Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída

Sofia Larissa da Costa Paiva

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: ______________________

Sofia Larissa da Costa Paiva

Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída

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

Orientador: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
Maio de 2016

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

P142a

Paiva, Sofia Larissa da Costa
Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída / Sofia Larissa da Costa Paiva; orientador Adenilso da Silva Simão. - São Carlos - SP, 2016.
151 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, 2016.

1. Teste baseado em modelos. 2. sistemas de transição com entrada/saída. 3. modelos de defeitos. 4. geração de casos de teste. 5. conjuntos de teste completos. I. Simão, Adenilso da Silva, orient. II. Título.

Sofia Larissa da Costa Paiva

Applying fault models in complete test suite generation from Input/Output Transition Systems

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

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
May 2016

À Deus, sem a qual não posso viver.

Ao meu esposo, Glesio.

Aos meus pais, Abner e Jeane.

AGRADECIMENTOS

Agradeço primeiramente à Deus, por ter me conservado com vida e por me dar forças para concluir essa etapa. Sem Ele nada posso.

Agradeço ao meu esposo Glesio pelo amor e compreensão, e especialmente por me incentivar e apoiar durante essa jornada. Sua presença foi fundamental para que eu chegasse até aqui.

A toda minha família, em especial meus pais, Abner e Jeane, minha eterna gratidão pelo apoio, atenção e presença, mesmo distantes. À minha irmã Deyse, pelo carinho e amizade sempre.

Agradeço ao meu orientador, professor Adenilso, pela orientação e confiança para realização deste trabalho.

Agradeço aos colegas de LabES de 2012 a 2016, não apenas pelas boas discussões em grupo, mas pelos momentos de bate-papo e distração que tornaram a jornada mais leve.

Aos professores do ICMC, especialmente às professoras Simone Senger e Rosana Braga, pelos ensinamentos e auxílio. Aos funcionários do ICMC pelo auxílio constante.

Ao professor Mohammad Reza Mousavi, pela colaboração no estudo de caso e pela recepção na visita técnica realizada na Universidade de Halmstad, Suécia.

Agradeço aqueles que auxiliaram na correção do texto: Ângela Giampedro, Carlos Diego Damasceno e Glesio Paiva.

Aos amigos que fiz em São Carlos, especialmente Yara, Erick, Jordanna e Iury pela companhia e amizade. Aos amigos que ficaram distantes na torcida, em especial a Carla Conte.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Processo 2012/09650-5, pelo apoio financeiro.

“Porque o SENHOR dá a sabedoria; da sua boca é que vem o conhecimento e o entendimento.”

Provérbios 2:6

Bíblia Sagrada, Edição Almeida Corrigida e Revisada.

RESUMO

PAIVA, S. L. C.. **Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída**. 2016. 155 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

O Teste Baseado em Modelos (TBM) emergiu como uma estratégia promissora para minimizar problemas relacionados à falta de tempo e recursos em teste de software e visa verificar se a implementação sob teste está em conformidade com sua especificação. Casos de teste são gerados automaticamente a partir de modelos comportamentais produzidos durante o ciclo de desenvolvimento de software. Entre as técnicas de modelagem existentes, Sistemas de Transição com Entrada/Saída (do inglês, Input/Output Transition Systems - IOTSs), são modelos amplamente utilizados no TBM por serem mais expressivos do que Máquinas de Estado Finito (MEFs). Apesar dos métodos existentes para geração de testes a partir de IOTSs, o problema da seleção de casos de testes é um tópico difícil e importante. Os métodos existentes para IOTS são não-determinísticos, ao contrário da teoria existente para MEFs, que fornece garantia de cobertura completa com base em um modelo de defeitos. Esta tese investiga a aplicação de modelos de defeitos em métodos determinísticos de geração de testes a partir de IOTSs. Foi proposto um método para geração de conjuntos de teste com base no método W para MEFs. O método gera conjuntos de teste de forma determinística além de satisfazer condições de suficiência de cobertura da especificação e de todos os defeitos do domínio de defeitos definido. Estudos empíricos avaliaram a aplicabilidade e eficácia do método proposto: resultados experimentais para analisar o custo de geração de conjuntos de teste utilizando IOTSs gerados aleatoriamente e um estudo de caso com especificações da indústria mostram a efetividade dos conjuntos gerados em relação ao método tradicional de Tretmans.

Palavras-chave: Teste baseado em modelos, sistemas de transição com entrada/saída, modelos de defeitos, geração de casos de teste, conjuntos de teste completos.

