lightweight and Portable Approach to Making Concurrent Failures Reproducible	Luo et al.	2010
A platform for search-based testing of concurrent software	Krena et al.	2010
A randomized scheduler with probabilistic guarantees of finding bugs	Burckhardt et al.	2010
Composable Specifications for Structured Shared-Memory Communication	Wood et al.	2010
ConMem: Detecting severe concurrency bugs through an effect-oriented approach	Zhang et al.	2010
Distributed reachability testing of concurrent programs	Carver et al.	2010
Kivati: Fast detection and prevention of atomicity violations	Chew et al.	2010
InstantCheck: Checking the determinism of parallel programs using on-the-fly incremental hashing	Nistor et al.	2010
MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code.	Gligoric et al.	2010
Preemption sealing for efficient concurrency testing	Ball et al.	2010
RACEZ : A Lightweight and Non-Invasive Race Detection Tool for Production Applications	Sheng et al.	2011
Testing Concurrent Programs on Relaxed Memory Models	Burnim et al.	2011
ConSeq: Detecting Concurrency Bugs through Sequential Errors	Zhang et al.	2011
Coverage guided systematic concurrency testing	Wang et al.	2011
Maximizing synchronization coverage via controlling thread schedule	Kim et al.	2011
MultiOtter : Multiprocess Symbolic Execution	Turpie et al.	2011
SOS: Saving Time in Dynamic Race Detection with Stationary Analysis	Li et al.	2011
Applications of Model Reuse When Using Estimation of Distribution Algorithms to Test Concurrent Software	Staunton et al.	2011
Detection of asynchronous message passing errors using static analysis	Christakis et al.	2011
Efficient data race detection for distributed memory parallel programs	Park et al.	2011
BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code	Nistor et al.	2012
CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications	Zhai et al.	2012
Location pairs: a test coverage metric for shared-memory concurrent programs	Tasiran et al.	2012
Automatic testing environment for multi-core embedded software - ATEMES	Koong et al.	2012
Coverage metrics for saturation-based and search-based testing of concurrent software	Krena et al.	2012
Exploiting parallelism in deterministic shared memory multiprocessing	Zhou et al.	2012
Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing	Lu et al.	2012
Maple: a coverage-driven testing tool for multithreaded programs	Yu et al.	2012
Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection	Nagarakatte et al.	2012
On a Technique for Transparently Empowering Classical Compiler Optimizations on Multithreaded Code	Joisha et al.	2012
DTAM: Dynamic Taint Analysis of Multi-threaded Programs for Relevancy	Ganai et al.	2012
SAM: Self-adaptive dynamic analysis for multithreaded programs	Chen et al.	2012
Sound Predictive Race Detection in Polynomial Time	Smaragdakis et al.	2012
TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs	Tasharofi et al.	2012
Verification and Coverage of Message Passing Multicore Applications	Deniz et al.	2012
A Modular Approach to Model-based Testing of Concurrent Programs	Carver et al.	2013
An Empirical Evaluation of the Cost and Effectiveness of Structural Testing Criteria for Concurrent Programs	Brito et al.	2013
Efficient mutation testing of multithreaded code	Gligoric et al.	2013
Generating effective tests for concurrent programs via AI automated planning techniques	Razavi et al.	2013
Generating unit tests for concurrent classes	Steenbuck et al.	2013
The Impact of Concurrent Coverage Metrics on Testing Effectiveness	Hong et al.	2013
An approach to testing commercial embedded systems	Yu et al.	2013
An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs	Xin et al.	2013
An empirical study of data race detector tools	Alowibdi et al.	2013
ASP: Abstraction Subspace Partitioning for Detection of Atomicity Violations with an Empirical Study	Wu et al.	2013
Automated Classification of Data Races Under Both Strong and Weak Memory Models	Kasikci et al.	2013
Runtime Prevention of Concurrency Related Type-state Violations in Multithreaded Applications	Zhang et al.	2013
Selective mutation testing for concurrent code	Gligoric et al.	2013
TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems	Kestor et al.	2013
Con2Colic Testing	Farzan et al.	2013
Test-Data Generation for Testing Parallel Real-Time Systems	Aziz et al.	2013
Variable and thread bounding for systematic testing of multithreaded programs	Bindal et al.	2013
On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Functional Faults in Concurrent Object-Oriented Software	Araujo et al.	2013
Efficient concurrency-bug detection across inputs	Deng et al.	2013
Synchronization identification through on-the-fly test	Yuan et al.	2013
ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs	Cai et al.	2014
Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs	Souza et al.	2014
RECONTEST: Effective Regression Testing of Concurrent Programs	Terragni et al.	2014
Toward a Methodology to Expose Partially Fixed Concurrency Bugs in Modified Multithreaded Programs	Tsui et al.	2014
Efficient Deterministic Multithreading Without Global Barriers	Lu et al.	2014
Efficient Execution Path Exploration for Detecting Races in Concurrent Programs	Setiadi et al.	2014
Pegasus: Automatic Barrier Inference for Stable Multithreaded Systems	Dhok et al.	2014
A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs	Hwang et al.	2014
An Efficient and Flexible Deterministic Framework for Multithreaded Programs	Kai et al.	2014
Bitra: Coverage-guided, automatic testing of actor programs	Tasharofi et al.	2014
CLAP: Recording Local Executions to Reproduce Concurrency Failures	Huang et al.	2014
ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs	Khoshnood et al.	2014
Embedding Semantics of the Single-Producer/Single-Consumer Lock-Free Queue into a Race Detection Tool	Dolz et al.	2014
Using SPIN for automated debugging of infinite executions of Java programs	Adalid et al.	2014
Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation	Hong et al.	2015
Assertion Guided Symbolic Execution of Multithreaded Programs	Guo et al.	2015
Concurrency Testing Using Schedule Bounding: an Empirical Study	Thomson et al.	2015
Detecting Event Anomalies in Event-based Systems	Safi et al.	2015
SATCheck: SAT-Directed Stateless Model Checking for SC and TSO	Demsky et al.	2015
Advances in noise-based testing of concurrent software	Fiedor et al.	2015
ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs	Zhai et al.	2015
Concurrency Debugging with Differential Schedule Projections	Machado et al.	2015
Effective and Precise Dynamic Detection of Hidden Races for Java Programs	Cai et al.	2015
Multithreaded Test Synthesis for Deadlock Detection	Samak et al.	2015
Race Detection for Android Applications	Maiya et al.	2015
Value-deterministic Search-based Replay for Android Multithreaded Applications	Jha et al.	2015
Synthesizing Racy Tests	Samak et al.	2015
Synthesizing Tests for Detecting Atomicity Violations	Samak et al.	2015
Unfolding based automated testing of multithreaded programs	Kahkonen et al.	2015
An Evaluation of Data Race Detectors Using Bug Repositories	Ganzha et al.	2016

3.4 *Design* de Experimentos na Área de Teste de Software Concorrente

O principal problema associado a transferência de tecnologia entre academia e indústria não é a falta de novas tecnologias e sim definir como e quando usar as tecnologias já existentes (BASILI, 1985). Dentre os principais problemas associados à transferência de tecnologia estão: 1) as novas tecnologias podem não ser adaptáveis às necessidades da indústria; 2) os interessados na tecnologia não conseguem enxergar os benefícios de sua aplicação em seus projetos e 3) o conhecimento obtido pela experiência de projetos anteriores não pode ser reaproveitado, pois não foi devidamente registrado a fim de evidenciar as vantagens do seu uso (LINKMAN; ROMBACH, 1997). Nesse contexto, uma alternativa é a utilização da experimentação para viabilizar a transferência tecnológica.

A Engenharia de Software Experimental pode funcionar como um mecanismo efetivo para conduzir a apresentação de novas tecnologias em ambiente industrial, mostrando-se como um componente tanto de pesquisa quanto educacional na Engenharia de Software. A troca de resultados experimentais entre indústria e academia pode fornecer ganhos para ambos os lados. Na indústria, por obter acesso a replicações, dados, pacotes de laboratório e artefatos de domínios genéricos que possibilitam avaliar a efetividade da nova tecnologia com a efetividade obtida em outras experiências, motivando-a, conseqüentemente, para a utilização da tecnologia e levando-a também a compartilhar os seus resultados, enriquecendo o corpo de conhecimento da área (HOHN *et al.*, 2004).

Os benefícios para a academia na interação com a indústria estão relacionados à possibilidade de validação externa dos estudos experimentais com a inclusão de artefatos de domínios de aplicações reais, tornando possível generalizar e evoluir pacotes de laboratório, motivando a pesquisa por novas metodologias.

Um *design* de experimentos fornece informações que guiam o leitor a especificar como avaliar uma técnica, como as questões de pesquisa podem ser definidas, quais variáveis podem ser medidas, quais ameaças à validade podem ser identificadas, entre outros fatores que compõem um estudo experimental (LOTT; ROMBACH, 1996).

Dada a importância de um *design* de experimentos para a ES destacada anteriormente, em Melo *et al.* (2016) é definido um *design* de experimentos na forma de um esquema estruturado com o objetivo de reduzir o esforço requerido para desenvolver e conduzir novos experimentos, bem como facilitar a comparação dos resultados para experimentos similares fornecendo um guia para auxiliar no desenvolvimento de estudos mais controlados e replicáveis em teste de software concorrente.

Comumente, as métricas usadas para avaliar as variáveis dependentes são diferentes e o contexto do experimento varia. Os estudos experimentais usam diferentes métricas para

medir a eficácia (o número de erros detectados, o número de mutantes mortos, a cobertura de código alcançada), o que torna os resultados difíceis de comparar. As variáveis dependentes, independentes e de contexto dos estudos precisam ser definidas em um mesmo *design* teórico a fim de obter informações úteis sobre os estudos e possibilitar a meta-análise e suporte ao desenvolvimento de teorias de causa e efeito (GALLIS *et al.*, 2003).

Nas próximas seções são descritos os passos para a definição do *design* de experimentos no contexto de teste de software concorrente. São descritos objetivos, variáveis e escolhas a serem tomadas durante o processo experimental. Usando o *design* proposto, famílias de estudos podem ser conduzidas podendo contribuir com hipóteses relevantes e importantes que não poderiam ser levantadas com base apenas em experimentos individuais (BASILI *et al.*, 1999; BASILI *et al.*, 2007).

O principal objetivo desse *design* de experimentos para o trabalho de doutorado foi oferecer uma alternativa à realimentação do corpo de conhecimento. Considerando que o corpo de conhecimento é formado por estudos experimentais na área de teste de software concorrente, a produção de experimentos sob um mesmo *design*, dadas as recomendações fornecidas pelo *design* proposto, podem oferecer objetivos concretos e reusáveis, planejamento e procedimentos os quais podem gerar evidência facilmente comparada e efetivamente agregada em estudos secundários.

3.4.1 Definição do experimento

A fase de definição do experimento determina como um experimento deve ser realizado e auxilia na construção e *design* do experimento.

1.1 Definição do Objetivo

A definição do objetivo é essencial para as atividades de planejamento e condução do experimento. Essa atividade envolve a escolha do aspecto a ser avaliado, a definição dos dados que serão coletados e como esses dados serão analisados.

O objetivo do experimento é definido usando os fundamentos do processo GQM (*Goal Question Metrics*) (BASILI, 1992). O modelo GQM é composto de cinco parâmetros: 1) Objeto do estudo: a entidade que será estudada no experimento; 2) Propósito: define a intenção do experimento; 3) Foco: o efeito primário sob estudo no experimento; 4) A perspectiva: o ponto de vista a partir do qual os resultados do experimento serão interpretados e 5) Contexto: define quais sujeitos estão envolvidos no experimento (BASILI *et al.*, 1999). Na Tabela 2 é mostrado o modelo proposto para a definição das questões de pesquisa do experimento para teste de software concorrente. Os valores na tabela são usados como referência para instanciar objetivos nesse contexto. Para ilustrar um possível objetivo para um estudo experimental no contexto de software concorrente pode ser definido como:

Analisar a técnica de teste de software concorrente (objeto de estudo) **com o propósito de avaliar** (propósito) **em relação à sua efetividade** (foco de qualidade) **do ponto de vista do pesquisador** (perspectiva) **no contexto de especialistas de teste encontrando erros em seu software concorrente** (contexto).

Tabela 2 – Modelo GQM para teste de software concorrente.

Definição do objetivo	
Objeto de estudo	técnica de teste de software concorrente (Tabela 3)
Propósito	avaliar, analisar, comparar, medir, estudar
Foco de qualidade	efetividade, eficiência, eficácia, aplicabilidade, usabilidade, viabilidade, reprodutibilidade, corretude, custo, desempenho (Tabela 4)
Perspectiva	pesquisador, profissional de teste, estudantes
Contexto	sujeitos, objetos, ambiente (Seção 3.4.2.1)

3.4.2 Planejamento do experimento

Como proposto por Wohlin *et al.* (2012), o *design* do experimento é o resultado da definição do planejamento do experimento e composto de sete etapas. Com o objetivo de oferecer um *design* de experimentos a ser usado como guia para a definição de experimentos em teste de software concorrente, cada etapa é apresentada em detalhes nas próximas seções.

3.4.2.1 Seleção do contexto

Baseado no objetivo definido, a seleção do contexto especifica o ambiente no qual o experimento será executado. O contexto do experimento pode ser classificado em termos do número de sujeitos e objetos envolvidos no estudo. Assim, o contexto do experimento pode ser caracterizado de acordo com quatro dimensões (WOHLIN *et al.*, 2012): 1) *off-line* vs. *on-line*; 2) estudante vs. profissional; 3) programas exemplo vs. aplicações reais; e 4) específico vs. geral.

A maioria dos estudos em teste de software concorrente presentes na literatura são conduzidos de maneira *off-line* por pesquisadores e usando programas provenientes de *benchmarks* (ou programas exemplos) em um contexto específico e controlado.

A comunidade de pesquisa tem argumentado que estudos de laboratório, conduzidos por estudantes, têm pouca qualidade e relevância se comparados a estudos conduzidos na indústria. No entanto, o estabelecimento de colaboração com a indústria é difícil e depender desta colaboração pode tornar o estudo inviável. A definição de projetos executados por estudantes pode ser uma boa alternativa, porque esse tipo de projeto é barato e fácil de controlar, porém é mais direcionado a certos contextos do que projetos executados por profissionais com vasta experiência na indústria (HÖST *et al.*, 2000).

Participantes

O participante do experimento é quem vai aplicar o experimento. Os participantes podem ser referidos como sujeitos (se humanos) ou unidades experimentais (se não humanos), nesse caso apenas máquinas executando *scripts* por exemplo. O tipo de participante pode ser (estudantes de ciência da computação, profissionais de teste, pesquisadores). Sujeitos, nesse caso, são as pessoas que executam o experimento, utilizando um método, ferramenta e programas, ou algoritmos nos quais os métodos ou ferramentas são aplicados (WOHLIN *et al.*, 2012).

Objetos

Objeto experimental é a unidade sob a qual os tratamentos são aplicados e os resultados avaliados. No contexto de teste de software, o objeto pode ser caracterizado como o SUT (*Software Under Testing*). Este software pode ser caracterizado como um exemplo didático (programa exemplo), aplicações industriais ou um conjunto de programas provenientes de *benchmarks*. Existe uma variedade de *benchmarks* que podem ser usados como objetos para avaliação experimental na área de teste de software concorrente (DOURADO *et al.*, 2016; RUNGTA; MERCER, 2009a; EYTANI; UR, 2004; YANG *et al.*, 2008; VALGRIND-DEVELOPERS, 2012).

3.4.2.2 Definição de hipóteses

Hipóteses são formalizações do estudo experimental proposto e podem ser usados para testar afirmações sobre os resultados esperados. Em geral, dois tipos de hipóteses são definidas (WOHLIN *et al.*, 2012):

Hipótese Nula

A hipótese nula é usada para testar as diferenças entre as observações, se elas são coincidentes e não resultantes dos padrões experimentais usados. Essa é a hipótese que o pesquisador quer rejeitar com maior nível de significância possível. Uma possível hipótese para o domínio de teste de software pode ser que uma técnica de teste A encontra em média o mesmo número de falhas que outra técnica B. Essa hipótese é definida como:

Hipótese nula 1: *A efetividade da técnica de teste A para encontrar falhas é a mesma da técnica de teste B.*

Hipótese alternativa

A hipótese alternativa representa a afirmação em favor da qual a hipótese nula é rejeitada. Um exemplo de hipótese é que uma técnica de teste A encontra mais falhas que outra técnica B. Essa hipótese pode ser definida como:

Hipótese alternativa 1.1: *A efetividade de encontrar falhas da técnica de teste A é maior que a da técnica de teste B.*

Para o objetivo estabelecido na definição, a hipótese nula, e sua correspondente hipótese alternativa, precisam ser definidas pelo pesquisador para extrair conclusões sobre o estudo.

3.4.2.3 Seleção de variáveis

Durante a seleção de variáveis são definidas as influências e efeitos que serão examinados no experimento. Para definir o *design* do experimento é necessário escolher dois tipos de variáveis: variáveis dependentes (variáveis de resposta) e variáveis independentes (variáveis preditoras).

Variáveis independentes

Variáveis independentes são fatores que são frequentemente variados no experimento para medir seu efeito nas variáveis dependentes. Variáveis independentes podem incluir tratamentos, materiais e algum fator relacionado com o contexto.

No contexto de avaliação das técnicas de teste de software concorrente, as variáveis independentes estão relacionadas ao objeto de estudo, que pode ser a abordagem de teste proposta. Nesse contexto, os possíveis valores de tratamentos podem ser categorizados de acordo com as técnicas apresentadas na Tabela 3.

Outras variáveis independentes no contexto de teste de software concorrente podem estar relacionadas às características dos sujeitos e objetos (software concorrente). Por exemplo, a complexidade do programa (número de *threads*/processos, sincronizações e primitivas de comunicação, conjunto de *interleavings* de eventos, tamanho do *SUT*).

Outro fator a ser considerado como variável independente é a origem da erro, que pode ser gerado aleatoriamente, ou com base em uma taxonomia de erros, ou usando uma técnica baseada em erros (como mutação por exemplo), ou apenas usando defeitos pré-existentes no software. Devido ao não determinismo de aplicações concorrentes, outros fatores que precisam ser considerados são: o número de vezes que o teste é executado e o limite de tempo para a execução do teste (considerando o problema da explosão do espaço de estados no qual o domínio de entrada aumenta exponencialmente e a cobertura de todos os domínios de entrada torna-se impraticável).

Tabela 3 – Classificação de técnicas de teste de software concorrente.

Técnica	Descrição
Teste estrutural	Utiliza o conhecimento da estrutura do código fonte para derivar entradas para o teste. Algumas abordagens incluem a cobertura de nós, arestas, e caminhos e o uso de grafos de fluxo de dados e controle
Teste baseado em modelos	Utiliza um modelo abstrato para especificar o comportamento do programa em teste e projetar e executar artefatos durante a execução do teste
Teste baseado em métodos formais	Utilizam técnicas matemáticas para especificar, desenvolver e verificar sistemas de hardware e software
Teste baseado em erros	Utiliza o conhecimento sobre erros comuns cometidos pelos programadores para derivar casos de teste. O conceito básico do teste baseado em erros é selecionar casos de teste que distinguem o programa em teste de programas alternativos que contém erros hipotéticos

Variáveis dependentes

A variável dependente é a característica avaliada para medir a influência dos fatores nos tratamentos do experimento.

No teste de software concorrente, as variáveis dependentes mais comuns são aquelas usadas para medir o quão boa é a técnica de teste para descobrir erros e quanto recurso computacional e de tempo ela consome. Na Tabela 4 são descritas algumas dessas variáveis e suas possíveis métricas. Essas variáveis dependentes são baseadas nos estudos apresentados nesta revisão e a descrição é baseada nas definições do autor sobre os estudos e o padrão de medida de usabilidade da ISO 9242-11.

Tabela 4 – Variáveis dependentes pra avaliação de técnicas de teste de software concorrente.

Variáveis dependentes	Descrição	Métricas (possíveis valores)
Efetividade	Habilidade de cada técnica em detectar falhas	Defeitos detectados, sequencias executadas, número de mutantes mortos, número de preempções, número de estados visitados, número de <i>interleavings</i> testados, número de execuções exploradas, cobertura alcançada, número de eventos e fluxo de dados explorados
Eficiência	Quão rapidamente a técnica é capaz de encontrar falhas	Tempo de computação, <i>overhead</i> , velocidade, número de caminhos explorados, tempo de resposta, tempo requerido para predição
Usabilidade	Satisfação do usuário	Análise da qualidade por meio de questionários
Custo	Quanto esforço é necessário para aplicar a técnica	Número de casos de teste para cobrir um critério, tempo para executar escalonamentos, número de <i>interleavings</i> explorados, número de execuções do programa.
Desempenho	Recursos que são gastos para executar a técnica	Tempo de execução, <i>overhead</i> de memória, <i>overhead</i> de espaço em disco, velocidade de execução, tempo de compilação, uso de memória, tempo de <i>overhead</i> , tempo de resposta
Acurácia	Número de falsos positivos/falsos negativos gerados	Número de falsos positivos, número de falsos negativos, número de elementos não executáveis, número de mutantes não equivalentes, número de caminhos/ <i>interleavings</i> não executáveis
Relevância	Relevância dos dados de teste gerados	Porcentagem de entradas relevantes do número total de entradas
Escalabilidade	Se a técnica funciona bem em um software e hardware mais complexo	Relação entre o número de <i>threads</i> e o <i>overhead</i> do tempo de execução, número de <i>threads</i> usadas

3.4.2.4 Seleção dos sujeitos

Os sujeitos são responsáveis pela execução do experimento. Idealmente, os sujeitos do experimento seriam uma amostra aleatória da população de profissionais de informática. No entanto, alcançar uma amostra aleatória é difícil. Em vez disso, os grupos de sujeitos são comumente uma amostra acidental dessa população, o que significa que eles são selecionados com base em sua participação em cursos universitários ou industriais (LOTT; ROMBACH, 1996). No contexto de teste de software concorrente, a maioria dos experimentos são desenvolvidos pelos pesquisadores que propõem uma técnica de teste e, em alguns casos, os alunos são utilizados como sujeitos experimentais.

3.4.2.5 Escolha do tipo de design

Depois de definir as variáveis de entrada é necessário definir o *design* (arranjo) experimental. O tipo de projeto de experiência, dependendo do objetivo da experiência, o número de fatores, as alternativas dos fatores, o número de variações indesejadas, etc (JURISTO; MORENO, 2010).

Com base na randomização, diferentes tratamentos são aplicados em diferentes objetos, mantendo outras variáveis constantes e medindo os efeitos nas variáveis dependentes. Em experimentos orientados a humanos, os sujeitos (humanos) aplicam diferentes tratamentos (técnicas) aos objetos (programas), enquanto em experimentos orientados à tecnologia, diferentes tratamentos (técnicas) são aplicados em objetos diferentes (WOHLIN *et al.*, 2012).

Os experimentos no contexto de teste de software concorrente apresentados no mapeamento sistemático (Apêndice B) são todos experimentos orientados à tecnologia, mas os pesquisadores não descrevem o tipo de *design* utilizado para realizar a avaliação experimental. Eles apenas descrevem o procedimento para testar a aplicação e não discutem como os tratamentos foram atribuídos aos objetos. Esse fator dificulta a replicação do experimento e pode trazer fontes de variação indesejadas para o estudo.

A Tabela 5 apresenta um resumo do tipo comum de *design* utilizado em experimentos de engenharia de software e pode ser útil para orientar o pesquisador a escolher o *design* apropriado para o experimento a partir de seus objetivos e variáveis instanciadas. Informações mais detalhadas sobre tipos e definições de *design* de experimentos podem ser encontradas em Wohlin *et al.* (2012) e Juristo e Moreno (2010).

Tabela 5 – Tipos de padrões de *design* (WOHLIN *et al.*, 2012).

Tipo de <i>design</i>	Condições do experimento	Aplicação
Um fator com dois tratamentos	<i>Design</i> completamente randomizado	Compara dois tratamentos e cada sujeito usa somente um tratamento em cada objeto
	Comparação pareada	Compara pares do experimento e cada sujeito usa ambos os tratamentos no mesmo objeto
Um fator com mais de dois tratamentos	<i>Design</i> bloqueado completamente randomizado	Sujeitos formam um ou mais unidades (blocos) homogêneos, no qual cada sujeito usa um objeto para todos os tratamentos e a ordem em que o sujeito usa o tratamento é definida randomicamente
	<i>Design</i> completamente randomizado	Usa um objeto para todos os tratamentos e os sujeitos são atribuídos randomicamente aos tratamentos
Dois fatores	2*2 <i>Design</i> fatorial	Sujeitos são randomicamente atribuídos a cada combinação de tratamentos
	<i>Design</i> aninhado de dois estágios	Têm dois fatores, cada qual com dois ou mais tratamentos. O <i>design</i> do experimento e análise são os mesmos de um <i>design</i> fatorial 2*2
Mais que dois fatores	2 ^k <i>Design</i> fatorial	Os sujeitos são atribuídos randomicamente para as combinações 2 ^k dos tratamentos
	2 ^k <i>Design</i> fatorial fraccionado	Executa somente uma fração do experimento completamente fatorial

3.4.2.6 Instrumentação

Durante a fase de instrumentação, são preparados todos os instrumentos usados na aplicação prática do experimento. Esses instrumentos podem ser de três tipos:

- Objetos do experimento: sob o qual o experimento é aplicado (software concorrente).
- Guias: recursos necessários para guiar os participantes do experimento (descrição do experimento, ferramenta usada pra guiar o experimento e *checklists*).
- Medidas: A medição no experimento é conduzida via coleta de dados. No experimento de teste, dados são gerados e coletados usando ferramentas automatizadas ou manualmente pelo testador.

O objetivo geral da instrumentação é fornecer meios para realizar o experimento e monitorá-lo, sem afetar o controle do experimento. Os resultados da experiência devem ser os mesmos independentemente da forma como o experimento é instrumentado. Se a instrumentação afeta o resultado do experimento, o estudo apresenta ameaças à sua validade (WOHLIN *et al.*, 2012).

3.4.2.7 Avaliação da validade

A validade de um experimento está relacionada à confiabilidade dos resultados obtidos. Durante a fase de planejamento de um experimento, as ameaças à validade devem ser identificadas para avaliar seu impacto sobre os dados. Todas as ameaças que possam ter um impacto na validade dos resultados precisam ser discutidas.

Validade de constructo/construção

Refere-se ao grau em que a manipulação das métricas em um estudo realmente representa as construções no mundo real.

A validade de construção em teste de software concorrente podem estar relacionadas à expectativa do pesquisador que pode afetar os resultados. Outras ameaças estão relacionadas ao uso incorreto de testes estatísticos. A configuração do tipo de falha, a escolha do conjunto de teste e a geração aleatória de dados de teste, precisam ser controladas e representativas do domínio de entrada para não se transformar em uma ameaça ao estudo.

Validade interna

Refere-se ao fato do tratamento ou as variáveis independentes serem realmente responsáveis pelos efeitos observados na variável dependente. Fatores desconhecidos podem ter influenciado os resultados e, portanto, colocar limitações na validade interna do estudo.

Algumas ameaças à validade no teste de software concorrente podem ser: a configuração incorreta da ferramenta usada que pode levar a conclusões incorretas; os defeitos injetados nos programas de exemplo podem ter sido inseridos com base no conhecimento e na compreensão do pesquisador sobre a técnica usada para manipular os resultados; o contexto de teste e o processo de instrumentação podem ser manipulados pelo experimentador para favorecer técnicas de teste gerando resultados não realistas.

Validade externa

Refere-se ao nível em que os resultados do estudo podem ser generalizados para outras populações ou configurações de participantes.

No teste de software concorrente, os autores relatam como principais ameaças à validade externa a representatividade dos programas utilizados nos experimentos (complexidade e granularidade da concorrência) e representatividade dos defeitos usados (usando taxonomias, erros pré-existentes, inserção de erros) e representatividade dos conjuntos de teste usados (tamanho e complexidade). Outros fatores como o tamanho das execuções de sequências de teste (devido ao comportamento não determinístico), o contexto em que o experimento é conduzido e o processo de randomização podem afetar os resultados.

Validade da conclusão

Refere-se ao fato das conclusões alcançadas em um estudo estarem ou não corretas. Para experimentos controlados, a validade da conclusão pode estar relacionada à aplicação de testes estatísticos aos dados.

No contexto de teste de software concorrente, as ameaças mais comuns à validade de conclusão estão relacionadas à corretude do método, técnica e ferramenta usados para realizar a atividade de teste. Outra ameaça comum está relacionada à seleção incorreta das métricas usadas para medir as variáveis dependentes, além do método usado para inserir defeitos nos objetos. A última ameaça está relacionada ao uso incorreto de testes estatísticos durante a avaliação de hipóteses.

A Tabela 6 resume as ameaças mais comuns à validade encontradas na literatura para testes de software concorrente. Na tabela, as ameaças à validade são classificadas por tipo e para cada tipo são apresentados exemplos de possíveis valores extraídos da literatura (Apêndice B).

Tabela 6 – Ameaças à validade no contexto de teste de software concorrente.

Validade da conclusão	Validade interna
O método usado não é correto	Configuração das ferramentas usadas
Medidas não são corretas	Seleção dos defeitos
A implementação das técnicas não é correta	Configuração do contexto
Processo de inserção de falhas não está correto	Processo de instrumentação
Suíte de teste não é representativa	
Análise estatística não é correta	
Validade de construção	Validade externa
Expectativa dos pesquisadores afetam os resultados	Representatividade do objeto do estudo
Configuração das falhas afetam os resultados	Representatividade das falhas semeadas
Escolha do <i>design</i> dos casos de teste	Representatividade das suítes de teste usadas
Heterogeneidade aleatória dos casos de teste	Tamanho da sequência de teste
Métrica de medida incorreta	Diferentes comportamentos
	Processo de randomização usado
	Configuração da ferramenta usada
	Generalização para outro tipo de software

O *design* de experimentos definido para a área de teste de software concorrente é sumarizado na Tabela 7. Ele auxilia nas etapas de definição e planejamento do experimento oferecendo objetivos concretos, planos e procedimentos reutilizáveis para definir e aplicar estudos experimentais na área de teste de software concorrente, com métricas específicas para este contexto. Sendo assim, o *design* definido orienta durante toda a fase de planejamento do experimento, porém as fases de operação, análise e interpretação e apresentação e empacotamento não são incluídas, pois o foco é a definição de maneira padronizada de experimentos, visando facilitar futuras comparações e não o processo de execução e análise que pode variar dado o contexto do estudo.

Como foi visto, o objetivo do *design* é fornecer métricas comuns para condução de estudos sob o mesmo escopo, que podem ser úteis para guiar a condução de novos estudos na área, de modo que facilite a replicação de estudos e que as evidências criadas sejam mais fáceis de comparar e, portanto, efetivamente agregadas em estudos secundários.

Convém observar que esse *design* de experimentos é uma estrutura preliminar para a condução de estudos experimentais na área. Porém, pressupõem-se que é generalizável para diferentes técnicas de teste de software concorrentes. No projeto de doutorado, esse *design* pode atuar como um guia para a evolução do corpo de conhecimento apresentado na Seção 3.3 possibilitando construir de forma progressiva e interativa um conjunto de conhecimento que traga informações quantitativas sobre a utilidade de técnicas de teste em diferentes ambientes.

Na próxima seção é apresentado o esquema de caracterização para seleção de técnicas de teste de software concorrente. Tanto o *design* de experimentos quanto o esquema de caracterização foram definidos com base nos resultados obtidos com o mapeamento sistemático apresentado no Apêndice B) e na análise das informações presentes no corpo de conhecimento apresentado na Seção 3.3.

Tabela 7 – *design* de experimentos para teste de software concorrente.

<p>Definição do Experimento</p> <ol style="list-style-type: none"> Objetivo do Experimento Modelo GQM: Analisar <i>técnicas de teste de software concorrente</i> para avaliar <i>efetividade</i> em um <i>SUT</i> do ponto de vista de <i>obter conhecimento</i> no contexto de <i>(uma variável particular de um conjunto de variáveis de contexto)</i> <ol style="list-style-type: none"> Objeto (técnica de teste de software concorrente, como na taxonomia definida na Tabela 3) Propósito (avaliar, comparar, analisar, medir, etc) Fator de qualidade (efetividade, eficiência, eficácia, usabilidade, custo, desempenho, acurácia, relevância, escalabilidade, etc) Ponto de vista (pesquisador, profissional de teste, estudantes, etc) Contexto (sujeitos, objetos, ambiente) <p>Planejamento do Experimento</p> <ol style="list-style-type: none"> Seleção de Contexto <ol style="list-style-type: none"> Sujeitos (participantes, pesquisadores, profissionais de teste, professores, estudantes, etc) Objetos (sistema em teste - SUT) Definição das Hipóteses <ol style="list-style-type: none"> Hipótese nula (A técnica de teste A tem a mesma efetividade da técnica de teste B) Hipótese alternativa (A técnica A é mais efetiva que a técnica B) Seleção de Variáveis <ol style="list-style-type: none"> Variáveis Dependentes (número de erros detectados, tempo de execução, cobertura alcançada, número de mutantes mortos, <i>overhead</i> de memória, número de falsos positivos, número de falsos negativos, número de <i>threads</i> usadas, etc.) Variáveis Independes (abordagem de teste, erros inseridos, complexidade da aplicação concorrente, número de execuções, etc.) Seleção de Sujeitos <ol style="list-style-type: none"> Seleção randômica de uma amostra da população de profissionais ou pesquisadores em ciência da computação Escolha do <i>Design</i> <p>Definido como os tipos padrões de <i>design</i> apresentados em Wohlin et al. (2012)</p> <ol style="list-style-type: none"> Um fator com dois tratamentos Um fator com mais de dois tratamentos Dois fatores com dois tratamentos Mais que dois fatores cada um com dois tratamentos Instrumentação <ol style="list-style-type: none"> Objetos (define qual objeto usar no experimento) Instrumentação (treinamento dos participantes, material de suporte, etc.) Conjunto de dados (questionários, saídas das ferramentas, etc.) Avaliação da Validade <ol style="list-style-type: none"> Validade interna (Define quais variáveis serão manipuladas e os seus efeitos nos resultados, por exemplo, enganos cometidos na configuração das ferramentas, seleção dos defeitos, configuração do contexto ou processo de instrumentação) Validade externa (Conclusões obtidas com a amostra do experimento e são generalizadas para a população de interesse, por exemplo, uso de objetos não representativos, erros inseridos ou dados de teste insuficientes) Validade de construção (Transforma as construções de causa-efeito em teoria e variáveis a serem observadas no experimento, por exemplo, os resultados podem ser afetados pelas expectativas do pesquisador e conhecimento durante a inserção dos erros, escolha do <i>design</i> de casos de teste e uso das métricas avaliadas) Validade de conclusão (Relacionada com o teste estatístico usado para gerar as conclusões, por exemplo, análise estatística incorreta) <ul style="list-style-type: none"> Todos os passos descritos acima fornecem o <i>design</i> de experimento, como definição e planejamento do experimento, tornando o experimento pronto para executar. Entretanto, é recomendado que o condutor do experimento revise o <i>design</i> com outros especialistas e execute um experimento piloto para fazer os ajustes necessários e validar o <i>design</i>.
--

3.5 Esquema de Caracterização e Seleção de Técnicas de Teste de Software Concorrente

A identificação das características para a seleção adequada das tecnologias de teste é um desafio para a área de teste de software (BASILI, 1985; VEGAS; BASILI, 2005; NETO; TRAVASSOS, 2009). Assim como na programação sequencial, as características do software concorrente criam obstáculos ao uso de metodologias para a seleção de tecnologias nesse contexto. Para evitar essa ameaça, com base nos dados contidos no corpo de conhecimento descrito na Seção 3.3, foram identificadas as características primárias do software concorrente que afetam a escolha da técnica de teste e um esquema de caracterização foi construído.

3.5.1 Atributos de caracterização

Na área de teste de software, a definição das técnicas mais adequadas para avaliar as características de um sistema específico permanece como uma questão em aberto (VEGAS; BASILI, 2005). Embora estudos secundários na área de teste concorrente existam (ARORA *et al.*, 2015; QAWY *et al.*, 2012; SOUZA *et al.*, 2011; BRITO *et al.*, 2010; MAMUN; KHANAM, 2009), esses estudos focam somente na categorização das técnicas de teste e não proporcionam informações que possam ser usadas para minimizar a possibilidade de escolha inadequada de uma técnica, o que pode gerar um efeito negativo na qualidade do software.

Nesse sentido, foram selecionados com base na literatura técnica da área, informações relevantes sobre as técnicas, atributos, e conceitos da programação concorrente que afetam o processo de teste. Essas informações são usadas para a construção de um esquema de caracterização para dar suporte a tomada de decisão durante o processo de seleção de técnicas de teste adequadas a um projeto de software. O esquema de caracterização é composto por 21 atributos conforme apresentado na Tabela 8. Esses atributos são utilizados para o cálculo da adequação das técnicas ao projeto de teste e consequentemente a seleção da técnica mais apropriada. Os atributos são classificados em quatro categorias conforme o tipo da informação que oferece, descritas a seguir:

- **Modelo de programação:** o modelo de programação usado pelo software em teste (plataforma de execução, paradigma de desenvolvimento, linguagem de programação);
- **Características gerais do teste:** o tipo de técnica, abordagem para geração de dados de teste, nível do teste, entrada e saída do caso de teste, atributos de avaliação de qualidade, e tipo de estudo empírico usado para validação (VEGAS; BASILI, 2005; NETO; TRAVASSOS, 2009);
- **Ferramenta de apoio ao teste:** identificação da ferramenta, linguagem e plataforma de operação (VEGAS; BASILI, 2005);

Tabela 8 – Atributos para a caracterização de técnicas de teste de software concorrente.

	Atributos de caracterização	Exemplo de valores
Modelo de programação	1. Plataforma em que o software em teste é executado	Linux, Windows, Android
	2. Paradigma de desenvolvimento adotado para a interação de processos/ <i>threads</i>	Memória compartilhada, passagem de mensagem, híbrido
	3. Linguagem de programação/ Bibliotecas de execução em que o software em teste é desenvolvido	Java, C/MPI, C/Pthreads, Fortran, Ada, Erlang
Características gerais do teste	4. Tipo de técnica de teste	Estrutural, funcional, baseado em erros, baseado em modelos
	5. Nível do desenvolvimento em que o técnica de teste é aplicada	Unitário, integração, sistema, aceitação, regressão
	6. Abordagem de geração de dados de teste	Rastreamento de execução, teste de alcançabilidade, escalonamento de <i>threads</i> , localização de pares
	7. Entrada esperada para o caso de teste	Código fonte, requisitos, modelos, <i>bytecode</i>
	8. Saída esperada para o caso de teste	Porcentagem de cobertura, tempo de execução, erros revelados
	9. Características de qualidade que a técnica é capaz de avaliar	Efetividade, eficiência, desempenho, escalabilidade, <i>overhead</i>
	10. Tipo de estudo empírico aplicado para a validação da técnica de teste	Estudo de caso, experimento controlado, <i>survey</i> , etnografia, pesquisa-ação
	11. Tipo de análise usada pela técnica de teste	Estática, dinâmica, híbrida
	12. Mecanismo para forçar a execução de diferentes sequências de execução	Teste de alcançabilidade, inserção de ruído
	13. Tipo de mecanismo de <i>replay</i> utilizado para a re-execução do programa durante o teste	Monitoramento, regressão, gravar-reexecutar, sequências de sincronização
Características do teste concorrente	14. Representação do programa que captura informações relevantes para o teste	Grafo de estados, redes de petri, modelo de predição
	15. Contexto/Objetivo do software concorrente	Computação de alto desempenho, sistemas distribuídos, aplicações móveis, sistemas embarcados, sistemas paralelos
	16. Técnica usada para tratar o problema de explosão de estados	Redução dinâmica de ordem parcial, função de adequação, limitação de preempção
	17. Tipo de erros concorrentes identificados pela técnica de teste	Condição de corrida, <i>deadlock</i> , violação de atomicidade, <i>livelock</i>
	18. Existência de uma ferramenta de suporte ao teste (representada pelo nome da ferramenta)	Contest, CHESS, JavaPathFinder
Ferramenta de apoio	19. Custo associado a ferramenta	Acadêmica, comercial, software livre
	20. Linguagem de programação suportada pela ferramenta	Java, C/MPI, C/Pthreads, C/OpenMP, Fortran, Ada, Erlang
	21. Plataforma de execução que a ferramenta opera	Linux, Windows, Android

- **Características do teste de software concorrente:** descreve os aspectos das tecnologias de teste específicas para programação concorrente, incluindo:

- *Tipo de análise usada pela técnica*, que pode ser estática ou dinâmica. Análise estática verifica o código sem executá-lo. Análise dinâmica verifica o código durante a execução. Ambas consideram a presença de vários tipos de primitivas de sincronização no software concorrente.
- *Paradigma de programação concorrente empregado para a interação entre processos/*threads**, que pode ser memória compartilhada ou passagem de mensagem. Memória compartilhada refere-se aos processos/*threads* que usam o mesmo espaço para o endereçamento de variáveis e realizam a comunicação por meio de variáveis compartilhadas. Passagem de mensagem refere-se ao uso de espaços de endereçamentos distintos onde a comunicação entre processos/*threads* é feita por meio de passagem de mensagens.

- *Tipo de mecanismo de replay*, devido ao não determinismo, múltiplas execuções de um programa concorrente com a mesma entrada podem exercitar diferentes sequências de sincronização e produzir saídas diferentes e corretas. No entanto, cada sequência de sincronização válida deve ser considerada durante o teste. Nesse cenário, mecanismos de *replay* são utilizados para controlar a execução de eventos de sincronização no programa em teste e repetir uma dada sequência de sincronização durante o teste, tornando a execução determinística.
- *Representação do programa*, algumas técnicas requerem representação em alto nível da aplicação, que capture informação relevante para a atividade de teste e represente abstrações como limitações de *buffers*, semáforos, e fila de *threads*. Uma representação gráfica de um programa provê uma visão independente da linguagem de programação e serve como entrada à técnica de teste.
- *Técnica que trata o problema da explosão do espaço de estados*, métodos de redução de espaço de estados enumeram e analisam os estados do sistema que podem ser alcançados para fazer possível a exploração do domínio de entrada. No contexto de programação concorrente, o espaço de *interleaving* de eventos e sequências de sincronização devem ser eficientemente explorados pela viabilidade do teste.
- *Erros concorrentes identificados pela técnica de teste*, aplicações concorrentes incluem erros que não existem em aplicações sequencias, como, *deadlock*, *livelock*, *starvation*, instanciação incorreta de *threads*, violação de atomicidade, código desprotegido, inconsistência de dados, erros de sincronização, erros de decomposição, uso incorreto de funções de passagem de mensagem, perda de sinais e erros de comunicação, que devem ser identificados durante a atividade de teste.

Esse esquema foi usado para extrair informações relevantes dos estudos primários identificados na literatura técnica. A análise dessas informações juntamente com a identificação das características do projeto de software que influenciam o processo de seleção de técnicas são usados como base para a construção do corpo de evidência para a área de teste de software concorrente descrito na Seção 3.3.