ABSTRACT

PAIVA, S. L. C.. **Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída**. 2016. 155 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Model-Based Testing (MBT) has emerged as a promising strategy for the minimization of problems related to time and resource limitations in software testing and aims at checking whether the implementation under test is in compliance with its specification. Test cases are automatically generated from behavioral models produced during the software development life cycle. Among the existing modeling techniques, Input/Output Transition Systems (IOTSs) have been widely used in MBT because they are more expressive than Finite State Machines (FSMs). Despite the existence of test generation methods for IOTSs, the problem of selection of test cases is an important and difficult topic. The current methods for IOTSs are non-deterministic, in contrast to the existing theory for FSMs that provides complete fault coverage guarantee based on a fault model. This manuscript addresses the application of fault models to deterministic test generation methods from IOTSs. A method for the test suite generation based on W method for FSMs is proposed for IOTSs. It generates test suites in a deterministic way and also satisfies sufficient conditions of specification coverage and all faults in a given fault domain. Empirical studies evaluated its applicability and effectiveness. Experimental results for the analyses of the cost of test suite generation by random IOTSs and a case study with specifications from the industry show the effectiveness of the test suites generated in relation to the traditional method of Tretmans.

Key-words: Model-based testing, input/output transition systems, fault models, test case generation, complete test suites.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelos utilizados em TBM 29
Figura 2 – Processo de TBM 40
Figura 3 – Exemplo de uma MEF 46
Figura 4 – Exemplo de um Sistema de Transição com Entrada e Saída (IOTS) 49
Figura 5 – $\triangle(S)$ - *Suspension Automata* obtido a partir do IOTS da Figura 4 51
Figura 6 – $det(\triangle(S))$ obtido pela determinização de $\triangle(S)$ da Figura 4 53
Figura 7 – Exemplo de um Caso de Teste para o IOTS da Figura 4 54
Figura 8 – Taxonomia estendida para o contexto de geração de testes a partir de IOTSs 62
Figura 9 – Distribuição dos estudos ao longo dos anos 63
Figura 10 – Tipo de publicação dos estudos 63
Figura 11 – Exemplo de IOTS 65
Figura 12 – Exemplo de IOLTS 66
Figura 13 – Exemplo de IOSTS 67
Figura 14 – Exemplo de TIOTS 70
Figura 15 – Exemplo de MIOTS 71
Figura 16 – Mapa dos estudos selecionados 75
Figura 17 – Modelos utilizados em TBM 78
Figura 18 – Um *IOTS(I,O)* M_1 82
Figura 19 – Árvore de teste para o IOTS M_1 88
Figura 20 – SCP representado como um IOTS 94
Figura 21 – Tela de geração de testes da ferramenta JTorX 96
Figura 22 – Resultados quando os estados estáveis e não-estáveis aumentam 103
Figura 23 – Resultados quando os estados estáveis aumentam 104
Figura 24 – Resultados quando os estados não-estáveis aumentam 105
Figura 25 – Resultados quando as entradas aumentam 106
Figura 26 – Resultados quando as saídas aumentam 107
Figura 27 – Resultados quando os estados estáveis e não-estáveis aumentam 108
Figura 28 – Resultados quando os estados estáveis aumentam 109
Figura 29 – Resultados quando os estados não-estáveis aumentam 110
Figura 30 – Resultados quando as entradas aumentam 111
Figura 31 – Resultados quando as saídas aumentam 112
Figura 32 – Especificação SBI 113
Figura 33 – Especificação EBI 114

Figura 34 – Especificação TIL 115
Figura 35 – Especificação EM 115
Figura 36 – Especificação AS 116
Figura 37 – Resultados para a especificação SBI 116
Figura 38 – Resultados para a especificação EBI 117
Figura 39 – Resultados para a especificação TIL 118
Figura 40 – Resultados para a especificação EM 119
Figura 41 – Resultados para a especficação AS 120
Figura 42 – Resultados para a especificação SCP 121
Figura 43 – Comparação da especificação SBI com a geração aleatória 121
Figura 44 – Comparação da especificação EBI com a geração aleatória 122
Figura 45 – Comparação da especificação TIL com a geração aleatória 122
Figura 46 – Comparação da especificação EM com a geração aleatória 123
Figura 47 – Comparação da especificação AS com a geração aleatória 123

LISTA DE ALGORITMOS

Algoritmo 1 – Construção do conjunto *transition cover* 86
Algoritmo 2 – Construção do conjunto de caracterização 89
Algoritmo 3 – Geração de conjuntos de teste *n*-completos 90
Algoritmo 4 – Algoritmo para geração aleatória de Mealy IOTSs 102

LISTA DE TABELAS

Tabela 1 – Lista de Controle 59
Tabela 2 – Estudos retornados e incluídos para cada fonte 60
Tabela 3 – Classificação quanto à geração de testes 67
Tabela 4 – Classificação das propriedades para Execução dos Testes 72
Tabela 5 – Resposta de cada um dos estados para a geração do conjunto W 89
Tabela 6 – Conteúdo de arquivo texto com especificação de IOTS 92
Tabela 7 – Conteúdo de arquivo texto com o conjunto de teste gerado pela ferramenta . 92
Tabela 8 – Comparação de resultados entre a JTorX e o Algoritmo 3 97
Tabela 9 – Condições para o funcionamento do monitor de velocidade máxima para trens 112
Tabela 10 – Características dos modelos utilizados no estudo de caso 117
Tabela 11 – Estudos selecionados 149

SUMÁRIO

1 INTRODUÇÃO **27**
1.1 Contextualização **27**
1.2 Definição do problema e justificativa para a pesquisa **30**
1.3 Objetivos do Trabalho **31**
1.4 Sumário dos Resultados Obtidos **32**
1.5 Organização da Tese **32**

2 FUNDAMENTAÇÃO TEÓRICA **35**
2.1 Fundamentos de Teste de Software **35**
2.1.1 Técnicas de Teste ***37***
2.2 Teste Baseado em Modelos **38**
2.2.1 Técnicas de Modelagem ***41***
2.3 Avaliação em Teste de Software **42**
2.3.1 Estudos Teóricos ***42***
2.3.2 Estudos Experimentais ***43***
2.4 Teste baseado em MEFs **45**
2.5 Teste baseado em IOTS **49**
2.5.1 Definição de IOTS ***49***
2.5.2 Framework para Teste de conformidade ***51***
2.5.2.1 Teste de conformidade com IOTSs *53*
2.6 Considerações Finais **56**

3 UM MAPEAMENTO SISTEMÁTICO SOBRE GERAÇÃO DE TESTES COM IOTS **57**
3.1 Planejamento e Condução do Estudo **58**
3.1.1 Questões de Pesquisa ***58***
3.1.2 Estratégia de Pesquisa ***58***
3.1.3 Processo de seleção e extração de dados dos estudos ***59***
3.2 Taxonomia para abordagens TBM **60**
3.3 Informações gerais sobre os estudos selecionados **61**
3.3.1 Distribuição dos estudos ao longo dos anos ***61***
3.3.2 Distribuição dos estudos nos fóruns ***61***
3.3.3 Grupos de pesquisa e autoria ***62***

3.3.4 Tipo de evidência *63*
3.3.5 Apoio computacional *64*
3.4 Classificação dos estudos **64**
3.4.1 QP1 - Características do modelo IOTS *64*
3.4.2 QP2 - Critérios de Seleção de Testes *66*
3.4.3 QP3 - Tecnologia *70*
3.4.4 QP4 - Execução dos Testes *71*
3.4.5 Mapa da geração de testes a partir de IOTS *74*
3.5 Considerações Finais **74**

4 MÉTODO W PARA MEALY IOTS **77**
4.1 Definições **79**
4.1.1 Definições sobre Teste *81*
4.2 Geração de conjuntos de teste completos para Mealy IOTS **82**
4.2.1 Condições para conjuntos de teste completos *83*
4.2.2 Algoritmo para geração de conjuntos de teste n-completos *85*
4.3 Aspectos de Implementação **91**
4.3.1 Geração de conjuntos de teste *91*
4.3.2 Execução de um conjunto de teste *92*
4.4 Estudo de caso: Protocolo de Controle de Sessão **93**
4.5 Considerações Finais **97**

5 AVALIAÇÃO DO MÉTODO PROPOSTO **99**
5.1 Resultados Experimentais **99**
5.1.1 Geração aleatória de Mealy IOTSs *101*
5.1.2 Análise dos Resultados *101*
5.1.2.1 Número de casos de teste *102*
5.1.2.2 Tamanho total dos conjuntos de teste *104*
5.2 Estudo de Caso **106**
5.2.1 Especificações *108*
5.2.1.1 Ceiling Speed Monitoring *109*
5.2.1.2 Indicadores de direção *112*
5.2.1.3 Body Comfort System *113*
5.2.2 Resultados *114*
5.3 Comparação do Estudo de Caso com Geração Aleatória **116**
5.4 Discussão dos Resultados **118**
5.5 Considerações Finais **119**