3.5.2 Definição dos critérios de adequação e avaliação do esquema

Após a definição dos atributos que irão compor o esquema de caracterização das técnicas, foi conduzido um *survey* a fim de avaliar qual a relevância desses atributos para a caracterização das técnicas e qual o seu grau de adequação ao projeto de teste. O *survey* foi conduzido com especialistas que tem trabalhado ativamente na área, tanto em pesquisa na academia como em projetos de software concorrente na indústria, a fim de definir pesos para quantificar a importância desses atributos. As etapas da condução do *survey* são apresentadas a seguir.

3.5.2.1 Planejamento e objetivo

Dentre os principais objetivos desse *survey*, estão: 1) Analisar se os atributos definidos no esquema de caracterização são adequados para caracterizar técnicas de teste para software concorrente e 2) Determinar o nível de relevância de cada atributo para o processo de seleção de técnicas de teste para um projeto de software. Os atributos foram avaliados do ponto de vista dos pesquisadores e profissionais que selecionam tecnologias de teste para um projeto de software específico.

3.5.2.2 Questões de pesquisa

Tendo como base os objetivos definidos para o estudo, a Tabela 9 lista as questões de pesquisa que guiam o estudo.

Tabela 9 – Questões de pesquisa do estudo.

RQ1	Os atributos definidos são adequados para caracterizar técnicas de teste de software concorrente?
RQ2	Existem atributos faltantes (atributos adicionais e relevantes (não apresentados na Tabela 8), que deveriam ser adicionados ao conjunto inicial de atributos)?
RQ4	Os atributos definidos são relevantes para o processo de seleção de técnicas de teste?

3.5.2.3 Instrumentação e execução

Foi desenvolvido um *survey online*² para coletar as respostas dos participantes sobre a avaliação do esquema. Esse *survey* é composto de três partes. A primeira parte coleta as informações sobre a experiência dos participantes, necessárias para o estabelecer o peso de resposta de cada participante. A segunda parte coleta os valores de adequação de cada atributo, como "Sim" e "Não". A terceira parte coleta a relevância de cada atributo, os valores possíveis são: (1) Sem relevância, (2) Relevância muito baixa, (3) Baixa relevância, (4) Média relevância, (5) Alta relevância e (6) Relevância muito alta.

Por meio do mapeamento sistemático (Apêndice B) foi possível identificar 109 estudos relacionados ao teste de software concorrente. A partir desses estudos, foram identificados 200 autores e extraídos seus endereços de e-mail. Foram enviados e-mails para esses 200 participantes em potencial com o link do *survey* online. O *survey* ficou disponível de 06 de junho a 06 de julho de 2017. Dos 200 participantes em potencial, somente 145 tinham e-mail válido (conforme apresentado no artigo). Desses 145, um total de 27 autores completaram o *survey* até o final. A Tabela 10 descreve os participantes que responderam ao *survey*.

3.5.2.4 Análise dos resultados

Para análise da adequação e relevância de cada atributo, foi atribuído a cada sujeito um peso com base no seu conhecimento e experiência. A Fórmula I define o peso, no qual, para

² <<https://goo.gl/forms/v5Knp3csrdCuZYxE2>>

Tabela 10 – Caracterização dos sujeitos do estudo.

Id	Formação Acadêmica	Número de Publicações	Nível de Experiência	Projetos na Indústria	Peso do Participante
1	Doutorado	1 a 5	Baixo	3	4.5
2	Doutorado	1 a 5	Baixo	2	4
3	Doutorado	1 a 5	Baixo	2	4
4	Doutorado	Mais de 20	Médio	2	8
5	Doutorado	1 a 5	Baixo	1	3.5
6	Doutorado	Mais de 20	Baixo	1	6.5
7	Doutorado	Mais de 20	Alto	4	10
8	Doutorado	6 a 10	Alto	1	6.5
9	Doutorado	1 a 5	Baixo	1	3.5
10	Doutorado	1 a 5	Médio	2	5
11	Doutorado	6 a 10	Alto	5	8.5
12	Mestrado	6 a 10	Baixo	2	4
13	Mestrado	1 a 5	Baixo	0	2
14	Doutorado	1 a 5	Baixo	2	4
15	Doutorado	11 a 20	Alto	10	12
16	Doutorado	1 a 5	Baixo	0	3
17	Doutorado	11 a 20	Baixo	2	6
18	Doutorado	11 a 20	Alto	10	12
19	Mestrado	1 a 5	Baixo	2	3
20	Doutorado	Mais de 20	Muito Alto	50	32
21	Doutorado	6 a 10	Médio	2	6
22	Doutorado	6 a 10	Baixo	5	6.5
23	Doutorado	Mais de 20	Médio	8	11
24	Doutorado	1 a 5	Baixo	1	3.5
25	Doutorado	Mais de 20	Baixo	10	11
26	Doutorado	11 a 20	Baixo	1	5.5
27	Doutorado	1 a 5	Alto	10	10

cada participante/sujeito (i), (F) é o mais alto grau acadêmico, (P) é a quantidade de artigos publicados na área, (E) é o nível de experiência na indústria (CTT) é o número de projetos que envolviam técnicas de teste de software concorrente na indústria que ele participou (NETO; TRAVASSOS, 2008). Essas informações e os pesos resultantes para cada sujeito são apresentados na Tabela 10.

$$(I) \text{Peso}(i) = F(i) + P(i) + E(i) + \frac{CTT(i)}{\text{Mediana}(\sum_{i=1}^T CTT(i))}$$

A Fórmula II calcula o nível de adequação (RQ1) dos atributos definidos no esquema de caracterização (Tabela 8). Para cada atributo (a), é calculada a soma para a opinião de cada participante sobre a adequação do atributo ($resposta(i,a)$) multiplicado pelo peso do participantes ($peso(i)$). Este valor varia de 0 a 100%.

$$(II) \text{Adequacao}(a) = \sum_{i=1}^T (resposta(i,a) * peso(i))$$

A Tabela 11 mostra a importância de cada atributo para caracterizar a técnica (adequação). Foi utilizado o teste de hipóteses Teste-t (CASELLA; BERGER, 2002) que gerou como resultado o $P\text{-value}=0.0000008$, que é menor que o nível de significância $\alpha=0.05$. Esses valores sugerem que a hipótese nula $NH01$ pode ser rejeitada em favor da hipótese alternativa $AH1$, considerando um limite de 50% para a eliminação de um atributo. Com base nesses dados, é possível concluir que pelo menos um atributo (tipo de estudo empírico) não é adequado para a caracterização das técnicas de teste de software concorrente segundo a opinião dos participantes do estudo.

Tabela 11 – Avaliação dos atributos de caracterização.

Categoria	Atributo	Adequação	Relevância
Modelo de programação	Plataforma de execução	88,75%	69,67%
	Paradigma de interação entre processos/threads	63,43%	61,07%
Características gerais do teste	Linguagem de programação	67,52%	58,57%
	Técnica de teste	95,65%	76,21%
	Nível do teste	94,37%	70,54
	Contexto/objetivo do software	89,00%	69,16%
	Entradas esperadas	76,21%	65,98%
	Saídas requeridas	81,33%	69,05%
	Atributo de qualidade avaliado	93,86%	79,85%
	Tipo de estudo empírico	34,02%	37,08%
Características do teste concorrente	Tipo de análise	100,00%	66,29%
	Mecanismo de geração de interleavings de sincronização	79,54%	71,05%
	Mecanismo de replay	67,77%	61,69%
	Representação do programa	64,71%	53,81%
	Mecanismo de geração de dados de teste	69,05%	54,27%
	Redução do espaço de estados	67,52%	60,51%
	Erros concorrentes revelados	100,00%	79,95%
	Existência de ferramenta de apoio	-	-
Ferramenta de apoio ao teste	Custo da ferramenta	78,52%	68,80%
	Linguagem de programação suportada pela ferramenta	-	-
	Plataforma que a ferramenta opera	-	-

A Fórmula III calcula a relevância (RQ3) para cada atributo (a) pela soma da pontuação dos níveis de relevância por atribuído para cada participante do estudo ($nivel(a)$) multiplicado pelo peso do participante ($peso(i)$).

$$(III) Relevancia(a) = \sum_{i=1}^T (nivel(i, a) * peso(i))$$

Os resultados do Teste-t para a análise da relevância mostraram um $P-value=0.000009$, que é menor que o intervalo de confiança definido $\alpha=0.05$, indicando a rejeição da hipótese nula $NH02$ em favor da hipótese alternativa $AH2$. Esses resultados sugerem que ao menos um dos atributos definidos no conjunto inicial não é relevante para o processo de seleção de técnicas de teste pois está abaixo do limite definido de 50%.

A Figura 14 mostra, para cada atributo de caracterização, o grau de adequação e o nível de relevância para o processo de seleção em um projeto de software.

Para avaliar se um atributo é adequado para caracterizar uma técnica de teste de software, um limite é estabelecido, valores acima do limite indicam que o atributo é adequado e deve ser incluído no esquema, valores abaixo do limite indicam que o atributo não é adequado e pode ser retirado do esquema. O limite estabelecido nesse estudo é de 50% do valor máximo que pode ser obtido para um atributo se todos os participantes responderem “Sim” a respeito da importância desse atributo para caracterizar uma técnica de teste concorrente (NETO; TRAVASSOS, 2008).

A Fórmula IV calcula o limite para a classificação dos atributos como adequados ou não adequados ao esquema de caracterização. O limite é calculado pela soma do peso atribuído ao participante do estudo ($peso(i)$) multiplicando por 0,5 (que equivale a 50%).

$$(IV) Limite = 0,5 * \sum_{i=1}^M peso(i)$$

Considerando os resultados do *survey*, somente um atributo está abaixo do limite. Então é possível concluir que todos os demais atributos são importantes para o esquema de caracterização. Como apresentado na figura, apenas o atributo tipo de estudo empírico, está abaixo do limite, sendo assim é considerado irrelevante e pode ser removido do esquema de caracterização.

Figura 14 – Porcentagem de adequação dos atributos de caracterização.

A Tabela 12 apresenta o resultado final do cálculo do peso de cada atributo considerando os valores obtidos com o cálculo da relevância dos atributos relacionados à opinião dos participantes do estudo. Ao atributo “tipo de estudo empírico” foi atribuído peso zero, pois os resultados do *survey* indicaram que este atributo deveria ser removido do esquema. Para os atributos relacionados à ferramenta de apoio ao teste, foi considerado o custo da ferramenta que já indica a existência da ferramenta, nesta versão da ferramenta SeleCTT não foram considerados os atributos de tipo de linguagem e plataforma em que a ferramenta opera, pois a inserção desses atributos no cálculo demanda uma alteração no algoritmo de seleção a fim de tratar hierarquia de critérios, esse fator deve ser estudado no desenvolvimento de novas versões da ferramenta.

Em relação à questão de pesquisa (RQ2), os participantes foram questionados sobre a necessidade de inclusão de novos atributos ao esquema de caracterização. Dentre os atributos listados e que deveriam ser incluídos ao esquema, estão: 1) a portabilidade da aplicação; 2) informações sobre o conhecimento e treinamento dos profissionais de teste; 3) reprodutibilidade dos erros; 4) esforço para aplicação da técnica (testadores/hora); 5) requisitos das técnicas; 6) facilidade de uso; 7) custo versus benefícios; 8) modelo de riscos; 9) erros relacionados à segurança; e 10) acurácia da ferramenta (falsos positivos e falsos negativos gerados). A inclusão desses atributos ao esquema de caracterização envolve a condução de novos estudos empíricos e poderão ser estudados como trabalhos futuros deste projeto de doutorado.

Tabela 12 – Pesos dos atributos de caracterização.

Categoria	Atributo	Peso
Modelo de programação	Plataforma de execução	6,15%
	Paradigma de interação entre processos/ <i>threads</i>	5,36%
	Linguagem de programação	5,18%
Características gerais do teste	Técnica de teste	6,68%
	Nível do teste	6,24%
	Contexto/objetivo do software	6,06%
	Entradas esperadas	5,80%
	Saídas requeridas	6,06%
	Atributo de qualidade avaliado	7,03%
	Tipo de estudo empírico	0,00%
	Tipo de análise	5,80%
Características do teste concorrente	Mecanismo de geração de <i>interleavings</i> de sincronização	6,24%
	Mecanismo de <i>replay</i>	5,45%
	Representação do programa	4,75%
	Mecanismo de geração de dados de teste	4,75%
	Redução do espaço de estados	5,36%
	Erros concorrentes revelados	7,03%
	Existência de ferramenta de apoio	-
	Custo da ferramenta	6,06%
Ferramenta de apoio ao teste	Linguagem de programação suportada pela ferramenta	-
	Plataforma que a ferramenta opera	-

3.5.2.5 Ameaças à validade do estudo

As possíveis ameaças à validade desse estudo podem ser divididas em:

- **Validade interna:** está relacionada ao fato dos resultados serem baseados na avaliação subjetiva dos participantes do experimento. Os especialistas foram questionados a indicar sua percepção sobre o esquema de caracterização. O esquema de caracterização não foi aplicado em um projeto real (em ambiente industrial). Portanto, é possível que os participantes tenham respondido de forma a não representar o verdadeiro estado dos projetos que são desenvolvidos na indústria.
- **Validade de construção:** está relacionado a maneira como os participantes interpretaram cada atributo listado no *survey*. Para reduzir essa ameaça, foi disponibilizado um glossário com a definição detalhada de cada atributo no *survey*. Porém, ainda assim é possível que os participantes não tivessem o mesmo conceito em mente que era esperado pelos condutores do estudo. No entanto, não há evidências da presença dessa ameaça nos resultados do estudo, se ela ocorreu, ela poderia reduzir a validade dos dados.
- **Validade externa:** está relacionada ao fato da amostra poder não ser representativa da população indicada para o *survey*. Para assegurar o nível adequado de conhecimento, os participantes foram selecionados do conjunto de autores que publicaram artigos científicos sobre técnicas de teste de software concorrente. Contudo, é possível que esse grupo de participantes não seja representativo da população como um todo.

3.5.2.6 Discussão

O esquema foi validado por meio de um *survey* com especialistas na área. O estudo avaliou o esquema de caracterização proposto para seleção de técnicas de teste de software concorrente. Com base no mapeamento sistemático, foram definidos um conjunto atributos para auxiliar os pesquisadores e profissionais da área a escolher a tecnologia de teste mais apropriada para sua situação em particular. O nível de relevância estabelecido para cada atributo definido no esquema de caracterização foi usado para o desenvolvimento de um mecanismo de recomendação que indica a técnica de teste mais apropriada para um projeto de software concorrente, por meio da comparação entre as características do projeto e da técnica de teste calculada por meio do nível de relevância de cada atributo.

Esse esquema define a estrutura do corpo de conhecimento apresentado na Seção 3.3 e é a base para a implementação da ferramenta de apoio (Seção 3.6) que automatiza o processo de seleção utilizando os atributos e pesos definidos nesse esquema.

3.6 SeleCTT: Uma Infraestrutura computacional de apoio à seleção de técnicas teste de software concorrente

Subordinado a este projeto de doutorado foi desenvolvido um projeto de conclusão de curso (MOURA *et al.*, 2017) com o intuito de automatizar a aplicação da metodologia de seleção apresentada na Seção 3.5. A ferramenta web desenvolvida implementa o esquema de caracterização proposto, considerando os atributos de caracterização e os pesos definidos no *survey* com especialistas na área (MELO *et al.*, 2017). A ferramenta foi desenvolvida utilizando PHP, MySQL, JavaScript, CSS3 e HTML e está disponível online no endereço: <<http://www.labes.icmc.usp.br/~selectt/home>>.

Essa seção descreve a ferramenta, seus principais módulos e a maneira como eles interagem a fim de gerar uma lista ordenada de técnicas adequadas a um projeto de software especificado.

3.6.1 Definição da ferramenta

A ferramenta de suporte desenvolvida é denominada SeleCTT (*An infrastructure for Selection of Concurrent Software Testing Techniques*) e funciona como uma página web com *cross-browser*, que utiliza informações extraídas do corpo de conhecimento (Seção 3.3), calcula e apresenta ao usuário quais são as melhores opções de técnicas de teste, considerando as características do projeto de software que está sendo desenvolvido. O objetivo é oferecer ao pesquisador e profissional de teste uma maneira automática de seleção de técnicas de teste, na forma de uma lista de recomendações que pode ser usada como auxílio à tomada de decisão durante o processo

de desenvolvimento e software, mais especificamente no processo de condução da atividade de teste de software.

Esse sistema oferece um formulário de entrada onde o usuário fornece as informações referentes ao projeto de software que será desenvolvido e com base nessas informações e análise do banco de dados de técnicas de teste (corpo de conhecimento) é retornado pelo sistema uma lista ordenada das técnicas mais adequadas a este projeto de software. Além de utilizar o sistema para consulta de técnicas de teste de software concorrente, o usuário pode também incluir novas técnicas ao corpo de conhecimento, possibilitando assim manter o corpo de conhecimento atualizado e em constante evolução.

3.6.2 Arquitetura da ferramenta

A ferramenta é composta por uma interface web de apresentação das informações ao usuário composta por páginas desenvolvidas em HTML5, CSS3 e JavaScript. A escolha dessas tecnologias teve como objetivo obter uma implementação mais simples, além de ser compatível com a maioria dos *browsers* disponíveis atualmente no mercado. O JavaScript juntamente com requisições em Ajax na API desenvolvida pela ferramenta viabilizaram o desenvolvimento de páginas dinâmicas, onde as solicitações de informações são feitas de maneira assíncrona ao servidor, não prejudicando a experiência do usuário. Para construção da infraestrutura web, foi utilizado o *framework* CodeIgniter, pois ele facilita o desenvolvimento de aplicações PHP, e torna a implementação rápida e eficaz, com uma baixa curva de aprendizado. Esse *framework* segue o padrão *Model-View-Controller (MVC)* (FOWLER, 2002) composto por três camadas distintas: 1) uma camada de visão, responsável pela interação com o usuário; 2) uma camada modelo, contendo as regras do sistema e responsável pela manipulação dos dados; e 3) uma camada de controle, responsável por gerenciar as duas outras camadas. Embora distintas, as três camadas são complementares entre si. A arquitetura da ferramenta é apresentado na Figura 15.

Figura 15 – Arquitetura da ferramenta SeleCTT (MOURA *et al.*, 2017).

O *framework* CodeIgniter foi usado para implementação dos módulos da ferramenta. A camada de visão é implementada usando HTML, CSS e JS oferecendo uma interface iterativa ao usuário. A camada modelo implementa os módulos do sistema em linguagem PHP e trata do gerenciamento do sistema (controle de usuários, inserção de técnicas de teste, busca por técnicas, gerenciamento de conteúdo e exibição de estatísticas de dados). A camada de controle é responsável pela interação entre os módulos da ferramenta. A base de dados oferece aos demais módulos as informações necessárias para condução do processo de seleção e recomendação de técnicas de teste.

3.6.2.1 *Classes de usuários*

O sistema oferece suporte a duas classes distintas de usuários:

- **Usuário padrão:** pesquisadores e profissionais da área, que utilizam a ferramenta para a seleção de técnicas de teste a serem aplicados em seus projetos de software. Esses usuários tem a função de fornecer à ferramenta informações específicas do projeto de software. Outra possível função atribuída a essa classe de usuários é a inserção de uma nova técnica de teste de software ao corpo e conhecimento.
- **Administrador do sistema:** esse tipo de usuário é responsável pela manutenção do sistema. O administrador do sistema é a pessoa interessada em gerenciar o sistema, controlando o cadastramento, alteração e remoção de novos usuários e informações inseridas no corpo de conhecimento. Eles são responsáveis também pelas possíveis manutenções que o sistema possa enfrentar.

3.6.2.2 *Módulos*

A ferramenta SeleCTT tem como objetivo central auxiliar o processo de seleção da técnica mais adequada a um projeto de software específico. A fim de atender essa tarefa, o usuário fornece à ferramenta dados que caracterizam o seu projeto de software. A ferramenta opera coletando esses dados, armazenando em uma lista de atributos e comparando esses dados com as informações das técnicas de teste já armazenados no banco de dados. A classificação das técnicas é feita com base no cálculo da distância euclidiana entre vetores, com base nos pesos estabelecidos a cada um dos atributos analisados. Os resultados são apresentados ao usuário na forma de uma lista ordenada de técnicas, onde é possível visualizar e expandir os resultados para cada uma das técnicas listadas.

Os módulos que compõem a infraestrutura web e são responsáveis pelo seu funcionamento são descritos em detalhes a seguir:

• Módulo de inserção

A Figura 16 ilustra as duas opções de funcionamento desse módulo. A primeira opção, está relacionada à entrada através de um formulário, que contém as informações dos atributos de uma nova técnica de teste de software concorrente para inserção no corpo de conhecimento. No entanto, a inserção desta nova técnica, utilizando esse módulo, aparecerá como pendente, permanecendo no corpo de conhecimento temporário, que necessita da aprovação de um administrador para realmente se consolidar no repositório e ser considerada no cálculo de adequação durante o processo de seleção. A segunda opção, está relacionada com a entrada de informações a respeito do projeto de software a ser desenvolvido através de um formulário. Os resultados das técnicas de teste mais adequadas para determinado projeto estão relacionados com as informações de entrada no módulo de inserção, pois estas informações serão utilizadas no algoritmo de seleção para serem cruzadas com os atributos das técnicas presentes no corpo de conhecimento. Todos os usuários registrados e com o e-mail confirmado tem acesso ao módulo de inserção.

Figura 16 – Módulo de inserção (MOURA *et al.*, 2017).

Se o usuário optar por buscar uma técnica adequada ao seu projeto de software, o módulo fornece ao usuário um formulário (Figura 25) onde são requisitadas como entrada as informações do projeto de software. No caso do usuário optar por inserir uma nova técnica de teste, um outro formulário (Figura 23) é exibido para obtenção das informações. Após a coleta dos dados, os mesmos são armazenados no corpo de conhecimento, para uso posterior durante o cálculo de adequação das técnicas no processo de seleção.

• Módulo de administração

Esse módulo possibilita o gerenciamento do conteúdo do corpo de conhecimento, das páginas que compõem a ferramenta e do acesso ao sistema (inclusão e exclusão de

usuários). Apenas usuários da classe “administrador do sistema” têm acesso a esse módulo. A Figura 17 apresenta o módulo de administração do sistema SeleCTT. Dentre as atividades providas por esse módulo estão: 1) o gerenciamento do corpo de conhecimento: responsável por controlar a inserção de novas técnicas, por meio da remoção das técnicas que não apresentem informações adequadas ao corpo de conhecimento e alteração das pendências nas técnicas para que elas façam parte do corpo de conhecimento, caso a informação inserida seja pertinente, além disso permite a edição de novos atributos, atualização de atributos existentes e informações das técnicas já cadastradas; 2) o gerenciamento dos usuários: responsável por controlar a inserção de novos usuários, a edição de usuários cadastrados, remoção de usuários, aprovação de usuários com verificação de e-mail pendente ou reenvio do e-mail de verificação, também possibilita o gerenciamento de permissões, onde é possível alterar a classe do usuário; e 3) o gerenciamento do conteúdo da ferramenta: responsável pela edição do conteúdo presente nas páginas comuns (página inicial, página de publicações, página de pessoas, etc).

Figura 17 – Módulo de administração.

• Módulo de controle

Esse módulo oferece funções de manipulação dos atributos e critérios utilizados para a caracterização das técnicas de teste.

A Figura 18 ilustra a interface do módulo de controle da ferramenta SeleCTT. Esse módulo gerencia os pesos que entram no cálculo da seleção da(s) técnica(s) de teste mais adequada(s) a um determinado projeto de software concorrente. Os valores dos pesos para cada um dos vinte e um atributos, que compõem parte da informação de uma técnica de teste de software concorrente, são representados na Tabela 12.

Os pesos apresentados para cada atributo nesse módulo são de natureza global na ferramenta. O gerenciamento dos pesos, de cada um dos vinte e um atributos, consiste em atualizar os valores para todas as técnicas presentes no corpo de conhecimento, pois esses

Figura 18 – Módulo de controle.

pesos são a base para o cálculo da adequação das técnicas ao projeto de software. Esses pesos foram derivados do estudo feito pelos autores [Melo et al. \(2017\)](#), por meio do *survey* apresentado anteriormente na Seção 3.5.2.

• Módulo de apresentação dos resultados

Esse módulo é responsável pela comparação das informações do projeto e das técnicas de teste, utilizando critérios de adequação, conforme definido na Seção 3.5, a fim de fornecer uma lista ordenada de técnicas relevantes ao projeto de software.

O Código-fonte 1 apresenta o algoritmo de seleção, que recebe como parâmetro as informações do projeto em desenvolvimento e ocorre a inicialização dos pesos e das técnicas presentes no corpo de conhecimento (linhas 3 a 9). A partir da inicialização, para cada técnica presente (linhas 11 a 24) são comparados os atributos da técnica “n” com os do projeto em desenvolvimento (linhas 15 a 21). Cada peso individual é calculado comparando os atributos (linha 16). O peso total é a soma de todos os pesos dos atributos (linhas 19 e 22). Por fim, ordena-se a lista com as técnicas com maior adequação nas primeiras posições (linha 26), retornando o resultado que será fornecido ao usuário (linha 28).

Na Figura 19 é apresentado o processo de seleção de técnicas de teste usando a ferramenta SeleCTT. O fluxo de atividades do processo é composto inicialmente pela caracterização do projeto pela equipe de teste, por meio do fornecimento das informações sobre cada um dos atributos de caracterização definidos. Informações sobre as técnicas de teste presentes

no corpo de conhecimento são recuperadas pela ferramenta SeleCTT. Essas informações são utilizadas no processo de seleção, através do cálculo da adequação pela comparação dos atributos do projeto com os atributos das técnicas de teste. Após a comparação, é realizada a classificação das técnicas de acordo com a pontuação alcançada na adequação dos critérios específicos. Os resultados gerados pela ferramenta são disponibilizados na forma de uma lista ordenada de acordo com a relevância de cada técnica e a recomendação é apresentada aos interessados na seleção. Cabe aos interessados avaliar a lista de técnicas e selecionar uma delas para utilizar no seu projeto de software.

```
1 <?php if ( ! defined('BASEPATH')) exit ('No direct script access allowed');
2
3     protected function resultadoAdequacaoTecnica ($projeto)
4     {
5         // Inicialização das variáveis
6         $todosPesos = getPesos();
7         $todasTecnica = getTodasTecnica();
8         $resultado = array ();
9         $count = 0;
10
11         foreach ($todasTecnica as $tecnica) {
12             //Inicialização do peso total
13             $pesoTotal = floatval(0.000);
14             $atributo = 0;
15             foreach($todosPesos['atributos'] as $peso) {
16                 $resultado[$count] [$atributo] =
17                 comparacaoAtributoDoProjetoComTecnica($projeto[$atributo],
18
19                     $tecnica[$atributo],
20
21                     $peso);
22                 $resultado[$count] ['resultado-peso'] = $pesoTotal;
23                 $count++;
24             }
25             // Ordena Tecnicas mais adequadas nas primeiras posicoes
26             $resultado = ordenaResultado ($resultado);
27
28             return $resultado;
29         }
30     }
31 }
```

Código-fonte 1 – Código fonte do algoritmo de seleção (MOURA *et al.*, 2017).

A Figura 20 ilustra a interface do módulo de apresentação dos resultados na ferramenta SeleCTT, que fornece uma lista ordenada de técnicas de teste adequadas a um projeto de software a fim de auxiliar o processo de seleção com base na comparação de atributos do projeto e das técnicas de teste. Usuários padrão e administradores têm acesso a esse módulo, ambos podem utilizar esse módulo para obtenção da lista com as técnicas de teste concorrente mais adequadas a determinado projeto em desenvolvimento. Para obter a lista com os resultados é necessário acessar primeiro o módulo de inserção e fornecer informações relevantes ao projeto em desenvolvimento, ou seja, a caracterização do

Figura 19 – Processo de operação da ferramenta SeleCTT.

projeto a fim de compará-las às características das técnicas de teste e obter os resultados referentes a essa comparação. Após submeter as informações do projeto, o servidor executa o algoritmo de seleção da ferramenta SeleCTT através dos critérios relevantes ao projeto. Os resultados do algoritmo de seleção são classificados e as técnicas mais apropriadas são ordenadas conforme sua porcentagem de adequação ao projeto em desenvolvimento.

Figura 20 – Módulo de apresentação dos resultados.

3.6.2.3 Base de dados

A base de dados implementa o corpo de conhecimento definido na Seção 3.3. Esse corpo de conhecimento contém todas as técnicas de teste de software concorrente, identificadas no mapeamento sistemático, e apresentadas na Tabela 2. O sistema de gerenciamento de banco de dados MySQL3 foi utilizado para a criação do banco de dados e armazenamento das informações sobre as técnicas de teste no corpo de conhecimento.

A Figura 21 ilustra o armazenamento das informações sobre as técnicas de teste no corpo de conhecimento. Para cada técnica de teste presente no corpo de conhecimento, vinte e um atributos foram considerados para a caracterização dessa técnica. Cada atributo é armazenado com o valor do tipo texto na base de dados. Os atributos são agrupados em quatro categorias que especificam o tipo da informação: 1) Modelo de programação: reúne atributos que representam as características relacionadas à implementação do software que será testado pela técnica; 2) Características gerais da técnica de teste: reúne atributos para as técnicas de teste de software em geral (não apenas técnicas para software concorrente); 3) Características de técnicas de teste de software concorrente: reúne atributos que descrevem características específicas da programação concorrente que influenciam na atividade de teste; e 4) Apoio ferramental: reúne informações sobre as ferramentas que oferecem suporte à técnica de teste de software.

Figura 21 – Armazenamento de informações no corpo de conhecimento (MOURA *et al.*, 2017).

O armazenamento das informações sobre cada técnica de teste considera esses atributos, podendo variar o conteúdo armazenado na base de dados, podendo conter um valor textual e no caso nenhum valor seja fornecido pelo usuário o valor armazenado será “sem informação”, a fim de não armazenar um valor nulo o que prejudicaria o cálculo da adequação. O corpo de conhecimento preliminar contém 109 técnicas de teste de software concorrente, podendo ser expandido com a inserção de novas técnicas de teste, seguindo a evolução da literatura científica na área.

Um usuário com conta válida (registrado e aprovado no sistema) pode acrescentar uma nova técnica na base de dados. Porém, essa técnica permanece como pendente até a validação do administrador do sistema. Uma técnica com o status “pendente” não é incluída durante o

cálculo de adequação, ela só é incluída quando consolidada na base de dados pelo administrador do sistema. Esse sistema de avaliação de novas técnicas visa garantir a integridade dos dados armazenados no corpo de conhecimento e a confiabilidade sobre as técnicas que o compõem, não afetando seu funcionamento.

3.6.3 Funcionalidades

Nessa seção são descritas em detalhes todas as funcionalidades oferecidas pela ferramenta SeleCTT e ao final da seção é descrito um exemplo de aplicação da ferramenta para auxiliar o processo de seleção em um projeto de software específico. A página inicial da ferramenta com as abas para cada uma das funcionalidades é apresentada na Figura 22.

Figura 22 – Interface da página inicial da ferramenta.

3.6.3.1 Busca por uma técnica específica

Além do cálculo de adequação para obtenção da lista de recomendação, a ferramenta oferece apoio à visualização das técnicas armazenadas no corpo de conhecimento. Essa função pode ser útil a profissionais que estejam procurando por uma técnica específica. Na aba *Content*, na seção *Techniques* é possível visualizar as técnicas presentes no corpo de conhecimento e filtrar por tipo de linguagem, nome da abordagem e tipo da técnica de teste. Essa informação pode ser útil em estudos secundários e também para a identificação de técnicas ainda não incluídas durante o processo de atualização da base de dados.

3.6.3.2 Inserção de uma técnica na base de dados

Se uma nova técnica de teste de software concorrente é proposta pela academia e ainda não compõe o banco de dados ou ainda se novos estudos secundários são conduzidos na academia e novas técnicas de teste identificadas, é possível sua inserção no corpo de conhecimento da ferramenta. A inserção é feita por meio do preenchimento de formulário sobre as principais características da técnica, como ilustrado na Figura 23. Os atributos requisitados no formulário para a inclusão da técnica estão relacionados à identificação da técnica (título do estudo, ano de publicação, referência), os demais atributos correspondem aos atributos de caracterização de técnicas de teste de software concorrente como definido na Tabela 8. Não é obrigatório o preenchimento de todos os atributos, porém o maior detalhamento torna a caracterização da técnica mais precisa durante o processo de seleção.

SELECTT Information - Content - Document - Contact Technique - Admin

Insert Technique

Please provide the information below.

Part 1 - Study identification

Title
Enter the Title of the project published
* Required

Select year of publication:
[Dropdown menu] ☐ No information

Bibliografic reference (Bibtex)
[Text area] ☐ No information

Link (URL)
Link to the article ☐ No information

Part 2 - Programming model

Execution platform ?
[Text field] ☐ No information
Example: Linux, Unix, Windows, Android

Figura 23 – Interface do formulário de inserção de técnicas de teste.

3.6.3.3 Seleção de técnicas de teste

No processo de seleção, as informações sobre um projeto em desenvolvimento são fornecidas pelo usuário. Estas informações envolvem os atributos presentes na Tabela 8. O servidor da ferramenta receberá esses atributos na forma de texto através do módulo de inserção. A partir dessa submissão, o sistema compara os atributos do projeto, fornecidos pelo usuário com os atributos das técnicas presentes no corpo de conhecimento.

A comparação é feita considerando os vinte e um atributos, um a um, considerando seus respectivos pesos, definidos na Tabela 12. Nessa etapa, são comparados os valores textuais presentes nos atributos do projeto em desenvolvimento com as n -técnicas (sendo n uma das técnicas presentes no corpo de conhecimento) seguindo a relação dada pelo vetor de adequação em 1, no qual cada atributo é comparado para cada técnica. Este cálculo de comparação baseia-se no estudo de Neto e Travassos (2009).

Para o cálculo de adequação, considera-se a equação em 1. O valor de adequação ($v_{adequacao}(i,n)$), sendo i o atributo e n a técnica, é calculado através da comparação das características presentes no atributo. Se o atributo possuir mais de uma característica (exemplo: Android e Windows) apenas as características que forem iguais entrarão nesse valor. Nesse caso, comparando o atributo do projeto em desenvolvimento, exemplo Android, com atributo de uma técnica do corpo de conhecimento, exemplo Android e Windows, o resultado será de 50% de adequação para este atributo. No entanto, a adequação total é dada pela soma do vetor adequação multiplicando-se pelo respectivo peso, essa relação está presente em 1.

3.6.3.4 Apresentação de dados estatísticos

Como apresentado na Figura 24 uma predição estatística sobre a quantidade de acessos e o uso da ferramenta pode ser visualizada na aba *Content* → *Statistics*. O objetivo é armazenar informações sobre o uso da ferramenta para possível evolução da mesma.

Figura 24 – Estatísticas sobre o uso da ferramenta.

Os dados referentes aos projetos inseridos e resultados gerados são armazenados no banco de dados e uma possível extensão de funcionalidades da ferramenta pode utilizar esses dados como critério durante o processo de recomendação, fornecendo ao usuário um *feedback* sobre o uso prático da técnica como experiências de uso.

3.6.4 Exemplo de aplicação

Nessa seção é descrito um exemplo de aplicação da ferramenta SeleCTT para a seleção de uma técnica de teste a ser aplicada a um projeto de software exemplo. São ilustradas cada uma das interfaces da ferramenta e as etapas até a geração da recomendação pela ferramenta.

3.6.4.1 Caracterização do projeto de software

Ao iniciar o processo de seleção da técnica de teste, os interessados na seleção acessam a interface web da infraestrutura SeleCTT, na página inicial da ferramenta (Figura 22), acessando o menu *Technique* → *Software Project Information*, é possível visualizar o formulário apresentado na Figura 25. Os atributos requisitados pelo formulário foram definidos seguindo o esquema de caracterização apresentado na Seção 3.5. A descrição de cada atributo é feita por meio de texto e é utilizada para comparação com os atributos das técnicas de teste.

The screenshot shows the SeleCTT web interface. At the top is a navigation bar with the logo and menu items: Information, Content, Document, Contact, Technique, and Admin. The main content area is titled 'Part 1 - Study identification' and contains a 'Title' field with a placeholder 'Enter the Title of the project published' and a label 'Just to label this technique search'. Below this is 'Part 2 - Programming model' with three sections: 'Execution platform ?' with a text input field and a 'No information' checkbox, 'Context/Objective ?' with a text input field and a 'No information' checkbox, and 'Programming Language/Runtime libraries ?' with a text input field and a 'No information' checkbox. Each section includes an example text below the input field. Finally, 'Part 3 - General testing characteristics' contains a 'Testing technique ?' field with a text input field and a 'No information' checkbox, with an example text below it.

Figura 25 – Interface do formulário de inserção de atributos do projeto de software.

3.6.4.2 Cálculo da adequação

Com as informações do projeto de teste, fornecidas por meio do formulário apresentado anteriormente, é possível comparar com os atributos de cada uma das técnicas que compõem o corpo de conhecimento. O cálculo é feito confrontando os atributos do projeto com os atributos das técnicas de teste conforme apresentado no Algoritmo 1, conforme os pesos definidos na Tabela 12 e setados conforme o módulo de controle (Figura 18).

3.6.4.3 Recomendação

Após o cálculo da adequação, a ferramenta oferece uma lista ordenada de acordo com a relevância de cada uma das técnicas presentes no corpo de conhecimento ao projeto de teste instanciado. A Figura 26 ilustra a interface de apresentação da recomendação das técnicas de teste. Na página de resultados (*Results*) além da visualização da porcentagem de adequação de cada técnica ao projeto é possível expandir os resultados para cada técnica sendo possível visualizar exatamente quais atributos são compatíveis entre o projeto e a técnica selecionada. Essa funcionalidade pode ajudar o pesquisador ou profissional de teste a escolher a técnica mais adequada segundo as restrições do projeto em desenvolvimento.

Figura 26 – Interface de apresentação da recomendação de técnicas de teste.

As informações sobre o projeto e os resultados gerados ficam armazenados na ferramenta

e é possível acessá-los a qualquer momento, por meio da conta do usuário que conduziu o processo de seleção. Essas informações podem ser usadas em projetos futuros ou armazenadas para futuras consultas, a fim de auxiliar a equipe de teste na implantação de uma nova técnica na atividade de teste e avaliar a sua eficácia no processo de desenvolvimento.

A ferramenta implementa o corpo de conhecimento, o esquema de classificação e a abordagem de auxílio à seleção de técnicas de teste, oferecendo uma infraestrutura web capaz de guiar pesquisadores e profissionais de teste durante o processo de escolha da técnica de teste a ser aplicada durante o projeto de desenvolvimento de software. Por estar disponível em ambiente web, é possível o acesso de qualquer pessoa interessada na área, bem como a contribuições dessas pessoas para a evolução do *framework*.

Dentre suas principais funcionalidades, a ferramenta oferece mecanismos de apoio ao cálculo automatizado do nível de adequação de técnicas de teste a um projeto de software previamente caracterizado. Além disso, oferece uma representação visual dos atributos que combinaram entre o projeto e as técnicas, assim o usuário pode selecionar a técnica com base nos atributos que julgar mais relevantes ao seu projeto.

Tendo em vista que a ferramenta oferece uma lista ordenada de técnicas de teste com base em atributos pré-definidos, fica a cargo do profissional de teste analisar mais profundamente cada técnica recomendada e suas particularidades e a partir daí optar pelo uso ou não em seu projeto de software.

Cabe ainda destacar que o objetivo da ferramenta é disponibilizar um conjunto ordenado de informações sobre as técnicas de teste disponíveis na literatura, tentando evitar que a equipe de teste permaneçam usando seu próprio conhecimento e hábitos adquiridos em projetos passados durante a seleção de uma técnica de teste para um novo projeto de software. Esse fator pode levar ao reúso de uma mesma técnica, sem considerar a existência de outras o que dificulta a transferência de novos conhecimentos obtidos na academia para a indústria.

3.7 Considerações finais

Este capítulo descreveu o *framework* proposto com o objetivo de sistematizar o processo de caracterização e seleção de técnicas de teste para o contexto de programação concorrente. Embora alguns métodos de caracterização existam para o contexto da programação sequencial, eles não são totalmente aplicáveis ao teste de software concorrente. Por este motivo, foi realizado um mapeamento sistemático a fim de identificar quais características do software concorrente afetam a atividade de teste e com base nessas informações foi construído um corpo de conhecimento para a área. Com base nesses resultados, foi desenvolvido um esquema de caracterização e uma abordagem de seleção específica para o contexto da programação concorrente. Com a abordagem instanciada, uma infraestrutura web foi implementada para automatizar o processo de seleção e disponibilizar seu uso online, ampliando o acesso aos dados e aos resultados obtidos. A proposta é avaliada por meio de um estudo experimental, o qual é descrito no próximo capítulo.

AVALIAÇÃO EXPERIMENTAL

4.1 Considerações Iniciais

Este capítulo descreve o experimento controlado conduzido a fim de analisar a efetividade, eficiência, completude e usabilidade, bem como a satisfação do usuário quando usando a abordagem de seleção de técnicas de teste proposta neste trabalho, denominada SeleCTT (um homônimo para a ferramenta apresentada na Seção 3.6, que implementa essa abordagem). A abordagem de seleção é composta pelo esquema e atributos de caracterização (Seção 3.5.1), os critérios de adequação (Seção 3.5.2) e o mecanismo de cálculo para a recomendação das técnicas de técnicas de teste mais adequadas a um projeto de software concorrente. A ferramenta não é utilizada durante a condução do experimentos, pois o objetivo era avaliar a abordagem de seleção proposta e não os benefícios oferecidos pela automatização do processo de seleção.

Ao longo desta seção é usado o termo “SeleCTT” para se referir à abordagem de teste, sempre que houver referência à ferramenta ela será especificada.

Na Seção 4.2 é descrito o planejamento do experimento. Na Seção 4.3 é apresentado o processo de execução do experimento. Na Seção 4.4 é mostrada a coleta e análise quantitativa e qualitativa dos dados. E por fim, na Seção 4.4.4 é apresentada uma discussão geral e algumas conclusões sobre os resultados obtidos no estudo.

4.2 Design do Experimento

O estudo experimental foi realizado com base nos princípios da Engenharia de Software Baseada em Evidência, a fim de obter evidências experimentais da eficácia e aplicabilidade prática da abordagem de seleção proposta.

O design do experimento é fatorial (2x4), no qual foram estudados os efeitos de dois fatores (abordagem de seleção e projeto de software), cada subdivisão do fator (nível do fator) e

os tratamentos (as combinações possíveis entre os diversos fatores nos seus diferentes níveis), conforme apresentado na Tabela 13. Para este estudo o Fator A representa a abordagem de seleção (SeleCTT ou Geral (VEGAS; BASILI, 2005)) e o Fator B corresponde ao tipo de projeto de software utilizado na condução do experimento (Banco, Estacionamento, Equação e Firewall) que são atribuídos aleatoriamente aos grupos de participantes (G1, G2, G3 e G4). O experimento foi executado em duas etapas: seleção 1 (S1) e seleção 2 (S2).

Tabela 13 – Design do experimento.

Fator A	Fator B	Grupos
Geral	Banco	G1-S1
	Estacionamento	G2-S1
	Equação	G3-S1
	Firewall	G4-S1
SeleCTT	Banco	G1-S2
	Estacionamento	G2-S2
	Equação	G3-S2
	Firewall	G4-S2

4.2.1 Objetivo

O objetivo deste estudo é avaliar se a abordagem “SeleCTT” (proposta neste trabalho) auxilia o processo de seleção de técnicas de teste, quando comparada ao uso da abordagem de seleção “Geral” (não específica para o contexto da programação concorrente) proposta por Vegas e Basili (2005).