6 CONCLUSÕES **125**
6.1 Contribuições **125**

6.2 Limitações e Trabalhos Futuros 126
6.2.1 Possíveis extensões 127
6.3 Publicações resultantes 127

Referências 129

APÊNDICE A ESTUDOS SELECIONADOS NO MAPEAMENTO SISTEMÁTICO 149

LISTA DE ABREVIATURAS E SIGLAS

AS	-	Alarm System
CSM	-	Ceiling Speed Monitoring
EBI	-	Emergency Brake Intervention
EM	-	Exterior Mirror
GFC	-	Grafo de Fluxo de Controle
HSI	-	Harmonized State Identification
ioco	-	Input/Output Conformance
IOLTS	-	Input/Output Labeled Transition System
IOSTS	-	Input/Output Symbolic Transition System
IOTS	-	Input/Output Transition System
IUT	-	Implementation Under Test
Lotos	-	Language Of Temporal Ordering Specification
LTS	-	Labeled Transition System
MEF	-	Máquinas de Estado Finito
MEFE	-	Máquinas de Estado Finito Estendidas
MIOTS	-	Multi Input/Output Transition System
MS	-	Mapeamento Sistemático
SBI	-	Service Brake Intervention
SCP	-	Session Control Protocol
STS	-	Symbolic Transition System
SUT	-	System Under Test
TBM	-	Teste Baseado em Modelos
TIL	-	Turn Indicator Light
TIOTS	-	Timed Input/Output Transition System
UML	-	Unified Modeling Language
VV&T	-	Verificação, Validação e Teste

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização

O software tem se tornado um componente importante em muitas atividades complexas nos diferentes ramos da sociedade. Assim, é fundamental que os métodos, ferramentas e procedimentos de construção de software tenham um processo controlado e produtivo para que ele tenha alta qualidade. Nesse sentido, a Engenharia de Software surgiu com o intuito de aplicar princípios de engenharia no desenvolvimento de software de alta qualidade e baixo custo (Pressman, 2003).

Apesar dos métodos, ferramentas e técnicas empregados no processo de desenvolvimento de software, erros podem ocorrer durante esse processo. Para minimizar a ocorrência de tais erros, além de outros riscos associados, atividades de Validação, Verificação e Teste (VV&T) são aplicadas para verificar se o software foi construído em conformidade com a especificação. Dentro dessas atividades de VV&T, a atividade de teste de software é uma das mais utilizadas por aumentar a confiança de que o software apresenta as funções especificadas (Rocha et al., 2001). Essa atividade tem como objetivo executar um produto para determinar se ele atende as especificações e funciona corretamente no ambiente para o qual foi projetado. O teste é uma das atividades mais onerosas no processo de desenvolvimento de software (Hierons et al., 2009).

Uma das principais etapas dessa atividade é a geração de casos de teste, que é uma tarefa que exige grande esforço. Um caso de teste é uma sequência de ações de entradas e saídas esperadas de um sistema. Obter casos de teste não é uma tarefa trivial, uma vez que se deve selecionar um conjunto específico e finito de casos de teste, já que na maioria dos casos é impraticável utilizar todo o domínio de entrada do software em teste (Delamaro et al., 2007; Tretmans, 2008; Utting et al., 2012; Dias Neto e Travassos, 2009). Nesse contexto, a geração de conjuntos de casos de teste eficientes tem sido um tema de estudo de diversos pesquisadores, por ser um dos pontos mais importantes e cruciais da atividade de teste (Delamaro et al., 2007).

Nessa tarefa, o conceito de critério de teste é útil, de forma a aumentar as possibilidades em revelar a presença de defeitos e estabelecer um nível elevado de confiança na correção do produto (Maldonado et al., 2004; Rocha et al., 2001; Hierons et al., 2009).

Tecnologias que visam automatizar a geração de conjuntos de teste de alta qualidade são essenciais para garantir o sucesso do processo de teste e a qualidade e confiança de aplicações complexas. O Teste Baseado em Modelos (TBM) surgiu neste contexto como uma estratégia promissora para minimizar problemas relacionados à falta de tempo e recursos, apoiando as fases de projeto e geração de testes automatizados (Utting et al., 2012).

Existem esforços da comunidade para o desenvolvimento de métodos de TBM que automatizem o processo de teste e o tornem sistemático e formal, já que o TBM oferece maior confiança nos resultados e menor custo em sua aplicação (Simao, 2007; Tretmans, 2008; Dias Neto e Travassos, 2009). A partir de um modelo de comportamento, são empregadas técnicas para derivação de casos de teste. É importante que o modelo de teste adotado seja bem definido para que os testes gerados sejam significativos e tenham semântica precisa. Neste contexto, modelos formais podem ser analisados algoritmicamente, já que a linguagem natural pode conter ambiguidades e inconsistências. Devem ser criados modelos que descrevem as sequências de ações que podem ocorrer durante o uso do sistema. Existem diversos formalismos para modelar rigorosamente uma aplicação e que são adotadas para apoiar o TBM, como as Máquinas de Estado Finito (MEFs), MEF Estendidas (MEFEs), Sistemas de Transição Rotulados (do inglês, *Labeled Transition Systems (LTS)*) e diagramas UML (*Unified Modeling Language*) (Dias Neto e Travassos, 2009).

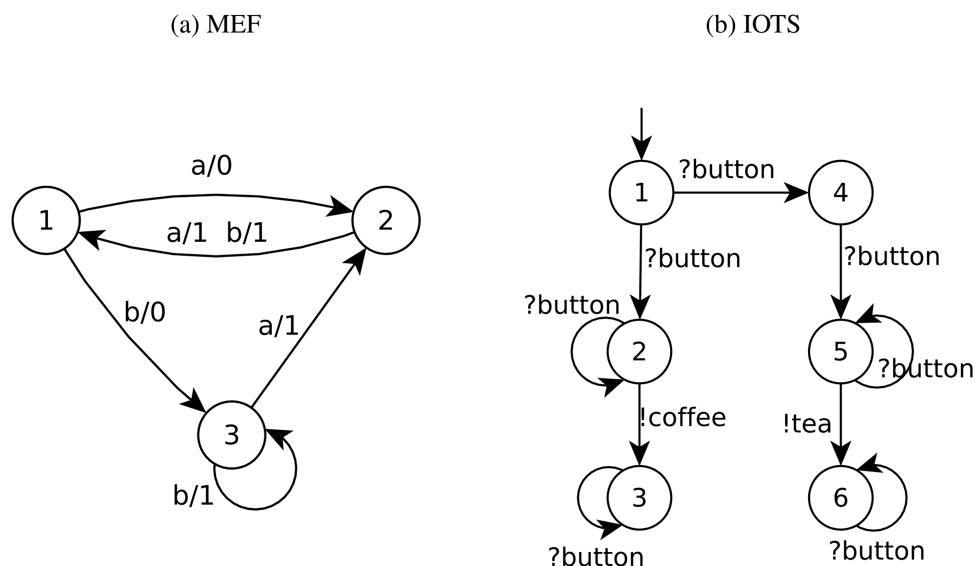
A indústria compreende o TBM como a execução automática de testes (Tretmans, 2008). Porém, da perspectiva acadêmica, TBM é uma extensão natural de métodos formais e técnicas de verificação que visam automatizar a geração adequada de casos de teste. O uso de modelos formais permite um modelo provido de embasamento matemático capaz de fornecer garantias significativas ao teste e que pode ser utilizado como a base para a automatização de partes do processo de teste, tornando-o mais eficiente e eficaz (Hierons et al., 2009). Esses modelos mais simples e precisos permitem fazer afirmações mais consistentes além de possibilitar uma melhor compreensão do sistema. Além disso, ambiguidades e inconsistências podem ser facilmente descobertas e corrigidas, proporcionando um software com menos defeitos (Simao, 2007; Gaudel, 2010a).

Um dos formalismos de modelagem amplamente utilizado é a Máquina de Estado Finito (MEF), um modelo simples e preciso que geralmente é expresso como uma máquina de Mealy (Lee e Yannakakis, 1996), como pode ser visto na Figura 1a (adaptada de (Simao e Petrenko, 2010b)). Este modelo é extensivamente utilizado por pesquisadores em métodos para geração de casos de teste para sistemas reativos e protocolos de comunicação (Hierons et al., 2009; Bochmann e Petrenko, 1994). Alguns destes métodos de geração destinam-se ao problema da geração de testes com cobertura garantida de todos os defeitos de um determinado conjunto de