No planejamento do estudo são utilizados alguns instrumentos derivados do experimento descrito em (VEGAS; BASILI, 2005) que avaliou a abordagem de seleção de técnicas de teste Geral. Embora nosso estudo use parte da instrumentação proposta, não é considerado uma replicação pois utiliza outro domínio de aplicação e outros objetos experimentais.

O objetivo da avaliação é analisar se a abordagem SeleCTT apresenta completude, efetividade, eficiência, usabilidade e satisfação do usuário, superior ao uso da abordagem Geral proposta em (VEGAS; BASILI, 2005), que foi a base para o desenvolvimento da abordagem proposta neste estudo. Além disso, essa comparação tem o intuito de observar se os atributos adicionais da abordagem proposta para programas concorrentes influenciam positivamente o processo de seleção de técnicas de teste se comparado ao uso de uma abordagem de seleção mais geral, que considera apenas o contexto de programação sequencial.

4.2.2 Seleção do Contexto

Os 64 participantes eram alunos da disciplina de programação concorrente com conhecimento em pelo menos duas áreas que abrangem o estudo (programação concorrente e engenharia de software). O estudo foi conduzido por meio de questionários online, utilizando a plataforma Google Forms¹. Os estudantes foram reunidos em sala de aula e sob o acompanhamento do

¹ <<https://www.google.com/forms/about/>>

pesquisador receberam um link para acesso ao experimento. O estudo foi executado em ambiente acadêmico, o qual é adequado para estudos de viabilidade e embora os resultados não possam ser diretamente aplicados ao contexto industrial, possibilitam testar conceitos propostos antes do seu uso prático (SHULL *et al.*, 2001).

4.2.3 Questões de Pesquisa

As questões de pesquisa foram definidas baseadas no estudo de Vegas e Basili (2005), a fim de avaliar a abordagem de seleção proposta quanto a sua relevância ao processo de caracterização e seleção de técnicas de teste. Para cada questão de pesquisa foram definidas métricas, a fim de obter respostas mensuráveis para essas questões, conforme descrito a seguir:

- **QP1: Qual a completude da abordagem de seleção?**

Métrica 1.1: Quantidade de informações (atributos) utilizados durante a seleção.

Métrica 1.2: Quantidade de informações (atributos) faltantes durante a seleção.

- **QP2: Qual é a efetividade da abordagem de seleção?**

Métrica 2.1: Porcentagem de acertos no processo de seleção da técnica adequada.

- **QP3: Qual é a eficiência da abordagem de seleção?**

Métrica 3.1: Tempo gasto na seleção.

Métrica 3.2: Porcentagem de acertos pela quantidade de atributos usados.

Métrica 3.3: Porcentagem de acertos pelo tempo gasto na seleção.

- **QP4: Qual é a usabilidade da abordagem de seleção?**

Métrica 4.1: Quantidade de problemas encontrados durante a seleção.

- **QP4: A abordagem de seleção satisfaz o usuário?**

Métrica 5.1: Qualitativa - Opinião do usuário (satisfação) em relação à abordagem de seleção.

4.2.4 Definição das Hipóteses

As hipóteses definidas para o experimento estão relacionadas com a avaliação da influência da abordagem de seleção proposta neste trabalho, em contrapartida à abordagem de seleção mais genérica existente na literatura. Além disso, é analisada a influência do tipo de projeto usado nos resultados obtidos. As hipóteses gerais para esse estudo são descritas a seguir:

- **Hipótese nula (HN1):** Não há diferença no uso da abordagem SeleCTT e da abordagem Geral para o processo de seleção de técnicas de teste de software concorrente para todos os projetos de software.

- **Hipótese alternativa (HA1):** A abordagem SeleCTT apresenta resultados superiores se comparado ao uso da abordagem Geral.
- **Hipótese alternativa (HA2):** A abordagem de seleção Geral apresenta resultados superiores ao uso da abordagem SeleCTT.
- **Hipótese alternativa (HA3):** Alguns projetos apresentam resultados superiores se comparados com outros, independentemente da abordagem usada.
- **Hipótese alternativa (HA4):** Dependendo do projeto de software, o processo de seleção apresenta melhores resultados usando a abordagem SeleCTT ou a abordagem Geral.

Considerando a existência de vários fatores a serem avaliados (completude, efetividade, eficiência, usabilidade e satisfação do usuário) as hipóteses gerais definidas são divididas em sub-hipóteses abrangendo cada um desses aspectos conforme apresentado na Tabela 14.

Tabela 14 – Hipóteses por aspecto estudado.

Aspecto avaliado	Hipótese nula	Hipótese alternativa
Completude	H.0.1. A completude dos atributos é independente da abordagem de seleção utilizada	H.1.1. A abordagem SeleCTT apresenta um conjunto de atributos mais completo que a abordagem de seleção Geral independentemente do projeto usado H.1.2. A abordagem Geral apresenta um conjunto de atributos mais completo que a abordagem SeleCTT independentemente do projeto usado H.1.3. A completude do conjunto de atributos está relacionado ao tipo de projeto usado
Efetividade	H.0.2. A efetividade do processo de seleção é independente da abordagem de seleção utilizada	H.2.1. A abordagem SeleCTT melhora a efetividade do processo de seleção independentemente do projeto usado H.2.2. A abordagem Geral melhora a efetividade do processo de seleção independentemente do projeto usado H.2.3. A efetividade da seleção está relacionada ao tipo de projeto usado
Eficiência	H.0.3. A eficiência do processo de seleção é independente da abordagem de seleção utilizada	H.3.1. A abordagem SeleCTT melhora a eficiência do processo de seleção independentemente do projeto usado H.3.2. A abordagem Geral melhora a eficiência do processo de seleção independentemente do projeto usado H.3.3. A eficiência da seleção está relacionada ao tipo de projeto usado
Usabilidade	H.0.4. A usabilidade do processo de seleção é independente da abordagem de seleção utilizada	H.4.1. A abordagem SeleCTT melhora a usabilidade do processo de seleção independentemente do projeto usado H.4.2. A abordagem Geral melhora a usabilidade do processo de seleção independente do projeto usado H.4.3. A usabilidade da seleção está relacionada ao tipo de projeto usado

4.2.5 Definição das Variáveis

O efeito dos tratamentos no resultado do estudo é mensurado por meio das variáveis dependentes e independentes. As relações entre variáveis (tratamentos) são definidas de maneira que as alterações de valores da variável independente afetem os valores da variável dependente. As variáveis dependentes e independentes definidas para esse estudo são apresentadas a seguir.

4.2.5.1 Variáveis dependentes

As variáveis dependentes para este experimento (saídas que refletem a relação entre os diferentes níveis dos fatores) são mostrados na Tabela 15. Os valores para essas variáveis mudam durante a aplicação do experimento a fim de avaliar sua influência no processo de seleção de técnicas de teste.

Tabela 15 – Variáveis dependentes.

Aspecto avaliado	Variáveis resposta	Objetivo
Compleitude	Total de informação usada durante a seleção	Avaliar quais atributos são usados durante a seleção
	Total de informação faltantes durante a seleção	Identificar atributos faltantes no processo de seleção
Usabilidade	Quantidade de problemas encontrados durante a seleção	Identificar problemas na abordagem de seleção
Efetividade	A ordem de adequação das técnicas ao projeto apresentado	Avaliar se a abordagem aumenta o número de acertos dos participantes
Eficiência	Tempo gasto na seleção das técnicas	Avaliar se a abordagem reduz o tempo de seleção
	Tempo de seleção/ quantidade de atributos usados	Avaliar se a abordagem reduz o tempo para análise dos atributos
	Tempo de seleção/ quantidade de acertos	Avaliar se a abordagem reduz o tempo e aumenta a quantidade de acertos
Satisfação do usuário	Quais as vantagens e desvantagens da abordagem usada?	Identificar pontos fortes e fracos na abordagem
	A abordagem facilita a seleção?	Avaliar se a abordagem auxilia o processo de seleção
	A abordagem é simples de ser entendida?	Avaliar se a abordagem é de fácil compreensão
	A abordagem é aplicável ao contexto industrial?	Avaliar se do ponto de vista do usuário a abordagem é aplicável à indústria
	Quais melhorias podem ser feitas?	Identificar pontos a serem aprimorados
	A visão do participante sobre o processo de seleção mudou?	Avaliar se a abordagem muda o modo de pensar do usuário quanto ao processo de seleção
	O que o participante aprendeu?	Identificar os ganhos ao participante

4.2.5.2 Variáveis independentes

As variáveis independentes do estudo são descritas a seguir. Os valores dessas variáveis não mudam durante a execução do experimento e não devem afetar os resultados obtidos para as variáveis dependentes.

- **Experiência dos participantes:** nesse estudo foram considerados alunos de graduação com conhecimento em programação concorrente e engenharia de software, alguns com conhecimento em teste de software e seleção de técnicas de teste, conforme apresentado

na Tabela 20. Embora a maioria possuía conhecimento técnico na área, eles são considerados inexperientes do ponto de vista da indústria, pois não atuaram em projetos de desenvolvimento de software concorrente na indústria.

- **Tarefa a ser realizada:** os participantes deviam, baseados em informações de projetos fornecidas, selecionar quais dentre as técnicas de teste apresentadas eram adequadas para o processo de teste, e ordená-las em relação a sua relevância ao projeto. O objetivo dessa tarefa foi avaliar a eficácia da abordagem SeleCTT para o processo de seleção de técnicas.
- **Documentação:** foram fornecidas aos participantes a especificação de um projeto de software bem como a descrição de cada uma das técnicas do conjunto de seleção. Além disso, um conjunto de documentos para a coleta de dados e respostas dos participantes (formulários online) foram utilizados para auxiliar a condução do experimento. Os formulários e demais instrumentos usados no estudo estão disponíveis no Apêndice A.

4.2.6 Fatores e alternativas

Os fatores (variáveis cujos valores serão alterados durante o experimento) definidos para esse experimento são: o processo de seleção de técnicas de teste e o projeto de software utilizado para condução do experimento. Os níveis (alternativas) para cada um desses fatores são apresentados nas Seções 4.2.6.1 e 4.2.6.2.

4.2.6.1 Abordagens de seleção

O fator abordagem de seleção tem duas alternativas possíveis: o abordagem de seleção proposta por (VEGAS; BASILI, 2005) (Geral) e a abordagem proposta neste trabalho (SeleCTT) que considera os critérios de adequação e os pesos definidos para os atributos específicos para o contexto da programação concorrente.

- **Abordagem Geral:** os sujeitos recebem o repositório com 4 técnicas de teste como descrito no Apêndice A, com os atributos organizados conforme a abordagem de Vegas e Basili (2005), os quais são usados como guia para selecionar as técnicas mais adequadas para os projetos de software considerados.
- **Abordagem SeleCTT:** os sujeitos recebem o mesmo repositório descrito no Apêndice A organizado de acordo com a versão completa da abordagem SeleCTT, com os novos atributos de caracterização e o indicador do nível de adequação de cada atributo e usando essas informações eles selecionam a técnica mais adequada ao projeto que receberam. Essa versão é usada para avaliar se os novos atributos de caracterização definidos para o contexto da programação concorrente trazem melhorias ao processo de seleção de técnicas.

4.2.6.2 Projeto de software

O fator projeto de software inclui o sistema concorrente a ser desenvolvido e as informações a respeito das características desse projeto.

Neste experimento são utilizados quatro projetos de software, com diferentes atributos. Os projetos de empréstimo bancário e de controle de vagas de estacionamento são derivados do estudo de [Vegas e Basili \(2005\)](#). O projeto de paralelização de equações de Poisson é um projeto desenvolvido na academia em parceria com a indústria ([ROGENSKI *et al.*, 2015](#)) e o projeto de correção do módulo autenticação de usuários é um projeto real desenvolvido na indústria, cedido a este projeto pela empresa Aker². Todos esses projetos são descritos brevemente a seguir:

- **Controle de empréstimo bancário:** o objetivo é desenvolver um sistema distribuído para aprovação e rejeição de empréstimos bancários e calcular os valores devidos no empréstimo, conforme os requisitos descritos na Tabela 40.
- **Controle de vagas de estacionamento:** O objetivo é desenvolver um sistema distribuído (cliente-servidor) para gerenciamento de vagas de estacionamento utilizando sensores sem fio, seguindo os requisitos apresentados na Tabela 41.
- **Projeto de paralelização de um método *multigrid* para solução de equação de Poisson:** o objetivo do projeto é paralelizar algoritmos para solução numérica da equação de Poisson para escoamento bidimensional incompressível de fluido newtoniano, a descrição e os requisitos do projeto são apresentados na Tabela 42.
- **Correção do módulo de autenticação de usuários para um sistema de Firewall para segurança em redes corporativas:** o objetivo é aumentar o desempenho do módulo de autenticação de usuários, já existente no sistema, por meio da distribuição da carga de processamento entre diversos servidores. O projeto desse software e seus requisitos são descritos na Tabela 43.

4.2.7 Seleção dos sujeitos

O experimento controlado foi conduzido no Instituto de Ciências Matemáticas e de Computação - ICMC/USP com alunos de graduação 3º ano do curso de Bacharelado em Ciências de Computação. O estudo apresenta exemplos de projetos de software que utilizam programação concorrente e alternativas de técnicas de teste (extraídas do corpo de conhecimento) a serem aplicadas para o teste de tais projetos.

Os sujeitos foram selecionados por conveniência. Foram convidados alunos de graduação que estão finalizando a disciplina de programação concorrente e que já tenham cursado a disciplina de engenharia de software. Os sujeitos foram divididos em quatro grupos com tamanho

² <<https://www.aker.com.br/aker-firewall-utm/>>

padrão (G1, G2, G3 e G4). A divisão dos sujeitos/participantes (P1, P2, ..., Pn) nos grupos é apresentada na Tabela 16. Essa divisão visa balancear a contribuição do tratamento e agrupamento das diferentes combinações de resultados.

Tabela 16 – Atribuição dos sujeitos por grupos e subgrupos.

	G1	G2	G3	G4
Seleção 1	P1	P17	P33	P49
	P2	P18	P34	P50
	P3	P19	P35	P51
	P4	P20	P36	P52
	P5	P21	P37	P53
	P6	P22	P38	P54
	P7	P23	P39	P55
	P8	P24	P40	P56
Seleção 2	P9	P25	P41	P57
	P10	P26	P42	P58
	P11	P27	P43	P59
	P12	P28	P44	P60
	P13	P29	P45	P61
	P14	P30	P46	P62
	P15	P31	P47	P63
	P16	P32	P48	P64

4.2.8 Design do experimento

Tendo em vista o objetivo de analisar a influência do uso da abordagem SeleCTT versus a abordagem Geral no processo de seleção de técnicas de teste, os sujeitos são divididos em quatro grupos conforme apresentado na Tabela 17. Os sujeitos foram atribuídos randomicamente aos quatro grupos estabelecidos.

Tabela 17 – Atribuição dos tratamentos aos grupos.

Projeto Abordagem	Banco	Estacionamento	Equação	Firewall
Geral	G1	G3	G4	G4
SeleCTT	G2	G1	G2	G3

A distribuição dos grupos entre as fatores de interesse são mostrados na Tabela 18. Os sujeitos do grupo 1 (G1) fazem a seleção por meio da abordagem Geral e posteriormente usando a abordagem SeleCTT. Os sujeitos do grupo 3 (G3) farão a seleção primeiro com o SeleCTT e depois com a abordagem Geral. Os grupos 2 e 4 (G2 e G4) utilizam apenas uma das abordagens SeleCTT ou Geral, respectivamente. De acordo com essa distribuição, os grupos 1 e 3 são usados para examinar o efeito do processo de seleção, e o propósito dos grupos 2 e 4 é obter o efeito de aprendizagem nos sujeitos inexperientes. O grupo 1 representa o contexto no qual a abordagem SeleCTT é introduzida na empresa, onde pessoas já tem experiência na seleção (Geral) e tentam a utilização de um novo método que é supostamente melhor que o antigo.

Tabela 18 – Distribuição dos grupos entre os fatores para análise de influência.

	G1	G2	G3	G4
Seleção 1	Geral	SeleCTT	SeleCTT	Geral
Seleção 2	SeleCTT	SeleCTT	Geral	Geral

4.2.9 Projeto do experimento

Para realização do experimento, cada participante foi responsável por fazer a seleção da técnica de teste mais adequada ao projeto de software para o qual foi alocado e ordenar por importância em relação às demais técnicas disponíveis. Os projetos foram distribuídos de maneira aleatória dentre os quatro grupos de participantes.

Como não era possível determinar anteriormente se o projeto a ser usado poderia influenciar os resultados do estudo, cada sujeito usou uma combinação diferente de abordagem de seleção e projeto de software, conforme apresentado na Tabela 17. Existem quatro projetos de software e cada grupo tem 8 sujeitos, assim é possível distribuir randomicamente os projetos de software em duas diferentes seleções e grupos.

Para cada projeto de software há uma técnica setada como adequada e as demais técnicas são ordenadas seguindo seu nível de adequação ao projeto. Um gabarito foi criado para auxiliar a análise dos resultados do experimento. Os projetos foram inseridos na ferramenta SeleCTT para o cálculo dos pesos dos atributos e ranqueamento das técnicas selecionadas. Os resultados sumarizados são mostrados na Tabela 48. Essa tabela não é de conhecimento dos participantes do experimento, apenas dos pesquisadores que desenvolveram o estudo e irão utilizar a tabela como gabarito para correção.

O design adotado no experimento é de dois fatores com replicação, no qual atribui-se aleatoriamente os sujeitos (alunos) a cada combinação de tratamento. Os tratamentos referem-se à: 1) utilização da abordagem SeleCTT para escolha da técnica de teste e 2) escolha da técnica baseada apenas nas informações fornecidas por meio da abordagem Geral.

4.2.10 Procedimento experimental

A execução do experimento segue a organização conforme a Tabela 17. O experimento é realizado em duas etapas uma de treinamento e uma de execução com os grupos. Na etapa de treinamento os fundamentos de teste de software concorrente são apresentados e uma explicação sobre a condução do experimento é fornecida. Após o treinamento os grupos são submetidos ao experimento, conforme a distribuição mostrada na Tabela 19.

Tabela 19 – Procedimento de execução do experimento.

Etapa 1	Etapa 2	Etapa 3
Treinamento	Execução 1	Execução 2
Apresentação de uma visão geral de teste de software no contexto da programação concorrente. Descrição e execução de atividade de preparação dos participantes para o experimento	G1-S1, G2-S1, G3-S1, G4-S1	G1-S2, G2-S2, G3-S2, G4-S2

4.2.11 Instrumentação

Um conjunto de formulários é usado para coleta de dados e fornecimento de informações aos participantes. Os formulários são apresentados no Apêndice A. Para construção dos formulários foi utilizado como apoio o estudo apresentado em (VEGAS; BASILI, 2005). Todos os sujeitos devem responder aos formulários 49, 50, 51, 52 e 53. O formulário 54 só é preenchido pelos sujeitos que utilizaram a abordagem de seleção SeleCTT.

4.3 Execução e Coleta de Dados

O experimento foi executado com alunos da disciplina de programação concorrente, que respondem um questionário sobre sua experiência com os temas de pesquisa, recebem treinamento sobre noções de teste de software concorrente e então são divididos em grupos para a execução completa do experimento. Os participantes alocados para o experimento são apresentados na Tabela 19. O experimento contou com a participação de 64 alunos com diferentes níveis de experiência. Na Tabela 20 são apresentados os níveis de experiência dos participantes conforme as áreas da pesquisa: Engenharia de Software (ES), Teste de Software (TS), Programação Concorrente (PC), Teste de Software Concorrente (TSC), Seleção de Técnicas de Teste de Software (STTS). Todos estudantes possuem conhecimento em Programação Concorrente e Engenharia de Software, alguns apresentam conhecimento em Teste de software e poucos apresentam conhecimento em Seleção de Técnicas de Teste de Software. A experiência dos participantes é medida usando os seguintes valores: (1) nenhum; (2) estudei em aula ou livro; (3) pratiquei em projetos em sala de aula; (4) pratiquei em um projeto na indústria; e (5) pratiquei em vários projetos na indústria.

4.3.1 Avaliação estatística dos dados

O relacionamento entre a variável resposta quantitativa e os fatores qualitativos é analisado por meio do teste estatístico de análise de variância de dois fatores com replicação - ANOVA (*Two-Way Analysis of Variance*) (MACFARLAND, 2011). Esse teste é aplicado para verificar se há diferença significativa entre as médias das variáveis de resposta nos grupos estabelecidos pela combinação dos diferentes níveis dos fatores. Além disso, analisa se há interferência entre os fatores. Assim, por meio desse teste é possível avaliar se o tipo de abordagem influencia no processo de seleção e se o tipo do projeto usado interfere nos resultados das abordagens dado o design do experimento.

Quando os resultados da análise de variância comprovam que há diferença significativa entre as médias das amostras não probabilísticas, ou seja, que é possível rejeitar a hipótese nula, o teste estatístico Teste-t unicaudal à direita (AHSANULLAH *et al.*, 2014), é usado para avaliar se a abordagem SeleCTT apresenta resultados superiores (melhores) aos da abordagem Geral para as variáveis avaliadas.

Tabela 20 – Caracterização dos participantes do experimento.

Grupo	Participante	Experiência em ES	Experiência em TS	Experiência em PC	Experiência em TSC	Experiência em STTS
G1	p1	3	1	3	1	1
	p2	3	1	3	1	1
	p3	3	2	3	2	2
	p4	3	2	3	2	2
	p5	3	2	3	2	1
	p6	3	3	3	3	3
	p7	3	2	3	1	1
	p8	3	2	3	2	1
	p9	2	3	3	2	3
	p10	3	3	3	2	2
	p11	4	5	3	2	1
	p12	3	3	3	2	3
	p13	3	2	3	2	2
	p14	4	4	3	2	2
	p15	4	4	3	2	4
	p16	3	2	3	1	2
G2	p17	3	2	3	1	1
	p18	3	1	3	1	1
	p19	3	2	3	2	2
	p20	3	3	3	2	2
	p21	3	2	3	2	1
	p22	3	2	3	2	1
	p23	3	2	3	2	1
	p24	2	2	3	2	2
	p25	5	5	5	5	5
	p26	3	2	3	2	2
	p27	3	2	3	2	2
	p28	3	2	3	3	1
	p29	3	3	3	2	2
	p30	3	3	3	2	2
	p31	2	2	2	1	1
	p32	3	2	3	2	2
G3	p33	2	2	3	1	1
	p34	2	2	3	1	1
	p35	2	1	3	1	1
	p36	2	2	2	2	2
	p37	3	2	2	2	2
	p38	3	3	3	1	3
	p39	3	2	3	1	1
	p40	3	2	2	1	2
	p41	3	2	3	2	2
	p42	3	3	3	2	2
	p43	3	2	3	2	2
	p44	5	5	2	2	2
	p45	3	2	3	2	2
	p46	3	2	3	1	1
	p47	3	2	3	2	2
	p48	3	3	3	3	2
G4	p49	3	3	2	2	1
	p50	3	2	3	1	1
	p51	2	1	3	2	1
	p52	3	2	3	2	2
	p53	2	2	2	1	2
	p54	5	4	5	2	1
	p55	3	2	3	2	2
	p56	3	2	3	2	2
	p57	3	3	3	1	1
	p58	3	2	3	1	1
	p59	3	2	3	1	1
	p60	3	3	3	3	1
	p61	4	2	3	1	1
	p62	3	2	3	2	2
	p63	3	2	3	1	2
	p64	3	2	3	1	1

4.4 Análise dos Resultados

Após a execução do experimento os dados coletados foram analisados utilizando análise descritiva e inferência estatística, conforme descrito nas próximas seções.

Para detecção dos *outliers* foi utilizado o recurso gráfico de construção de *box plots* (WILLIAMSON *et al.*, 1989), utilizando a ferramenta *MiniTab*³ (MEYER; KRUEGER, 1997). Foram analisados para cada variável do estudo os *outliers*, conforme os gráficos (*box plots*) apresentados nas próximas seções. Nesse estudo consideramos os *outliers* em todas as análises. Como a amostra é relativamente pequena e bastante diversificada em termos da experiência de cada participante, eram esperados resultados diferentes para cada participante o que não deve afetar a avaliação e qualidade dos dados coletados. *Outliers* nesse estudo representam valores individuais atípicos causados por variações nos grupos amostrais (GRUBBS, 1969).

Como são avaliados dois fatores e são usados vários grupos com diversos indivíduos para cada fator, esta amostra é considerada uma amostra com repetição. Portanto, para a análise de variância é utilizado o teste ANOVA de dois fatores com replicação. A análise de variância (ANOVA) é usada para avaliar se há diferença no uso das duas abordagens e se essa diferença ocorre devido ao tipo de abordagem de seleção (amostras) ou devido ao tipo de projeto usado (colunas). Neste tipo de teste de hipóteses é analisada se há diferença entre as abordagens de seleção, entre os projetos de software usados e se a presença de interação entre os fatores está associada à mudança de comportamento de um fator nos diferentes níveis do outro fator, com relação às características de interesse, ou seja, se o tipo do projeto de software interfere nos resultados da abordagem e vice-versa.

Neste estudo são definidas três hipóteses alternativas para cada hipótese nula. A fim de selecionar a hipótese alternativa após a rejeição da hipótese nula utilizando ANOVA, é utilizado o *Teste-t unicaudal à direita* (ZABELL, 2008), pois a hipótese alternativa de interesse para esse estudo é que a abordagem SeleCTT é superior a abordagem Geral durante o processo de seleção de técnicas de teste.

4.4.1 Análise quantitativa dos resultados

Nesta seção são apresentadas as análises quantitativas para cada variável avaliada no estudo (Tabela 15), considerando os resultados obtidos com os participantes durante a execução da seleção.

4.4.1.1 Completude - Porcentagem de atributos usados

- **Análise de outliers**

Os resultados do teste de análise de *outliers* (Figura 27) mostram que nenhum dos participantes foi considerado *outlier* para a variável completude (em relação à quantidade de

³ <www.minitab.com/>

informação utilizada na seleção) em nenhum dos fatores avaliados (abordagem de seleção e tipo do projeto de software).

Figura 27 – Box plots para a variável completude dos atributos usados durante a seleção, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

• Análise de variância e teste de hipóteses

A análise das médias é mostrada na Figura 28, a fim de verificar se existem diferenças significativas dentre essas médias é utilizado o teste de análise de variância ANOVA de dois fatores com replicação (MACFARLAND, 2011). Os resultados da análise de variância são apresentados na Tabela 21. O valor de F é maior que o F-crítico ($7,674 > 4,013$) e o valor-P é menor que o nível de significância estabelecido de 5% ($0,008 < 0,05$) para as abordagens (Geral e SeleCTT), o que indica que existe uma diferença significativa na mudança do número de atributos usados para cada tipo de abordagem. O valor não significativo de F e do valor-P para as colunas (Banco, Estacionamento, Equação e Firewall) indica que a mudança teve o mesmo efeito em todas os projetos, ou seja, não existe diferença significativa entre o uso de diferentes projetos considerando a quantidade de atributos usados na seleção. Além disso, o valor não significativo de F e do valor-P para as interações indica que não existe um efeito de interação nas variáveis, ou seja o tipo do projeto não afeta nos resultados do tipo de abordagem de seleção usada.

Figura 28 – Análise da média (porcentagem de informação usada na seleção), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Tabela 21 – Anova (Complettude - atributos usados).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	0,145	1,000	0,145	7,674	0,008	4,013
Colunas	0,068	3,000	0,023	1,201	0,318	2,769
Interações	0,039	3,000	0,013	0,690	0,562	2,769
Dentro	1,058	56,000	0,019			
Total	1,310	63,000				

Como o teste de variâncias mostrou que existe uma diferença significativa entre as médias relacionadas à completude dos atributos usados durante o processo de seleção (considerando o tipo da abordagem de seleção), foi aplicado o teste de hipóteses (Teste-t unicaudal à direita) sobre a amostra, para analisar se a hipótese alternativa H.1.1 (Tabela 14) pode ser aceita, com base na rejeição da hipótese nula H.0.1. Os resultados do Teste-t (Tabela 22) mostram que não é possível aceitar a hipótese alternativa H.1.1. pois o valor Stat-t é menor que t-crítico uni-caudal ($-2.778 < 1.670$) para o nível de 5% de significância. Portanto a abordagem SeleCTT não é mais completa que a abordagem Geral considerando a porcentagem de atributos usados durante a seleção, pode-se afirmar apenas que as abordagens requerem quantidades de atributos diferentes.

Tabela 22 – Teste-t (Complettude - atributos usados).

Teste-t	SeleCTT	Geral
Média	0.382102273	0.477272727
Variância	0.019442732	0.018128499
Observações	32	32
Hipótese da diferença de média	0	0.477272727
gl	62	
Stat t	-2.777470279	
P(T<=t) uni-caudal	0.003618833	
t crítico uni-caudal	1.669804163	

4.4.1.2 Complettude - Quantidade de atributos faltantes

• Análise de outliers

A Figura 29 apresenta os resultados do teste de análise de *outlier* para a variável completude considerando a quantidade de atributos faltantes no esquema de caracterização.

Figura 29 – *Box plots* para a variável completude em relação aos atributos faltantes, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Conforme os resultados observados na Figura 29, existem 5 *outliers* (8%) considerando o tipo do projeto de software e 8 *outliers* (13%) considerando o tipo de abordagem de seleção. Esses valores são mantidos durante a condução dos testes estatísticos, dado o tamanho da amostra e a diversidade dos sujeitos.

• Análise de variância e teste de hipóteses

A Figura 30 mostra a distribuição das médias para cada abordagem de seleção e para cada projeto de software usado. Na Tabela 23 são apresentados os resultados do teste de análise de variância dessas médias, considerando a quantidade de atributos faltantes relatados pelos participantes para os fatores: abordagem de seleção e tipo do projeto de software. A análise dos resultados do teste estatístico mostram que não é possível rejeitar a hipótese nula $H_0.1$ (Tabela 14) para a quantidade de atributos faltantes, pois a aplicação da ANOVA para essa amostra resultou no valor de F menor que o valor de F crítico ($1,263 < 4,013$) e o valor-P é maior que o nível de significância ($0,266 > 0,05$) para o tipo de abordagem usada. Então não é possível afirmar que exista uma diferença significativa entre as médias de atributos faltantes para o tipo de abordagem de seleção. Ao mesmo tempo, a interpretação do valor de F é também menor que o F crítico ($1,055 < 2,769$) e valor-P maior que o nível de significância ($0,266 > 0,05$) relacionado as colunas (projetos de software), indicando que esse fator também não afeta os resultados da seleção. E por fim o valor F relacionado ao efeito de uma interação sobre a outra também é menor que o F-crítico ($0,681 < 2,769$) e valor-P é maior que o nível de significância ($0,568 > 0,05$). Em outras palavras, não existe uma diferença significativa entre a quantidade de atributos faltantes entre as abordagens de seleção Geral e SeleTT e não há interação entre os fatores.

Figura 30 – Análise da média (porcentagem de informação faltantes na seleção), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Tabela 23 – Anova (Compleitude - atributos faltantes).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	1,266	1,000	1,266	1,263	0,266	4,013
Colunas	3,172	3,000	1,057	1,055	0,376	2,769
Interações	2,047	3,000	0,682	0,681	0,568	2,769
Dentro	56,125	56,000	1,002			
Total	62,609	63,000				

4.4.1.3 Efetividade - Porcentagem de escolhas corretas

- **Análise de outliers**

Os resultados do teste de análise de *outliers* apresentados na Figura 31 mostram que nenhum participante foi considerado *outlier* para o fator abordagem de seleção e apenas dois participantes (3%) foram considerados *outliers* para o fator tipo do projeto para a variável efetividade considerando a porcentagem de escolhas corretas pelos participantes do estudo.

Figura 31 – *Box plots* para a variável efetividade, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

- **Análise de variância e teste de hipóteses**

A Figura 32 mostra a distribuição das médias entre os fatores (tipo de abordagem de seleção e tipo de projeto de software) para a variável efetividade em relação à porcentagem de escolhas corretas. Na Tabela 24 são apresentados os resultados do teste de análise de variância para essa variável. A análise dos resultados do teste estatístico ANOVA para a amostra resultou no valor de F maior que o valor de F crítico ($21,000 > 4,013$) e valor-P menor que o nível de significância ($0,000 < 0,05$) considerando o tipo de abordagem de seleção utilizada, então podemos rejeitar a hipótese nula H.0.2 (Tabela 14) e afirmar que existe uma diferença significativa entre a efetividade das diferentes abordagens de seleção. Para o fator tipo de projeto de software, o valor de F é também maior que o valor F crítico ($4,559 > 2,769$) e valor-P menor que o nível de significância ($0,006 < 0,05$) mostrando que existe uma diferença significativa entre os resultados quando se usam tipos projetos de software diferentes durante a seleção das técnicas. Analisando a interação entre os fatores, é possível concluir que um fator interfere nos resultados do outro fator pois o valor de F é também maior que o valor F crítico ($5,704 > 2,769$) e valor-P é menor que o nível de significância ($0,006 < 0,05$). Em outras palavras, pode-se afirmar que existe uma diferença significativa entre as abordagens usadas no processo de seleção, porém o tipo de projeto de software pode influenciar nos resultados para cada uma das abordagens.

Como a hipótese nula H.0.2 foi rejeitada com significância estatística, foi avaliada a possibilidade de aceitação da hipótese alternativa H.2.1. (Tabela 14). O resultado do Teste-

Figura 32 – Análise da média (porcentagem de escolhas corretas), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Tabela 24 – Anova (Efetividade - porcentagem de escolhas corretas).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	1,266	1,000	1,266	21,000	0,000	4,013
Colunas	0,824	3,000	0,275	4,559	0,006	2,769
Interações	1,031	3,000	0,344	5,704	0,002	2,769
Dentro	3,375	56,000	0,060			
Total	6,496	63,000				

t para a variável efetividade apresentado na Tabela 25 mostra que é possível rejeitar a hipótese nula H.0.2 que a efetividade das abordagens não difere entre as amostras e aceitar a hipótese alternativa H.2.1., que afirma que a abordagem SeleCTT melhora a efetividade do processo de seleção, pois o valor P é menor que o nível de 5% de significância do teste ($0,0001 < 0,05$) indicando uma diferença significativa das médias em função da hipótese alternativa H.2.1.

Tabela 25 – Teste-t (Efetividade - porcentagem de escolhas corretas).

Teste-t	SeleCTT	Geral
Média	0.5234375	0.2421875
Variância	0.106287802	0.062436996
Observações	32	32
Hipótese da diferença de média	0	
gl	58	
Stat t	3.87327258	
P(T<=t) uni-caudal	0.000137722	
t crítico uni-caudal	1.671552762	

4.4.1.4 Eficiência - Tempo de resposta

• Análise de outlier

Os resultados do teste de análise de *outlier* para a variável eficiência considerando o tempo de resposta, são apresentados na Figura 33. Esses resultados mostram que três participantes (5%) foram considerados *outliers* considerando o tipo de abordagem e também o tipo de projeto de software utilizado. Esses valores atípicos são considerados durante os demais testes estatísticos.

Figura 33 – *Box plots* para a variável eficiência relacionada ao tempo de resposta, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

• Análise de variância e teste de hipóteses

A Figura 34 ilustra a distribuição das médias para a variável eficiência em relação ao tempo gasto para a seleção da técnica de teste. Na Tabela 26 são apresentados os resultados do teste de análise de variância para essa variável. A análise dos resultados do teste estatístico ANOVA para essa amostra resultou no valor de F menor que o valor de F crítico para o tipo de abordagem de seleção ($0,292 < 4,013$) e para o tipo de projeto de software ($1,693 < 2,769$) respectivamente, o valor-P também foi maior que o nível de significância para os dois fatores ($0,590 > 0,05$) e ($0,179 > 0,05$). Então não podemos rejeitar a hipótese nula $H_{0.3}$ (Tabela 14), ou seja, não há diferença significativa na eficiência das abordagens (SeleCTT e Geral) considerando o tempo gasto pelos participantes durante o processo de seleção. Porém, nessa amostra existe um efeito de interação de valor-P menor que o nível de significância ($0,012 < 0,05$), o que mostra que existe uma interação entre os fatores, ou seja, que o tipo abordagem utilizada é também influenciada pelo tipo do projeto de software usado durante a seleção.

Figura 34 – Análise da média (tempo de resposta em minutos), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Tabela 26 – Anova (Eficiência - tempo de resposta).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	19,141	1,000	19,141	0,294	0,590	4,013
Colunas	330,922	3,000	110,307	1,693	0,179	2,769
Interações	779,047	3,000	259,682	3,986	0,012	2,769
Dentro	3648,125	56,000	65,145			
Total	4777,234	63,000				

4.4.1.5 Eficiência - Porcentagem de informações usadas/porcentagem de escolhas corretas

• Análise de outliers

Os resultados do teste de análise de *outliers* para a variável eficiência considerando a porcentagem de informações usadas pela porcentagem de escolhas corretas são apresentados na Figura 35. Os resultados mostram que cinco participantes (8%) foram considerados *outliers* para o fator tipo de abordagem de seleção e quatro participantes (6%) foram considerados *outliers* para o tipo de projeto de software (Banco, Estacionamento e Firewall). Esses valores são mantidos durante os teste estatísticos.

Figura 35 – Box plots para a porcentagem de informações usadas/porcentagem de escolhas corretas, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

• Análise da variância e teste de hipóteses

A distribuição das médias para a variável eficiência considerando a porcentagem de informações usadas pela porcentagem de escolhas corretas são apresentadas na Figura 36. Na Tabela 27 são apresentados os resultados de análise de variância dessas médias. Os valores obtidos com o teste ANOVA mostram que a hipótese nula H.0.3 (Tabela 14) pode ser rejeitada pois o valor de F é maior que o F crítico ($17,611 > 4,013$) e o valor-P é menor que o nível de significância ($0,000 < 0,05$), ou seja existe diferença entre as médias, o que mostra que a eficiência depende do tipo de abordagem de seleção utilizada. A análise do fator tipo de projeto não mostrou resultados significativos de diferença entre as médias e a análise da interferência entre os fatores também não comprova que um fator interfere nos resultados do outro.

Figura 36 – Análise da eficiência (porcentagem de informações usadas/porcentagem de escolhas corretas).

Tabela 27 – Anova (Eficiência - porcentagem de informações usadas/porcentagem de escolhas corretas).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	12,886	1,000	12,886	17,611	0,000	4,013
Colunas	3,254	3,000	1,085	1,483	0,229	2,769
Interações	5,005	3,000	1,668	2,280	0,089	2,769
Dentro	40,974	56,000	0,732			
Total	62,119	63,000				

Após rejeitar a hipótese nula, foi avaliado se é possível aceitar a hipótese alternativa H.3.1 que afirma que a abordagem SeleCTT melhora a eficiência do processo de seleção. Como mostram os resultados do Teste-t na Tabela 28 o valor de t-crítico é menor que o valor de t ($1,674 < 4,028$) e o valor-P menor que o nível de significância ($0,000 < 0,05$) portanto, podemos aceitar a hipótese alternativa H.3.1, ou seja, a eficiência considerando a porcentagem de informações usadas/porcentagem de escolhas corretas é maior usando a abordagem SeleCTT em contrapartida ao uso da abordagem Geral.

Tabela 28 – Teste-t (Eficiência - porcentagem de informações usadas/porcentagem de escolhas corretas).

Teste-t	SeleCTT	Geral
Média	1.486290684	0.588876
Variância	1.128294346	0.459882
Observações	32	32
Hipótese da diferença de média	0	
gl	53	
Stat t	4.028270158	
P(T<=t) uni-caudal	9.01369E-05	
t crítico uni-caudal	1.674116237	

4.4.1.6 Eficiência - Porcentagem de escolhas corretas/tempo de seleção

• Análise de outliers

Os resultados do teste de análise de outliers apresentados na Figura 37 mostram que seis participantes (9%) foram considerados outliers considerando o tipo de abordagem de seleção e quatro (6%) para o tipo de projeto em relação a variável eficiência considerando a porcentagem de escolhas corretas pelo tempo de seleção.

Figura 37 – Box plots para a porcentagem de escolhas corretas/tempo de seleção, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

• Análise da variância e teste de hipóteses

A Figura 38 ilustra a distribuição das médias da variável eficiência para a métrica porcentagem de escolhas corretas/tempo de seleção. Os resultados da análise de variância para essa variável são apresentados na Tabela 29. O valor F é menor que o F-crítico tanto para o tipo de abordagem de seleção ($3,166 < 4,013$) quanto para o tipo de projeto de software ($2,357 < 2,769$) e o valor-P é maior que o intervalo de significância para o fator abordagem de seleção ($0,081 > 0,05$) e para o tipo de projeto ($0,081 > 0,05$). Esses resultados mostram que não existe diferença significativa entre as médias tanto considerando a abordagem de seleção quanto o tipo do projeto. A análise da interação entre os fatores também mostra que os fatores não são influenciados entre si, pois F é menor que F crítico ($2,207 < 2,769$) e valor-P é maior que o nível de significância ($0,097 > 0,05$). Sendo assim não é possível rejeitar a hipótese H.0.3 (Tabela 14) de que a eficiência do processo de seleção é independente da abordagem de seleção usada (considerando a porcentagem de escolhas corretas/tempo de seleção).

Figura 38 – Análise da média (porcentagem de escolhas corretas/tempo de resposta), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Tabela 29 – Anova (Eficiência - porcentagem de escolhas corretas/tempo de resposta).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	0,044	1,000	0,044	3,166	0,081	4,013
Colunas	0,099	3,000	0,033	2,357	0,081	2,769
Interações	0,092	3,000	0,031	2,207	0,097	2,769
Dentro	0,781	56,000	0,014			
Total	1,016	63,000				

4.4.1.7 Usabilidade - Quantidade de problemas reportados

• Análise de outliers

Os resultados do teste de análise de *outliers* (Figura 39) mostram que doze participantes (19%) foram considerados *outliers* para o fator abordagem de seleção e nove participantes (14%) foram considerados *outliers* considerando o tipo do projeto de software para a variável usabilidade, considerando a porcentagem de problemas reportados pelos participantes. A grande quantidade de valores atípicos para essa variável está relacionada ao fato que essa questão era aberta (livre) no formulário usado pelos participantes e não existia limite de problemas que poderiam ser citados. Esses valores atípicos não foram eliminados pois todos os problemas reportados são considerados relevantes para as análises.

Figura 39 – Box plots para a variável usabilidade, considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

• Análise da variância e teste de hipóteses

Os resultados obtidos pela análise das médias conforme apresentados na Figura 40 e análise de variância na Tabela 30 mostram que não existe diferença significativa entre a usabilidade das abordagens de seleção considerando a porcentagem de problemas reportados pelos participantes em relação ao tipo de abordagem de seleção usada, pois o valor de F é menor que o valor de F crítico ($1,101 < 4,013$) e o valor-P é maior que o nível de significância ($0,299 > 0,05$). Assim não é possível rejeitar a hipótese nula H.0.4 (Tabela 14) que afirma que a usabilidade do processo de seleção é independente da abordagem de seleção utilizada. Porém a análise do tipo do projeto de software apresenta diferenças significativas para esta variável considerando o valor F é maior que o F crítico ($2,979 > 2,769$) e o valor-P menor que 5% ($0,039 < 0,05$), mostrando que há diferença entre as médias para esse fator. A análise da interação mostra que não há influência entre os fatores.