defeitos, chamado de domínio de defeitos, mas sob certas condições, caracterizando um modelo de defeitos (Chow, 1978; Fujiwara et al., 1991; Luo et al., 1995). Além disso, tais métodos satisfazem certos critérios, como a cobertura de todos os estados e transições da especificação. Condições de teste, que são similares ao conceito de hipótese de teste (Gaudel, 1995), podem ser encontradas em modelos de defeitos no teste a partir de MEF. Tais métodos geralmente restringem o número de estados no sistema sob test (do inglês, *System Under Test (SUT)*). Casos de teste gerados para uma especificação com n estados são chamados de testes n-completos (Petrenko e Yevtushenko, 2005; Dorofeeva et al., 2005; Simao e Petrenko, 2010b; Hierons et al., 2009). Mesmo sendo um tópico bem estabelecido há décadas (Chow, 1978; Fujiwara et al., 1991; Bochmann e Petrenko, 1994), muitos avanços recentes no teste a partir de MEFs têm sido publicados nessa linha (Petrenko e Yevtushenko, 2002; Dorofeeva et al., 2005; Petrenko e Yevtushenko, 2005; Hierons e Ural, 2006; Ural e Zhang, 2006; Simao e Petrenko, 2010a; Hierons e Ural, 2010; Petrenko e Yevtushenko, 2011; Hwang et al., 2012; Kushik et al., 2014; Petrenko e Yevtushenko, 2014; Petrenko e Simao, 2015).

Figura 1 – Modelos utilizados em TBM

(a) MEF (b) IOTS



Embora classes importantes de sistemas possam ser modeladas e especificadas apropriadamente por MEFs, Sistemas de Transição com Entrada/Saída (do inglês, *Input/Output Transition Systems* - IOTS) (Tretmans, 2008) tem atraído interesse da comunidade de pesquisadores. Esse interesse ocorre porque os IOTSs são modelos mais expressivos do que as MEFs. IOTSs não impõem nenhuma restrição em pares de entrada/saída como ocorre em MEFs, e um IOTS pode alcançar um estado na qual nenhuma ação de saída é produzida (Petrenko et al., 2003; Petrenko e Yevtushenko, 2002), como visto na Figura 1b (adaptada de (van der Bijl e Peureux, 2005)). IOTSs possuem a habilidade de aceitar entradas em qualquer estado. Isso não é adequado para modelar alguns tipos de sistemas, como sistemas interativos, pois esse tipo de sistema recusa a entrada em algumas situações. Entretanto, a habilidade para entrada pode ser utilizada para

modelar comportamento de processos, tais como implementações, especificações e casos de teste, além de servir como um modelo semântico para várias linguagens formais (Tretmans, 2008; Gaudel, 2010b).

1.2 Definição do problema e justificativa para a pesquisa

IOTS é um modelo que vem sendo utilizado para especificação formal de software porque descreve o comportamento baseado em uma álgebra de processos e fornece um formalismo elegante que se concentra na comunicação entre as entidades. Além disso, a presença de uma especificação em MEF ou IOTS simplifica a geração de testes, uma vez que a maioria dos problemas relevantes são decidíveis. Trabalhos em teste a partir de tais especificações tem focado em teste a partir de IOTSs (Hierons et al., 2009).

TBM a partir de IOTSs foi proposto por Tretmans (1996), que estabeleceu a teoria de teste de conformidade de entrada/saída (do inglês *Input/Output Conformance (ioco)*). Essa teoria verifica com precisão se uma implementação está em conformidade com uma determinada especificação. O autor também propôs um dos algoritmos mais utilizados para conduzir o processo de seleção de testes a partir de IOTSs (Hierons et al., 2009; Tretmans, 2008; van der Bijl et al., 2004; Weiglhofer e Aichernig, 2010; Weiglhofer e Wotawa, 2009; Noroozi et al., 2011). No entanto, esse processo, que é mostrado em (Tretmans, 2008; van der Bijl e Peureux, 2005), produz conjuntos de teste completos de forma não-determinística, ou seja, aleatória (Hessel et al., 2008; Jard e Jéron, 2005; Jeannet et al., 2007). Isso significa que a completude é um resultado mais teórico do que prático. Em relação à comunicação, considera-se que a interação entre o testador e o caso de teste é síncrona na teoria *ioco*. Porém, na prática muitas interações são baseadas em comunicação assíncrona ou na troca de mensagens através de *buffers* e podem ser modeladas como filas, como em sistemas *Web* e sistemas que se comunicam através de redes. Isso significa que pode ocorrer um atraso na comunicação, levando à observação de uma sequência de ações diferente daquela que o sistema produziu. Assim, é importante levar esse fato em consideração na realização dos testes.

A maioria dos métodos de geração de casos de teste encontrados relata extensões da teoria de teste estabelecida por Tretmans (2008), que estabelece um framework para geração automatizada de casos de teste a partir de IOTSs. Entre os métodos e ferramentas existentes, vários contextos de aplicação são definidos, como o tipo de sistema (sistemas de tempo-real, sistemas de protocolo de comunicação e aplicações *Web*) e o tipo de teste realizado (teste síncrono, teste assíncrono com a utilização de canais de comunicação ou teste distribuído).

Apesar do número crescente de trabalhos sobre geração de testes a partir de IOTSs utilizando a teoria *ioco*, não há mecanismos sistemáticos para a geração de conjuntos de teste completos a partir de IOTSs com garantia de cobertura de defeitos, ao contrário do teste a partir de MEFs que possui uma teoria sólida e bem estabelecida (Chow, 1978; Simao, 2007; Hierons et

al., 2009). IOTSs são modelos mais expressivos do que as MEFs, especialmente por lidarem com não-determinismo e por terem uma noção de conformidade mais rica do que MEFs. Os trabalhos clássicos em IOTSs limitam-se a indicar que o conjunto de teste deve ser gerado de forma não-determinística (Tretmans, 2008; Jard e Jéron, 2005), não havendo garantia de que o conjunto de teste gerado atende a determinado critério de teste e não possuindo a qualidade requerida para revelar defeitos. Hipóteses de teste são utilizadas de um modo mínimo, considerando apenas que o sistema pode ser modelado com o mesmo formalismo que a especificação. Consequentemente, a completude de um conjunto de testes para IOTSs é garantida apenas na teoria através da reexecução dos testes um número não limitado de vezes.

Por outro lado, os métodos de geração de testes a partir de MEFs fornecem garantia de cobertura completa de defeitos para um determinado domínio de defeitos, um conjunto de teste de tamanho reduzido e gerado em tempo finito e focam no aprimoramento da aplicabilidade dos métodos (Simao e Petrenko, 2010b; Dorofeeva et al., 2010; Endo e Simao, 2013). Desse modo, adaptar e estender conceitos e características já utilizados no teste a partir de MEFs para o teste a partir de IOTSs parece ser promissor, como já mostrado por pesquisas recentes (Hierons, 2012a, 2013; Huo e Petrenko, 2009; Simao e Petrenko, 2014). Porém, ao reformular os conceitos do teste a partir de MEFs para o contexto de IOTSs, alguns problemas emergem: IOTSs são mais genéricos, não sendo requerida a alternância entre entradas e saídas, sendo que o testador deve saber quando aplicar as entradas e quando observar saídas; a comunicação entre o testador e a implementação pode ser assíncrona, não sendo possível garantir que a sequência observada foi aquela produzida pelo sistema sob test (do inglês, *System Under Test* - SUT); um sistema pode entrar em quietude, e então não há como proceder. Além disso, Hierons (2013) demonstrou que é indecidível definir se um elemento do domínio de defeitos está em conformidade com sua especificação no teste assíncrono a partir de IOTSs.

Portanto, a questão de seleção de testes é uma das importantes questões abertas no teste baseado em IOTSs (Tretmans, 2008; Hierons et al., 2009). Relacionada à questão de seleção de testes, está a questão de como a completude, cobertura ou qualidade de um conjunto de testes gerados automaticamente podem ser expressos, medidos e controlados (Tretmans, 2008).

1.3 Objetivos do Trabalho

Apesar do teste a partir de IOTSs ser um tópico em crescente investigação, mais contribuições podem ser alcançadas para torná-lo um processo sistemático, promovendo um alto nível de confiabilidade. O teste a partir de MEFs auxilia nessa investigação, já que os métodos de geração de testes a partir de MEFs são determinísticos e empregam modelos de defeitos que garantem a completude dos conjuntos de teste gerados.

Neste contexto, esta tese investiga a seguinte questão: *É possível aplicar domínios de defeitos em métodos determinísticos de geração de casos de teste a partir de IOTSs?* Com base

nisso, os seguintes objetivos específicos nortearam a execução dos passos da pesquisa realizada:

- *Condições de suficiência:* definição de condições mínimas que os conjuntos de teste devem conter a fim de garantir a satisfação de critérios de teste.
- *Método para geração de casos de teste a partir de IOTSs:* definição de um algoritmo determinístico para geração de casos de teste considerando um modelo de defeitos.
- *Construção de ferramentas que auxiliem a aplicação do método proposto:* implementação do método proposto auxiliando na automatização do processo de teste.
- *Estudos empíricos:* realização de estudos experimentais a partir de IOTSs gerados aleatoriamente e de estudos de caso reais que mostram a viabilidade e aplicabilidade do método proposto.