Tabela 30 – Anova (Usabilidade - quantidade de problemas reportados).

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Amostra	0,391	1,000	0,391	1,101	0,299	4,013
Colunas	3,172	3,000	1,057	2,979	0,039	2,769
Interações	1,422	3,000	0,474	1,335	0,272	2,769
Dentro	19,875	56,000	0,355			
Total	24,859	63,000

Figura 40 – Análise da média (quantidade de problemas reportados), considerando a abordagem de seleção (esquerda) e o tipo do projeto de software (direita).

Na Tabela 31 é resumida a análise de dados para as variáveis avaliadas no experimento, considerando cada fator e os resultados da análise de variância em (valor-P) a fim de rejeitar a hipótese nula de que não existe diferença entre as abordagens de seleção de técnicas de teste.

Tabela 31 – Síntese dos resultados da análise quantitativa.

Variável	Métrica	Fator	Outlier	Média	Valor-P	Influência
Compleitude	% informações usadas	Abordagem	Não	G: 0,42 S: 0,38	0,008	Sim
		Projeto	Não	B: 0,45 Es: 0,46 Eq: 0,38 F: 0,43	0,318	Não
		Interação	-	-	0,562	Não
	% informações faltantes	Abordagem	Sim	G: 1,22 S: 0,94	0,266	Não
		Projeto	Sim	B: 1,00 Es: 1,31 Eq: 1,25 F: 0,75	0,376	Não
		Interação	-	-	0,568	Não
Efetividade	% acertos	Abordagem	Não	G: 0,24 S: 0,52	0,000	Sim
		Projeto	Sim	B: 0,58 Es: 0,30 Eq: 0,33 F: 0,33	0,006	Sim
		Interação	-	-	0,002	Sim
Eficiência	Tempo	Abordagem	Sim	G: 14,94 S: 13,84	0,590	Não
		Projeto	Sim	B: 11,44 Es: 15,81 Eq: 17,25 F: 13,06	0,179	Não
		Interação	-	-	0,012	Sim
	%acertos / %informação usada	Abordagem	Sim	G: 0,59 S: 1,49	0,000	Sim
		Projeto	Sim	B: 1,41 Es: 0,84 Eq: 1,03 F: 0,87	0,229	Não
		Interação	-	-	0,089	Não
	%acertos / tempo	Abordagem	Sim	G: 0,02 S: 0,08	0,081	Não
		Projeto	Sim	B: 0,12 Es: 0,02 Eq: 0,03 F: 0,04	0,081	Não
		Interação	-	-	0,097	Não
Usabilidade	Quantidade de problemas reportados	Abordagem	Sim	G: 1,03 S: 0,88	0,299	Não
		Projeto	Sim	B: 1,19 Es: 0,88 Eq: 1,13 F: 0,63	0,039	Sim
		Interação	-	-	0,272	Não

4.4.2 Análise qualitativa dos dados

Nesta seção são sumarizados os dados qualitativos para cada uma das variáveis analisadas de maneira quantitativa na Seção 4.4.1.

4.4.2.1 Atributos de caracterização usados durante a seleção

A análise da Tabela 32 mostra que o atributo "tipo de erro revelado" é o mais usado durante a seleção, seguido pelo "tipo de saída gerada". Os atributos relacionados ao contexto da programação concorrente como "contexto/objetivo do software", "tipo de análise", "paradigma de interação entre os processos", também estão entre os mais usados quando se aplica a abordagem SeleCTT.

De maneira geral, pode-se concluir que todos os atributos foram usados durante o processo de seleção de técnicas de teste, o que é um bom resultado e demonstra que o esquema de caracterização definido é realmente útil e auxilia o processo de seleção.

Tabela 32 – Porcentagem de uso dos atributos de caracterização para a seleção.

Atributo	Geral	SeleCTT
Plataforma de execução	38,46%	52,27%
Contexto/Objetivo	66,67%	68,18%
Linguagem de programação	43,59%	40,91%
Técnica de teste	-	65,91%
Critério de geração dos casos de teste	-	15,91%
Nível de teste	-	59,09%
Mecanismo de geração de <i>interleavings</i>	-	4,55%
Entradas requeridas	-	50,00%
Saídas (Resultados gerados)	87,18%	75,00%
Características de qualidade	69,23%	59,09%
Tipo de estudo (avaliação empírica)	15,38%	11,36%
Tipo de análise	46,15%	54,55%
Paradigma concorrente de iteração de processos	-	52,27%
Mecanismo de <i>replay</i>	-	2,27%
Representação do programa	-	6,82%
Técnica de redução do espaço de estados	-	2,27%
Tipos de erros revelados (erros concorrentes)	92,31%	88,64%
Ferramenta de apoio	30,77%	27,27%
Custo associado à ferramenta	-	38,64%
Plataforma em que a ferramenta opera	-	22,73%
Compreensibilidade	15,38%	-
Dependências	47,55%	-

4.4.2.2 Informações faltantes para a caracterização das técnicas

A Tabela 33 apresenta as informações listadas pelos participantes do experimento como relevantes e que não foram encontradas nas abordagens de seleção (Geral ou SeleCTT).

É importante ressaltar que alguns dos participantes que utilizaram a abordagem Geral listaram a falta de uma métrica para avaliar cada atributo a fim de definir sua importância para o processo de seleção. Esse método é implementado na abordagem SeleCTT, por meio da definição de pesos aos atributos a fim de calcular sua adequação ao projeto.

Tabela 33 – Informações faltantes para cada abordagem de seleção.

Informações faltantes	Geral	SeleCTT
Definição da arquitetura usada no projeto	-	1
Conhecimento necessário da equipe de teste	2	1
Definição mais detalhada de cada atributo	4	8
Descrição detalhada do método usado para seleção das técnicas de teste	-	1
Descrição do esforço necessário para empregar a técnica (tempo, custo)	11	4
Descrição mais detalhada das ferramentas de apoio	3	3
Descrição mais detalhada do projeto de software	-	1
Descrição sobre o impacto da falta de um critério durante a seleção	-	1
Especificação dos diagramas disponíveis para a representação do programa	-	1
Falta de compatibilidade entre os atributos do projeto e das técnicas	-	3
Falta de informação sobre a efetividade de cada técnica	1	-
Falta de informações sobre os recursos disponíveis para o desenvolvimento dos testes (custo e pessoal)	3	-
Falta de um método de automatização da comparação entre os atributos para tornar a seleção mais rápida, consequentemente a decisão também	2	2
Falta de uma métrica para definir quais atributos são mais relevantes na escolha da técnica	6	-
Limitações de cada técnica (pontos fracos)	1	-
Mais informações sobre a aplicabilidade das técnicas de teste	2	-
Referências de outros projetos em que as técnicas foram usadas	-	1
Requisitos necessários para execução das técnicas	1	1
Tipo de entradas aceitáveis para cada técnica	3	-
Um exemplo de aplicação da técnica	-	1
Uma métrica da qualidade do artigo onde essa técnica foi inicialmente publicada	-	1

Outro ponto, também relevante, é o relato de alguns participantes em ambas as abordagens, sobre a falta de um mecanismo que automatize o processo de seleção utilizando as informações fornecidas. A abordagem de seleção SeleCTT foi implementada como uma ferramenta web que automatiza o processo de seleção oferecendo uma lista de recomendação das técnicas de teste ordenadas de acordo com o cálculo de sua adequabilidade ao projeto. A ferramenta proposta é descrita em detalhes na Seção 3.6. Durante o processo de seleção não foi utilizada a ferramenta SeleCTT, pois o objetivo deste estudo foi avaliar a eficiência da abordagem de teste e não os benefícios da automatização do processo proporcionados pela ferramenta.

4.4.2.3 Acertos por abordagem de seleção

A quantidade de acertos para a abordagem geral foi 24% já para a abordagem SeleCTT foi de 52% em média. A análise das médias juntamente com os resultados da avaliação quantitativa apresentados na Seção 4.4.1.3 mostram que a abordagem SeleCTT é mais efetiva durante o processo de escolha de uma técnica de teste de software para um projeto de software concorrente considerando a porcentagem de acertos obtidos pelos participantes.

4.4.2.4 Análise do tempo gasto com a seleção

O tempo de resposta não teve variação significativa mesmo a quantidade de atributos usados sendo maior para a abordagem SeleCTT. Esse resultado é bastante positivo do ponto de

vista de que a abordagem SeleCTT possui um maior número de atributos a serem analisados e mesmo assim o tempo de resposta dos participantes foi similar ao da abordagem Geral.

4.4.2.5 Problemas reportados pelos participantes

A Tabela 34 lista os principais problemas/dúvidas reportadas pelos participantes durante a condução do estudo experimental.

Tabela 34 – Problemas/dúvidas reportados pelos participantes durante a condução do estudo.

Problemas reportados	Geral	SeleCTT
Falta de informação sobre o custo da ferramenta de teste	2	-
Falta de uma explicação detalhada de como foram definidos os pesos	-	3
Falta de compatibilidade entre os atributos do projeto e das técnicas	1	-
Falta de conhecimento amplo dos participantes sobre o assunto	3	3
Dúvida sobre qual a flexibilidade do não atendimento de algumas especificações	7	8
Não conhecimento dos participantes sobre as ferramentas de apoio	-	2
Não entendimento dos participantes sobre o processo de seleção	-	1
Necessidade da definição mais detalhada de certos atributos	3	6
Problemas com a maneira como o experimento foi instrumentado	-	1
Dificuldade em estipular qual a duração estimada de cada técnica	1	-
Falta de compreensão sobre a hierarquia de importância das informações para a escolha da técnica de teste	-	4
Dificuldade em definir a prioridade dos requisitos e o peso a ser atribuído a cada um deles	14	-
Dúvida sobre quão pertinentes são os fatores de compatibilidade do teste com a aplicação do ponto de vista da integração e do ponto de vista de negócios	1	-
Dificuldade em definir se haveria tempo hábil para aplicar mais de uma técnica de teste, pois nenhum deles supria completamente a necessidade do sistema	1	-

Dentre os principais problemas listados pelos participantes que utilizaram a abordagem Geral de seleção está a dificuldade em definir a prioridade dos atributos e estabelecer uma métrica (peso) para cada um deles. Esse fator é suprido pela abordagem SeleCTT que disponibiliza um esquema de caracterização com pesos para cada um dos atributos apresentados, que facilita o processo de ordenação das técnicas quanto sua adequabilidade ao projeto.

O principal problema relacionado a abordagem SeleCTT listado pelos participantes do estudo é a dificuldade de estabelecer uma estratégia de seleção quando algumas especificações não são atendidas. Os participantes relatam não entender qual a flexibilidade do não atendimento de alguns atributos durante o processo de seleção. Esse problema pode estar relacionado ao fato do projeto usado ser um projeto exemplo e os participantes não serem desenvolvedores reais, no caso de um projeto no qual o profissional de teste seleciona a técnica a ser usada no projeto que está desenvolvendo, a priorização de atributos é intrínseca às necessidades do projeto e não deve causar esse tipo de dúvida à equipe de teste.

4.4.2.6 Satisfação do usuário

- **Capacidade da abordagem facilitar o processo de seleção**

A Tabela 35 mostra a opinião dos participantes quando questionados sobre a facilidade do uso da abordagem proposta durante o processo de seleção de técnicas de teste de software. Na tabela foram agregadas as respostas similares dos participantes, a fim de tornar as informações mais intuitivas ao leitor.

Tabela 35 – Satisfação do usuário - facilidade da seleção.

A abordagem facilita a seleção de técnicas de teste? Por quê?	Respostas
Sim, pois permite avaliar separadamente cada requisito do projeto e definir uma pontuação de acordo com o peso de cada atributo.	2
Sim, pois torna possível a comparação simultânea de várias técnicas de modo mais objetivo.	4
Sim, porque ele apresenta bem quais são as necessidades do projeto, quais são os critérios de escolha, seus pesos e o que cada técnica oferece.	14
Sim, deixa conciso e claro o que deve ser priorizado.	6
Sim, pois facilita a visualização das exigências do usuário.	3
Não, pois é necessário ter uma ideia prática de como as técnicas funcionam.	1
Não facilita a tomada de decisão a não ser sob o contexto de que, quem está sendo responsável por esta decisão já possua amplo conhecimento prévio das técnicas dadas como opções.	2

A maioria dos participantes, 91% consideram que a abordagem SeleCTT facilita o processo de seleção, por diversos motivos: permitir estabelecer uma pontuação por meio de pesos para cada atributo, permitir a comparação simultânea de várias técnicas, definir de maneira simples as necessidade do projeto, tornar clara e concisa a informação que deve ser priorizada e facilitar a visualização dos requisitos do usuário.

Poucos participantes afirmaram que a abordagem não facilita a seleção, 9% alegaram que por meio da abordagem não é possível entender o funcionamento prático das técnicas e que a abordagem só pode ser útil se o conhecimento prévio da equipe de teste sobre as técnicas for muito alto. Embora pertinente o ponto de vista dos participantes, destacamos que a abordagem visa auxiliar o processo de tomada de decisão oferecendo um guia inicial na identificação de uma nova técnica de teste. A decisão da equipe em adotar a técnica sugerida implica no estudo detalhado da técnica (a referência sobre a técnica é oferecida na ferramenta) para a decisão final sobre sua implantação.

- **Simplicidade da abordagem**

A Figura 41 ilustra a opinião dos participantes em julgar a abordagem de fácil entendimento (sim) ou de difícil compreensão (não). A grande maioria dos participantes, 94% declararam que a abordagem é fácil de ser entendido, enquanto apenas 6% julgaram ter alguma dificuldade na compreensão da abordagem. As principais dificuldades são relatadas na Tabela 34.

- **Possibilidade de uso da abordagem na indústria**

Os participantes foram questionados sobre a possibilidade do uso da abordagem SeleCTT na indústria. A Figura 41 ilustra a posição de cada participante sobre a questão.

A maioria dos participantes (84%) declaram ser possível o uso da abordagem na indústria, principalmente para a redução do tempo de seleção da técnica e aumento de possibilidade de técnicas a serem implantadas.

Alguns participantes são apreensivos quanto ao uso da abordagem em ambiente industrial (16%), principalmente por julgarem necessário o conhecimento amplo sobre o assunto e conhecimento adicional para entendimento do processo de seleção, além do conhecimento prévio da técnica de teste que será selecionada, caso a equipe de teste não esteja familiarizada com a mesma.

Figura 41 – Satisfação do usuário - simplicidade da abordagem e possível uso na indústria.

- **Vantagens da abordagem**

Na Tabela 36 são apresentadas as principais vantagens oferecidas pela abordagem SeleCTT, listadas pelos participantes do experimento. Essas vantagens estão relacionadas à sumarização e apresentação intuitiva das informações, o que torna o processo de seleção rápido e dinâmico. A especificação de atributos para caracterização das técnicas prioriza informações mais importantes (definidas com base em um *survey* com especialistas na área) o que segundo os participantes torna o processo mais objetivo, eliminando informações pouco relevantes ao processo, tornando a comparação mais eficiente. Por ser um processo sistemático e simples não é exigido conhecimento prévio aprofundado do usuário para gerar a lista ordenada de técnicas adequadas ao projeto de software. Porém vale ressaltar que a escolha final da técnica a ser aplicada no projeto de software em desenvolvimento cabe à equipe de teste que deve avaliar mais profundamente se as características das técnicas recomendadas pela abordagem se adequem as restrições de seu projeto de software.

Tabela 36 – Satisfação do usuário - vantagens do uso da abordagem.

Quais as vantagens da abordagem de seleção SeleCTT?	Respostas
O conhecimento prévio exigido é muito menor.	7
O tempo de absorver as diferentes literaturas é menor.	3
O processo se torna mais ágil e objetivo, pois não é precisamos buscar informações que são relevantes no meio de informações consideradas irrelevantes.	6
É mais simples, rápido, dinâmico e intuitivo.	14
Padronização das informações o que torna a comparação mais eficiente.	2

• Desvantagens da abordagem

A Tabela 37 lista as principais desvantagens reportadas pelos participantes sobre a abordagem de seleção SeleCTT. Dentre elas estão:

Tabela 37 – Satisfação do usuário - desvantagens da abordagem.

Quais as desvantagens da abordagem de seleção SeleCTT?	Respostas
Problemas que surgem no projeto como um todo podem ser deixados de lado.	1
Pode omitir informações mais profundas e úteis para o avaliador	16
Por vezes não há uma equivalência direta entre os atributos e, portanto, é preciso avaliar se é preciso considerá-lo ou não.	2
Não detalha informações sobre ferramentas e outros atributos como mecanismos de <i>replay</i> , por exemplo.	3
Não possui uma relação clara de prós e contras.	3
Para o uso, é necessário ter um conhecimento amplo da área já que os termos são colocados de modo mais direto. Seria interessante algum modelo de visualização como fluxograma ou algo do tipo.	1
Muitas variáveis que não tem grande impacto são consideradas durante a seleção.	1
Nenhuma.	5

- Omissão de informações que podem ser úteis ao avaliador e falta de detalhamento de certos atributos

Esse problema é difícil de ser tratado, visto que a abordagem é criada justamente para tentar sumarizar as informações mais importantes durante o processo de seleção. Porém a referência sobre a literatura técnica, que define a técnica, sempre é fornecida, possibilitando que maiores informações possam ser consultadas.

- Não apresentar os prós e contras de cada técnica

A definição de prós e contras é bastante subjetiva, portanto, nesse estudo optou-se pela utilização de atributos mais objetivos que sejam mais facilmente identificados e comparados durante o processo de seleção. Além disso, os prós e contras de cada técnica estão relacionados aos requisitos do projeto em desenvolvimento que não são previsíveis anteriormente ao fornecimento dessas informações pelo usuário.

- Não apresentar equivalência direta entre os atributos

Nem sempre há relação de total adequação entre um projeto e uma técnica então a existência de atributos diferentes não pode ser evitada quando se instancia uma abordagem aplicável a diferentes técnicas de teste.

- Não abrange problemas que podem surgir mais tarde no desenvolvimento do projeto. Tendo em vista que a seleção da técnica de teste é definida após a definição dos requisitos do projeto é esperado que não ocorram grandes mudanças nos atributos.
- É necessário ter conhecimento prévio da área para entendimento dos termos usados. Quanto ao conhecimento prévio dos participantes, é esperado que a equipe de desenvolvimento auxilie a equipe de teste e possa sanar qualquer problema com a terminologia empregada na abordagem.
- Algumas variáveis com pouco impacto parecem ser pouco relevantes para a abordagem.

Um *survey* foi conduzido com especialistas na área, que tem trabalhado propondo e utilizando técnicas de teste de software concorrente tanto da academia quanto na indústria, os pesos dos atributos foram definidos com base na opinião desses especialistas. Porém, maiores validações podem ser conduzidas a fim de reavaliar os atributos do esquema.

- **Melhorias a serem implementadas na abordagem**

A Tabela 38 apresenta as sugestões de melhorias listadas pelos participantes do estudo. Como principal melhoria a ser adicionada a abordagem está o desenvolvimento de uma plataforma para automatizar a escolha das técnicas usando o esquema de caracterização e os pesos definidos. Foi implementada uma infraestrutura computacional para este fim que é apresentada na Seção 3.6. Esse resultado mostra a importância da ferramenta para automatizar a aplicação da abordagem e facilitar o processo de seleção.

Relacionada a proposta anterior, outra sugestão levantada pelos participantes é a implantação de um esquema para melhor visualização dos dados, o que é também oferecido pela ferramenta, com a disponibilização de gráficos e elementos visuais para exposição dos resultados.

Há também a sugestão de possibilitar ao usuário a opção de setar os pesos dos atributos conforme julgar mais importante ao seu processo de desenvolvimento. Atualmente os pesos do estudo estão definidos com base na opinião de especialistas na área que tem trabalhado ativamente com abordagens de teste e acreditamos que esse pode ser um bom ponto de partida para a seleção. Porém, essa sugestão é vista como uma possível melhoria e evolução da ferramenta.

Quanto ao fato dos participantes julgarem necessário um amplo conhecimento da equipe de teste sobre as técnicas de teste disponíveis, esse é justamente o ponto que a abordagem tenta atacar, oferecendo uma lista ordenada de possibilidades a partir da qual técnicas mais relevantes ao projeto podem ser identificadas (com base na comparação de atributos comuns) e posteriormente estudadas em profundidade e então adotadas pela equipe de teste.

Tabela 38 – Satisfação do usuário - sugestões de melhoria.

Você acredita que a abordagem possa ser melhorada para tornar seu uso mais simples? Como?	Respostas
Sim, com uma plataforma para automatizar a escolha da técnica se os dados forem discretizados, gerando uma pontuação de cada técnica para auxiliar a escolha da mesma.	9
Sim, expondo uma lista de prioridades sobre o que o cliente prioriza, dentro das especificações dadas.	3
Sim, exemplificar melhor o peso de cada atributo para a tomada de decisão.	5
Melhorar o modo de visualizar as informações. Embora já exista um grande grau de resumo nas informações, o modo de visualização usando várias tabelas ainda está muito poluído. Talvez um fluxograma ou algo mais visual tornaria o uso deste técnica mais prazerosa.	5
Não há necessidade de melhorias.	10

• Aprendizado obtido com o uso da abordagem

Os participantes foram questionados quanto ao conhecimento adquirido durante o uso da abordagem, dentre os principais podemos destacar:

- Identificar quais fatores são importantes na escolha das técnicas de teste de software.
- Mensurar a importância de cada atributo.
- Compreender quais características da programação concorrente afetam o teste de software.
- Selecionar informações importantes a fim de comparar as técnicas de teste de software.
- Priorizar a escolha de requisitos de acordo com sua importância.
- Usar um método sistemático para escolha da técnica de teste.

A Tabela 39 apresenta as principais lições aprendidas reportadas pelos participantes do estudo.

De uma maneira geral os resultados apresentados são bastante animadores, considerando que a maioria dos estudantes estavam se deparando com o problema da seleção de técnicas de teste pela primeira vez.

Os participantes, perceberam que existem características da programação concorrente que afetam o teste de software e que alguns fatores devem ser considerados durante a atividade de seleção de técnicas de teste. Eles puderam utilizar uma abordagem que oferecia uma maneira de mensurar a relevância desses atributos, utilizando pesos. Além disso, compreenderam a importância de uma abordagem sistemática para o processo de seleção de técnicas de teste.

Tabela 39 – Satisfação do usuário - lições aprendidas.

O que você aprendeu com a utilização da abordagem de seleção?	Respostas
Que a experiência de escolha da técnica é influenciada pela experiência do usuário com a abordagem, em que a forma como são exibidas as informações do projeto e das técnicas tem um impacto enorme na facilidade e agilidade do processo para o usuário.	1
Selecionar uma técnica para o teste de software não é uma atividade simples.	1
Identificar o que é relevante no software e na atividade de teste, considerando não apenas os requisitos a serem atendidos, mais também os pesos associados a eles.	2
Identificar características a serem consideradas para o teste de aplicações concorrentes orientado à indústria. Que não existe apenas uma forma de testar programas concorrentes e que se deve ter cuidado na escolha da técnica de teste.	2
Que projetos de código concorrente possuem algumas características que podem ser bem delimitadas e avaliadas na hora de se escolher uma técnica de teste.	1
Que escolher uma técnica adequada pode ser uma tarefa simples, basta organizar as características desejadas e proporcionadas pelas técnicas. A quantidade de critérios utilizados para auxiliar a escolha de uma técnica de teste é bem grande.	3
Que devem ser levados em consideração diversas etapas da produção de software, desde o modelo inicial, até linguagens de programação e S.O. utilizados.	4
Que existem muitos atributos a serem considerados para escolher uma técnica de teste. Fazer a comparação de diversas técnicas usando as informações mais importantes.	2
A comparar diferentes técnicas de maneira dinâmica juntamente com os atributos que deveriam atender em relação à aplicação.	1
A ter um olhar mais sistemático na escolha de uma técnica de teste de software.	1
Formas de comparar e decidir qual técnica de teste de software usar.	1
Que pode ser que nenhuma técnica se encaixe de forma completa aos requisitos do sistema e que prioridades deverão ser feitas.	3

4.4.3 Avaliação da validade

- **Ameaças à validade interna:**

- **Duplicação de resultados:** o experimento é executado em sala de aula, onde as carteiras ficam próximas umas das outras, além disso o experimento é conduzido online (onde os estudantes tem livre acesso à internet), assim é possível que eles compartilhem os resultados obtidos. A fim de evitar essa ameaça, os participantes são notificados no início do experimento que não é permitido o compartilhamento dos resultados entre si e que o objetivo do estudo é medir o esforço de cada um para execução da atividade e não a quantidade de respostas corretas obtidas.
- **Conhecimento prévio (experiência) do participante:** como os estudantes possuem nível de conhecimento variado, a sua habilidade para condução do experimento também varia e pode afetar os resultados. A fim de minimizar essa ameaça, os estudantes foram divididos em grupos de maneira randômica, assim cada grupo possui heterogeneidade de sujeitos.

- **Ameaças à validade externa:**

- **Experiência em seleção:** A grande maioria dos participantes não possuem experiência em seleção de técnicas de teste e software concorrente. O que pode resultar em vantagens para alguma das estratégias. Sendo assim, talvez o resultado não possa ser generalizado para todos participantes.

- **Projetos de software:** Os projetos usados no experimento podem não ser representativos do total de projetos de software existentes na área. Selecionamos quatro diferentes projetos a fim de representar diferentes projetos de software. Dois projetos são provenientes do estudo conduzido por [Vegas e Basili \(2005\)](#), possibilitando possíveis comparações de resultados. Um estudo é baseado em um projeto desenvolvido na academia para resolver um problema da indústria e um projeto é proveniente diretamente da indústria, esses projetos foram escolhidos a fim de contemplar projetos reais de desenvolvimento de software. Contudo, esse conjunto pode ainda assim, não ser representativo de todos os tipos de projetos de software concorrente existentes para esse contexto, o que pode tornar os resultados obtidos não generalizáveis para toda a população de projetos de software concorrente.
- **Técnicas de teste:** O conjunto de técnicas escolhidas para condução do estudo podem não ser representativas da população como um todo. A fim de contemplar a variedade de técnicas de teste existentes, foram selecionadas para o experimento técnicas de teste com diferentes atributos (paradigma de interação de processos, linguagem de programação, tipo de análise, níveis de teste, dentre outros atributos).

4.4.4 Discussão e Conclusão

A avaliação quantitativa apresentada na Seção 4.4.1 mostrou que a abordagem SeleCTT e a abordagem Geral apresentam comportamento similares (sem variação significativa) para a maioria das variáveis estudadas. O que mostra que de maneira geral as abordagens são equivalentes quanto a sua aplicabilidade. Porém, quando analisando a variável efetividade em relação à porcentagem de acertos obtidos pelos participantes, a abordagem SeleCTT se mostrou superior a abordagem Geral. O que ressalta a importância dos novos atributos relacionados à programação concorrente para a escolha da técnica adequada a um projeto de software.

Os resultados mostraram também que a abordagem SeleCTT melhora a eficiência do processo de seleção quando considerando a porcentagem de informações usadas e a porcentagem de escolhas corretas, utilizando uma quantidade menor de atributos e aumentando a quantidade de acertos obtidos pelos participantes.

Esses resultados evidenciam a relevância de uma abordagem específica para a área de programação concorrente a fim de melhorar o processo de seleção de técnicas de teste nesse contexto.

Na avaliação qualitativa ficam claras as vantagens e desvantagens da abordagem de seleção. São destacados alguns problemas encontrados durante o uso da abordagem e os participantes são questionados sobre o possível uso da abordagem na indústria.

Os resultados obtidos mostram que a abordagem SeleCTT é mais intuitiva e simples de ser entendida que a abordagem geral, apresentando uma métrica eficiente para avaliar a

adequação das técnicas de teste ao projeto de software, o que torna mais rápida sua aplicação e a comparação das técnicas mais eficiente.

Dentre as principais desvantagens estão a sumarização dos dados apresentados, a falta de uma relação direta entre os dados, a falta de conhecimento prévio da área pelo usuário, esse fator pode ter influenciado os outros fatores citados, pois os participantes do estudo possuíam baixo ou nenhum conhecimento sobre seleção de técnicas de teste.

Em geral os resultados mostraram que a abordagem SeleCTT pode trazer vantagens ao processo de seleção e parece ser recomendável sua aplicação na indústria. Maiores estudos nesse sentido devem ser conduzidos a fim de avaliar o real ganho no uso dessa abordagem em ambiente industrial. Planejamos o desenvolvimento de trabalhos acadêmicos, como estudos futuros nessa direção.

4.5 Considerações Finais

Este capítulo apresentou o estudo experimental conduzido a fim de avaliar a abordagem de seleção proposta neste trabalho em relação à abordagem de seleção para técnicas de teste de software no contexto geral (sem considerar as características da programação concorrente).

Foram utilizadas análises estatísticas e teste de hipóteses, bem como avaliações de variáveis qualitativas a fim de explorar a opinião dos participantes sobre o uso da abordagem.

Em síntese, os resultados são bastante promissores e mostram os ganhos obtidos e as vantagens associadas ao uso da nova abordagem de seleção de técnicas de teste definida para o contexto da programação concorrente.

Na próxima seção descrevemos além dos resultados do experimento, os resultados gerais alcançados por este trabalho, as contribuições científicas, tecnológicas e de formação de recursos. Além disso são listadas algumas limitações e alternativas a trabalhos futuros.

CONCLUSÕES

5.1 Considerações Iniciais

Este capítulo descreve os principais resultados obtidos com a realização do trabalho. As principais contribuições científicas, tecnológicas e de formação de recursos humanos abrangidas por esta pesquisa são apresentadas. Além disso, alguns fatores que limitam este trabalho e que podem ser levados em consideração na interpretação dos resultados obtidos são discutidos. Este capítulo apresenta também as produções científicas geradas no período de desenvolvimento do trabalho. Por fim, alternativas a trabalhos futuros são relacionadas, a fim de dar continuidade a esta pesquisa e prover maiores recursos na área de teste de software e programação concorrente, bem como no estudo de evidências nesse contexto.

5.2 Resultados Obtidos

Dentre os principais resultados alcançados por meio da condução deste trabalho, estão:

1. Uma visão geral do estado da arte da área de teste de software concorrente e dos estudos experimentais nesse contexto, por meio de uma lista de técnicas, ferramentas disponíveis, e análises de evidências nesse domínio.
2. A disponibilização de um corpo de conhecimento extensível e disponível online. A construção de um corpo de conhecimento web que armazena as técnicas de teste de software concorrentes identificadas por meio de um mapeamento sistemático da área e contribui para a disponibilização dessa informação à comunidade interessada.
3. A definição de um esquema de caracterização para o contexto da programação concorrente com o objetivo de identificar informações relevantes ao processo de seleção de técnicas de

- teste. São determinados atributos de caracterização e pesos são atribuídos a eles a fim de quantificar a adequação e relevância das técnicas para um projeto de software específico.
4. Um design de experimentos para guiar novos estudos empíricos na área e fornecer resultados mais comparáveis e agrupáveis em estudos secundários.
 5. A instanciação de uma abordagem de seleção de técnicas de teste auxiliando o processo de tomada de decisão durante o projeto de software.
 6. O desenvolvimento de uma infraestrutura computacional em parceria com um projeto de conclusão de curso, para automatizar o processo de seleção de técnicas visando eliminar o erro humano e diminuir o tempo gastos nessa atividade, implementando a abordagem proposta e oferecendo uma lista de recomendação de técnicas de teste à comunidade interessada.

5.3 Contribuições do Trabalho

Dentre as principais contribuições deste trabalho pode-se ressaltar:

- Desenvolvimento e disponibilização de um *framework* para auxiliar a avaliação sistemática de técnicas de teste para o contexto de programas concorrentes.
- Disponibilização do material gerado nas avaliações conduzidas por meio de uma interface web a fim de facilitar a realização de estudos secundários e comparáveis entre tecnologias de teste.
- Disponibilização de uma ferramenta web que possibilita a escolha sistemática de técnicas de teste de software disponíveis na literatura técnica, com base em atributos de caracterização pré-definidos.
- Disponibilização de um esquema de caracterização a fim de transformar os dados obtidos na literatura técnica em informações úteis a equipe de teste durante o processo de seleção de técnicas de teste, a fim de possibilitar a ampliação o uso do teste de software, hoje fortemente concentrado na academia, permitindo com isso a geração de novas demandas de teste, as quais realimentarão as pesquisas feitas formando-se um ciclo no desenvolvimento de novos conhecimentos.
- Prover subsídios para a transferência de tecnologia gerada na academia para centros desenvolvedores de software concorrente na academia e indústria.
- Possibilitar o desenvolvimento de aplicações concorrentes com maior qualidade, através da instanciação de estudos empíricos seguindo o design de experimentos proposto para avaliação de técnicas de VV&T para software concorrente, considerando diversos fatores como satisfação, custo, eficácia e eficiência.

- Desdobramento e expansão do projeto de mestrado conduzido anteriormente a este projeto de doutorado, possibilitando usar os conceitos da Engenharia de Software Empírica, não apenas como guia para condução de estudos de avaliação de novas tecnologias propostas, mas também oferecer informações relevantes ao processo de tomada de decisão no processo de desenvolvimento de software e ampliar o uso das tecnologias já existentes.
- Desenvolvimento acadêmico da aluna de doutorado, aumentando conhecimento em suas áreas de interesse, que envolvem teste de software, programação concorrente e a intersecção dessas áreas, teste de software concorrente, bem como estudo de uma nova área de domínio a de engenharia de software empírica e sua aplicação nesse contexto.
- Formação de recursos humanos em VV&T de aplicações concorrentes considerando o desenvolvimento de um projeto de conclusão de curso atrelado ao projeto em questão e possibilidade de desdobramento para projetos de iniciação científica, mestrado, doutorado e pós doutorado.

5.4 Limitações

Dentre os fatores que limitam este trabalho estão:

- As técnicas selecionadas para a composição do corpo de conhecimento são provenientes de mapeamento sistemático da literatura técnica, considerando bases de dados científicas e não todas as bases de informações disponíveis, como relatórios da indústria e páginas de empresas da área, o que pode limitar as técnicas interpretadas como estado da arte da área. Além disso, foram consideradas apenas técnicas de teste que apresentavam um estudo empírico para sua validação, pois esses estudos oferecem informações sobre os benefícios e limitações de cada técnica.
- A ferramenta desenvolvida não considera a possibilidade de escolha de mais de uma técnica para ser usada em conjunto a fim de encontrar mais erros. Como já conhecido as técnicas de teste são complementares e seu uso em conjunto pode trazer benefícios ao projeto de teste.
- A abordagem de seleção e a ferramenta não foram avaliadas na indústria, com uma equipe de teste desenvolvendo um projeto real de software. A avaliação conduzida nesse estudo utilizou alunos dos cursos de graduação em Bacharelado em Ciência da Computação com conhecimento em programação concorrente e engenharia de software. O uso de profissionais na indústria talvez gerasse diferentes resultados, considerando sua experiência prática na área. Novos estudos nessa direção devem ser conduzidos como trabalhos futuros.

5.5 Trabalhos Futuros

Como recomendações a trabalhos futuros é possível, entre outras atividades:

- Estender a ferramenta SeleCTT para que ela possibilite a seleção combinada de técnicas de teste a fim de oferecer maior cobertura ao processo de teste e uma maior qualidade do conjunto de teste. Um estudo alternativo também está relacionado ao uso de algoritmos baseados em inteligência artificial para a atualização dos pesos definidos para os atributos, a fim de adequar a seleção a um contexto de desenvolvimento de software específico.
- Prover um mecanismo de alimentação automática do corpo de conhecimento que possibilitaria manter o repositório sempre atualizado e eliminaria a possibilidade de inserção de erros humanos durante o processo, diminuindo também a força de trabalho dessa atividade. Um possível trabalho de mestrado, com alternativa a construção de uma ontologia para o domínio de teste de software concorrente e desenvolvimento do mecanismo de re-alimentação para o corpo de conhecimento.
- Desenvolver novos estudos empíricos em ambiente real de produção de software pode ajudar a avaliar de maneira mais realista a metodologia e a ferramenta. Esse tipo de estudo pode ser conduzido como parte de projetos de conclusão de curso ou mesmo de iniciação científica na área.
- Definir um projeto de pós doutorado relacionado a este projeto de doutorado, baseado no estabelecimento de parceria com a indústria a fim de validar esse trabalho e aprimorar a abordagem de seleção e a ferramenta desenvolvida. Considerando as demandas geradas pela indústria é possível tornar o trabalho mais próximo da realidade e mais amplamente aceito pela comunidade de pesquisa como alternativa a avaliação prática da validade de técnicas propostas para o contexto do software concorrente.

5.6 Produção Científica

As contribuições deste trabalho de doutorado deram origem a artigos submetidos e publicados em periódico e congressos, conforme apresentado a seguir:

- **Trabalho publicado em periódico**
 - MELO, S. M.; DE SOUZA, S. R. S. ; SARMANHO, F. S. ; SOUZA, P. S. L. *Contributions for the structural testing of multithreaded programs: coverage criteria, testing tool, and experimental evaluation*. SOFTWARE QUALITY JOURNAL, v. 1, p. 1-39, 2017.

- **Trabalho submetido a periódico**

- MELO, S. M.; CARVER, J. C. ; DE SOUZA, S. R. S. ; SOUZA, P. S. L. *Mapping Empirical Research in Concurrent Software Testing*. INFORMATION AND SOFTWARE TECHNOLOGY JOURNAL, v. 1, p. 1-29. (Submetido em 7 de Dezembro de 2017).

- **Trabalhos publicados em anais de congressos**

- MELO, S. M.; DE SOUZA, S. R. S. ; DE SOUZA, P. S. L. ; CARVER, J. C. . *How to test your concurrent software: an approach for the selection of testing techniques*. In: the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems - SEPS. 2017. Vancouver. ACM Press, New York. 2017. p. 42.
- MELO, S. M.; SOUZA, P. S. L. ; Souza, S. R. S. . *Towards an Empirical Study Design for Concurrent Software Testing*. In: 2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SEHPCCSE), Salt Lake City. 2016. p. 49-49.
- MELO, S. M.; SOUZA, S. R. S. ; SILVA, R. A. ; SOUZA, P. S. L. . *Concurrent software testing in practice: a catalog of tools*. In: Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation - A-TEST, Bergamo, 2015. p. 31-40.
- MELO, S. M.; SOUZA, S. R. S. . *Definição de um Framework para Avaliação Sistemática de Técnicas de Teste no Contexto de Programação Concorrente*. In: IV WORKSHOP DE TESES E DISSERTAÇÕES DO CBSOFT - WTDSOFT, Maceió, 2014. v. 2. p. 79-85.

- **Capítulo de livro publicado**

- SOUZA, S. R. S. ; SOUZA, P. L. S. ; MELO, S. M. ; SILVA, R. A. ; VERGILIO, S. R. . *Teste de Programas Concorrentes*. In: Marcio Eduardo Delamaro, José Carlos Maldonado, Mario Jino. (Org.). *Introdução ao teste de software*. 2ed. Rio de Janeiro: Elsevier, 2016, p. 261-296.

REFERÊNCIAS

ABADI, M.; FLANAGAN, C.; FREUND, S. N. Types for safe locking: Static race detection for java. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 28, n. 2, p. 207–255, mar. 2006. ISSN 0164-0925. Citado na página [52](#).

AHSANULLAH, M.; KIBRIA, B. M. G.; SHAKIL, M. Characterizations of student's t distribution. In: **Normal and Student's t Distributions and Their Applications**. Paris: Atlantis Press, 2014. p. 129–141. ISBN 978-94-6239-061-4. Citado na página [114](#).

AICHERNIG, B.; GRIESMAYER, A.; SCHLATTE, R.; STAM, A. Modeling and testing multi-threaded asynchronous systems with Creol. **Electronic Notes in Theoretical Computer Science**, v. 243, p. 3–14, 2009. Citado na página [27](#).

ALMASI, G. S.; GOTTLIEB, A. **Highly parallel computing**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1. Citado nas páginas [26](#) e [47](#).

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2008. I-XXII, 1-322 p. ISBN 978-0-521-88038-1. Citado nas páginas [25](#) e [35](#).

AMTOFT, T.; NIELSON, H. R.; NIELSON, F. Behavior analysis for validating communication patterns. **International Journal on Software Tools for Technology Transfer**, Springer-Verlag, v. 2, n. 1, p. 13–28, 1998. Citado na página [52](#).

ANDREWS, G. R. **Concurrent programming: principles and practice**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991. Citado nas páginas [48](#) e [49](#).

ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurrency and Computation: Practice and Experience**, John Wiley and Sons Ltd., Chichester, UK, p. 1572–1611, 2015. ISSN 1532-0634. Citado nas páginas [26](#), [28](#), [29](#), [53](#), [57](#), [58](#), [62](#) e [81](#).

BALL, T.; BURCKHARDT, S.; COONS, K. E.; MUSUVATHI, M.; QADEER, S. **Preemption Sealing for Efficient Concurrency Testing**. [S.l.], 2009. Citado nas páginas [27](#) e [52](#).

BARBOSA, E. F.; CHAIM, M. L.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Teste estrutural. In: **José Carlos Maldonado; Mario Jino; Márcio Eduardo Delamaro. (Org.). Introdução ao Teste de Software**. 2. ed. Rio de Janeiro, RJ: Elsevier Ltda, 2016. cap. 4, p. 59–87. Citado nas páginas [38](#) e [41](#).

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the determination of sufficient mutant operators for C. In: **Software Testing, Verification Reliability**. [S.l.: s.n.], 2001. v. 11, n. 2, p. 113–136. Citado na página [38](#).

BASILI, V. Quantitative evaluation of software methodology. In: **Proceedings of the First Pan-Pacific Computer Conference**. [S.l.: s.n.], 1985. Citado nas páginas [62](#), [70](#) e [81](#).

BASILI, V. R. **Software Modeling and Measurement: The Goal/Question/Metric Paradigm**. College Park, MD, USA, 1992. Citado na página 71.

BASILI, V. R.; ROMBACH, H. D.; SCHNEIDER, K.; KITCHENHAM, B. A.; PFAHL, D.; SELBY, R. W. (Ed.). **Empirical Software Engineering Issues. Critical Assessment and Future Directions, International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers**, v. 4336 de **Lecture Notes in Computer Science**, (Lecture Notes in Computer Science, v. 4336). [S.l.]: Springer, 2007. ISBN 978-3-540-71300-5. Citado nas páginas 62 e 71.

BASILI, V. R.; SHULL, F.; LANUBILE, F. Building knowledge through families of experiments. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 25, n. 4, p. 456–473, jul. 1999. ISSN 0098-5589. Citado nas páginas 28, 57 e 71.

BECHINI, A.; CUTAJAR, J.; PRETE, C. A tool for testing of parallel and distributed programs in message-passing environments. In: **Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean**. [S.l.: s.n.], 1998. v. 2, p. 1308–1312 vol.2. Citado na página 52.

BEIZER, B. **Software testing techniques**. 2. ed. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0. Citado na página 35.

BELLI, F.; BUDNIK, C. J.; HOLLMANN, A.; TUGLULAR, T.; WONG, W. E. Model-based mutation testing - approach and case studies. **Sci. Comput. Program.**, v. 120, p. 25–48, 2016. Disponível em: <<https://doi.org/10.1016/j.scico.2016.01.003>>. Citado na página 38.

BIANCHI, F. A.; MARGARA, A.; PEZZÈ, M. A survey of recent trends in testing concurrent software systems. **IEEE Transactions in Software Engineering**, p. 1–40, 2017. To appear. Citado na página 29.

BIEMAN, J. M.; GHOSH, S.; ALEXANDER, R. T. A technique for mutation of java objects. In: **Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE '01**. Washington, DC, USA: IEEE Computer Society, 2001. p. 337–. Citado na página 38.

BINDER, R. V. **Testing object-oriented systems: models, patterns, and tools**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-80938-9. Citado na página 34.

BRADBURY, J. S.; JALBERT, K. Defining a catalog of programming anti-patterns for concurrent java. In: **Proc. of SPAQu'09**. [S.l.: s.n.], 2009. p. 6–11. Citado na página 53.

BRIAND, L.; LABICHE, Y. Empirical studies of software testing techniques: Challenges, practical strategies, and future research. **SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 29, n. 5, p. 1–3, set. 2004. ISSN 0163-5948. Citado na página 58.

BRITO, M. A. S. **Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes**. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, ICMC/USP, São Carlos, SP, abril 2011. Citado nas páginas 28 e 29.

BRITO, M. A. S.; FELIZARDO, K.; SOUZA, P. S. L.; SOUZA, S. R. S. Concurrent software testing: A systematic review. **22nd IFIP International Conference on Testing Software and Systems**, p. 79–84, 2010. Short paper. Citado nas páginas 28, 57, 58 e 81.

CAI, Y.; CHAN, W.; YU, Y. Taming deadlocks in multithreaded programs. In: **13th International Conference on Quality Software (QSIC'13)**. [S.l.: s.n.], 2013. p. 276–279. Citado na página 52.

CASELLA, G.; BERGER, R. **Statistical Inference**. 2. ed. Pacific Grove, PA: Duxbury Resource Center, 2002. Textbook Binding. Citado na página 85.

CHAKI, S.; CLARKE, E.; OUAKNINE, J.; SHARYGINA, N.; SINHA, N. Concurrent software verification with states, events, and deadlocks. **Formal Aspects of Computing**, V17, n. 4, p. 461–483, December 2005. Citado na página 52.

CHEN, Q.; WANG, L.; YANG, Z.; STOLLER, S. Have: Detecting atomicity violations via integrated dynamic and static analysis. In: CHECHIK, M.; WIRSING, M. (Ed.). **Fundamental Approaches to Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5503). p. 425–439. ISBN 978-3-642-00592-3. Citado na página 52.

CHEW, L.; LIE, D. Kivati: Fast detection and prevention of atomicity violations. In: **Proceedings of the 5th European Conference on Computer Systems, EuroSys '10**. New York, NY, USA: ACM, 2010. p. 307–320. ISBN 978-1-60558-577-2. Citado na página 52.

CHOI, B. J.; DEMILLO, R. A.; KRAUSER, E. W.; MATHUR, A. P.; MARTIN, R. J.; OFFUTT, A. J.; PAN, H.; SPAFFORD, E. H. The mothra toolset. In: **Twenty-Second Annual Hawaii International Conference on System Sciences**. HI: [s.n.], 1989. Citado na página 38.

CHOI, J.-D.; ZELLER, A. Isolating failure-inducing thread schedules. In: **Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02**. New York, NY, USA: ACM, 2002. p. 210–220. ISBN 1-58113-562-9. Citado na página 52.

CHRISTAKIS, M.; SAGONAS, K. Detection of asynchronous message passing errors using static analysis. **Lecture Notes in Computer Science (LNCS)**, v. 6539, p. 5–18, 2011. Citado na página 52.

CORDEIRO, L. Smt-based bounded model checking for multi-threaded software in embedded systems. In: **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE '10 - Volume 2**. New York, NY, USA: ACM, 2010. p. 373–376. Citado na página 27.

CORDEIRO, L.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. In: **Proceedings of the 33rd International Conference on Software Engineering, ICSE '11**. New York, NY, USA: [s.n.], 2011. p. 331–340. ISBN 978-1-4503-0445-0. Citado na página 28.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas distribuídos: conceitos e projeto**. 4. ed. Porto Alegre, RS: Bookman Editora Ltda, 2007. Citado na página 47.

DANTAS, A. Improving developer's confidence in test results of multi-threaded systems: avoiding early and late assertions. In: **Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)**. Orlando, Florida: [s.n.], 2008. p. 899–900. Citado na página 52.

DANTAS, A.; GAUDENCIO, M.; BRASILEIRO, F.; CIRNE, W. **Obtaining Trustworthy Test Results in Multi-threaded Systems**. [S.l.], 2007. Citado na página 52.

- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. 2. ed. [S.l.]: Elsevier Brasil, 2016. v. 394. (Campus, v. 394). Citado nas páginas [25](#), [34](#), [35](#) e [43](#).
- DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface mutation: an approach for integration testing. **IEEE Transactions on Software Engineering**, v. 27, n. 3, p. 228–247, Mar 2001. ISSN 0098-5589. Citado na página [25](#).
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, April 1978. Citado nas páginas [25](#), [37](#) e [38](#).
- DENG, L.; OFFUTT, J.; SAMUDIO, D. Is mutation analysis effective at testing android apps? In: **2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017**. [S.l.: s.n.], 2017. p. 86–93. Citado na página [38](#).
- DIJKSTRA, E. W. **Cooperating Sequential Processes, Technical Report, EWD-123**. 1965. Citado na página [48](#).
- DING, Z.; ZHANG, K.; HU, J. A rigorous approach towards test case generation. **Information Sciences**, v. 178, n. 21, p. 4057–4079, 2008. Citado na página [27](#).
- DOKMANIC, I.; PARHIZKAR, R.; RANIERI, J.; VETTERLI, M. Euclidean distance matrices: A short walk through theory, algorithms and applications. **CoRR**, abs/1502.07541, 2015. Citado nas páginas [28](#) e [64](#).
- DOURADO, G. G. M.; SOUZA, P. S. L.; PRADO, R. R.; BATISTA, R. N.; SOUZA, S. R. S.; ESTRELLA, J. C.; BRUSCHI, S. M.; LOURENÇO, J. A suite of java message-passing benchmarks to support the validation of testing models, criteria and tools. In: **International Conference on Computational Science, ICCS**. [S.l.: s.n.], 2016. p. 2226–2230. Citado na página [73](#).
- EASTERBROOK, S. Empirical research methods for software engineering. In: **Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2007. (ASE '07), p. 574–574. ISBN 978-1-59593-882-4. Citado nas páginas [55](#) e [56](#).
- EMMI, M.; QADEER, S.; RAKAMARIC, Z. Delay-bounded scheduling. **ACM SIGPLAN Notices (POPL 2011)**, ACM, New York, United States, v. 46, n. 1, p. 411–422, 2011. ISSN 0362-1340. Citado na página [52](#).
- EYTANI, Y.; UR, S. Compiling a benchmark of documented multi-threaded bugs. In: **IPDPS**. [S.l.]: IEEE Computer Society, 2004. Citado na página [73](#).
- FARCHI, E.; NIR, Y.; UR, S. Concurrent bug patterns and how to test them. **Parallel and Distributed Processing Symposium, International**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 286b, 2003. Citado na página [53](#).
- FAROOQ, S. U.; QUADRI, S. Empirical evaluation of software testing techniques in an open source fashion. In: **Proceedings of the 2Nd International Workshop on Conducting Empirical Studies in Industry**. New York, NY, USA: ACM, 2014. (CESI 2014), p. 21–24. ISBN 978-1-4503-2843-2. Citado na página [25](#).

FERRARI, F. C.; MALDONADO, J. C.; RASHID, A. Mutation testing for aspect-oriented programs. In: **2008 1st International Conference on Software Testing, Verification, and Validation**. [S.l.: s.n.], 2008. p. 52–61. ISSN 2159-4848. Citado na página 25.

FLYNN, M. J. Very high-speed computing systems. In: **Proc. IEEE**. [S.l.: s.n.], 1966. v. 54, p. 1,901–1,909. Citado na página 45.

FLYNN, M. J.; RUDD, K. W. Parallel architectures. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 28, n. 1, p. 67–70, mar. 1996. ISSN 0360-0300. Citado nas páginas 15 e 46.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420. Citado na página 90.

FRANKL, P. G. **The use of data flow information for the selection and evaluation of software test data**. Tese (Doutorado) — New York University, New York, NY, USA, 1987. Citado nas páginas 35 e 38.

GALLIS, H.; ARISHOLM, E.; DYBA, T. An initial framework for research on pair programming. In: **International Symposium on Empirical Software Engineering**. [S.l.: s.n.], 2003. p. 132–142. Citado na página 71.

GILL, A. **Introduction to the Theory of Finite State Machines**. New York: McGraw-Hill, 1962. Citado na página 36.

GLASS, R.; VESSEY, I.; RAMESH, V. Research in software engineering: an analysis of the literature. **Information and Software Technology**, v. 44, n. 8, p. 491 – 506, 2002. ISSN 0950-5849. Citado na página 56.

GLIGORIC, M.; JAGANNATH, V.; LUO, Q.; MARINOV, D. Efficient mutation testing of multithreaded code. **Softw. Test., Verif. Reliab.**, v. 23, n. 5, p. 375–403, 2013. Citado na página 27.

GLIGORIC, M.; JAGANNATH, V.; MARINOV, D. Mutmut: Efficient exploration for mutation testing of multithreaded code. In: **Third International Conference on Software Testing, Verification and Validation (ICST)**. Paris, France: [s.n.], 2010. p. 55 –64. Citado na página 27.

GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. **Introduction to Parallel Computing**. 2. ed. [S.l.]: Addison Wesley, 2003. Hardcover. ISBN 0201648652. Citado nas páginas 43, 44, 45, 47 e 49.

GRUBBS, F. E. Procedures for detecting outlying observations in samples. **Technometrics**, Taylor & Francis, v. 11, n. 1, p. 1–21, 1969. Citado na página 116.

HALEY, A.; ZWEBEN, S. H. Development and application of a white box approach to integration testing. **Journal of Systems and Software**, v. 4, n. 4, p. 309–315, 1984. Citado na página 25.

HANSEN, P. B. **Operating system principles**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1973. ISBN 0-13-637843-9. Citado na página 49.

HAREL, D. Statecharts: A visual formalism for complex systems. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 8, n. 3, p. 231–274, jun. 1987. ISSN 0167-6423. Citado na página 41.

- HAVELUND, K.; PRESSBURGER, T. Model checking java programs using java pathfinder. **International Journal on Software Tools for Technology Transfer**, Springer-Verlag, v. 2, n. 4, p. 366–381, 2000. ISSN 1433-2779. Citado na página 27.
- HERMAN, P. M. A data flow analysis approach to program testing. **Australian Computer Journal**, v. 8, n. 3, p. 92–96, 1976. Citado na página 25.
- HOARE, C. A. R. Monitors: an operating system structuring concept. **Commun. ACM**, ACM, New York, NY, USA, v. 17, p. 549–557, October 1974. ISSN 0001-0782. Citado na página 49.
- HÖFER, A.; TICHY, W. F. Status of empirical research in software engineering. In: _____. **Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 10–19. Citado na página 56.
- HOHN, E. N.; MALDONADO, J. C.; MENDONCA, M.; FABBRI, S.; BOAS, A. L. V.; TAMBASCIA, C.; FREITAS, M. E. D.; PAGLIUSO, P. Pbr: Transferência de tecnologia baseada em pacotes de experimentação. In: **III Simposio Brasileiro de Qualidade de Software**. [S.l.: s.n.], 2004. p. 161–175. Citado na página 70.
- HONG, S.; AHN, J.; PARK, S.; KIM, M.; HARROLD, M. J. Testing concurrent programs to achieve high synchronization coverage. In: **Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA'12**. New York, NY, USA: ACM, 2012. p. 210–220. Citado na página 52.
- HÖST, M.; REGNELL, B.; WOHLIN, C. Using students as subjects: A comparative study of students and professionals in lead-time impact assessment. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 5, n. 3, p. 201–214, nov. 2000. ISSN 1382-3256. Citado na página 72.
- HOWDEN, W. E. Reliability of the path analysis testing strategy. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 2, p. 208–215, May 1976. ISSN 0098-5589. Citado nas páginas 25 e 53.
- IEEE, I. A. **IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology**. [s.n.], 1990. Disponível em: <<http://standards.ieee.org/findstds/standard/610.12-1990.html>>. Citado na página 34.
- IYENGAR, S. R.; DEBNATH, N. C. A general abstract representation for the study of concurrent programs. In: **Proceedings of the 1993 ACM Conference on Computer Science, CSC '93**. New York, NY, USA: ACM, 1993. p. 131–136. Citado nas páginas 52 e 53.
- JURISTO, N.; MORENO, A. M. **Basics of Software Engineering Experimentation**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441950117, 9781441950116. Citado nas páginas 58 e 76.
- JURISTO, N.; MORENO, A. M.; VEGAS, S. Reviewing 25 years of testing technique experiments. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 9, n. 1-2, p. 7–44, mar. 2004. ISSN 1382-3256. Citado nas páginas 57 e 58.
- KAMIL, A.; YELICK, K. Enforcing textual alignment of collectives using dynamic checks. **Lecture Notes in Computer Science (LNCS)**, v. 5898, p. 368–382, 2010. Citado nas páginas 27 e 52.

- KATAYAMA, T.; FURUKAWA, Z.; USHIJIMA, K. Event interactions graph for test-case generations of concurrent programs. In: **Asia Pacific Software Engineering Conference**. Brisbane, Queensland, Australia: [s.n.], 1995. p. 29–37. Citado nas páginas 52 e 53.
- KIM, H.; CHO, D.; MOON, S. Maximizing synchronization coverage via controlling thread schedule. In: **Consumer Communications and Networking Conference (CCNC), 2011 IEEE**. [S.l.: s.n.], 2011. p. 1186–1191. Citado na página 52.
- KITCHENHAM, B. A.; DYBAA, T.; JORGENSEN, M. Evidence-Based Software Engineering. In: **Proceedings of ICSE 2004**. [S.l.]: IEEE Computer Society Press, 2004. p. 273–281. Citado na página 28.
- KOJIMA, H.; KAKUDA, Y.; TAKAHASHI, J.; OHTA, T. A model for concurrent states and its coverage criteria. In: **International Symposium on Autonomous Decentralized Systems (ISADS 2009)**. Athens, Greece: [s.n.], 2009. p. 1–6. Citado na página 53.
- KRAWCZYK, H.; WISZNIEWSKI, B. **Systematic Testing of Parallel Programs**. [S.l.], 1995. Citado na página 27.
- KRAWCZYK, H.; WISZNIEWSKI, B.; MORK, P. Classification of software defects in parallel programs. **Relatório Técnico, Faculty of Eletronics, Technical University of Gdansk, Poland**, 1994. Citado na página 53.
- LASTOVETSKY, A. L. **Parallel Computing on Heterogeneous Networks**. 1. ed. New York, NY, USA: John Wiley & Sons, Inc., 2003. Citado na página 49.
- LEMOS, O. A. L. **Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ**. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, ICMC-USP, São Carlos, SP, 2005. Citado na página 36.
- LEMOS, O. A. L.; FRANCHIN, I. G.; MASIERO, P. C. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for java. **Science of Computer Programming**, v. 74, n. 10, p. 861–878, 2009. ISSN 0167-6423. Citado na página 25.
- LEWIS, B.; BERG, D. J. **Threads Primer: A Guide to Multithreaded Programming**. [S.l.]: SUNSOFT, 1996. xxvi–319 p. (Sun BluePrints Program). ISBN 0-13-443698-9. Citado na página 53.
- LI, J.; HEI, D.; YAN, L. Correctness analysis based on testing and checking for OpenMP programs. In: **ChinaGrid Annual Conference (ChinaGrid 09)**. Beijing, China: [s.n.], 2009. p. 210–215. Citado na página 27.
- LINKMAN, S.; ROMBACH, H. Experimentation as a vehicle for software technology transfer-a family of software reading techniques. **Information and Software Technology**, v. 39, n. 11, p. 777–780, 1997. Citado na página 70.
- LINKMAN, S. P.; VICENZI, A. M. R.; MALDONADO, J. **An evaluation of systematic functional testing using mutation testing**. 2003. VII International Conference on Empirical Assessment in Software Engineering, Keele, UK. Citado na página 37.
- LOTT, C. M.; ROMBACH, H. D. Repeatable software engineering experiments for comparing defect-detection techniques. **Empirical Software Engineering**, Springer Netherlands, v. 1, n. 3, p. 241–277, 1996. Citado nas páginas 70 e 75.

- LU, S.; JIANG, W.; ZHOU, Y. A study of interleaving coverage criteria. In: **Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering**. New York, NY, USA: ACM, 2007. p. 533–536. Citado nas páginas 27 e 53.
- MACFARLAND, T. **Two-Way Analysis of Variance: Statistical Tests and Graphics Using R**. Springer New York, 2011. (SpringerBriefs in Statistics). ISBN 9781461421337. Disponível em: <<https://books.google.com.br/books?id=nGxbnDJ9tEwC>>. Citado nas páginas 114 e 117.
- MAHESWARA, G.; BRADBURY, J. S.; COLLINS, C. Tie: An interactive visualization of thread interleavings. In: **Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10**. New York, NY, USA: ACM, 2010. p. 215–216. ISBN 978-1-4503-0028-5. Citado na página 27.
- MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Bridging the gap in the presence of infeasible paths: Potential uses testing criteria. In: **XII International Conference of the Chilean Computer Science Society**. Santiago, Chile: [s.n.], 1992. p. 323–340. Citado nas páginas 25, 34 e 40.
- MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. **Introdução ao testes de software**. Instituto de Ciências Matemáticas e de Computação - ICMC - USP: [s.n.], 2004. Nota didática do ICMC/USP n. 65. Citado nas páginas 37 e 42.
- MAMUN, A. A.; KHANAM, A. **Concurrent software testing: a systematic review and an evaluation of static analysis tool**. Dissertação (Mestrado) — School of Computing, Blekinge Institute of Technology, Sweden, 2009. Citado nas páginas 26, 28, 29, 57, 58, 62 e 81.
- MAMUN, M. A. A.; KHANAM, A.; GRAHN, H.; FELDT, R. Comparing four static analysis tools for java concurrency bugs. In: **Proc. of the Third Swedish Workshop on Multi-Core Computing (MCC-10)**. [S.l.: s.n.], 2010. p. 143–146. Citado na página 28.
- MATHUR, A. P.; KRAUSER, E. W. Modeling mutation on a vector processor. In: **ICSE**. [S.l.: s.n.], 1988. p. 154–161. Citado na página 38.
- MCCABE, T. A complexity measure. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, p. 308–320, 1976. ISSN 0098-5589. Citado nas páginas 25 e 39.
- MELO, S. M.; SOUZA, P. S. L.; SOUZA, S. R. S. Towards an empirical study design for concurrent software testing. In: **Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering**. Piscataway, NJ, USA: IEEE Press, 2016. (SE-HPCCSE '16), p. 1–1. ISBN 978-1-5090-5224-0. Citado nas páginas 63 e 70.
- MELO, S. M.; SOUZA, S. R. S. Definição de um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente. In: **IV Workshop de Teses e Dissertações do CBSOft - WTDSOft**. Maceió, AL, Brazil: [s.n.], 2014. p. 79–85. Citado na página 65.
- MELO, S. M.; SOUZA, S. R. S.; SILVA, R. A.; SOUZA, P. S. L. Concurrent software testing in practice: A catalog of tools. In: **Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation**. New York, NY, USA: ACM, 2015. (A-TEST 2015), p. 31–40. Citado nas páginas 26, 67 e 68.

MELO, S. M.; SOUZA, S. R. S.; SOUZA, P. S. L. Structural testing for multithreaded programs: An experimental evaluation of the cost, strength and effectiveness. In: **24th International Conference on Software Engineering and Knowledge Engineering (SEKE)**. [S.l.: s.n.], 2012. p. 476–479. Citado na página 28.

MELO, S. M.; SOUZA, S. R. S.; SOUZA, P. S. L.; CARVER, J. C. How to test your concurrent software: An approach for the selection of testing techniques. In: **Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems**. New York, NY, USA: ACM, 2017. (SEPS 2017), p. 42–43. ISBN 978-1-4503-5517-9. Citado nas páginas 62, 89 e 94.

MEYER, R.; KRUEGER, D. **MINITAB Guide to Statistics**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN 0137842325. Citado na página 116.

MOLINARI, L. **Testes Funcionais de Software**. 2008. Visual Books, Florianópolis. Citado na página 36.

MOURA, F. M.; MELO, S. M.; SOUZA, S. R. S. **SeleCTT: Ferramenta para automatização do processo de seleção de técnicas de teste de software concorrente**. São Carlos, SP, Brasil, 2017. Citado nas páginas 15, 17, 89, 90, 92, 95 e 97.

MYERS, G. J.; SANDLER, C. **The Art of Software Testing**. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122. Citado nas páginas 25, 34, 35, 36 e 37.

NETO, A. C. D.; TRAVASSOS, G. H. Supporting the selection of model-based testing approaches for software projects. In: **Proceedings of the 3rd International Workshop on Automation of Software Test**. New York, NY, USA: ACM, 2008. (AST '08), p. 21–24. ISBN 978-1-60558-030-2. Citado nas páginas 57, 85 e 86.

NETO, A. C. D.; TRAVASSOS, G. H. Model-based testing approaches selection for software projects. **Information & Software Technology**, v. 51, n. 11, p. 1487–1504, 2009. Citado nas páginas 29, 81 e 100.

NEUMANN, J. V. The principles of large-scale computing machines. **IEEE Ann. Hist. Comput.**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 10, n. 4, p. 243–256, out. 1988. ISSN 1058-6180. Citado na página 45.

NTAFOS, S. On required element testing. **Software Engineering, IEEE Transactions on**, SE-10, n. 6, p. 795–803, Nov 1984. Citado na página 25.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 5, p. 99–118, April 1996. ISSN 1049-331X. Citado na página 38.

PERRY, D. E.; PORTER, A. A.; VOTTA, L. G. Empirical studies of software engineering: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE '00), p. 345–355. ISBN 1-58113-253-0. Citado nas páginas 27 e 59.

PETERSON, J. L. Petri nets. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 9, n. 3, p. 223–252, set. 1977. ISSN 0360-0300. Citado nas páginas 41 e 42.

- PETRENKO, A.; BORODAY, S.; GROZ, R. Confirming configurations in efsm testing. **IEEE Transactions on Software Engineering**, v. 30, n. 1, p. 29–42, Jan 2004. Citado nas páginas 41 e 42.
- PILAR, M.; SIMMONDS, J.; ASTUDILLO, H. Semi-automated tool recommender for software development processes. **Electronic Notes in Theoretical Computer Science**, v. 302, n. 0, p. 95 – 109, 2014. Proceedings of the Latin American Computing Conference (CLEI 2013). Citado na página 29.
- PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. 5. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681. Citado nas páginas 34, 35, 37, 38 e 39.
- QAWY, D. A.; KAMEL, A.; OMARA, F. A survey on testing concurrent and multi-threaded applications tools and methodologies. In: **The International Conference on Informatics and Application (ICIA2012)**. Malaysia: [s.n.], 2012. p. 459–471. Citado nas páginas 28, 29, 57 e 81.
- QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656. Citado nas páginas 15, 43, 46 e 49.
- RAKAMARIC, Z. Storm: static unit checking of concurrent programs. In: **32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)**. New York, United States: ACM, 2010. (ICSE '10), p. 519–520. ISBN 978-1-60558-719-6. Citado na página 27.
- RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: **Proceedings of the 6th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982. (ICSE '82), p. 272–278. Citado na página 25.
- _____. Selecting software test data using data flow information. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 367–375, April 1985. ISSN 0098-5589. Citado nas páginas 39 e 40.
- RICCA, F.; TONELLA, P. Analysis and testing of web applications. In: **Proceedings of the 23rd International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2001. (ICSE '01), p. 25–34. ISBN 0-7695-1050-7. Citado na página 25.
- ROGENSKI, J. K.; PETRI, L. A.; SOUZA, L. F. de. Effects of parallel strategies in the transitional flow investigation. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, v. 37, n. 3, p. 861–872, May 2015. Citado na página 111.
- RUNGTA, N.; MERCER, E. G. Clash of the titans: Tools and techniques for hunting bugs in concurrent programs. In: **7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging**. [S.l.: s.n.], 2009. p. 9:1–9:10. Citado na página 73.
- _____. A meta heuristic for effectively detecting concurrency errors. **Lecture Notes in Computer Science (LNCS)**, v. 5394, p. 23–37, 2009. Citado nas páginas 27 e 52.
- SAINI, G. **An empirical evaluation of adequacy criteria for testing concurrent programs**. Dissertação (Mestrado) — Faculty of Graduate School of The University of Texas at Arlington, Arlinton, Texas, 2005. Citado na página 28.
- SALEH, K.; BOUJARWAH, A. A.; AL-DALLAL, J. Anomaly detection in concurrent java programs using dynamic data flow analysis. **Information & Software Technology**, v. 43, n. 15, p. 973–981, 2001. Citado na página 52.

SARMANHO, F. S.; SOUZA, P. S. L.; SOUZA, S. R. S.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: **ICCS (1)**. [S.l.: s.n.], 2008. p. 337–346. Citado na página 53.

SEO, H.-S.; CHUNG, I. S.; KWON, Y. R. Generating test sequences from statecharts for concurrent program testing. **IEICE - Trans. Inf. Syst.**, Oxford University Press, Oxford, UK, E89-D, p. 1459–1469, April 2006. ISSN 0916-8532. Citado na página 28.

SESHU, S. Introduction to the theory of finite-state machines. **Proceedings of the IEEE**, v. 51, n. 9, p. 1275–1275, Sept 1963. Citado na página 41.

SHENG, T.; VACHHARAJANI, N.; ERANIAN, S.; HUNDT, R.; CHEN, W.; ZHENG, W. Racez: a lightweight and non-invasive race detection tool for production applications. In: TAYLOR, R. N.; GALL, H.; MEDVIDOVIC, N. (Ed.). **ICSE**. [S.l.]: ACM, 2011. p. 401–410. Citado na página 52.

SHIRES, D. R.; POLLOCK, L. L.; SPRENKLE, S. Program flow graph construction for static analysis of mpi programs. In: ARABNIA, H. R. (Ed.). **PDPTA**. [S.l.]: CSREA Press, 1999. p. 1847–1853. Citado nas páginas 52 e 53.

SHULL, F.; CARVER, J.; TRAVASSOS, G. H. An empirical methodology for introducing software processes. In: **In Proceedings of the 8 th European Software Engineering Conference**. [S.l.: s.n.], 2001. p. 10–14. Citado nas páginas 27, 55 e 107.

SILVA, J. D. P.; SOUZA, S. R. S.; SOUZA, P. S. L. Geração automática de dados de teste para programas concorrentes com o uso de meta-heurística. In: **Brazilian Workshop on Systematic and Automated Software Testing, co alocado ao Congresso Brasileiro de Software (CBSOFT)**. Maceió, AL, Brazil: [s.n.], 2014. p. 71–80. Citado na página 27.

SILVA, R. A.; SOUZA, S. R. S.; SOUZA, P. S. L. Mutation operators for concurrent programs in MPI. In: **13th Latin American Test Workshop (LATW)**. Quito, Equador: [s.n.], 2012. v. 1, p. 1–6. Citado na página 27.

SIMÃO, A. Teste baseado em modelos. In: **José Carlos Maldonado; Mario Jino; Márcio Eduardo Delamaro. (Org.). Introdução ao Teste de Software**. 2. ed. Rio de Janeiro, RJ: Elsevier Ltda, 2016. cap. 3, p. 39–56. Citado nas páginas 36, 41, 42 e 43.

SIMÃO, A.; PETRENKO, A. Generating checking sequences for partial reduced finite state machines. In: **Proceedings of the 20th IFIP TC 6/WG 6.1 International conference on Testing of Software and Communicating Systems: 8th International Workshop, TestCom '08 / FATES '08**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 153–168. ISBN 978-3-540-68514-2. Citado na página 36.

SINGH, M.; SHARMA, A. K.; SAXENA, R. An uml+z framework for validating and verifying the static aspect of safety critical system. **Procedia Computer Science**, v. 85, n. Supplement C, p. 352 – 361, 2016. ISSN 1877-0509. International Conference on Computational Modelling and Security (CMS 2016). Citado na página 43.

SJOBERG, D. I. K.; DYBA, T.; JORGENSEN, M. The future of empirical methods in software engineering research. In: **2007 Future of Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 358–378. ISBN 0-7695-2829-5. Citado na página 55.

SOUZA, P. S. L.; SOUZA, S. R. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In: **In: International Conference on Computational Science (ICCS 2013)**. Barcelona, Espanha: [s.n.], 2013. p. 1–10. Citado na página 27.

SOUZA, S. R. S.; BRITO, M. A. S.; SILVA, R. A.; SOUZA, P. S. L.; ZALUSKA, E. Research in concurrent software testing: A systematic review. In: **Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging**. New York, NY, USA: ACM, 2011. (PADTAD '11), p. 1–5. ISBN 978-1-4503-0809-0. Citado nas páginas 26, 28, 51, 57, 58, 62 e 81.

SOUZA, S. R. S.; MALDONADO, J. C.; FABBRI, S. C. P.; MASIERO, P. C. Statecharts specifications: A family of coverage testing criteria. In: **XXVI Conferência Latinoamericana de Informática - CLEI'2000**. Tecnológico de Monterrey, México: [s.n.], 2000. Citado na página 42.

SOUZA, S. R. S.; MALDONADO, J. C.; FABBRI, S. C. P. F.; SOUZA, W. L. Mutation testing applied to estelle specifications. **Software Quality Journal**, Kluwer Academic Publishers, Hingham, MA, USA, v. 8, n. 4, p. 285–301, dez. 1999. ISSN 0963-9314. Citado na página 27.

SOUZA, S. R. S.; SOUZA, P. S. L.; MELO, S. M.; SILVA, R. A. Teste de programas concorrentes. In: **José Carlos Maldonado; Mario Jino; Márcio Eduardo Delamaro. (Org.). Introdução ao Teste de Software**. 2. ed. Rio de Janeiro, RJ: Elsevier Ltda, 2016. cap. 9, p. 231–248. Citado nas páginas 49 e 53.

SOUZA, S. R. S.; SOUZA, P. S. L.; SIMÃO, A. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 20, p. 1893–1916, November 2008. ISSN 1532-0626. Citado na página 53.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMÃO, A. S.; GONÇALVES, T. B.; LIMA, A. M.; HAUSEN, A. C. Valipar: A testing tool for message-passing parallel programs. In: **17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005)**. [S.l.: s.n.], 2005. p. 386–391. Citado na página 53.

SUGETA, T. **Proteum-RS/ST : Uma Ferramenta para Apoiar a Validação de Especificações Statecharts Baseada na Análise de Mutantes**. Dissertação (Mestrado) — ICMC-USP, São Carlos, SP, Brasil, November 1999. Citado na página 38.

SUGETA, T.; MALDONADO, J. C.; WONG, W. E. Mutation testing applied to validate sdl specifications. In: GROZ, R.; HIERONS, R. M. (Ed.). **Testing of Communicating Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 193–208. Citado na página 27.

TAKAHASHI, J.; KOJIMA, H.; FURUKAWA, Z. Coverage based testing for concurrent software. **Distributed Computing Systems Workshops, International Conference on**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 533–538, 2008. ISSN 1545-0678. Citado na página 53.

TAN, R. P.; NAGPAL, P.; MILLER, S. Automated black box testing tool for a parallel programming library. In: **2nd International Conference on Software Testing, Verification, and Validation (ICST 2009)**. Denver, Colorado, United States: [s.n.], 2009. p. 307–316. Citado na página 27.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms (2nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275. Citado nas páginas 43, 45, 47, 48 e 49.

TYAGI, M.; MALHOTRA, S. A review of empirical evaluation of software testing techniques with subjects. **International Journal of Advanced Research in Computer Engineering & Technology**, IJARCET, v. 3, p. 519–523, fev. 2014. ISSN 2278 – 1323. Citado na página 58.

VALGRIND-DEVELOPERS. **Valgrind-3.6.1**. 2012. Disponível em: <<http://valgrind.org/>>. Citado na página 73.

VEGAS, S.; BASILI, V. R. A characterisation schema for software testing techniques. **Empirical Software Engineering**, v. 10, n. 4, p. 437–466, 2005. Citado nas páginas 26, 28, 29, 57, 64, 65, 81, 106, 107, 110, 111, 114 e 137.

VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. Coverage testing criteria for message-passing parallel programs. In: **6th IEEE Latin-American Test Workshop (LATW 2005)**. Salvador, Bahia, Brazil: [s.n.], 2005. p. 161–166. Citado na página 53.

VOS, T. E. J.; MARÍN, B.; ESCALONA, M. J.; MARCHETTO, A. A methodological framework for evaluating software testing techniques and tools. In: **QSIC**. [S.l.: s.n.], 2012. p. 230–239. Citado nas páginas 29 e 57.

WEYUKER, E. J.; WEISS, S. N.; HAMLET, R. G. Comparison of program testing strategies. In: **Symposium on Testing, Analysis, and Verification'91**. [S.l.: s.n.], 1991. p. 1–10. Citado nas páginas 25 e 38.

WILLIAMSON, D. F.; PARKER, R. A.; KENDRICK, J. S. The box plot: A simple visual method to interpret data. **Annals of Internal Medicine**, v. 110, p. 916–921, 1989. Citado na página 116.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. Heidelberg, New York, USA: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435. Citado nas páginas 19, 28, 55, 56, 57, 72, 73, 76, 77 e 80.

WOJCICKI, M. A.; STROOPER, P. An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies. In: **Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on**. [S.l.: s.n.], 2007. p. 9–9. Citado na página 28.

WOOD, B. P.; SAMPSON, A.; CEZE, L.; GROSSMAN, D. Composable specifications for structured shared-memory communication. In: **OOPSLA**. [S.l.]: ACM, 2010. p. 140–159. Citado na página 27.

XIAOAN, B.; NA, Z.; ZUOHUA, D. Test case generation of concurrent programs based on event graph. In: **5th International Joint Conference on INC, IMS, and IDC (NCM 2009)**. Seoul, Korea: [s.n.], 2009. p. 143–149. Citado na página 27.

YANG, C. S.; POLLOCK, L. L. The challenges in automated testing of multithreaded programs. In: **In Proceedings of the 14th International Conference on Testing Computer Software**. [S.l.: s.n.], 1997. p. 157–166. Citado na página 50.

- YANG, C. S. D.; POLLOCK, L. L. All-uses testing of shared memory parallel programs. **Software Testing, Verification, And Reliability Journal**, v. 13, p. 3–24, 2003. Citado na página 53.
- YANG, C. S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 23, p. 153–162, March 1998. ISSN 0163-5948. Citado nas páginas 27 e 53.
- YANG, R.-D.; CHUNG, C.-G. Path analysis testing of concurrent programs. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 34, p. 43–56, January 1992. ISSN 0950-5849. Citado na página 27.
- YANG, Y.; CHEN, X.; GOPALAKRISHNAN, G. **Inspect: A Runtime Model Checker for Multithreaded C Programs**. [S.l.], 2008. Citado na página 73.
- YANG, Z.; SAKALLAH, K. Trace-driven verification of multithreaded programs. **Lecture Notes in Computer Science (LNCS)**, v. 6447, p. 404–419, 2010. Citado na página 27.
- ZABELL, S. L. On student's 1908 article "the probable error of a mean". **Journal of the American Statistical Association**, Taylor & Francis, v. 103, n. 481, p. 1–7, 2008. Citado na página 116.
- ZELKOWITZ, M. V.; WALLACE, D. R.; BINKLEY, D. W. Lecture notes on empirical software engineering. In: JURISTO, N.; MORENO, A. M. (Ed.). River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2003. cap. Experimental Validation of New Software Technology, p. 229–263. ISBN 981-02-4914-4. Citado na página 56.
- ZHAI, K.; XU, B.; CHAN, W. K.; TSE, T. H. Carisma: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In: **Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012**. New York, NY, USA: ACM, 2012. p. 221–231. Citado na página 52.
- ZHANG, L.; MARINOV, D.; KHURSHID, S. Faster mutation testing inspired by test prioritization and reduction. In: **Proceedings of the 2013 International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2013. (ISSTA 2013), p. 235–245. ISBN 978-1-4503-2159-4. Disponível em: <<http://doi.acm.org/10.1145/2483760.2483782>>. Citado na página 38.
- ZHAO, J. Multithreaded dependence graphs for concurrent java programs. In: **Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on**. [S.l.: s.n.], 1999. p. 13–23. Citado nas páginas 52 e 53.
- ZHU, H. A formal analysis of the subsume relation between software test adequacy criteria. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 22, p. 248–255, 1996. ISSN 0098-5589. Citado na página 35.

INSTRUMENTAÇÃO DO EXPERIMENTO

Tabela 40 – Projeto 1.

Sistema de empréstimo bancário	
Descrição do sistema a ser desenvolvido	
<p>O objetivo é construir um software para aprovação e rejeição de empréstimos bancários e calcular os valores a serem pagos no empréstimo. O sistema deverá permitir ao usuário realizar empréstimo para um cliente selecionado. O valor do empréstimo é autorizado pelo gerente operacional que possui autoridade para isso. O usuário preenche o valor do empréstimo e a quantidade de meses de pagamento e o dia do pagamento de cada mês. Após o preenchimento dessas informações, o sistema deverá apresentar uma interface que contenha a simulação mostrando os meses e o valor de cada prestação a ser paga pelo cliente e o cliente confirma a operação.</p>	
Atributos de caracterização	Valor
Características do software a ser desenvolvido	
Plataforma de execução do software	Windows e Linux
Contexto/Objetivo	Sistema concorrente, Sistema distribuído
Linguagem de programação para construir o software	C/MPI
Paradigma concorrente de interação de processos	Passagem de mensagem
Complexidade do projeto	Alta (muitas fórmulas)
Duração estimada do projeto	1 mês para os testes
Tamanho estimado da aplicação	Pequeno
Frequência de mudança nos requisitos	Alta de acordo com a legislação
Características desejadas para os testes	
Nível de teste desejado para o projeto	Teste de unidade
Tipo de técnica de teste a ser aplicada no software	Teste estrutural
Tipo de análise	Dinâmica
Características de qualidade avaliadas no projeto	Funcionalidade, segurança, efetividade
Tipos de falhas a serem reveladas	Erro de arquivo de dados, erro de Shell script, erro de exceção, erro de atomicidade, deadlock
Custo desejado para os testes	A empresa possui verba para a aquisição de novas ferramentas
Ferramentas para realização dos testes	A empresa não possui ferramentas para realização de testes
Entradas disponíveis para a técnica de teste	Código fonte
Saídas esperadas	Número de erros identificados, tempo de execução
Representação do programa	Diagramas e código fonte

Tabela 41 – Projeto 2.

Sistema de controle de vagas de estacionamento	
Descrição do sistema a ser desenvolvido	
O objetivo é construir um software que gerencie vagas de estacionamento, utilizando sensores, para verificar se está ou não ocupado e uma central que informe os motoristas sobre informações do estacionamento. O sistema deve ser composto por dois módulos principais: (1) Módulo de vagas, que utiliza um micro controlador e um sensor que capta, processa os dados da vaga e envia a uma central, através de uma rede sem fio. (2) Central de controle, que capta as informações recebidas pelo módulo 1 e converte em informações apresentadas ao usuário em uma tela/monitor.	
Atributos de caracterização	Valor
Características do software a ser desenvolvido	
Plataforma de execução do software	Windows
Contexto/Objetivo	Sistema distribuído, Sistema embarcado
Linguagem de programação para construir o software	Ada
Paradigma concorrente de interação de processos	Memória compartilhada
Complexidade do projeto	Alta (muitas fórmulas)
Duração estimada do projeto	2 meses para os testes
Tamanho estimado da aplicação	Pequeno
Frequência de mudança nos requisitos	Baixa
Características desejadas para os testes	
Nível de teste desejado para o projeto	Teste de sistema, teste de integração
Tipo de técnica de teste a ser aplicada no software	Teste funcional, Teste baseado em modelo
Tipo de análise	Dinâmica
Características de qualidade avaliadas no projeto	Funcionalidade, segurança, eficiência
Tipos de falhas a serem reveladas	Código faltando, erro de navegação, erro de exceção, tipo de dados incorreto, deadlock, livelock, condição de corrida
Custo desejado para os testes	A empresa não possui verba para a aquisição de novas ferramentas
Ferramentas para realização dos testes	A empresa não possui ferramentas para realização de testes
Entradas disponíveis para a técnica de teste	Código fonte
Saídas esperadas	Quantidade de erros detectados, tempo de execução
Representação do programa	Modelo em linguagem lustre

Tabela 42 – Projeto 3.

Projeto de paralelização de equações de Poisson	
Descrição do sistema a ser desenvolvido	
O objetivo do projeto é paralelizar (dividir a execução em dois ou mais processadores) os algoritmos para solução numérica da equação de Poisson. A implementação do código deve utilizar a linguagem Fortran juntamente com a biblioteca MPI. Os algoritmos desenvolvidos são usados na solução numérica das equações de Navier-Stokes para um escoamento bidimensional incompressível de fluido newtoniano. O objetivo da paralelização de códigos numéricos é permitir o acesso mais rápido aos resultados numéricos. Desta forma pode-se acessar mais rapidamente os resultados de simulações que levariam um tempo computacional muito grande se fossem executados em um único microprocessador.	
Atributos de caracterização	Valor
Características do software a ser desenvolvido	
Plataforma de execução do software	Linux
Contexto/Objetivo	Computação de alto desempenho
Linguagem de programação para construir o software	Fortran, MPI
Paradigma concorrente de interação de processos	Passagem de mensagem
Complexidade do projeto	Alta (muitas fórmulas)
Duração estimada do projeto	1 mês para os testes
Tamanho estimado da aplicação	Médio
Frequência de mudança nos requisitos	Baixa
Características desejadas para os testes	
Nível de teste desejado para o projeto	Teste de sistema, teste de unidade
Tipo de técnica de teste a ser aplicada no software	Teste funcional, teste baseado em modelos
Tipo de análise	Dinâmica
Características de qualidade avaliadas no projeto	Eficiência, efetividade
Tipos de falhas a serem reveladas	Bugs no código fonte, deadlock
Custo desejado para os testes	A empresa não possui verba para a aquisição de novas ferramentas
Ferramentas para realização dos testes	A empresa não possui ferramentas para realização de testes
Entradas disponíveis para a técnica de teste	Código fonte
Saídas esperadas	Número de erros identificados, tempo de execução
Representação do programa	Independente

Tabela 43 – Projeto 4.

Módulo de autenticação para sistema de firewall	
Descrição do sistema a ser desenvolvido	
Definição de um módulo de autenticação (permissão de acesso) para um sistema de Firewall que garante segurança, disponibilidade e controle de redes corporativas. O Firewall configura automaticamente as prioridades do tráfego da rede. O módulo de autenticação já está implementado no firewall, porém devido a um problema de lentidão quando uma alta demanda de acesso é requerida, uma otimização desse módulo é requerida. O objetivo é distribuir o controle de acesso entre diferentes servidores a fim de aumentar o desempenho desse módulo.	
Atributos de caracterização	Valor
Características do software a ser desenvolvido	
Plataforma de execução do software	Linux
Contexto/Objetivo	Computação de alto desempenho
Linguagem de programação para construir o software	C/C++/Pthreads
Paradigma concorrente de interação de processos	Memória compartilhada
Complexidade do projeto	Média (software legado)
Duração estimada do projeto	4 meses para os testes
Tamanho estimado da aplicação	Médio
Frequência de mudança nos requisitos	Baixa
Características desejadas para os testes	
Nível de teste desejado para o projeto	Teste de integração, regressão, carga e desempenho
Tipo de técnica de teste a ser aplicada no software	Teste baseado em erros
Tipo de análise	Dinâmica
Características de qualidade avaliadas no projeto	Desempenho, efetividade
Tipos de falhas a serem reveladas	Gargalo, deadlock, condição de corrida
Custo desejado para os testes	A empresa possui verba para a aquisição de novas ferramentas
Ferramentas para realização dos testes	A empresa não possui ferramentas para realização de testes
Entradas disponíveis para a técnica de teste	Código fonte
Saídas esperadas	Quantidade de erros detectados, tempo de execução, quantidade de memória usada
Representação do programa	Independente

Tabela 44 – Técnica de teste 1.

Técnica 1 (Teste estrutural – Critérios de fluxo de controle e de dados) [Souza et al., 2013]	
Atributo de caracterização	Valores
Plataforma de execução	Linux
Contexto/Objetivo	Sistemas distribuídos
Linguagem de programação	C/MPI
Técnica de teste	Teste estrutural
Critério de geração dos casos de teste	Critérios de fluxo de controle e dados
Nível de teste	Teste unitário
Mecanismo de geração de interleavings	Trace de execução
Entradas requeridas	Código fonte concorrente
Saídas (Resultados gerados)	Porcentagem de cobertura dos critérios, quantidade de erros detectados
Características de qualidade avaliada	Efetividade, custo, esforço para satisfação do critério
Tipo de estudo (avaliação empírica)	Experimento
Tipo de análise	Dinâmica
Paradigma concorrente de iteração de processos	Passagem de mensagem
Mecanismo de replay	Execução controlada
Representação do programa	Grafo de fluxo de controle paralelo
Técnica de redução do espaço de estados	Não informado
Tipos de erros revelados (erros concorrentes)	Erros de fluxo de controle e de dados
Ferramenta de apoio	ValiMPI
Custo associado à ferramenta	Sem custo
Plataforma em que a ferramenta opera	Linux
Compreensibilidade	Conhecimento sobre a linguagem em que o software é implementado
Dependências	Dependente da instrumentação

Tabela 45 – Técnica de teste 2.

Técnica 2 (Teste baseado em modelos – Teste baseado em busca) [Staunton et al., 2011]	
Atributo de caracterização	Valores
Plataforma de execução	Independente
Contexto/Objetivo	Sistemas distribuídos
Linguagem de programação	Java
Técnica de teste	Teste baseado em modelos
Critério de geração dos casos de teste	Teste baseado em busca
Nível de teste	Teste de sistema
Mecanismo de geração de interleavings	Reuso de modelos, EDA (estimation distributed algorithm)
Entradas requeridas	Especificação do sistema em LTL
Saídas (Resultados gerados)	Tempo gasto para encontrar um erro
Características de qualidade do software avaliadas	Eficiência
Tipo de estudo (avaliação empírica)	Estudo de caso
Tipo de análise	Dinâmica
Paradigma concorrente de iteração de processos	Memória compartilhada
Mecanismo de replay	Reuso de modelos
Representação do programa	Grafo de espaço de estados, Modelo em LTL
Técnica de redução do espaço de estados	Função de aptidão (fitness function), mecanismos heurísticos
Tipos de erros revelados (erros concorrentes)	Deadlock, erros inseridos pelo pesquisador
Ferramenta de apoio	ECJ, HSF-SPIN
Custo associado à ferramenta	Sem custo
Plataforma em que a ferramenta opera	Independente
Compreensibilidade	Conhecimento em análise de modelos
Dependências	Não depende de outra abordagem

Tabela 46 – Técnica de teste 3.

Técnica 3 (Teste baseado em erros – Detecção de condição de corrida) [Kasikci et al., 2013]	
Atributo de caracterização	Valores
Plataforma de execução	Linux
Contexto/Objetivo	Computação de alto desempenho
Linguagem de programação	C/C++
Técnica de teste	Teste baseado em erros
Critério de geração dos casos de teste	Detecção de condição de corrida
Nível de teste	Teste unitário
Mecanismo de geração de interleavings	Execução simbólica
Entradas requeridas	Código fonte
Saídas (Resultados gerados)	Quantidade de erros detectados, número de falsos positivos, overhead do tempo de execução
Características de qualidade do software avaliada	Efetividade, acurácia, desempenho
Tipo de estudo (avaliação empírica)	Estudo de caso
Tipo de análise	Dinâmica
Paradigma concorrente de iteração de processos	Memória compartilhada
Mecanismo de replay	Classificação baseada em replay
Representação do programa	Grafo Happens-before
Técnica de redução do espaço de estados	Caminhos primários de limite superior anteriormente explorados, redução de arestas na árvore de execução
Tipos de erros revelados (erros concorrentes)	Condição de corrida
Ferramenta de apoio	Portend+
Custo associado à ferramenta	Sem custo
Plataforma em que a ferramenta opera	Linux
Compreensibilidade	Conhecimento da linguagem em que o software é implementado
Dependências	Não depende de outra abordagem

Tabela 47 – Técnica de teste 4.

Técnica 4 (Teste baseado em métodos formais – checagem de modelos com estados, eventos e deadlocks) [Chaki et al., 2005]	
Atributo de caracterização	Valores
Plataforma de execução	Linux
Contexto/Objetivo	Sistemas distribuídos
Linguagem de programação	C/C++/Java
Técnica de teste	Teste baseado em erros
Critério de geração dos casos de teste	Abordagem baseada em barreiras dinâmicas
Nível de teste	Teste de sistema
Mecanismo de geração de interleavings	Escalonamento randômico
Entradas requeridas	Código fonte
Saídas (Resultados gerados)	Quantidade de erros detectados.
Características de qualidade do software avaliadas	Efetividade
Tipo de estudo (avaliação empírica)	Estudo de caso
Tipo de análise	Estática e dinâmica
Paradigma concorrente de iteração de processos	Passagem de mensagem
Mecanismo de replay	Replay determinístico
Representação do programa	Não informado
Técnica de redução do espaço de estados	ASN
Tipos de erros revelados (erros concorrentes)	Deadlock
Ferramenta de apoio	Não possui
Custo associado à ferramenta	Não possui
Plataforma em que a ferramenta opera	Independente
Compreensibilidade	Conhecimento em análise de modelos
Dependências	Dependente da geração de modelos formais

Tabela 48 – Porcentagem de adequação das técnicas ao projeto.

Técnica	T1	T2	T3	T4
Projeto				
Banco	0,6851	0,2680	0,4673	0,5758
Estacionamento	0,3214	0,5476	0,3991	0,4010
Equação	0,5341	0,2680	0,8043	0,3321
Firewall	0,5292	0,4292	0,6118	0,5467

Tabela 49 – Formulário de consentimento (F0).

<p>Identificação do participante: _____</p> <p>Você está sendo convidado a participar deste estudo de pesquisa, que é chamado de Experimento sobre seleção de técnicas de teste de software concorrente. Este estudo está sendo conduzido por Silvana Morita Melo, que é aluna de doutorado na Universidade de São Paulo sob orientação da Profª. Drª. Simone do Rocio Senger de Souza, professora na Universidade de São Paulo.</p> <p>Sobre o que é esse estudo?</p> <p>Este estudo está sendo realizado para avaliar duas abordagens de seleção de técnicas de teste de software concorrente pra um projeto de software.</p> <p>Por que este estudo é importante? Que bom os resultados terão?</p> <p>Este estudo fornecerá uma lista de características que ajudarão os profissionais de testes e pesquisadores a selecionar uma técnica de teste adequada para o projeto de software. Entre os objetivos deste estudo é ajudar a entender, caracterizar e escolher técnicas de teste no contexto de programação concorrente. Esperamos reduzir a barreira entre a proposta de técnicas de teste na academia e seu uso prático na indústria.</p> <p>Por que me pediram para participar desse estudo?</p> <p>Você foi solicitado a estar neste estudo porque você possui conhecimento em alguma das áreas abrangidas por este estudo (Engenharia de software, Programação concorrente, Teste de software) e pode fornecer dados importantes para a validação de nossa pesquisa.</p> <p>Eu li este formulário de consentimento. Eu entendo o que eu o que devo fazer.</p> <p>1. Você aceita participar desse estudo?</p> <p>☐ Sim.</p> <p>☐ Não.</p>
--

Tabela 50 – Formulário de caracterização do participante (F1).

Nome: _____	
Curso: _____	Período: _____
Assinale as alternativas abaixo de acordo com a sua experiência em cada uma das áreas:	
2. Como você avalia sua experiência em engenharia de software?	
☐ 1 = nenhuma	
☐ 2 = estudei em aula ou livro	
☐ 3 = pratiquei em 1 projeto em sala de aula	
☐ 4 = usei em 1 projeto na indústria	
☐ 5 = usei em vários projetos na indústria	
3. Como você avalia seu grau de conhecimento em teste de software?	
☐ 1 = nenhuma	
☐ 2 = estudei em aula ou livro	
☐ 3 = pratiquei em 1 projeto em sala de aula	
☐ 4 = usei em 1 projeto na indústria	
☐ 5 = usei em vários projetos na indústria	
4. Como você avalia seu grau de conhecimento em programação concorrente?	
☐ 1 = nenhuma	
☐ 2 = estudei em aula ou livro	
☐ 3 = pratiquei em 1 projeto em sala de aula	
☐ 4 = usei em 1 projeto na indústria	
☐ 5 = usei em vários projetos na indústria	
5. Como você avalia seu grau de conhecimento teste de software concorrente?	
☐ 1 = nenhuma	
☐ 2 = estudei em aula ou livro	
☐ 3 = pratiquei em 1 projeto em sala de aula	
☐ 4 = usei em 1 projeto na indústria	
☐ 5 = usei em vários projetos na indústria	
6. Você já teve a experiência de selecionar técnicas de teste de software para um projeto de software? Como você avalia sua experiência?	
☐ 1 = nenhuma (nunca selecionei técnicas para um projeto de software)	
☐ 2 = estudei em aula ou livro	
☐ 3 = pratiquei em 1 projeto em sala de aula	
☐ 4 = usei em 1 projeto na indústria	
☐ 5 = usei em vários projetos na indústria	

Tabela 51 – Formulário de seleção de técnicas (F2).

Identificação do participante: _____

Antes de prosseguir com o experimento você deve:

- 1) Acessar os documentos auxiliares que serão utilizados para condução do experimento no link abaixo:
https://drive.google.com/open?id=1PP08ETym_pIEU0Cs3TNPmw4E2z807hsD
- 2) Em posse dos documentos, ler o glossário de termos e a especificação do projeto de software.
- 3) Marcar o tempo que iniciou o processo de seleção.
- 4) Ler a especificação de cada uma das quatro técnicas de teste apresentadas para selecionar a mais adequada ao projeto de software apresentado.

7. Horário em que você iniciou a seleção das técnicas de teste: _____

8. Ordene as quatro técnicas em relação a sua adequação ao projeto apresentado (primeira é a mais adequada):

	Primeira	Segunda	Terceira	Quarta
Técnica 1	☐	☐	☐	☐
Técnica 2	☐	☐	☐	☐
Técnica 3	☐	☐	☐	☐
Técnica 4	☐	☐	☐	☐

9. Marque quais informações você utilizou para a seleção (ordenação) das técnicas de teste

Plataforma de execução	☐
Contexto/Objetivo	☐
Linguagem de programação	☐
Técnica de teste	☐
Critério de geração de casos de teste	☐
Nível de teste	☐
Mecanismo de geração de interleavings	☐
Entradas requeridas	☐
Saídas (resultados gerados)	☐
Característica de qualidade avaliada	☐
Tipo de estudo (avaliação empírica)	☐
Tipo de análise	☐
Paradigma de interação de processos	☐
Mecanismo de replay	☐
Representação do programa	☐
Técnica de redução de espaço de estados	☐
Tipo de erros revelados	☐
Ferramenta de apoio	☐
Custo associado à ferramenta	☐
Plataforma que a ferramenta opera	☐
Compreensibilidade	☐
Dependências	☐

10. Horário em que você terminou a seleção das técnicas de teste: _____

Tabela 52 – Formulário de informações faltantes durante a seleção (F3).

[illegible]

Tabela 53 – Formulário de dúvidas geradas durante a seleção (F4).

[illegible]

Tabela 54 – Formulário de avaliação da abordagem de seleção (F5).

Identificação do participante: _____
Esse formulário tem o objetivo de avaliar a opinião dos usuários sobre a abordagem utilizada para seleção de técnicas de teste de software (conjunto de informações sobre as técnicas de teste apresentados em forma de tabelas).
13. Você acha que a abordagem facilita a seleção de técnicas de teste? Por quê?

14. Quais as vantagens da abordagem se comparada a seleção das técnicas de teste baseados no seu atual método de seleção?

15. Quais as desvantagens da abordagem?

16. Você acha que a abordagem é simples de ser entendida?

17. Se você tivesse que usar a abordagem em um projeto na indústria, você acha que seria simples usá-la?

18. Você acredita que a abordagem possa ser melhorada para tornar seu uso mais simples? Como?

19. O que você aprendeu com a utilização da abordagem de seleção?

20. Escreva abaixo a hora em que você terminou o experimento:

MAPEAMENTO SISTEMÁTICO SOBRE EVIDÊNCIAS EM TESTE DE SOFTWARE CONCORRENTE

Mapping Empirical Research in Concurrent Software Testing

Silvana M. Melo^{a,*}, Jeffrey C. Carver^b, Paulo S. L. Souza^a, Simone R. S. Souza^a

^a*Institute of Mathematics and Computer Sciences, University of Sao Paulo
Avenida Trabalhador Sao-carlense, 400 - Centro
CEP: 13566-590 - Sao Carlos - SP, Brazil*

^b*Department of Computer Science, University of Alabama
3441 SEC - Box 870290 - 35487 - Tuscaloosa - AL, USA*

Abstract

[Background]: Testing concurrent software is difficult and expensive, especially due to the non-determinism that results in an exponential increase in potential test inputs. To address this complication, researchers develop testing techniques that select a subset of the input domain that has a high probability of revealing faults. Even though a significant number of techniques for concurrent software testing have been proposed, a lot of techniques remain little used at the academia and industry. One of the problems is the lack of research that provide information about the practical applicability of these techniques. Empirical evidence can provide an important scientific basis to improve the research and contribute with information on the strengths and weaknesses of each technique to move the researchers/practitioners towards well-founded decision during the testing technique selection. **[Aim]:** This study aim to gather and synthesize empirical research to provide an overview of the field and identify relevant information about the studies that has been conducted. **[Method]:** We conducted a systematic mapping to identify and analyze empirical research on concurrent software testing techniques. We also provide a detailed analysis of the studies and the design choices made. **[Results]:** Our primary results include: there is a general lack of empirical validation of concurrent software testing techniques, the type of evaluation varies with the type of technique, there are some key challenges to conducting empirical studies in this domain, and there is a dearth of controlled experiments. **[Conclusions]:** There is a need for more empirical work to validate the claims about concurrent software testing. Researchers also need to devote more effort to analyzing and synthesizing the results of the evidences that exist. This work is a preliminary initiative in this direction.

Keywords: Empirical Software Engineering, Concurrent Programming, Software Testing, Systematic Mapping

1. Introduction

The availability of multicore processors and inexpensive clusters has resulted in the increase of concurrent software. Modern business applications use concurrency to improve overall system performance. Consequently, researchers have proposed a variety of testing techniques for concurrent software.

Although several testing techniques have been proposed for concurrent software testing, the selection of one for a specific testing process remains based on the testing team's knowledge or habits acquired from previous projects, which has resulted in the constant use of a single technique and no use of other. The difficult transfer of knowledge from the academy to a practical application at the industry is due to the lack of evidences of applicability of those techniques in software projects [1].

The information required to select the most appropriate technique(s) for a given context should come from empirical software engineering (ESE) methods that provide insights into the benefits and limits of each technique [2].

Although secondary studies on concurrent software testing exist [3, 4, 5, 6, 7] (see Section 2 for more details), their focus

has been on the categorization of testing techniques, methodologies, and tools used. Thus, still lacking information on how these techniques have been validated in terms of the type of empirical evidence presented as sequential testing context [2, 1].

Empirical methods such as controlled experiments, case studies, and surveys are needed to help us evaluate and validate the research results. These methods allows to scientifically state whether something is better than something else. Thus, empirical methods provide one important scientific basis for software engineering [8]. In addition, empirical studies are a key way to get information to understand the specific factors that cause techniques/tools and methods to be more or less cost-effective and move the researchers/practitioners towards well-founded decisions [9].

By means of a systematic mapping study, we collected and compared existing evidence on concurrent software testing. The goal of this study was to determine whether the technical literature on concurrent software techniques provides a uniform and rigorous base of empirical evidence. If several studies evaluate the same techniques under similar conditions on different subject programs, there is a possibility to perform an aggregation of findings and thus strengthen the ability to draw general conclusions. In this mapping however we found that the existing studies were diverse, thus hindering such aggrega-

*Corresponding author.

Email address: morita@icmc.usp.br (Silvana M. Melo)

tion. Instead we present a qualitative analysis of the findings, an overview of existing techniques for concurrent software testing and of the amount and quality of empirical evidence.

The Systematic Mapping (SM) was gather together existing empirical evidence regarding concurrent software testing techniques to allow developers to answer questions about what techniques has been evaluated and how the strength of the evidences presented. The specific goals of this SM are:

1. provide an overview of the empirical studies area on concurrent software testing;
2. identify the concurrent software testing techniques that has been evaluated empirically and explore how these validation has been made;
3. analyze the strength of the empirical studies selected in the review and discuss the findings and the limitations of the evidence; and
4. present how empirical studies have been designed in concurrent testing context, challenges and research opportunities in this direction.

As the first secondary study that analyses the available empirical research on concurrent software testing, we aim to provide a snapshot of the use of empirical methods to study concurrent software testing techniques and aid in the identification of gaps in this area. This paper presents the results of a systematic, comprehensive review that aims at characterizing how empirical studies have been designed to investigate concurrent software testing cost-effectiveness and what empirical evidence is available in the literature for this field. In addition, we identify resources to increase the information available about testing concurrent applications, which will provide a guide to assists on how concurrent software testing techniques can be empirically assessed.

The remainder of the paper is structured as follows: Section 2 provides an overview of the prior related work. Section 3 explains the systematic mapping planning defined to conduct the research. Section 4 discusses the strength of the empirical studies identified in the mapping. Section 5 presents the study results, answering the research questions defined. Section 6 describes how empirical studies are conducted in concurrent software testing context and providing a guide for planning new studies. Section 7 discusses the limitations and threats to validity of the study. Finally, Section 8 concludes the paper and describes avenues of future work.

2. Related Work

This section first presents some fundamentals of concurrent software testing to provide an overview and context for this work. Then it provides an overview of the existing secondary studies on concurrent software testing to situate our current paper among the existing studies.

2.1. Concurrent Software Testing

Compared with sequential programming, concurrent programming enables a smart use of features for increasing the efficiency (in terms of execution time), avoiding the idleness of resources (that occurs with sequential processes), and reducing computational costs [10]. However, the testing of concurrent software raises some challenges. Concurrent execution leads to non-deterministic behavior because of the communication and synchronization of concurrent (or parallel) processes (or threads). The non-determinism introduced by concurrent execution means that the same input could lead to multiple different, but correct, outputs. Therefore, the testing activity must analyze the outputs that result from the execution of different synchronization sequences. The deterministic execution technique [11] can perform this analysis. This technique works by forcing the execution of a specific synchronization sequence for a given input to obtain the same output (result) in a non-deterministic environment.

A number of other features related to communication and synchronization between processes (or threads) impose challenges on concurrent software testing [12], including: development of techniques for static analysis, detection of errors related to synchronization, communication, data flow, deadlocks, livelocks, data race, and atomicity violation, adaptation of testing techniques for sequential programming to concurrent programs, definition of a data flow criterion that considers message passing and shared variables, automatic test data generation, efficient exploration of *interleaving* events, reduction of costs in testing activities, deterministic reproduction for a given synchronization sequence, and representation of a concurrent program that captures relevant information to the test.

Testing techniques provide different criteria to select the set of test cases that will be used to exercise the software. Testing techniques can be grouped into families based upon these criteria. In this way, techniques that belong to the same family are similar regarding the information needed to generate the test cases (source code or specifications) or the aspect of the code exercised by the test cases (e.g., control flow, data flow, or typical errors) [2].

Researchers have adapted testing techniques, such as structural, model-based, formal-method based, and fault-based, developed for sequential software for use with concurrent software [13, 14, 15, 16]. In addition, researchers have developed other methods specially for concurrent software that take into account features like non-determinism, synchronization, and communication of concurrent/parallel processes [17, 18, 19, 20, 21].

Due to the high number of studies that proposed methods to assist the concurrent software testing, some secondaries studies have been conducted, trying to identify and categorize this research area [3, 4, 5, 6, 7].

2.2. Secondary Studies on Concurrent Software Testing

This section details existing literature reviews and systematic mapping studies on concurrent software testing. The objective is to compare and contrast them and to show the limitations that our study aims to address.

Table 1: Secondary studies on concurrent software testing research.

Author/Reference	Year	Period	# Papers		Focus
			Returned	Included	
Mamun and Khanam [7]	2009	1989 - 2009	4352	109	Mapping studies about testing in shared-memory programs. Evaluation of static analysis tools to detect Java concurrent bugs.
Brito et al. [6] and Souza et al. [5]	2011	1988 - 2011	1166	175	Identify published studies related to concurrent testing techniques, bug classification, and testing tools.
Arora et al. [3]	2015	1983 - 2014	4175	179	Classifying testing approaches in eight categories: reachability testing, model-based testing, mutation-based testing, slicing-based testing, structural testing, search-based testing, formal testing, and random testing.
Our Study	2016	1978 - 2016	7101	109	Gather and classify the empirical evidence in concurrent software testing.

There are several systematic literature reviews on the individual classes of testing approaches (e.g. search-based, regression testing, and model-based testing) available at technical literature that not make a distinction about the type of the SUT (e.g. sequential or concurrent applications) however, there are specific characteristics of concurrent applications that affect the testing activity and need to be considered. Thus, in this study we consider the SLRs of approaches on concurrent software testing.

SLRs synthesize existing evidence concerning a specific research question, topic area or phenomenon of interest, and identify areas where research is lacking [22]. The existence of unexplored topics holds true in the area of concurrent software testing justifying a need for more secondary studies of software testing methods, techniques, and approaches.

Although, few and disconnected, some SLRs have been conducted to gather and classify research in this field. However, unlike the previous SLRs that focus on identifying and classifying concurrent software testing techniques, our study is the first that focuses on collecting evidence about the empirical studies used to validate concurrent software testing techniques.

Mamun and Khanam [7] present a systematic review of research on concurrent software testing in shared memory systems. The review covers: concurrent software issues, bugs types, testing techniques and tools, test case generation techniques and tools, and benchmarks of programs. The paper classifies the testing techniques as either static or dynamic. The paper also notes the programming languages supported by the testing tools and benchmarks. The contributions of this paper include: an identification of 51 testing techniques, 38 testing tools, and 87 unique Java multithreaded bug patterns.

Brito et al. [6] and Souza et al. [5] present a systematic review of studies related to concurrent software testing approaches, bug classification, and testing tools. This review reuses some of the classifications of the Mamun and Khanam [7] review and adds a new category related to programming paradigm (message-passing vs. shared-memory). The paper presents an overview of the state-of-the-art of concurrent software testing into the following categories: 1) 166 papers that propose a new approach, mechanism or framework for concurrent software testing; 2) 6 papers that present a taxonomy, classification or discussion about concurrent bugs; and 3) 50 papers that present a tool or automated methodology to support concurrent software testing. Note that the total number of papers exceeds the

175 listed in the table because some papers contain more than one contribution.

Likewise, the previous studies the review of Arora et al. [3] aimed to summarize the available research for concurrent software testing. But this is the first study which has classified the techniques under a particular approach. In this study, the authors classifying the selected works into eight categories: 1) reachability testing; 2) structural testing; 3) model-based testing; 4) mutation-based testing; 5) slicing-based testing; 6) formal methods; 7) random testing; and 8) search-based testing. This review was conducted in 2014 and obtained 4175 papers from research databases, and in the final inclusion process were selected 179 studies. These works focus on the testing techniques and were classified into: 8 testing approaches, 80 algorithms, 11 graphical representations used under a specific technique, and 83 tools available that implement an specific methodology.

Table 1 summarizes these reviews doing a comparison with our study (presented in this paper). These existing systematic reviews focus on describing the state-of-the-art of concurrent software testing. They categorize techniques, approaches, methods, kinds of bugs revealed, and supporting tools. The primary deficiency of these reviews is the lack of discussion about how the techniques are evaluated (e.g. by means of empirical studies). This information is important for analyzing the maturity level of knowledge about the testing techniques, including their strengths and weaknesses [2]. Although papers have examined empiricism in software testing (e.g. [2, 23, 24, 25]), they did not focus on concurrent software. It is this deficiency that our current review seeks to remedy by gathering all empirical evidence about concurrent software testing techniques.

Our systematic mapping is similar to previous reviews, taking into account that gathers available literature to concurrent software testing. However, our study have a different scope because we are focusing in collect evidences about the empirical studies existent in concurrent software testing. An overview of the aspects of the testing techniques that has been tested empirically allow researchers more easily to identify the gaps that can be covered in future research. Results constructed based on ESE can be useful to creating predictive models that can be applied in similar industrial applications.

3. Systematic Mapping Plan

We defined our mapping process according to the protocol model defined by Petersen et al. [26]. The following subsections explain the primary steps in the mapping protocol.

3.1. Research questions

This review aims to summarize the current state of empirical research about concurrent software testing. To drive this review, Table 2 lists a series of research questions.

The following information on population, intervention and comparison has been extracted:

- Population: Concurrent software testing techniques.
- Intervention: Contributions of empirical studies to validate testing techniques for concurrent software.
- Comparison: Not applicable.
- Results: Empirical evidences on concurrent software testing.

3.2. Selection criteria

Table 3 lists the inclusion/exclusion criteria we used to ensure we selected only the most relevant studies.

3.3. Control group of primary studies

A set of relevant primary studies are considered as a control group aim to verify the accuracy of the keywords chosen to compose the search string. In this study the control group was generated based on an ad hoc search. The control group is composed of six studies included in previously literature reviews and that present a high number of citations. These studies are shown in Table 4.

3.4. Definition of the search string

To obtain the relevant studies, we have executed automated searches on the following databases: IEEEExplore, ACM, Scopus, Web of Science, CiteSeerX, and ScienceDirect. These databases were chosen due to their comprehensiveness and extensive use in several software engineering secondary studies [22]. For each database, we developed a search string as shown in Table 5. In IEEE database has a limit to 15 search terms and, therefore, we have constructed two strings for this.

This string consists of the keywords *concurrent application* and *software testing*, along with their synonyms.

This study has been conducted during 2015 and 2016, the first semester of the year 2016 and before have been considered during the search.

3.5. Review Execution

Figure 1 illustrates the selection process, divided in four main phases:

1. **Phase 1: Papers identified in electronic database searching** – We executed the search string on each electronic database and retrieved the resulting studies.
2. **Phase 2. Title and abstract selection** – We analyzed the titles and abstracts of each study from Phase 1 to identify those that met the inclusion/exclusion criteria.
3. **Phase 3. Paper screening** – We quickly scanned the full paper for all studies included in Phase 2 to identify those papers that included empirical evaluation.
4. **Phase 4. Full text reading** – We carefully reviewed each paper included in Phase 3 to select only those studies that were truly relevant to our research questions.

For phases 3 and 4, the first author performed the task followed by a test/re-test validation by the second and third authors. This approach helped reduce researcher bias. During the validation process, the second and third authors analyzed the results of a randomly selected set of studies, using the same criteria as the first author. We then compared the results and discussed in online meetings to resolve any disagreements. The sample used in the validation process was 5% in phase 3 and 10% in phase 4 of the total number of papers selected on phase 2. Cohen's Kappa statistic method was used to measure the ability of different raters to classify subjects into one of several groups. The observed proportional agreement was defined as: $Kappa = (observed\ accuracy - expected\ accuracy) / (1 - expected\ accuracy)$. The proportional agreement for this study was $Kappa = .93$, which is classified as a "very good" agreement [32].

This process resulted in 109 relevant papers from the initial set of 7262 returned by the database search. Table 6 shows the results of the selection process for each database. The duplicate studies were eliminated considering the order presented in the table. Thus, the database with more relevant studies was ACM because we maintain the duplicates on it and remove subsequently from the others databases.

3.6. Data extraction

We used the data extraction form defined by Dybå and Dingsøyr [33], as shown in Appendix A, to gather the main information about the empirical studies. We chose to reuse this form, rather than creating our own, because it included all of the data items necessary to answer the research questions for this study.

4. Strength of empirical evidence

This section discusses the strength of the empirical studies conducted to validate concurrent software testing approaches. The following subsections organize this discussion around four testing techniques which provide rules to the systematic project

Table 2: Research questions

Research questions	Motivational factor
RQ-1 What are the general trends of empirical evaluation of concurrent testing approaches?	To understand and provide an overview of empirical research in the concurrent software testing area.
RQ-2 What empirical method have been used in the concurrent software testing area?	To identify the empirical methods adopted to conduct the evaluation in the concurrent software testing area.
RQ-3 Which testing technique and approach were empirically evaluated?	To identify testing techniques that present empirical evidence of your applicability.
RQ-4 Which languages are supported by the testing techniques evaluated?	To classify the languages covered by the concurrent software testing techniques.
RQ-5 Which aspects and response variables are analyzed for the empirical method?	To analyses which quality aspects of the testing technique has been evaluated by the empirical studies.
RQ-6 Which studies provide comparison between different techniques approaches?	To obtain information about the effectiveness between different techniques.
RQ-7 What kind of target programs are tested in the empirical evaluation?	To define what kind of software has been used as subject in the empirical evaluations.
RQ-8 What types of statistical analysis are used in the empirical evaluation?	To analyses the kind of statistical analysis are applied to obtain the conclusions about the empirical studies.
RQ-9 Which threats to validity are discussed in the empirical studies?	To evaluate how trustworthiness are the empirical study conclusions.
RQ-10 What are the publication channels most relevant in context of concurrent software testing?	To list the most relevant publication channels in this field. Providing a guide to researchers looking for references or publication alternatives.

Figure 1: Study selection procedure.

Table 3: Inclusion and exclusion criteria.

Inclusion criteria
IC-1 Studies that propose testing techniques for concurrent software and apply empirical studies to evaluate the proposed technique.
Exclusion criteria
EC-1 Studies that do not propose concurrent software testing techniques, studies that propose software testing techniques but does not related to concurrent applications, and studies that propose techniques just for hardware testing.
EC-1.1 Studies that propose concurrent program testing without an empirical evaluation.
EC-1.2 Studies in which the author claims that has an empirical evaluation, but the process is not shown in the paper.
EC-2 Studies not written in English and studies not accessible in full-text.

Table 4: Control group of primary studies

Reference	Title
Hong et al. [27]	Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation
Alowibdi and Sten-neth [28]	An empirical study of data race detector tools
Souza et al. [14]	Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs
Gligoric et al. [29]	Efficient mutation testing of multithreaded code
Lu et al. [30]	Learning from mistakes: a comprehensive study on real world concurrency bug characteristics
Koppol et al. [31]	Incremental integration testing of concurrent programs

of test cases for exercises both the internal logic of the software components and their domains of input and output.

We consider in this study the testing technique classification in Table 7 [34] where the testing techniques differ in their source of the information used for the test data selection.

The following subsections provide an overview of each type

of technique and the main conclusions drawn from the studies. These subsections also provide a list of the primary limitations of the set of studies. Instead of repeating the same limitations multiple times, Section 4.1 describes the limitations common across techniques. The individual sections then refer back to this list.

To better organization of the paper, the tables with details

Table 5: Search strings

Database	String
ACM	(Abstract:(("concurrent code") OR ("concurrent software") OR ("concurrent program") OR ("concurrent application") OR ("concurrent algorithm") OR ("parallel code") OR ("parallel software") OR ("parallel program") OR ("parallel application") OR ("parallel algorithm") OR ("multi-thread*") OR ("multi-thread*")) AND ("test*"))
IEEE	1(("Abstract":concurrent code" OR "Abstract":concurrent software" OR "Abstract":concurrent program" OR "Abstract":concurrent application" OR "Abstract":concurrent algorithm" OR "Abstract":parallel code") AND ("Abstract":test*)) 2(("Abstract":parallel software" OR "Abstract":parallel program" OR "Abstract":parallel application" OR "Abstract":parallel algorithm" OR "Abstract":multi-thread*" OR "Abstract":multithread*") AND ("Abstract":test*))
CiteSeerX	((abstract:(("concurrent code") OR abstract:(("concurrent software") OR abstract:(("concurrent program") OR abstract:(("concurrent application") OR abstract:(("concurrent algorithm") OR abstract:(("parallel code") OR abstract:(("parallel software") OR abstract:(("parallel program") OR abstract:(("parallel application") OR abstract:(("parallel algorithm") OR abstract:(("multi-threaded") OR abstract:(("multithreaded")) AND (abstract:(("test") OR abstract:(("testing"))
Web of Science	TS=(((("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*")) NOT ("simulation" OR "extensive test*" OR "hardware test*" OR "performance test*" OR "GPU"))
Scopus	ABS (((("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*")) AND NOT ("simulation" OR "extensive test*" OR "hardware test*" OR "performance test*" OR "GPU")) AND (LIMIT-TO (SUBJAREA , "COMP"))
ScienceDirect	Abstract(("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*))

Table 6: Electronic databases considered during the systematic mapping and amount of returned/included papers.

Database	# Papers		
	Returned	Included	Percentage
ACM	457	31	6.8%
IEEE	574	7	1.2%
CiteSeerX	2312	38	1.6%
Web of Science	1486	15	1.0%
Scopus	1790	15	0.8%
ScienceDirect	482	3	0.6%
Total	7101	109	1.5%

about the review is presented in Appendix A.

4.1. Common limitations

The limitations common across empirical evidence on concurrent software testing techniques are:

1. The programs and benchmarks used in the studies are smaller than 1000 LOC and may be not representative of real industrial applications, reducing the generalization of the results.

Table 7: Classification of testing techniques.

Technique	Definition
Structural testing	Uses knowledge of source code internal structure to derive test cases for testing. Some testing criteria include node, edge and path coverage.
Model based testing	Uses an abstract model to specify the behavior of the program under test and extract of it information to assist the software testing.
Fault-based testing	Uses knowledge of typical errors committed by programmers to guide the generation of test cases. The basic concept of fault-based testing is to select test cases that would distinguish the program under test from derived programs that contain hypothetical faults.
Formal-method based testing	Use mathematical techniques for specification, development, and verification of software and hardware system.

2. The empirical studies are mostly individual case studies. Due to confounding factors, it is difficult to draw cause-effect conclusions from individual case studies without devoting effort to cross-case analysis.
3. When determining the strengths and weaknesses of the techniques, most papers use *random testing* as the comparison technique rather than using other similar concurrent software testing techniques. Although this was quite expected, comparison among different systematic testing techniques may establish an inclusion relation that is really important in the context of testing coverage.
4. The conclusions are based primarily upon descriptive rather than inferential statistics. Descriptive statistic provide basis of the initial description of the data as part of a more extensive statistical analysis, or they may be sufficient in and of themselves for a particular investigation. But even more, the use of inferential statistic allow to infers properties about the population, including testing hypotheses and deriving estimates.
5. The dependent variables and the measures used to evaluate them differ across studies, hindering the ability to compare results.
6. Most papers do not describe the threats to validity of the study, which are important for assessing the trustworthiness of the conclusions. The conference paper size (number of pages) can be contribute with this limitation.

4.2. Structural testing

Structural (white-box) testing uses the knowledge of the internal structure of the software to derive test cases. Researchers have adapted control and data-flow testing criteria from sequential programming for use in concurrent programming [13, 14, 35]. In addition, researchers have developed criteria and coverage specifically for concurrent software, focusing on communication and synchronization [17, 36, 37, 38, 27]. Table A.12 list the papers that describe empirical studies used to validate properties of structural testing techniques. Analysis of these studies leads to the following conclusions:

- Similar to sequential software, testing of concurrent software based on control and communication flows is generally cheaper than testing based on data-flow or message passing [13].
- A composite approach combining structural and reachability testing has lower computational cost than structural testing alone and obtains maximum coverage and higher defect detection with fewer test cases [14].
- The existing concurrent test coverage metrics are often moderate to strong predictors of concurrent testing effectiveness. Furthermore, classical sequential coverage metrics are not useful for improving testing of concurrent programs [38]. Combinations of coverage metric provide more reliable performance across different programs, specifically for the test generation, fault detection, and the difficult-to-cover test requirements [37].

However, these conclusions are limited by common limitation factors (2, 4, 5, and 6) described in Section 4.1.

In summary, the empirical studies in concurrent structural software testing present more controlled data (half the studies are controlled experiments). Although currently, studies between different techniques are scarce and is difficult to compare the results between testing approaches because the metrics used are different and the design of studies vary.

4.3. Fault based testing

The non-determinism of concurrent applications introduces errors that are not found in sequential applications, e.g. atomicity violation, deadlock, and data race. To identify these types of errors research proposed specific testing approaches that consider communication and synchronization among threads [39, 40, 41, 42, 43, 44, 45]. Atomicity-violation bugs occur when concurrent execution violates the atomicity of certain code region [46, 47, 48, 49, 50]. Table A.13 shown the main empirical studies found in this literature review for approaches to detection of concurrency bugs. The main conclusions that can be drawn from the studies are:

- Tools such as Ctrigger [46, 47, 48], Kivati [49], and Intruder [50] were proposed to detect concurrency bugs. Empirical results show that these tools are able to detect atomicity violation bugs with reasonable overhead.
- Tools based on the *locking discipline* or the *happens-before* relations were proposed to detect races in multithread applications such as RACEZ [39], DrFinder [40], ThreadSanitizer [41], Portend+ [42], rccjava [43], NARADA [51], SAM [52], Deva [53], DROIDRACER [54], Dialyzer [55], TRADE [56], CLAP [57]. The empirical evaluations show that these tools are effective for detecting data race concurrency bugs.
- Many tools are proposed to analyze multithreaded programs and predict deadlocks, such as ConLock [44], OMEN1 [45], ASN [58]. Empirical results with these

tools show that effectiveness increases for detecting deadlocks while decreasing the number of false positives [30].

A commonly used fault-based technique is the mutation testing which uses knowledge about common mistakes made by developers to generate mutant programs, which are similar to the original but with minor syntactical changes produced by the mutant operators. Concurrent software requires additional mutation operators to handle communication and synchronization mistakes and to seek alternatives for deterministic re-execution [16, 59, 29, 60]. The main conclusions reached by this group of studies are:

- Operator-based selection (ConMAN operators) is slightly better than random mutant selection, reducing the number of mutants generated for testing tools [60].
- Selective mutation may optimize the execution of mutants by reducing the state space explored by each mutant [16].
- Results show that sequential and concurrent mutation operators are independent which demonstrates the importance of studying concurrent mutation operators [16]. An orthogonal optimization to selective mutation called MuT-MuT [59, 29] for mutation testing of concurrent code optimizes the execution of mutants up to 77% by reducing the state space explored.

However, the previous results are limited by the common limitation factors (2, 4, and 6) described at Section 4.1 the results show that the main concern of researchers on mutation testing execution time rather than effectiveness. This focus reflects the high cost of mutation testing caused by the increased non-determinism.

4.4. Formal method based testing

In technical literature in computer science field a variety of theoretical research has been devoted to formal methods. Formal languages and verification techniques are proposed and supported by tools, aim to verify properties of formal system descriptions. Although methods based on sound mathematical theories, there are not many systems developed nowadays for which correctness is completely formally verified using these methods. Formal methods can be applied to automatize the testing process, in particular the automated derivation of tests from formal specifications [61].

Aim to assists in testing support, researchers have explored the application of formal methods [62, 63, 64, 65, 66, 67, 41, 68, 69, 70, 43, 71, 72, 73, 74]. Table A.14 shown the empirical studies conducted on these approaches. The main conclusions reached by this group of studies are:

- The state-space coverage increases at an even faster rate when partial-order reduction is performed using the iterative context bounding approach than with other search methods because it limits the number of scheduler choices without limiting the depth of the execution [71].

- The schedule bounding approach is superior to depth-first search. Many, but not all, defects can be found using a small schedule bound [71].
- Delay bounding beats preemption bounding and schedule bounding is more effective at finding bugs than unbounded depth-first search [63, 68, 73, 70].
- Preemption sealing uses scheduling strategy to increase the efficiency and efficacy of run-time tools for detecting concurrency errors [74].
- Dynamical partial order reduction (TransDpor) technique leverages transitivity to speed-up exploration by up to two orders of magnitude compared to existing partial order reduction (POR) techniques [71].
- Compared with state-based or event-based frameworks, the use of state/event formalism facilitates the formulation of appropriate specifications and yields substantial improvements in verification time and memory usage [67].
- Transactional memory schedules for unit testing existing application or to a synthetic description written in a domain-specific language [69] can either execute this workload dynamically to reduce the runtime overhead.
- AI automated planning techniques to the formal development of the encoding for a subset of synchronization primitives [62] can successfully predict feasible runs for most run selection criteria and predict a run for most of the violation patterns.

However, the previous results are limited by the common limitation factors (2, 3, 4, and 6) described in Section 4.1.

In summary, the results have demonstrated that formal techniques are useful with good results for the effectiveness of software testing.

4.5. Model based testing

Models (normally formal models) specify the program behavior and testers use them to generate test sequences [75]. Model checking is the most common model-based testing for concurrent software [75, 76, 44, 77, 78, 19, 79, 80, 81, 82, 83, 84]. The drawback to model checking is that it can lead to a state-space explosion problem. To avoid this problem, researchers have explored state-space reduction. Tables A.15 show the empirical studies conducted on these approaches. The main conclusions reached by this group of studies are:

- The use of labeled transition systems (LTS) with a modular approach can significantly reduce the number of test sequences generated and executed during model-based testing. Modular testing may require significantly fewer test sequences than non-modular testing [75].
- Techniques which analyzes the predictive run, and generate a set of scheduling constraints with respect to the given cycle (constraint-based approach) [44] to dynamic checking on deadlocks can be both effective and efficient.

- Value-schedule-based technique [78] could achieve a higher coverage of program behaviors while testing thread schedules.
- Reachability testing [80] ensures that every partially ordered synchronization sequence of a program with a given input is exercised exactly once. Empirical studies showed a good speedup the number of sequences exercised during reachability testing while preserving important fault detection capabilities.
- Dynamic symbolic execution with unfoldings typically requires less test executions but is slower per test execution when compared to DPOR [84].
- Estimation of Distribution Algorithms (EDA) [83] based model checking saving computational and manual effort when building and debugging concurrent systems. This is achieved by reusing information, specifically information from the models constructed, from an earlier execution to aid the search in a future execution.
- The model reusing approach, which uses information from the models constructed from earlier executions, may save computational effort and improve fault detection [19].

However, the previous results are limited by the common limitation factors (2, 3, 4, and 5) described in Section 4.1.

Model checkers systematically control the thread scheduler to explore all possible behaviors of a program. Being systematic and exhaustive in nature, model checkers can often prove that a concurrent system satisfies its desired properties. Unfortunately, a key problem with model checking is that it does not scale with program size. This is because the number of possible interleavings of a concurrent program often grows exponentially with the length of execution. This is the well-known state-explosion problem. In summary, the empirical results showed that the main concern of researchers is the state-space explosion and the advance of each solution to avoid this problem. More empirical research is necessary to evaluate the effectiveness of this approach in terms of its capacity to reveal faults, making the results more comparable with other testing techniques [77].

4.6. Summary

The results shown in this section demonstrate that, in general, there is a need for additional empirical studies about concurrent software testing. There is specifically a need for studies that compare the performance of one technique against other techniques. In addition, researchers need to perform more replication of studies to allow the community to draw broader and deeper conclusions about the various approaches. To help establish stronger causal relationships, researchers need to conduct more controlled experiments, which provide stronger internal validity due to the increased control exercised [85]. These controlled experiments can complement the findings of the more frequent case studies. Finally, a complete discussion of validity threats is important for the reader to make sure the study results are reasonable and trustworthy. The absence of

this information hinders the replication of studies and the aggregation of data across studies.

5. Results and Synthesis

The previous section provided an overview of the empirical evidence about each concurrent testing technique. This section answers the research questions introduced in Table 2.

5.1. General Trends in Empirical Research on Concurrent Testing (RQ-1)

From the 393 studies identified, which propose concurrent software testing techniques, 109 (28%) presented an empirical evaluation. This percentage is good considering software engineering as a whole has empirical validation in only 17%-20% of the paper [86, 87]. Figure 2 shows the number of empirical studies on concurrent testing published each year. There is an increasing trend starting in 2006. This increase could be attributed to the importance of concurrent testing after the availability of dual core and hyper-threaded processors in the personal computers, which started in 2006 [88]. We do not consider the year of 2016 in the chart because the search was finished in March 2016 and the results for this year are incomplete and may appear unrealistic to the reader.

Figure 2: Publications per year.

5.2. Empirical Methods Used in Concurrent Software Testing (RQ-2)

The goal of this question was to examine the frequency of different types of empirical evaluation approaches. Authors often use inconsistent terminology when describing their empirical research methods. To provide a consistent view on the empirical methods used in each study, we classified them based on the following definitions:

- **Case study:** A case study answers a how or why question, is based on a research question and has data that links back to its research questions and provide a deep understanding of the reasons for and motivation behind phenomena of interest. A case study also occurs in a real world situation [89]. However, in this study, exploratory studies are classified as case studies in accordance with the other research in SE [90]. Exploratory studies may lack hypotheses, and/or a real world case, but has the same aim of case studies.

- **Controlled experiment and quasi-experiment:** In an experiment, the researcher manipulates one factor or variable to test their effects on another factor or variable. An experiment is typically driven by a formal hypothesis. In a controlled experiment, the researcher randomly assigns the subjects to the treatments, whereas in a quasi-experiment, the researcher assigns subjects to treatments in some other controlled way [91].
- **Literature Survey:** An examination of published literature to characterize or analyze research about a phenomenon of interest.

Using these definitions, we classified each empirical study as either a case study, an experiment, or a survey. The first author classified each study into the appropriate type. Then, the second and third authors each reviewed a random sample of approximately 20% of the papers to validate the first author's categorization.

As Figure 3 shows, case studies are the most common empirical method used to evaluate or demonstrate the functionality of proposed concurrent software testing techniques or tools. A relatively small number of studies contain an experimental evaluation and literature survey. Although case study are important to provide insight into potential cause-effect relations. The use of controlled experiments can help establish causal relationships to support or refute hypotheses.

Figure 3: Distribution of studies per kind of empirical method.

5.3. Testing techniques and approaches that were empirically evaluated (RQ-3)

In Figure 4 is shown the distribution of empirical studies for each concurrent software testing technique. The results indicated that the technique with the highest number of empirical studies in the concurrent software testing field is the structural testing technique. This technique has been widely used in sequential programming and has significant results in this field, maybe this is the factor that motivates researchers in concurrent programming to instantiate this technique to their context. Fault based testing was highly used to, this technique covers all

approaches to detect concurrent testing errors (data race, deadlock, atomicity violation, etc) and it is not restricted to a systematic model or methods to execute the test, because of this there is a lot of work, but the majority of the empirical evidence are case studies. Formal-method based testing technique followed by model based testing has not presented a high number of studies if compared to the previous two others, this fact can be attributed to the complexity of the formal method increased by the complexity of concurrent applications and the non-determinism characteristic of concurrent applications which insert the state explosion problem to the model based testing.

Figure 4: Distribution of empirical studies by kind of testing technique.

5.4. Programming Languages supported by Empirically Evaluated Testing Techniques (RQ-4)

Figure 5 shows the distribution of empirical studies by programming language supported by the testing techniques evaluated. The results showed the most commonly used programming languages are Java and C/C++. Though, there are testing techniques proposed for a variety of other programming languages but, in fact, most of the contributions on concurrent software testing as testing tools are related to these programming languages [92].

5.5. Characteristics of the Empirical Studies (RQ-5 – RQ-9)

This section summarizes the attributes of the empirical studies. Figure 6 shows the distribution of studies relative to each aspect of the empirical study definition:

- a) **Dependent variables:** the dependent variable most evaluated is the effectiveness to reveal faults, which is a variable that better describes what is expected of the testing techniques [2].
- b) **Comparison with other techniques:** more than half of empirical studies presents comparisons between the testing approaches. Although these results seem good, we need to consider that the majority of comparisons are between approaches inside the same technique family or comparisons with random testing, which not aggregates

Figure 5: Distribution of empirical studies per programming language supported.

real value to the analysis of the cost and possibility of aggregation between testing techniques to improve the testing coverage.

c) **Type of sample:** in the majority of the works, the authors state that they use objects that represent real programs and benchmarks samples. Although with a more realistic analysis we can observe that the authors claim using real world programs, but in fact they use only examples programs or parts of real applications.

d) **Type of statistical analysis:** the majority of evaluations has used only descriptive analysis to obtain conclusions. As discussed early in Section 5 the use of inferential statistics is more trustful font of conclusions.

e) **Discussion about threats to validity:** only few studies showed the evaluation of validity of their empirical studies, which introduces a negatively impact in the quality of the results and conclusions obtained. Validity evaluation guarantees that the results can be better understood and replications can be performed with more confidence.

5.6. Publication Channels (RQ-10)

Figure 7 shows the conferences and journals researchers have published the empirical studies about concurrent software testing. This information can be useful for testing practitioners who are looking for evaluations of testing techniques and for researchers who are studying concurrent software testing.

The results show that empirical research in concurrent software testing has been published in journals that focus on software testing, verification and validation (*Software Testing, Verification and Reliability*), on general software engineering (*IEEE Transactions on Software Engineering*), on systems and programming (*Journal of Systems and Software* and *ACM Transactions on Programming Languages and Systems*), and on parallel programming and distributed systems (*IEEE Transactions*

Figure 6: Characteristics of the selected empirical studies.

on Parallel and Distributed Systems). Among the conferences, the research is published in conferences focused on programming languages and systems implementation (*PLDI*, *OOPSLA*, and *ASPLOS*), on software testing, verification and validation (*ISSTA*, *ICST*, *HVC*), on general software engineering (*ESEC*, *ICSE*, *ASE*), on parallel programming (*PPoPP*) and on parallel and distributed system testing and debugging (*PADTAD*). Overall, the majority of papers appear in conferences and journals that focus on general software testing, programming languages, and software engineering. A smaller number of papers appear in venues specifically targeting concurrent software, possibly because of the lack of such venues.

5.7. State-of-the-art of Empirical Research in Concurrency Software Testing

The bubble chart in Figure 8 summarizes the *state-of-the-art* of empirical research in concurrent software testing. In the figure, the size of the bubble represents the number of studies included for each testing technique relative to sample type and research method. We can make some general observations from this graph. First, *fault based testing* technique use more realistic applications, so not surprisingly they also use a high number of case studies. Second, *structural testing* techniques used the most case studies, but used real applications and example programs too.

The chart also identifies some gaps in the space. First, there are few empirical studies about *model based testing*. Second, there is a dearth of controlled experiments about *formal method based testing*, and *model based testing*. Third, most studies are

Figure 7: Main channels for publication.

case studies which focus on a “proof-of-concept” for the new technique or tool and do not analyze across cases.

6. Designing empirical studies in Concurrent Software Testing

This section summarizes the results of the analysis of the finding evidences in concurrent software testing. The objective is understand how empirical studies have been designed (in terms of subject programs, study design, metrics, goals, variables and procedures that are used to construct the studies) in the concurrent programming context. Results obtained from this analysis are used as the basis to provide a guideline for experiments definition [93]. Besides to offer a comprehension on how empirical studies are designed, this section can work as a guideline containing a basic infrastructure for experimental studies in the context of concurrent programming, defining goals, variables, and choices to do during the experimental process.

In the following sections we describe all steps of the empirical design used for definition of experiments in the concurrent software testing context. We believe that similar experiments (studies with same research scope) offer concrete and reusable goals, plans and procedures which may create evidence that can be easily compared and effectively aggregated in secondary studies.

6.1. Goal Definition

The goal definition is essential for the activities of planning and conduction of the experiment. Since the choice of an aspect to be evaluated define what data will be collected, and how these data can be analyzed.

We describe how to define the objective of the experiment using the fundamentals of GQM process [94]. The GQM goal

template is composed of five parameters: 1) Object of study: the entity that is studied in the experiment; 2) Purpose: defines the intention of the experiment; 3) Focus: it is the primary effect under study in the experiment (the aspect of the object of study that is of interest); 4) The perspective: it tells the view-point from which the experiment results are interpreted; 5) Context: defines which subjects are involved in the experiment and which software artifacts (objects) are used in the experiment [95]. In Table 8 is shown the proposed template for the definition of experiment research question for concurrent software testing. The values in the table can be used as a reference to help researchers to instantiate goals in this context. For example, possible goal of an experimental study in the concurrent software testing can be defined as:

Analyze the concurrent software testing technique (object of study) **for the purpose of evaluation** (purpose) **with respect to their effectiveness** (quality focus) **from the point of view of the of the researcher** (perspective) **in the context of testing specialists finding defects in concurrent software** (context).

Table 8: GQM template for concurrent software testing

Goal definition	
Object of study	concurrent software testing technique (Table 7)
Purpose	evaluate, analyze, compare, measure, study
Quality focus	effectiveness, efficiency, efficacy, applicability, usability, feasibility, reproducibility, correctness, cost, performance (Table 9)
Perspective	researcher, testing practitioner, students
Context	subjects, objects, environment

6.2. Experiment Planning

As proposed by Wohlin et al. [8] the design of experiment is a result of the experiment planning definition and composed of seven steps. In order to provide a design of experiment to

Figure 8: State-of-the-art for empirical studies of concurrent software testing.

be used as a guide for definition of experiments in concurrent software testing, we detailed each of these steps on the next sections.

6.3. Context Selection

Based on the goal defined, the context selection specifies the environment in which the experiment will be executed. The experiment context can be classified in terms of the number of subjects and objects involved in the study. Hence, the context of the experiment can be characterized according to four dimensions [8]: 1) Off-line vs. on-line; 2) Student vs. professional; 3) Academic examples vs. real problems; and 4) Specific vs. general.

The majority of studies in concurrent software testing presented in this review are conducted off-line by researchers using benchmark programs and real programs in a specific and controlled context.

The research community has argued that laboratory studies performed by students have less quality and relevance instead of the software industry. Nevertheless, the establishment of collaboration with industry is hard and, depending on this collaboration, the study can be unfeasible. The definition of project staffed by students can be a good alternative, because this kind of project is cheaper and easier to control, but more directed to a certain context than projects staffed by professionals with more and various experience [96].

6.3.1. Subjects

The subjects of the experiment are who will apply the experiment. The participants can be referred to subjects (if humans) or experimental units (machines). The kind of participants can be (computer science students, testing practitioners, researchers). All participant characteristics that may affect the experiment results should be described (experience, schooling, knowledge of technique). For all the experimental studies in this mapping the

subjects are researchers who proposed the concurrent software testing techniques.

6.3.2. Objects

Experimental objects are the units where the experiment is executed. In the context of software testing the object can be characterized as the SUT (Software Under Testing). This software can be a didactic example with less than 1000 LOC (example programs), real industrial practical applications (real applications) or a set of programs from a benchmark. There are a variety of benchmarks that can be used as objects for empirical evaluation in the concurrent software testing area [97, 98, 99, 100, 101]. The majority are derived from benchmarks of programs aiming guarantees control and more replicate results to the experiments.

6.4. Hypotheses Formulation

Hypotheses are the formalization of the experimental study purpose and are used to test statements about the expected results. In general, two hypotheses should be formulated [8]:

6.4.1. Null Hypothesis

The null hypothesis is used to state that the differences between the observations are coincidental, therefore there are not underlying patterns in the experimental setting. This is the hypothesis that the researcher wants to reject with as high significance as possible. A possible hypothesis for the software testing domain may be that a testing technique *A* finds on average the same number of faults as the technique *B*. This hypothesis can be defined as:

Null hypothesis 1: *The effectiveness to find faults of the technique A is the same of the technique B.*

6.4.2. Alternative Hypothesis

The alternative hypothesis represents the statement in favor of which the null hypothesis is rejected. An example hypothesis is that a testing technique *A* finds more faults than the technique *B*. A possible hypothesis definition can be:

Alternative hypothesis 1.1: *The effectiveness to find faults of the testing technique A is greater than the technique B.*

For each goal stated in the research objective, the null hypothesis, and their corresponding alternative hypotheses, need to be reported by the researcher to draw conclusions about the study.

6.5. Variable Selection

In the variable selection is defined the influences and effects which will be examined in the experiment.

To define a design of experiment is necessary choose two types of variables: the dependent variable (response variable) and the independent variable (predictor variables).

6.5.1. Independent variables

Independent variables are the factors that are frequently varied in the experiment to measure their effect on the dependent variables. Independent variables can include treatments, materials and some context factors.

In the context of evaluation of concurrent testing techniques the independent variable related to the object of study can be a proposed testing approach. In this context, the possible values of treatments may be categorized according to the techniques showed in Table 7.

Other independent variables in concurrent software testing may be related to the subjects and objects (programs) characteristics. For example, the complexity of the programs (number of threads/processes, synchronizations and communication primitives, LOC).

Another factor to be considered as independent variable is the fault origin, that may be generated randomly, or based on an error taxonomy, or using an error based technique (as mutation), or just using faults pre-existent on the software. Because of the non-determinism of concurrent applications, other factors that need to be considered are the number of times that the testing is executed, and the time limit for test execution (considering the problem of state space explosion where the input domain increases exponentially and the coverage of all input domain is not practicable).

6.5.2. Dependent variables

The dependent variable is the characteristic evaluated to measure the influence of the factors on the treatments in the experiment.

To answer the questions about how good is the testing technique to discover bugs and to spend computational resources and time. The most common dependent variables in concurrent software testing and their possible values to measure are described in Table 9. This dependent variables are based on the studies presented in this review and the description is based on the author's definitions on the studies and the pattern of usability measured ISO 9242-11.

6.6. Selection of Subjects

Subjects are who is responsible for execution of the experiment. Ideally, the subjects of the experiment would be a random sample of the population of computer science professionals. However, attaining a random sample is difficult. Instead, subject groups are commonly an accidental sample of this population, meaning that they are selected based on their participation in university or industrial courses [102]. In the context of concurrent software testing the most of the experiments are developed by the researchers that propose a technique and in some cases, students are used as experimental subjects.

6.7. Choice of Design Type

After setting the input variables is necessary to define the experimental design (arrangement). The kind of experiment design depending on the goal of the experiment, the number of factors, the alternatives of the factors, the number of undesired variations, etc [103].

Based on the randomization, different treatments are applied to or by different subjects, while keeping other variables constant, and measuring the effects on dependent variables. In human-oriented experiments, subjects (humans) apply different treatments (techniques) to objects (programs), while in technology-oriented experiments, different technical treatments are applied to different objects [8].

The experiments for concurrent software testing presented in this review are all technology-oriented experiments, but the researchers not describe the kind of design was used to conduct the empirical evaluation. They only describe the procedure to testing application, and they not discuss how the treatments were assigned to the objects. This factor difficult the experiment replication and can bring in undesired sources of variation to the study.

Table 10 presents a summary of the common kind of design used on software engineering experiments and can be useful to guide the researcher to choose the appropriate design to the experiment from their objectives and variables instantiate. More detailed information about experiment design types and definitions can be found in [8, 103].

6.8. Instrumentation

During the instrumentation phase, is prepared all the instruments used in the practical application of the experiment, these instruments are of three types:

- Experiment object: where the experiment is applied (the program code).
- Guidelines: resources needed to guide the participants in the experiment (experiment descriptions, tool user guide, and checklists).
- Measurements: The measure in an experiment is conducted via data collection. In testing experiments, data are generally collected via automatic tool's output or manually by the tester.

Table 9: Dependent variables to concurrent software testing evaluation

Dependent variable	Description	Metrics (possible values)
Effectiveness	Ability of each technique to detect faults	defects detected, sequences exercised, number of mutants killed, number of preemptions, number of states visited, number of interleaving tested, number of executions explored, coverage archived, number of program events and data flow explored
Efficiency	How quickly the technique is capable of find faults	computation time, time and storage overhead, speedup, number of path explored, response time, time required for prediction
Usability	User satisfaction	quality analysis by questionnaires
Cost	How much effort is need to apply the technique	number of test cases to cover a criterion, time to execute schedules, number of interleaving explored, number of program executions
Performance	Resources that are expended when execute a technique	execution time, running time, memory overhead, disk space overhead, speedup, compilation time, memory usage, overhead time, response time
Accuracy	Number of false positives/false negatives generated	number of false positives, number of false negatives, number of infeasible elements, number of equivalent mutants, number of non executable paths/ interleavings
Scalability	If the technique works well in more complex software and hardware	relation between number of threads and runtime overhead, number of executions per time, number of threads used

Table 10: Standard Design Types

Kind of design	Conditions of the experiment	Application
One factor two treatments	Complete randomized design	Compare two treatment and each subject uses only one treatment on each object
	Paired comparison	Compare pairs of experiment material and each subject uses both treatments on the same object
One factor with more than two treatments	Randomized complete block design	Subjects form a more homogeneous experiment unit (block) each subject uses one object to all treatments and the order in which the subjects use the treatments are assigned randomly
	Completely randomized design	Uses one object to all treatments and the subjects are assigned randomly to the treatments
Two factors	2*2 factorial design	Subjects are randomly assign to each combination of the treatments
	Two-stage nested design	Has two factors, each with two or more treatments. The experiment design and analysis are the same as for the 2*2 factorial design
More than two factors	2 ^k factorial design	The subjects are randomly assigned to the 2 ^k combinations of treatments
	2 ^k fractional factorial design	Run just a fraction of the complete factorial experiment

The overall goal of the instrumentation is to provide means for performing the experiment and to monitor it, without affecting the control of the experiment. The results of the experiment shall be the same independently of how the experiment is instrumented. If the instrumentation affects the outcome of the experiment, the study presents threats to your validity [8].

6.9. Validity Evaluation

The validity of an experiment is related to the trustworthiness of the results obtained. During the planning phase of an experiment, the threats to validity must be identified in order to assess their impact over the data. All threats that might have an impact on the validity of the results need to be discussed. This includes at least:

6.9.1. Construct validity

Refers to the degree to which the operationalization of the measures in a study actually represents the constructs in the real world.

Construct threats in concurrent software testing can be resulted of the expectations of the researcher that can affect the results. Other threats are related to the incorrect use of statistical tests. The setting of the kind of fault, test case set choice and random generation need to be controlled and representative of the input domain to avoid construct threats to the experiment.

6.9.2. Internal validity

Refers to the extend to which the treatment or independent variables were actually responsible for the effects seen to the dependent variable. Unknown factors may have had an influence on the results and therefore put limitations on the internal validity of the study.

Some threats to validity on the concurrent software testing can be: the incorrect setting of the tool used that could lead to incorrect conclusions; the defects injected into the example programs, could have be inserted based on the knowledge and understanding of the researcher about the technique used to manipulate the results; the testing context and instrumentation process can be manipulate by the experimenter to favor you testing techniques resulting in non realistic results.

6.9.3. External validity

Refers to degree to which the findings of the study can be generalized to other participant populations or settings.

In the concurrent software testing the authors reports as mainly threats to external validity the representativeness of the programs used in the experiments (complexity and concurrency), and representativeness of the used faults (using taxonomies, pre-existent bugs, faults seeding), and the representativeness of the test suites used (size and complexity). Other factors as the size of the test sequences executions (due to non-deterministic behavior), the context in that the experiment is conducted and the randomization process can affect the results.

6.9.4. Conclusion validity

Refers to whether the conclusions reached in a study are correct. For controlled experiments conclusion validity may be related to the application of statistical tests to the data.

In the context of concurrent software testing the most common threats to conclusion validity are related to the correctness

Table 11: Threats to validity in concurrent software testing

Conclusion validity	Internal validity
Method used is not correct	Setting of tools used
Measure used is not correct	Selection of defects
Technique implementation is not correct	Configuration of the context
Process to fault insertion is not correct	Instrumentation process
Test suite size is not representative	
Incorrect statistical analysis	
Construct validity	External validity
Researcher expectancies affect the result	Representativeness of the object programs
Setting of faults affect the results	Representativeness of the faults seeded
Choice of test case design	Representativeness of the test suites used
Random heterogeneity of test cases	Test sequences size
Incorrect measure metric	Different behaviours
	Randomization process used
	Configuration of the tools used
	Generalization to other kinds of software

of the method, technique and tool used to conduct the testing activity. Other common threat is related to the correct selection of the metrics used to measure the dependent variables. Further, the method used to insert faults on the objects, can cause a threat. The last threat is related to the correct use of statistical tests in the hypothesis evaluation.

Table 11 summarize the most common threats to validity founding on the literature to concurrent software testing.

The survey from experiments with same research scope presented in this review was used to instantiate the empirical design of experiments, offering concrete, reusable, goals, plans and procedures to define and apply empirical studies in the concurrent software testing area.

In [93] is presented the detailed process to experiments definitions and we hopefully that it can be useful for researchers and testers that need to evaluate their proposals or compare different techniques searching for a most suitable to their needs.

Most of the studies in concurrent software testing are so far case studies. However, a major deficiency of case studies is that many variables vary from one case study to another, so that comparing the results to detect cause-effect relationships is difficult [104]. Therefore, we analyze the design for quasi/controlled experiments that should be conducted to complement case studies in empirical software engineering. It provide basis to replicate studies and the created evidence easier compared and hence effectively aggregated in secondary studies.

7. Threats to validity

Threats to the validity which can affect and limit the systematic mapping are analyzed below.

The first potential limitation is related to construct validity of this work relates to the papers chosen for inclusion in the mapping study. It is possible that our search string, choice of databases, or paper selection process inadvertently omitted relevant papers. However, by using the systematic process and having a second author double-check the selection, we do not believe we have omitted important papers.

The second potential limitation relates to the internal validity refers to the accuracy of conclusions about cause and effect data extracted from each study. To reduce this limitation, we used the predefined extraction form (Appendix A). The conclusions are drawn from the data collected in the extraction form. Even with this process in place, it is still possible that we missed the presence of an empirical study in a paper or misidentified the type of empirical study.

To reduce the potential for both limitations, we used a multistage process that involved all three authors. First, one author performed each step of the review process. Then, to mitigate single-researcher bias, a second author reviewed and validated the work of the first author. We used a random sampling process to perform this second review. We obtain a kappa value of 0.93 (very good agreement) [32] for the primary study selection.

8. Conclusion

This paper describes an systematic mapping that identifies and classifies empirical studies about concurrent software testing techniques. None of the existing secondary studies on concurrent software testing explicitly focus on the empirical validation of the proposed techniques. Therefore, this review fills a void in the literature. This SM includes 109 studies that contain empirical validation of concurrent software testing techniques. Based on the findings we draw the following conclusions:

1. **There is a general lack of empirical validation of concurrent software testing techniques.** The 109 papers included in this study represent on 28% of the papers on concurrent software testing techniques. The lack of empirical validation in almost 3/4 of the papers leads to a set of techniques devoid of any formal foundation that are risky to transfer to practice application. There is a need for more empirical study, reproduction, replication and re-analysis of studies [23, 105].
2. **The frequency of empirical validation varies with the type of concurrent software testing technique.** *Structural testing* have the most empirical validation, followed by *fault based testing*, and *formal method based testing*. In addition to these techniques, we did identify examples of empirical validation in all families of concurrent software testing techniques. This observation is a good indication that researchers are attempting to employ empirical validation methods to proposed concurrent software testing techniques.
3. **There are challenges in conducting empirical validation of concurrent software testing techniques:** The most common challenges are:
 - The subject programs used in the empirical studies are small and may be not representative of real industrial applications, thereby decreasing the generality of the results.
 - Case studies are the most common empirical method. Due to confounding factors, it is difficult to draw

cause-effect conclusions from individual case studies without devoting effort to cross-case analysis.

- Studies primarily compare the proposed technique against *random testing*, rather than against similar concurrent software testing techniques, thereby reducing the opportunity to establish relationships among the concurrent software testing techniques.
- The conclusions are based primarily upon descriptive rather than inferential statistics, thereby reducing the trustworthiness of the results.
- The dependent variables and the measures used to evaluate them differ across studies, thereby comparison among study results is difficult.
- Most papers do not describe the threats to validity of the study, therefore readers cannot assess the trustworthiness of the conclusions or perform more reliable replications.

4. **There is a dearth of controlled experiments:** The snapshot of the empirical studies in concurrent software testing indicates the need for more controlled experiments, especially related to: *formal method based testing* and *model based testing*.

In addition to those conclusions, the primary contributions of this work are:

1. **Identification of the main characteristics of the empirical studies.** We described the main factors (empirical method, dependent variable, type of sample, kind of statistical analysis, threats to validity) that compose the empirical studies for concurrent software testing, which can be used as a basis for comparisons and strength evaluation of these studies.
2. **Presentation of an empirical design.** This paper presented an empirical study design to experiments that evaluate concurrent software testing techniques. This scheme aim to reduce the effort required to develop and conduct further instances of this class of experiments as well as ease the comparison of results from similar experiments. In addition, its provide a guide to assists in the development of more controlled and replicated studies in concurrent software testing.
3. **Collection of empirical evidence about concurrent software testing.** The findings of this review can be used as a preliminary foundation to construct a body of evidence in concurrent software testing area. We identified and extracted relevant information about empirical evidence from the studies. Additional information can be extracted to provide evidence about important characteristics for technique selection.
4. **Identification of the need for more empirical validation.** This review provides an overview of the empirical studies on concurrent software testing and classifies them based on their empirical methodology. The results shown

that there is a need for more experiments to help establish stronger cause-effect relationships as well as more aggregation of case studies to draw deep conclusions. This review also identified gaps in the empirical research results to provide a starting point for additional future research.

In the future, we plan to use the results of this SLR to define criteria, based on empirical evidence, to guide developers in choosing testing techniques. In addition, we plan to encode these results in a tool that will automatically recommend the most appropriate techniques based on the characteristics of a given project.

Acknowledgment

The authors acknowledge the São Paulo Research Foundation, FAPESP, for the financial support under process no. 2015/23653-5 and 2013/05046-9.

References

- [1] N. Juristo, A. M. Moreno, S. Vegas, Towards Building a Solid Empirical Body of Knowledge in Testing Techniques, ACM SIGSOFT Software Engineering Notes 29 (5) (2004) 1–4.
- [2] N. Juristo, A. M. Moreno, S. Vegas, Reviewing 25 Years of Testing Technique Experiments, Empirical Software Engineering 9 (1-2) (2004) 7–44, ISSN 1382-3256.
- [3] V. Arora, R. Bhatia, M. Singh, A systematic review of approaches for testing concurrent programs, Concurrency and Computation: Practice and Experience (2015) 1572–1611 ISSN 1532-0634.
- [4] D. A. Qawy, A. Kamel, F. Omara, A Survey on Testing Concurrent and Multi-Threaded Applications Tools and Methodologies, in: The International Conference on Informatics and Application (ICIA2012), Malaysia, 459–471, 2012.
- [5] S. R. S. Souza, M. A. S. Brito, R. A. Silva, P. S. L. Souza, E. Zaluska, Research in Concurrent Software Testing: A Systematic Review, in: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '11, ACM, New York, NY, USA, ISBN 978-1-4503-0809-0, 1–5, 2011.
- [6] M. A. S. Brito, K. Felizardo, P. S. L. Souza, S. R. S. Souza, Concurrent Software Testing: A Systematic Review, 22nd IFIP International Conference on Testing Software and Systems (2010) 79–84 Short paper.
- [7] A. A. Mamun, A. Khanam, Concurrent software testing: a systematic review and an evaluation of static analysis tool., Master's thesis, School of Computing, Blekinge Institute of Technology, Sweden, 2009.
- [8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, ISBN 0-7923-8682-5, 2000.
- [9] D. E. Perry, A. A. Porter, L. G. Votta, Empirical Studies of Software Engineering: A Roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, ACM, New York, NY, USA, ISBN 1-58113-253-0, 345–355, 2000.
- [10] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, Addison Wesley, 2^o edn., ISBN 0201648652, 2003.
- [11] Y. Lei, R. H. Carver, Reachability Testing of Concurrent Programs, IEEE Trans. Software Eng. 32 (6) (2006) 382–403.
- [12] C.-S. D. Yang, Program-based, Structural Testing of Shared Memory Parallel Programs, Ph.D. thesis, University of Delaware, Newark, DE, USA, 1999.
- [13] M. A. S. Brito, S. R. S. Souza, P. S. L. Souza, An Empirical Evaluation of the Cost and Effectiveness of Structural Testing Criteria for Concurrent Programs., in: International Conference on Computational Science, ICCS, vol. 18 of *Procedia Computer Science*, Elsevier, 250–259, 2013.
- [14] S. R. S. Souza, P. S. L. Souza, M. A. S. Brito, A. S. Simao, E. J. Zaluska, Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs, Software Testing, Verification and Reliability 25 (3) (2015) 310–332.

- [15] D. Kester, M. Mwebesa, J. S. Bradbury, How Good is Static Analysis at Finding Concurrency Bugs?, in: SCAM, IEEE Computer Society, 115–124, URL <http://dblp.uni-trier.de/db/conf/scam/scam2010.html#KesterMB10>, 2010.
- [16] M. Gligoric, L. Zhang, C. Pereira, G. Pokam, Selective mutation testing for concurrent code., in: M. Pezzé, M. Harman (Eds.), International Symposium on Software Testing and Analysis, ISSTA, ACM, 224–234, 2013.
- [17] E. Sherman, M. B. Dwyer, S. G. Elbaum, Saturation-based testing of concurrent programs., in: H. van Vliet, V. Issarny (Eds.), Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/SIGSOFT FSE, ACM, 53–62, 2009.
- [18] J. Yu, S. Narayanasamy, C. Pereira, G. Pokam, Maple: A Coverage-driven Testing Tool for Multithreaded Programs, SIGPLAN Not. 47 (10) (2012) 485–502, ISSN 0362-1340.
- [19] K. Lu, X. Zhou, T. Bergan, X. Wang, Efficient Deterministic Multithreading Without Global Barriers, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, ISBN 978-1-4503-2656-8, 287–300, 2014.
- [20] Y. Lei, R. H. Carver, R. Kacker, D. Kung, A Combinatorial Testing Strategy for Concurrent Programs, Software Testing Verification and Reliability 17 (4) (2007) 207–225.
- [21] K. Lu, X. Zhou, X. Wang, T. Bergan, C. Chen, An Efficient and Flexible Deterministic Framework for Multithreaded Programs, Journal of Computer Science and Technology 30 (1) (2015) 42–56.
- [22] B. Kitchenham, S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, Tech. Rep., Keele University and Durham University Joint Report, 2007.
- [23] N. Juristo, A. M. Moreno, Basics of Software Engineering Experimentation, Springer Publishing Company, Incorporated, 1st edn., ISBN 1441950117, 9781441950116, 2010.
- [24] L. Briand, Y. Labiche, Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research, SIGSOFT Software Engineering Notes 29 (5) (2004) 1–3, ISSN 0163-5948.
- [25] M. Tyagi, S. Malhotra, A Review of Empirical Evaluation of Software Testing Techniques with Subjects, International Journal of Advanced Research in Computer Engineering & Technology 3 (2014) 519–523, ISSN 2278-Å\$ 1323.
- [26] K. Petersen, S. Vakkalanka, L. Kuzniarz, Guidelines for conducting systematic mapping studies in software engineering : An update, Information and Software Technology 64 (2015) 1–18.
- [27] S. Hong, M. Staats, J. Ahn, M. Kim, G. Rothermel, Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation, Software Testing Verification and Reliability 25 (4) (2015) 334–370.
- [28] J. S. Alowibdi, L. Stenneth, An empirical study of data race detector tools, in: 2013 25th Chinese Control and Decision Conference (CCDC), 3951–3955, 2013.
- [29] M. Gligoric, V. Jagannath, Q. Luo, D. Marinov, Efficient mutation testing of multithreaded code, Software Testing, Verification and Reliability 23 (5) (2013) 375–403.
- [30] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, ACM, New York, NY, USA, ISBN 978-1-59593-958-6, 329–339, 2008.
- [31] P. V. Koppol, R. H. Carver, K.-C. Tai, Incremental Integration Testing of Concurrent Programs, IEEE Transactions on Software Engineering 28 (6) (2002) 607–623, ISSN 0098-5589.
- [32] J. Cohen, A Coefficient of Agreement for Nominal Scales, Educational and Psychological Measurement 20 (1960) 37–46.
- [33] T. Dybå, T. Dingsøy, Empirical Studies of Agile Software Development: A Systematic Review, Information and Software Technology 50 (9–10) (2008) 833–859, ISSN 0950-5849.
- [34] I. C. Society, P. Bourque, R. E. Fairley, Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edn., ISBN 0769551661, 9780769551661, 2014.
- [35] P. G. Joisha, R. S. Schreiber, P. Banerjee, H.-J. Boehm, D. R. Chakrabarti, On a Technique for Transparently Empowering Classical Compiler Optimizations on Multithreaded Code, ACM Transactions on Programming Languages and Systems 34 (2) (2012) 9:1–9:42, ISSN 0164-0925.
- [36] M. Ganai, D. Lee, A. Gupta, DTAM: Dynamic Taint Analysis of Multithreaded Programs for Relevancy, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, ISBN 978-1-4503-1614-9, 46:1–46:11, 2012.
- [37] S. Tasiran, M. E. Keremoglu, K. Muslu, Location pairs: a test coverage metric for shared-memory concurrent programs, Empirical Software Engineering 17 (3) (2012) 129–165.
- [38] S. Hong, M. Staats, J. Ahn, M. Kim, G. Rothermel, The Impact of Concurrent Coverage Metrics on Testing Effectiveness, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 232–241, 2013.
- [39] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications, in: International Conference on Software Engineering ICSE, 401–410, 2011.
- [40] Y. Cai, L. Cao, Effective and Precise Dynamic Detection of Hidden Races for Java Programs, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, NY, USA, ISBN 978-1-4503-3675-8, 450–461, 2015.
- [41] M. F. Dolz, D. del Río Astorga, J. Fernández, J. D. García, F. García-Carballeira, M. Danelutto, M. Torquati, Embedding Semantics of the Single-Producer/Single-Consumer Lock-Free Queue into a Race Detection Tool, in: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16, ACM, New York, NY, USA, ISBN 978-1-4503-4196-7, 20–29, 2016.
- [42] B. Kasikci, C. Zamfir, G. Candea, Automated Classification of Data Races Under Both Strong and Weak Memory Models, ACM Transactions on Programming Languages and Systems 37 (3) (2015) 8:1–8:44, ISSN 0164-0925.
- [43] M. Abadi, C. Flanagan, S. N. Freund, Types for Safe Locking: Static Race Detection for Java, ACM Transactions on Programming Languages and Systems 28 (2) (2006) 207–255, ISSN 0164-0925.
- [44] Y. Cai, S. Wu, W. K. Chan, ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 491–502, 2014.
- [45] M. Samak, M. K. Ramanathan, Multithreaded Test Synthesis for Deadlock Detection, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, ACM, New York, NY, USA, ISBN 978-1-4503-2585-1, 473–489, 2014.
- [46] S. Park, S. Lu, Y. Zhou, CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places, in: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, ACM, New York, NY, USA, ISBN 978-1-60558-406-5, 25–36, 2009.
- [47] S. Lu, S. Park, Y. Zhou, Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing., IEEE Transactions on Software Engineering 38 (4) (2012) 844–860.
- [48] S. Lu, J. Tucek, F. Qin, Y. Zhou, AVIO: Detecting Atomicity Violations via Access Interleaving Invariants, ACM Special Interest Group on Operating Systems SIGOPS 40 (5) (2006) 37–48, ISSN 0163-5980.
- [49] L. Chew, D. Lie, Kivati: Fast Detection and Prevention of Atomicity Violations, in: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, ACM, New York, NY, USA, ISBN 978-1-60558-577-2, 307–320, 2010.
- [50] M. Samak, M. K. Ramanathan, Synthesizing Tests for Detecting Atomicity Violations, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 131–142, 2015.
- [51] M. Samak, M. K. Ramanathan, S. Jagannathan, Synthesizing Racy Tests, SIGPLAN Notes 50 (6) (2015) 175–185, ISSN 0362-1340.
- [52] Q. Chen, L. Wang, Z. Yang, SAM: Self-adaptive Dynamic Analysis for Multithreaded Programs, in: Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification

- and Testing, HVC'11, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-34187-8, 115–129, 2012.
- [53] G. Safi, A. Shahbazian, W. G. J. Halfond, N. Medvidovic, Detecting Event Anomalies in Event-based Systems, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, ISBN 978-1-4503-3675-8, 25–37, 2015.
- [54] P. Maiya, A. Kanade, R. Majumdar, Race Detection for Android Applications, SIGPLAN Notes 49 (6) (2014) 316–325, ISSN 0362-1340.
- [55] M. Christakis, K. Sagonas, Detection of Asynchronous Message Passing Errors Using Static Analysis, in: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages, PADL'11, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-18377-5, 5–18, 2011.
- [56] G. Kestor, O. S. Unsal, A. Cristal, S. Tasiran, TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems, ACM Transactions on Programming Languages and Systems 2 (2) (2015) 11:1–11:23, ISSN 2329-4949.
- [57] J. Huang, C. Zhang, J. Dolby, CLAP: Recording Local Executions to Reproduce Concurrency Failures, SIGPLAN Not. 48 (6) (2013) 141–152, ISSN 0362-1340.
- [58] K. Zhai, Y. Cai, S. Wu, C. Jia, W. Chan, ASN: A Dynamic Barrier-based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs, IEEE Transactions on Parallel and Distributed Systems 99 (PrePrints) (2014) 1, ISSN 1045-9219.
- [59] M. Gligoric, V. Jagannath, D. Marinov, MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code, in: 2010 Third International Conference on Software Testing, Verification and Validation, 55–64, 2010.
- [60] J. S. Bradbury, J. R. Cordy, J. Dingel, Comparative Assessment of Testing and Model Checking Using Program Mutation, in: Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007), 210–219, 2007.
- [61] G. Tretmans, CONCUR'99 - Concurrency Theory: 10th International Conference, chap. Testing Concurrent Systems: A Formal Approach, Lecture Notes in Computer Science, Springer Verlag, 46–65, 1999.
- [62] N. Razavi, A. Farzan, S. A. McIlraith, Generating effective tests for concurrent programs via AI automated planning techniques, International Journal on Software Tools for Technology Transfer: STTT 16 (1) (2014) 49–65.
- [63] P. Thomson, A. F. Donaldson, A. Betts, Concurrency Testing Using Schedule Bounding: An Empirical Study, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '14, ACM, New York, NY, USA, ISBN 978-1-4503-2656-8, 15–28, 2014.
- [64] B. Demsky, P. Lam, SATCheck: SAT-directed Stateless Model Checking for SC and TSO, SIGPLAN Notes 50 (10) (2015) 20–36.
- [65] B. P. Wood, A. Sampson, L. Ceze, D. Grossman, Composable Specifications for Structured Shared-memory Communication, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, ACM, New York, NY, USA, ISBN 978-1-4503-0203-6, 140–159, 2010.
- [66] S. Khoshnood, M. Kusano, C. Wang, ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, ACM, New York, NY, USA, 165–176, 2015.
- [67] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, N. Sinha, Concurrent software verification with states, events, and deadlocks, Formal Aspects of Computing V17 (4) (2005) 461–483.
- [68] M. Musuvathi, S. Qadeer, Iterative Context Bounding for Systematic Testing of Multithreaded Programs, SIGPLAN Notes 42 (6) (2007) 446–455, ISSN 0362-1340.
- [69] D. Harmanci, P. Felber, V. Gramoli, M. SuBkraut, C. Fetzer, TMU-NIT: A Transactional Memory Unit Testing and Workload Generation Tool, Tech. Rep. RR-I-08-08.1, Informatics Institute, University of de Neuchâtel, Switzerland, 2008.
- [70] S. Bindal, S. Bansal, A. Lal, Variable and thread bounding for systematic testing of multithreaded programs, in: Proceedings of the 22nd International Symposium on Software Testing and Analysis, ACM, 145–155, 2013.
- [71] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha, TransDPOR: A Novel Dynamic Partial-order Reduction Technique for Testing Actor Programs, in: Proceedings of the International Conference on Formal Techniques for Distributed Systems, FMOODS'12/FORTE'12, Springer-Verlag, Berlin, Heidelberg, 219–234, 2012.
- [72] D. Adalid, A. Salmerón, M. D. M. Gallardo, P. Merino, Using SPIN for Automated Debugging of Infinite Executions of Java Programs, Journal of Systems and Software 90 (2014) 61–75, ISSN 0164-1212.
- [73] M. Musuvathi, S. Qadeer, Fair Stateless Model Checking, SIGPLAN Notes 43 (6) (2008) 362–371, ISSN 0362-1340.
- [74] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, S. Qadeer, Preemption Sealing for Efficient Concurrency Testing, in: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'10, Springer-Verlag, Berlin, Heidelberg, ISBN 3-642-12001-6, 978-3-642-12001-5, 420–434, 2010.
- [75] R. Carver, Y. Lei, Multicore Software Engineering, Performance, and Tools. MUSEPAT, C11 2013, St. Petersburg, Russia., in: J. M. Lourenço, E. Farchi (Eds.), Multicore Software Engineering, Performance, and Tools, chap. A Modular Approach to Model-Based Testing of Concurrent Program, Springer, Berlin, Heidelberg, 85–96, 2013.
- [76] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, D. Marinov, BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, ISBN 978-1-4673-1067-3, 727–737, 2012.
- [77] K. Sen, Effective Random Testing of Concurrent Programs, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, NY, USA, 323–332, 2007.
- [78] J. Chen, S. Macdonald, Incremental testing of concurrent programs using value schedules, in: IEEE/ACM International Conference on Automated Software Engineering, 313–322, 2008.
- [79] N. Machado, B. Lucia, L. Rodrigues, Concurrency Debugging with Differential Schedule Projections, SIGPLAN Notes 50 (6) (2015) 586–595, ISSN 0362-1340.
- [80] R. H. Carver, Y. Lei, Distributed reachability testing of concurrent programs., Concurrency and Computation: Practice and Experience 22 (18) (2010) 2445–2466.
- [81] X. Zhou, K. Lu, X. Wang, X. Li, Exploiting parallelism in deterministic shared memory multiprocessing., Journal of Parallel and Distributed Computing 72 (5) (2012) 716–727.
- [82] J. Turpie, E. Reisner, J. S. Foster, M. Hicks, MultiOtter: Multiprocess Symbolic Execution, Tech. Rep. CS-TR-4982, Department of Computer Science, University of Maryland, College Park, MD, 2011.
- [83] J. Staunton, J. A. Clark, Applications of Model Reuse when Using Estimation of Distribution Algorithms to Test Concurrent Software, in: Proceedings of the Third International Conference on Search Based Software Engineering, SSBSE'11, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-23715-7, 97–111, 2011.
- [84] K. Kähkönen, O. Saarikivi, K. Heljanko, Unfolding Based Automated Testing of Multithreaded Programs, Automated Software Engineering 22 (4) (2015) 475–515.
- [85] S. Easterbrook, Empirical Research Methods for Software Engineering, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, ISBN 978-1-59593-882-4, 574–574, 2007.
- [86] D. I. K. Sjöberg, T. Dyba, M. Jorgensen, The Future of Empirical Methods in Software Engineering Research, Future of Software Engineering 0 (2007) 358–378.
- [87] W. F. Tichy, P. Lukowicz, L. Prechelt, E. A. Heinz, Experimental Evaluation in Computer Science: A Quantitative Study, Journal of Systems and Software 28 (1) (1995) 9–18, ISSN 0164-1212.
- [88] Y. Eytani, K. Havelund, S. D. Stoller, S. Ur, Towards a framework and a benchmark for testing tools for multi-threaded programs., Concurrency and Computation: Practice and Experience 19 (3) (2007) 267–279.
- [89] R. K. Yin, Case Study Research: Design and Methods, 3rd Edition (Applied Social Research Methods, Vol. 5), SAGE Publications, 3rd edn., ISBN 0761925538, 2002.
- [90] C. Zannier, G. Melnik, F. Maurer, On the Success of Empirical Studies in the International Conference on Software Engineering, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, ISBN 1-59593-375-1, 341–350,

- 2006.
- [91] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, Norwell, MA, USA, ISBN 0-7923-8682-5, 2000.
 - [92] S. M. Melo, S. R. S. Souza, R. A. Silva, P. S. L. Souza, *Concurrent Software Testing in Practice: A Catalog of Tools*, in: *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation, A-TEST 2015*, ACM, New York, NY, USA, 31–40, 2015.
 - [93] S. M. Melo, P. S. L. Souza, S. R. S. Souza, *Towards an Empirical Study Design for Concurrent Software Testing*, in: *Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering, SE-HPCSE '16*, IEEE Press, Piscataway, NJ, USA, ISBN 978-1-5090-5224-0, 1–1, 2016.
 - [94] V. R. Basili, *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*, Tech. Rep., University of Maryland, College Park, MD, USA, 1992.
 - [95] V. R. Basili, F. Shull, F. Lanubile, *Building Knowledge Through Families of Experiments*, *IEEE Trans. Softw. Eng.* 25 (4) (1999) 456–473, ISSN 0098-5589.
 - [96] M. Höst, B. Regnell, C. Wohlin, *Using Students As Subjects: A Comparative Study of Students and Professionals in Lead-Time Impact Assessment*, *Empirical Software Engineering* 5 (3) (2000) 201–214, ISSN 1382-3256.
 - [97] G. G. M. Dourado, P. S. L. de Souza, R. R. Prado, R. N. Batista, S. R. S. Souza, J. C. Estrella, S. M. Bruschi, J. Lourenço, *A Suite of Java Message-Passing Benchmarks to Support the Validation of Testing Models, Criteria and Tools*, in: *International Conference on Computational Science, ICCS*, 2226–2230, 2016.
 - [98] N. Rungta, E. G. Mercer, *Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs*, in: *7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 9:1–9:10, 2009.
 - [99] Y. Eytani, S. Ur, *Compiling a Benchmark of Documented Multithreaded Bugs*, in: *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, ISBN 0-7695-2132-0, 0–0, 2004.
 - [100] Y. Yang, *Inspect: A Framework for Dynamic Verification of Multithreaded C Programs*, URL <http://www.cs.utah.edu/~yuyang/inspect/>, accessed July, 2016.
 - [101] Valgrind-Developers, *Valgrind-3.6.1*, URL <http://valgrind.org/>, accessed July, 2016.
 - [102] C. M. Lott, H. D. Rombach, *Repeatable software engineering experiments for comparing defect-detection techniques*, *Empirical Software Engineering* 1 (3) (1996) 241–277.
 - [103] N. Juristo, A. M. Moreno, *Basics of Software Engineering Experimentation*, Springer Publishing Company, Incorporated, 1st edn., ISBN 1441950117, 9781441950116, 2010.
 - [104] D. I. K. Sjöberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, M. Vokac, *Conducting realistic experiments in software engineering*, in: *International Symposium on Empirical Software Engineering, ISESE '02*, IEEE Computer Society, Washington, DC, USA, 17–26, 2002.
 - [105] I. da Costa Araújo, W. O. da Silva, J. B. de Sousa Nunes, F. O. Neto, *AR-RETT: A Framework to Create Reproducible Experiments to Evaluate Software Testing Techniques*, in: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing, SAST*, ACM, New York, NY, USA, ISBN 978-1-4503-4766-2, 1:1–1:10, 2016.
 - [106] E. Deniz, A. Sen, J. Holt, *Verification and Coverage of Message Passing Multicore Applications*, *ACM Transactions on Design Automation of Electronic Systems* 17 (3) (2012) 23:1–23:31, ISSN 1084-4309.
 - [107] S. Steenbuck, G. Fraser, *Generating Unit Tests for Concurrent Classes*, in: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 144–153, 2013.
 - [108] N. Rungta, E. G. Mercer, *Hardware and Software: Verification and Testing*: 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27–30, 2008. *Proceedings*, chap. A Meta Heuristic for Effectively Detecting Concurrency Errors, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-01702-5, 23–37, 2009.
 - [109] B. Krena, Z. Letko, T. Vojnar, S. Ur, *A Platform for Search-based Testing of Concurrent Software*, in: *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '10*, ACM, New York, NY, USA, ISBN 978-1-4503-0136-7, 48–58, 2010.
 - [110] B. Krena, Z. Letko, T. Vojnar, *Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software*, in: *Proceedings of the Second International Conference on Runtime Verification, RV'11*, Springer-Verlag, Berlin, Heidelberg, 177–192, 2012.
 - [111] D. Deng, W. Zhang, S. Lu, *Efficient Concurrency-bug Detection Across Inputs*, *SIGPLAN Notes* 48 (10) (2013) 785–802, ISSN 0362-1340.
 - [112] S. Tasharofi, M. Pradel, Y. Lin, R. E. Johnson, *Bitar: Coverage-guided, automatic testing of actor programs*, in: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, Silicon Valley, CA, USA, November 11–15, 2013, 114–124, 2013.
 - [113] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, C.-T. Yang, *Automatic Testing Environment for Multicore Embedded software-ATEMES*, *J. Syst. Softw.* 85 (1) (2012) 43–60, ISSN 0164-1212.
 - [114] C. Artho, A. Biere, S. Honiden, *Enforcer - Efficient Failure Injection*, in: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, Hamilton, Canada, August 21–27, 2006, *Proceedings*, 412–427, 2006.
 - [115] T. Yu, A. Sung, W. Srisa-An, G. Rothermel, *An Approach to Testing Commercial Embedded Systems*, *J. Syst. Softw.* 88 (2014) 207–230, ISSN 0164-1212.
 - [116] S. Guo, M. Kusano, C. Wang, Z. Yang, A. Gupta, *Assertion Guided Symbolic Execution of Multithreaded Programs*, in: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, ACM, New York, NY, USA, ISBN 978-1-4503-3675-8, 854–865, 2015.
 - [117] V. Terragni, S.-C. Cheung, C. Zhang, *RECONTEST: Effective Regression Testing of Concurrent Programs*, in: *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, IEEE Press, Piscataway, NJ, USA, 246–256, 2015.
 - [118] T. Tsui, S. Wu, W. K. Chan, *Toward a Methodology to Expose Partially Fixed Concurrency Bugs in Modified Multithreaded Programs*, in: *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, ACM, NY, USA, 49–56, 2014.
 - [119] Y. Lei, R. H. Carver, *Reachability Testing of Concurrent Programs*, *IEEE Transactions on Software Engineering* 32 (6) (2006) 382–403.
 - [120] G.-H. Hwang, C.-S. Lin, T.-S. Lee, C. Wu-Lee, *A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs*, *Software Testing, Verification and Reliability* 24 (8) (2014) 706–737.
 - [121] K. Sen, G. Agha, *A Race-detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs*, in: *Proceedings of the 2Nd International Haifa Verification Conference on Hardware and Software, Verification and Testing, HVC'06*, Springer-Verlag, Berlin, Heidelberg, 166–182, 2007.
 - [122] A. Farzan, A. Holzer, N. Razavi, H. Veith, *Con2Colic Testing*, in: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, ACM, New York, NY, USA, 37–47, 2013.
 - [123] T. E. Setiadi, A. Ohsuga, M. Maekawa, *Efficient Execution Path Exploration for Detecting Races in Concurrent Programs*, *International Journal of Computer Science* 40 (3) (2013) 15 – 34.
 - [124] M. Olszewski, J. Ansel, S. Amarasinghe, *Kendo: Efficient Deterministic Multithreading in Software*, *SIGPLAN Notes* 44 (3) (2009) 97–108, ISSN 0362-1340.
 - [125] Q. Luo, S. Zhang, J. Zhao, M. Hu, *A Lightweight and Portable Approach to Making Concurrent Failures Reproducible*, in: *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, FASE'10*, Springer-Verlag, Berlin, Heidelberg, 323–337, 2010.
 - [126] A. Nistor, D. Marinov, J. Torrellas, *InstantCheck: Checking the Determinism of Parallel Programs Using On-the-Fly Incremental Hashing*, in: *IEEE/ACM International Symposium on Microarchitecture*, 251–262, 2010.
 - [127] M. Dhok, R. Mudduluru, M. K. Ramanathan, *Pegasus: Automatic Barrier Inference for Stable Multithreaded Systems*, in: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISTA 2015*, ACM, New York, NY, USA, ISBN 978-1-4503-3620-8, 153–164, 2015.
 - [128] X. Yuan, Z. Wang, C. Wu, P.-C. Yew, W. Wang, J. Li, D. Xu, Syn-

- chronization Identification Through On-the-fly Test, in: Proceedings of the 19th International Conference on Parallel Processing, Euro-Par'13, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-40046-9, 4–15, 2013.
- [129] R. Xin, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, H. Guan, An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs, in: IEEE International Conference on Software Maintenance (ICSM), 100–109, 2013.
- [130] N. Rungta, E. G. Mercer, Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs, in: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '09, ACM, New York, NY, USA, 9:1–9:10, 2009.
- [131] S. Nagarakatte, S. Burckhardt, M. M. Martin, M. Musuvathi, Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, ACM, NY, USA, ISBN 978-1-4503-1205-9, 543–554, 2012.
- [132] S. Burckhardt, P. Kothari, M. Musuvathi, S. Nagarakatte, A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs, SIGPLAN Notes 45 (3) (2010) 167–178, ISSN 0362-1340.
- [133] H. Kim, D. Cho, S. Moon, Maximizing synchronization coverage via controlling thread schedule, in: 2011 IEEE Consumer Communications and Networking Conference (CCNC), 1186–1191, 2011.
- [134] C.-S. Park, K. Sen, P. Hargrove, C. Iancu, Efficient Data Race Detection for Distributed Memory Parallel Programs, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, ISBN 978-1-4503-0771-0, 51:1–51:12, 2011.
- [135] J. Burnim, K. Sen, C. Stergiou, Testing Concurrent Programs on Relaxed Memory Models, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, ACM, New York, NY, USA, ISBN 978-1-4503-0562-4, 122–132, 2011.
- [136] J. Fiedor, V. Hrubá, B. Krena, Z. Letko, S. Ur, T. Vojnar, Advances in noise-based testing of concurrent software., *Softw. Test., Verif. Reliab.* 25 (3) (2015) 272–309.
- [137] K. Zhai, B. Xu, W. K. Chan, T. H. Tse, CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications., in: M. P. E. Heimdahl, Z. Su (Eds.), International Symposium on Software Testing and Analysis, ISSTA, ACM, ISBN 978-1-4503-1454-1, 221–231, 2012.
- [138] C. Wang, M. Said, A. Gupta, Coverage Guided Systematic Concurrency Testing, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, ISBN 978-1-4503-0445-0, 221–230, 2011.
- [139] A. K. Jha, S. Jeong, W. J. Lee, Value-deterministic Search-based Replay for Android Multithreaded Applications, in: Proceedings of the 2013 Research in Adaptive and Convergent Systems, RACS '13, ACM, New York, NY, USA, ISBN 978-1-4503-2348-2, 381–386, 2013.
- [140] S. Wu, C. Yang, C. Jia, W. K. Chan, ASP: Abstraction Subspace Partitioning for Detection of Atomicity Violations with an Empirical Study, *IEEE Transactions on Parallel and Distributed Systems* 27 (3) (2016) 724–734.
- [141] L. Zhang, C. Wang, Runtime Prevention of Concurrency Related Type-state Violations in Multithreaded Applications, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, ISBN 978-1-4503-2645-2, 1–12, 2014.
- [142] K. Sen, Race Directed Random Testing of Concurrent Programs, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, ACM, NY, USA, 11–21, 2008.
- [143] L. Wang, S. D. Stoller, Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs, in: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '06, ACM, New York, NY, USA, ISBN 1-59593-189-9, 137–146, 2006.
- [144] J. Schimmel, K. Molitorisz, W. F. Tichy, An Evaluation of Data Race Detectors Using Bug Repositories, in: M. Ganzha, L. A. Maciaszek, M. Paprzycki (Eds.), FedCSIS, 1349–1352, 2013.
- [145] W. Zhang, C. Sun, S. Lu, ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach, *SIGPLAN Notes* 45 (3) (2010) 179–192.
- [146] L. Wang, S. D. Stoller, Runtime analysis of atomicity for multithreaded programs, *IEEE Transactions on Software Engineering* 32 (2) (2006) 93–110.
- [147] D. Li, W. Srisa-an, M. B. Dwyer, SOS: Saving Time in Dynamic Race Detection with Stationary Analysis, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, ACM, NY, USA, 35–50, 2011.
- [148] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, C. Flanagan, Sound Predictive Race Detection in Polynomial Time, *SIGPLAN Not.* 47 (1) (2012) 387–400, ISSN 0362-1340.
- [149] M. W. Aziz, S. A. B. Shah, Test-Data Generation for Testing Parallel Real-Time Systems, in: International Conference Testing Software and Systems, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, 211–223, 2015.
- [150] W. Araujo, L. C. Briand, Y. Labiche, On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Functional Faults in Concurrent Object-Oriented Software, *IEEE Transactions on Software Engineering* 40 (10) (2014) 971–992.

Appendix A.

Appendix A.1. Data extraction form

Appendix A.2. Empirical evidence on concurrent software testing.

Study description	Study identifier	Unique numeric id for the study.
	Study title	Article title.
	Date of data extraction	Date of the study analysis.
	Bibliography reference	Bibliography reference.
	Type of article	Kind of publication (journal article, conference paper, PhD thesis, technical report, among others).
	Publication channel	Name of the conference, journal or institution where the study was published.
	Study aims	General objectives of the paper (purpose of the research).
Empirical evaluation	Objectives	Goals of the empirical evaluation.
	Design of study	Research method adopted by the author to conduct the empirical study.
	Research question	Describe a research question, if any.
	Research hypothesis	Describe the hypothesis of the empirical study, if any.
	Testing technique	Type of testing technique evaluated by the empirical study.
	Comparison	Verify if the empirical study presents a comparison of the proposed testing technique/tool with others.
	Sample description	Information about the sample used in the study.
	Language	Programming language supported by the testing technique evaluated in the empirical study.
	Control group	If the study presents a control group and the number of groups and sample size.
	Data analysis	If the data analysis is qualitative or quantitative.
	Metrics	The metrics used to measure the dependent variable.
	Statistical analysis	If the study presents a statistical analysis.
	Aspects evaluated	Aspect or technique evaluated by the empirical study.
	Data collection	How the data was obtained.
	Response variable	Dependent variable of the empirical study.
Study findings	Findings and conclusions	Mainly finds and conclusions of the empirical study.
	Validity	If the study discuss threats to validity.
	Relevance	If the empirical study has useful results for research or industry.

Table A.12: Empirical evidence on structural testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Chen et al. [52]	Data race detection, detecting data race and atomicity violations, dynamic testing	Case study	Performance overhead and effectiveness	Run time, number of redundant contexts, number of total context checks performed, atomicity violation coverage	9 real world programs Java	Compare SAM with Artemis	Descriptive	N
Deniz et al. [106]	Mutation-based coverage metrics	Case study	Performance	Running time (seconds) and memory usage (megabytes)	5 benchmarks, 6 author examples MCAPI	N	Descriptive	N
Steenbuck and Fraser [107]	GA algorithm, coverage criterion	Case study	Effectiveness	Number of data races found	12 example programs Java	N	Descriptive	Y
Rungta and Mercer [108]	Greedy depth-first search	Case study	Effectiveness	Density of error discovery	3 benchmarks, 3 real world programs Java	Compare meta heuristic to randomized depth-first search	Descriptive	N
Krena et al. [109]	Searchbestie (concurrency search based testing)	Case study	Capability to find bugs	Percentage of code coverage	4 example, 1 real world Java	Compare a simple hill climbing algorithm with a random search algorithm	Descriptive	N
Krena et al. [110]	Coverage metrics (ConcurPairs, DUPair, Sync)	Case study	Accuracy	Percentage of code coverage	5 real world programs Java	Compare the proposed metrics with previously proposed metrics gathered by several dynamic analyses: Eraser, GoldILocks, Avio, GoodLock	Descriptive	N
Brito et al. [113]	Data and control flow criteria	Experiment	Cost, efficiency, effectiveness	Number of test cases to cover a criterion, probability to satisfy a testing criterion using a test set adequate to another testing criterion, percentage of defects detected	8 example programs C/MPI	N	Inferential	Y
Joisha et al. [35]	Data flow criteria	Case study	Performance	Compilation time	8 benchmarks C/C++	N	Descriptive	N
Souza et al. [14]	Data, control and communication flow criteria	Experiment	Efficiency, effectiveness	Number of faults found, number of faults injected, number of SYNsequences (or paths) executed for each approach	9 example programs C/MPI	Compare composite approach to both structural testing and reachability testing	Inferential	Y
Ganai et al. [36]	Dynamic taint analysis for control flow	Case study	Relevancy	Ratio of relevant inputs to the total number of inputs	9 benchmarks C/C++	Compare the result of relevancy analysis for different DTAM approaches	Descriptive	N
Sherman et al. [17]	Synchronization, wait-notify, inter-thread and data flow	Case study	Cost and effectiveness	Coverage values	8 example programs Java	N	Descriptive	N
Hong et al. [38]	Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF	Experiment	Effectiveness, coverage achieved, test suite size	Coverage achieved, test suite size, and fault detection	9 example programs Java	N	Inferential	Y
Hong et al. [27]	Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF	Experiment	Effectiveness, coverage achieved, test suite size	Coverage achieved, test suite size, and fault detection	9 example programs Java	Compare Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF coverage metrics	Inferential	Y
Tasiran et al. [37]	Location-Pairs	Case study	Effectiveness, coverage achieved, test suite size	Percentage of iterations for which a metric was successful	9 example programs Java	Compare concurrency bug detection approach (CFP) with the traditional approach (Full)	Descriptive	N
Deng et al. [111]	Concurrent Function Pairs CFP	Case study	Effectiveness, performance, false negatives, bug-report duplication rate, and trace size	Number of inputs that report each bug, number of selected inputs and functions, number of races and single-variable atomicity violations overlap across inputs, trace size and trace-analysis time	5 real applications C/C++	Compare concurrency bug detection approach (CFP) with the traditional approach that applies full-blown detection to every input (Full)	Descriptive	N
Tasharofi et al. [112]	Schedule coverage criteria	Case study	Effectiveness	Number of bugs detected, number of schedules generated, number of executions until detect the bug, average time to find a bug	5 real world Scala Actor programs	Compare Bita with random scheduling	Descriptive	N
Koong et al. [113]	Multi-round coverage ATEMES	Survey and Case study	Performance, reliability, usability	line coverage, response time, branch coverage, executing time, and failed test case data	13 example programs C/C++	N	Descriptive	N
Artho et al. [114]	Failure injection	Case study	Usefulness	Exception coverage after Enforcer was used	9 real world programs Java	Compare to manual approaches	Descriptive	N
Yu et al. [115]	Coverage-adequate criteria and test oracle	Experiment	Effectiveness	Number of faults in object programs detected by the different test suites	3 real world programs C	N	Descriptive	Y
Koppol et al. [31]	ALTS-based coverage criteria all-int-T-synchronizations and all-int-L-synchronizations	Case study	Effectiveness, effort, state space reduction	Number of transitions and states of a particular reduced ALTS, number of states in the reduced ALTS as a percentage of the number of states in the unreduced ALTS, average sizes of the program slices produced using algorithm, the number of int-T-synchronizations and int-L-synchronizations in the reduced models compared the level of effort required to satisfy the all-T-synchronization and all-L-synchronization coverage criteria, number of test paths	3 example programs CSS	N	Descriptive	N

Continued on next page

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Yu et al. [18]	Thread interleaving coverage-driven testing	Case study	Efficiency, effectiveness	Average performance overhead, number of successfully exposed iRoots/ number, total predicted iRoots	13 benchmarks C/C++	Compare Maple with: PCT, PCT-Large, RandDelay and CHES	Descriptive	N
Guo et al. [116]	Assertion guided, symbolic execution	Case study	Performance, effectiveness, scalability	Number of explored executions, run-time	50 benchmark and real world programs C/C++	N	Descriptive	N
Lu et al. [19]	Deterministic lazy release consistency (DLRC)	Case study	Accuracy, execution time	Number of synchronization operations, number of memory operations and the memory footprint	16 benchmark C/C++/Pthreads	Compare RFDet with DThreads	Descriptive	N
Lu et al. [21]	Deterministic multi-threading (DMT)	Case study	Performance	Execution time, memory overhead, disk space overhead	5 benchmarks Pthread	Compare DTM with a state-of-the-art DMT system — DThreads	Descriptive	N
Terragni et al. [117]	Recontest, dynamic CIA	Case study	Effectiveness, efficiency, correctness	Number of potential violations detected by the number or violations predicted, computation time, number of violations reported by Recontest	13 real world programs Java	Compare Recontest with CAPP (stress testing)	Descriptive	Y
Tsui et al. [118]	Similarity-based active testing strategy	Case study	Performance	Testing time	3 benchmarks Pthreads	Compare Maple and Regression-Maple	Descriptive	N
Lei et al. [20]	SYN-sequences	Case study	Effectiveness, adequacy	Number of synchronization sequences exercised during reachability testing	5 example programs Java	Compare the performance of RichTest and VeriSoft	Descriptive	N
Lei and Carver [119]	SYN-sequences	Case study	Performance, effectiveness	Number of sequences exercised, number of times the programs are executed and the number of transitions that are explored	4 example programs C++	Compare Richtest and VeriSoft	Descriptive	N
Hwang et al. [120]	SYN-sequences	Case study	Feasibility	Deadlocks detected, time, max memory usage, number of states, number of threads used	7 benchmarks Java	Compare depth-first search and breadth-first search	Descriptive	N
Sen and Agha [121]	Concolic testing	Case study	Effectiveness	Run time in seconds, number of Paths, number of races/deadlocks/infinite loops/exceptions, number of threads, number of functions tested	13 real world benchmarks Java	Compare Jcute and JPF	Descriptive	N
Farzan et al. [122]	Con2colic testing	Case study	Effectiveness, scalability	Diagnosis time, number of iterations required to complete the diagnosis, average size of the root cause returned by the proposed method	14 example programs Benchmarks Java	Compared Concrest with Poirot	Descriptive	N
Setiadi et al. [123]	Thread-Pair-Interleaving (TPAIR) criterion	Case study	Effectiveness	Number of test cases, number of interleavings	5 benchmarks Java	Compare the number of test cases against an existing test case reduction method based on the TPAIR criterion	Descriptive	N
Olszewski et al. [124]	Deterministic multi-threading	Case study	Performance	Percentage of nondeterministic (pthreads), execution time	9 real world benchmarks Pthreads	Compare the performance of the proposed lazy read API to using deterministic locks	Descriptive	N
Luo et al. [125]	Replay-based	Case study	Effectiveness, efficiency, performance overhead, crash reproducibility	Effectiveness: number of generated test cases and reproducible test cases; overhead: execution time, failure reproducible	6 benchmarks Java	N	Descriptive	Y
Nistor et al. [126]	On-the-fly incremental hashing	Case study	Performance, determinism checking	Overhead, detected bug, number of determinism checks	17 real world programs C++	N	Descriptive	N
Dhok et al. [127]	Automatic Barrier Inference	Case study	Effectiveness, performance	Execution time	5 real world programs C/C++	Compare the ability of Pegasus to address performance problems across versions of manually inserted barriers	Descriptive	N
Yuan et al. [128]	Data race detection, synchronization identification	Case study	Effectiveness, efficiency, running time	Identified sync pairs, number of false positives, number of testing threads, testing points, running time of SyncTester, throughput	15 real world benchmarks Pthreads	Compare SyncTesters with two existing dynamic test schemes, ISSA08 and Helgrind+	Descriptive	N
Xin et al. [129]	Synchronization coverage	Experiment	Lock manifestation, lock pattern and lock usage evolution	Lock manifestation, lock pattern and lock usage evolution	4 real world programs Java	N	Descriptive	N
Rungta and Mercer [130]	Stateless random walk, randomized depth-first search, guided search	Case study	Effectiveness	Hardness, bound, trials	3 benchmarks Java/C#	Compare tools: CalFuzzer, ConTest, CHES, and Java Pathfinder	Descriptive	N
Nagarakatte et al. [131]	PPCT randomized scheduler	Case study	Effectiveness, performance	Runs with the bug found, execution time overhead	9 real world programs C++	Compare PCCT with PCT	Descriptive	N
Burckhardt et al. [132]	Randomized Scheduler	Case study	Effectiveness	Percentage of bugs detected	8 real world programs C/C++	Compare PCT with stress, synchronization based random sleeps	Descriptive	N
Kim et al. [133]	Synchronization coverage, active scheduling	Case study	Effectiveness	Time to achieve full coverage	6 real world Pthread	Compare with basic random testing	Descriptive	N
Park et al. [134]	Active testing	Case study	Data race detection, scalability	Runtime, overhead of the program with race prediction enabled and the average runtime and overhead for program re-execution with race confirmation, total number of potential races, speedup	14 benchmarks C/UPC	N	Descriptive	N
Burnim et al. [135]	Active testing	Case study	Feasibility	Average runtime (seconds), cycles predicted, cycles confirmed, number of bugs identified, probability of confirming a cycle	7 benchmarks C	N	Descriptive	N
Fiedor et al. [136]	Noise-based testing	Survey	Efficiency of detecting concurrency-related errors	Time, number of test cases	8 Java, 4 C benchmarks	N	Descriptive	N
Zhai et al. [137]	Context sensitive	Case study	Effectiveness	Detection rate, time overhead: slowdown of the strategy and space overhead: memory consumption	9 benchmarks Java	N	Descriptive	Y

Continued on next page

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Wang et al. [138]	Coverage-guided selective search	Case study	Performance, effectiveness; bug detection capability	Number of interleavings tested, run time in seconds, number of bugs detected	6 real world programs C/C++/Pthreads	Compare HaPSet, DPOR, and PCB	Descriptive	N
Jha et al. [139]	Value-deterministic search-based replay	Case study	Efficiency	Compilation Time (CT) in second. Number of Line of Code (LOC). Number of Classes (NOC). Number of Reported Data Race (RDR). Number of Actual Data Race (ADR). Type of Data Race (TDR). Output Result (OR). Tool Usage (TU)	8 real world applications Java	N	Descriptive	N
Christakis and Sagonas [55]	Message passing errors in Erlang	Case study	Effectiveness, performance	Number of message passing problems identified by the analysis, elapsed wall clock time (in seconds), and memory requirements (in MB)	15 real world programs Erlang	N	Descriptive	N

Table A.13: Empirical evidence on fault based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Bradbury et al. [60]	ConMan operators	Experiment	Effectiveness	Number of faults detected	4 example programs Java	N	Descriptive	Y
Gligoric et al. [29]	MutMut, selective mutation	Experiment	Performance	Execution time	12 example programs Java	N	Descriptive	Y
Kester et al. [15]	Static analysis	Survey	Effectiveness	Number of bugs detected and percentage of spurious results produced by the tool	12 benchmarks Java	N	Descriptive	Y
Alowibdi and Stenneth [28]	Data race detection	Survey	Correctness, performances, and effectiveness	Compilation Time (CT) in second. Number of Line of Code (LOC). Number of Classes (NOC). Number of Reported Data Race (RDR). Number of Actual Data Race (ADR). Type of Data Race (TDR). Output Result (OR). Tool Usage (TU)	5 benchmarks Java	Compare tools - active, testing, static analysis, aspect-oriented approach, static and dynamic, program analysis, model checking	Descriptive	N
Wu et al. [140]	Partition-based	Experiment	Cost-effectiveness	Percentage of bugs detected versus the percentage of suspicious instances confirmed	10 benchmarks Java	Comparisons among the untreated ordering (U), random ordering (R), and our family of techniques (F)	Inferential	Y
Kasicki et al. [42]	Data race detection	Case study	Effectiveness, Accuracy and Precision, Performance (Time to Classify)	Number of Bugs finding and states coverage, classification accuracy, number of executions per time, running time memory overhead, time to classify	11 real world programs C/C++	Compare Portend+ to the Record/Replay Analyzer technique	Descriptive	N
Safi et al. [53]	Data race detection, Event-based interaction	Case study	Accuracy and execution time	Execution time, number of false positives, number of false negatives	20 real world programs Java	Compare DEvA with CAFA and Chord.	Descriptive	N
Lu et al. [30]	Deadlock detection, transactional memory	Survey	Bug pattern, manifestation, fix strategy	Number of bugs detected	4 real world programs, 105 bugs C/C++	Compare strategy to deadlock detection and transactional memory	Descriptive	Y
Sheng et al. [39]	Dynamic data race detection	Case study	Effectiveness, efficiency	Overhead and false positives	2 real world programs C/C++	N	Descriptive	N
Zhang and Wang [141]	Prevention of low level bugs	Case study	Effectiveness, performance, scalability	Effectiveness: suppression of type-state violations performance: overhead of runtime failure mitigation scalability: increasing the number threads and runtime overhead	11 real world programs C/C++/Pthreads	Compare the proposed approach to other methods for runtime prevention of low-level concurrency bugs	Descriptive	N
Gligoric et al. [16]	MutMut, selective mutation	Experiment	Effectiveness and cost	Execution time	14 real world programs Java	Compare concurrent mutation operators and sequential mutation operators	Descriptive	Y
Sen [142]	Randomized dynamic analysis	Case study	Effectiveness	Average runtime, number of potential races detected, number of real races reported, number of exceptions	14 real world programs Java	N	Descriptive	N
Kestor et al. [56]	Data race detection, relaxed transactional data race	Case study	Effectiveness, performance	Number of existing and injected data races detected by tools, ratio of the execution time, run-time overhead	8 real world programs C/C++	Compares TRADE with s-TRADE	Descriptive	N
Wang and Stoller [143]	Atomicity violation detection	Case study	Performance, accuracy	Running time	12 example programs Java	Compare the proposed algorithm for view-atomicity with the off-line reduction-based algorithm and the pairwise block-based algorithm	Descriptive	N
Schimmel et al. [144]	Data race detection	Case study	Efficiency	Bugs detected	25 real world bugs Java	Compare four data race, detectors, three dynamic: MTRAT, ConTest, Jinx, and one static tool: Jchord	Descriptive	N
Zhai et al. [58]	Deadlock detection	Case study	Effectiveness	Deadlocks detected	15 real world bugs C/C++/Java	Compare ASN to PCT, Magic-Scheduler and DeadlockFuzzer	Descriptive	N
Huang et al. [57]	Reproduce concurrent bugs (CLAP)	Case study	Effectiveness to reproduce bugs, performance	Runtime and space overhead, number of schedules, number of bugs reproduced	20 real world programs Java	Comparison between CLAP and LEAP	Descriptive	N
Zhang et al. [145]	Data race detection, consequence oriented approach ConSeq	Case study	Effectiveness, accuracy, performance	Number of bugs detected, number of false positives, runtime overhead	7 real world programs C/C++	Compare race detector and atomicity-violation detector	Descriptive	N
Park et al. [46]	Atomicity violation detection	Case study	Effectiveness, efficiency, reproducibility	Time spent to reproduce an exposed bug	7 real world programs C/C++	Compare CTrigger with stress testing	Descriptive	N
Cai and Cao [40]	Data race detection, detect hidden races	Case study	Effectiveness, performance,	Thrashing Rate Number of races detected, overhead time, Thrashing Rate	1 real world, 7 benchmarks Java	Compare DrFinder, FastTrack, ConTest, and PCT	Descriptive	N
Lu et al. [47]	Atomicity violation detection	Case study	Effectiveness, efficiency, reproducibility	Bug-exposing time, CTrigger bug-exposing time breakdown, unserializable interleaving coverage	7 real world benchmarks C/C++	Compare CTrigger with four other bug-exposing mechanisms on the same platform: Stress, Pure-Pin, Race-based	Descriptive	N
Chew and Lie [49]	Atomicity violation detection	Case study	Effectiveness, performance	Number of bugs detected, number of false positives, run-time overhead	5 example programs Java	N	Descriptive	N
Samak and Ramanathan [45]	Deadlock detection	Case study	Effectiveness	Number of deadlocks synthesized, numbers corresponding to the synthesized multithreaded tests and the deadlocks detected	8 real world classes Java	N	Descriptive	N
Maiya et al. [54]	Data race detection, happens before reasoning	Case study	Performance	Number of data races reported, number of reports generated by DROIDRACER, number of true and false positives	10 real world open-source applications, 5 proprietary applications Java	N	Descriptive	N
Wang and Stoller [146]	Atomicity violation detection	Case study	Usability, accuracy, performance	Atomicity violations reported, the number of missed errors for the online reduction algorithm, running time	12 example programs Java	Compare just the three algorithms proposed: online reduction-based, offline reduction-based, and block-based	Descriptive	N
Li et al. [147]	Data race detection	Case study	Performance, effectiveness	Runtime, race detected, overhead	6 benchmarks Java	Compared the effectiveness of approaches to detect races in deployed environments with Pacer (a sampling based race detector based on FastTrack)	Descriptive	N

Continued on next page

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Lu et al. [48]	Atomicity violation detection	Case study	Effectiveness	Number of bugs detected, number of false positives	3 real world, 5 benchmark programs C/C++	N	Descriptive	N
Park et al. [134]	Active testing	Case study	Data race detection, scalability	Runtime, overhead of the program with race prediction enabled and the average runtime and overhead for program re-execution with race confirmation, total number of potential races, speedup	14 benchmarks C/UPC	N	Descriptive	N
Gligoric et al. [59]	MutMut, selective mutation	Experiment	Performance	Execution time	8 benchmarks Java	N	Descriptive	N
Smaragdakis et al. [148]	Data race detection, causally-precedes (CP)	Case study	Performance, prediction capability	Number of races detected, running time	12 real world programs Java	N	Descriptive	N
Samak et al. [51]	Data race detection	Case study	Effectiveness, efficacy	Races detected, synthesized test count, synthesis time	7 real world Java library	N	Descriptive	N
Samak and manathan [50]	Atomicity violation detection	Case study	Effectiveness	Number of synthesized tests, number of true atomicity violations, time	7 real world java libraries Java	Compare Intruder with ConTeGe, Omen and Narada	Descriptive	N

Table A.14: Empirical evidence on formal method-based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Razavi et al. [62]	AI planning	Case study	Effectiveness, performance, predictive	Number of predicted runs, time required for each prediction, number of bugs found during the tests for each violation type	8 real world programs Java	N	Descriptive	N
Thomson et al. [63]	Scheduling bounding	Case study	Effectiveness	Number of schedules, time, bugs found	52 benchmarks C/C++/Java	Compare straightforward depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB) and the use of a naive random scheduler (Rand)	Descriptive	Y
Demsky and Lam [64]	Model checking	Case study	Performance	Execution time	5 real world benchmarks C	Comparing SAT-Directed Stateless Model Checking performance to the original DPOR stateless model checking algorithm implemented in CDSChecker, the source DPOR algorithm implemented in Nidhugg, and Check-Fence.	Descriptive	N
Wood et al. [65]	Composable Specifications	Case study	Precision and conciseness of annotations as well as the tool performance	Size of annotation, number of specified or exercised communication, run-time overhead (time and space), number or annotation	3 real world, 8 example programs Java	N	Descriptive	Y
Khoshnood et al. [66]	Logical constraint based symbolic analysis	Case study	Effectiveness	Reduction ratio, i.e., the number of constraints in the root cause divided by the average number of constraints in a bad schedule	34 benchmarks C	N	Descriptive	N
Chaki et al. [67]	Model checking	Case study	Effectiveness	Deadlock; maximum number of states; number of reachable states; number of iterations; time in seconds; memory usage in MB; effectiveness: number of states and transitions in the, number of states in the model; model construction time; model checking time; total verification time	3 real world programs C	N	Descriptive	N
Dolz et al. [41]	Data race detection	Case study	Effectiveness	Number of data races detected	39 real world programs C/C++/Pthreads	N	Descriptive	N
Musuvathi and Qadeer [68]	Model checking, iterative context-bounding	Case study	Efficacy	Number of states visited for the context-bounded and depth-first strategies both with and without fairness.	5 benchmarks C/C++/C#	N	Descriptive	N
Aziz and Shah [149]	GA Algorithms, test data generation	Case study	Performance and scalability	Execution time	1 real world benchmark C/Pthreads	Compare to randomly generated input data	Descriptive	N
Harmanci et al. [69]	Model checking, variable and thread bounding	Case study	Performance	Overhead of the dynamic execution	6 example programs C	Compare hand-written C code and TMUNIT	Descriptive	N
Bindal et al. [70]	Model checking	Case study	Performance, effectiveness	Running time, average number of executions and time required to capture a bug	9 benchmarks Java/C#	N	Descriptive	N
Abadi et al. [43]	Data race detection	Case study	Effectiveness	Number of annotations and time required to annotate each program, the number of race conditions found in each program, the time spent by the programmer inserting annotations and the time to run the tool	5 example programs Java	N	Descriptive	N
Tasharofi et al. [71]	Dynamic partial order reduction	Case study	Effectiveness	Number of paths, number of transitions, exploration time in seconds, and memory usage in MB	8 example programs ActorFoundry	Compare TransDPOR and DPOR	Descriptive	N
Adalid et al. [72]	Model checking	Case study	Efficiency	Hash projection, number or transitions, time, number of executions, number of states	4 real world servers Java applications	Compare TJT and the LTL extension for JPF (Java Path Finder)	Descriptive	N
Musuvathi and Qadeer [73]	Fair Stateless Model Checking	Case study	Efficacy, performance, effectiveness	Number of states visited for the context-bounded and depth-first strategies both with and without fairness	7 real world C++	N	Descriptive	N
Ball et al. [74]	Preemption sealing	Case study	Efficacy and efficiency of preemption sealing	Number of executions explored during testing, time, speedup	5 .NET libraries .Net	N	Descriptive	Y
Araujo et al. [150]	Designing contracts	Case study	Effectiveness	Probability of oracles detecting faults, average effort to diagnose faults normalized with the size of an execution thread	1 real world application Java	compare to manual analysis	Inferential	Y

Table A.15: Empirical evidence on model based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Carver and Lei [75]	LTS MODELS	Case study	Effectiveness	Number of: partially-ordered sequences, totally-ordered sequences, modular sequences	3 example programs Java	N	Descriptive	Y
Nistor et al. [76]	Clustering, bug detection	Experiment	Effectiveness	Number of transitions	14 real world bugs Java	N	Descriptive	Y
Cai et al. [44]	Constraint based	Case study	Effectiveness, performance	Performance: time, effectiveness on real deadlocks: number of real deadlock confirmed, effectiveness on false positives: number of false positive reported	3 benchmarks C/C++/Java	Comparisons among PCT, MagicScheduler (MS), DeadlockFuzzer (DF), and ConLock (CL)	Descriptive	Y
Sen [77]	Random partial order sampling & Randomized dynamic analysis	Case study	Effectiveness	Number of executions required to find a defect, number of partial orders sampled	12 example programs Java	Compare simple randomized algorithm and RAPOS (random partial order sampling)	Descriptive	Y
Chen and Macdonald [78]	Value-schedule-based technique	Case study	Functionality, applicability	Number of transitions and intermediate states required to cover all relevant behaviors of the program or discover a bug	9 example programs Java	N	Descriptive	N
Lu et al. [19]	Deterministic lazy release consistency (DLRC)	Case study	Accuracy, execution time	Number of synchronization operations, number of memory operations and the memory footprint	16 benchmark C/C++/Pthreads	Compare RFDet with DThreads	Descriptive	N
Machado et al. [79]	SMT solver, differential schedule projection	Case study	Efficiency, efficacy	Efficiency: execution time, efficacy: the number of program events and data-flow edges in the full execution	4 real world, 3 example programs C/C++/Java	N	Descriptive	N
Carver and Lei [80]	Interleaving free concurrency	Case study	Performance	Speedup	4 benchmarks, 1 Lotus specification Java/Lotus	N	Descriptive	N
Zhou et al. [81]	FPDet deterministic runtime	Case study	Determinism, performance, effectiveness of ABA, stability	Time, scalability, redistribution granularity and maximum execution budget	15 real world benchmark Java	Compare FPDet with three different deterministic runtime strategies: (1) the Nondet strategy which does not provide determinism of programs, (2) the DMP strategy which uses serial execution to eliminate data races, and (3) the CoreDet strategy	Descriptive	N
Turpie et al. [82]	Multiprocess symbolic execution	Case study	Performance	Line coverage, number of paths explored	2 real world programs C	N	Descriptive	N
Staunton and Clark [83]	Model reuse, estimation distributed algorithm EDA	Case study	Efficiency	Time to finding an error	3 example programs LTL	N	Inferential	N
Kähkönen et al. [84]	Dynamic symbolic execution with unfolding	Case study	Effectiveness, performance	Number of tests needed to fully cover the benchmarks and execution time	10 benchmarks Java	Compare unfolding approach to DPOR	Descriptive	N