1.4 Sumário dos Resultados Obtidos

As contribuições deste trabalho são descritas a seguir:

- Condições de suficiência foram definidas baseadas na cobertura da especificação de modo que os métodos que satisfazem tais condições garantem a construção de conjuntos de teste completos com relação ao domínio de defeitos definido. Tais condições são apresentadas na Seção 4.2.1.
- Um método para geração de conjuntos de teste completos a partir de Mealy IOTSs foi proposto com base no método W para MEFs. Tal método satisfaz as condições de suficiência propostas empregando um modelo de defeitos e gera conjuntos de teste de forma determinística e repetível. O método proposto encontra-se no Capítulo 4.
- Estudos empíricos foram realizados com o método proposto e apresentados no Capítulo 5: resultados experimentais de conjuntos de teste de Mealy IOTSs gerados aleatoriamente forneceram evidências do custo de geração dos conjuntos de teste; um estudo de caso avaliou a eficácia do método proposto em comparação com o método clássico de Tretmans (2008); e por fim uma comparação entre os IOTSs do estudo de caso com os Mealy IOTSs gerados aleatoriamente mostrou que os conjuntos de teste obtidos pelo método W são semelhantes em relação ao número de casos de teste e ao tamanho dos conjuntos.

1.5 Organização da Tese

O restante da tese está organizado da forma a seguir:

- **Capítulo 2:** Este capítulo apresenta uma visão geral da fundamentação teórica que apoia a investigação apresentada nesta tese. Primeiramente, conceitos fundamentais de teste de software são revisados. O TBM bem como seu processo e técnicas de modelagem são discutidos. O capítulo também apresenta uma visão geral do teste a partir de MEFs, que auxiliaram no delineamento desta investigação. Finalmente, a base teórica e a notação empregada no teste a partir de IOTSs são detalhadas nesse capítulo.

- **Capítulo 3:** Este capítulo resume os resultados do mapeamento sistemático de literatura sobre métodos de geração de casos de teste para IOTSs em geral. Os estudos foram categorizados de acordo com a taxonomia para abordagens TBM (Utting et al., 2012). Este capítulo é baseado no artigo de Paiva e Simao (2015), e foi atualizado para ser incluído na tese.

- **Capítulo 4:** Este capítulo descreve um método para geração de casos de teste a partir de Mealy IOTSs, denominado de método W_{IOTS}. São apresentadas condições de suficiência na qual o método deve satisfazer para garantir a cobertura de todos os defeitos do domínio de defeitos. Um algoritmo determinístico é proposto com base em um algoritmo de geração de teste para MEFs. Uma especificação real foi utilizada para realizar uma comparação entre o conjunto de teste gerado pelo método proposto e conjuntos gerados pelo método clássico desta área (Tretmans, 2008). Os resultados indicam que o método proposto garante a cobertura completa da especificação com um conjunto de teste bem menor que o método clássico. Este capítulo é baseado no artigo de Paiva e Simao (2016).

- **Capítulo 5:** Este capítulo apresenta estudos empíricos realizados para validação do método proposto no capítulo anterior. Resultados experimentais de conjuntos de teste para IOTSs gerados aleatoriamente avaliam o custo de geração do método proposto. Também é apresentado um estudo de caso mostrando a viabilidade de aplicação do método em especificações reais. Os modelos reais foram utilizados para uma comparação entre o método proposto e o método clássico da área (Tretmans, 2008), e os resultados indicam que o método proposto é mais eficiente em detectar defeitos em transições mais difíceis de serem alcançadas na implementação. Uma comparação dos resultados do estudo de caso com resultados de IOTSs gerados aleatoriamente evidencia a semelhança dos resultados obtidos.

- **Capítulo 6:** Este capítulo conclui a tese destacando as principais contribuições e sumarizando as limitações e direções para trabalhos futuros.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

Teste de Software é uma atividade fundamental para a garantia da qualidade de software por fornecer diversos critérios propostos por diferentes técnicas. Abordagens que empregam modelos formais, conhecidas como Teste Baseado em Modelos (TBM), visam automatizar o processo de teste a fim de torná-lo sistemático e formal. Este capítulo apresenta uma visão geral dos conceitos de Teste de Software, especialmente o Teste Baseado em Modelos.

Este capítulo está organizado da seguinte forma: A Seção 2.1 revisa os conceitos principais de Teste de Software. A Seção 2.2 apresenta uma visão geral do Teste Baseado em Modelos. A Seção 2.4 detalha o teste baseado em Máquinas de Estados Finitas (MEFs). Por fim, a Seção 2.5 fornece uma visão geral do teste baseado em IOTSs.

2.1 Fundamentos de Teste de Software

Apesar dos procedimentos e técnicas adotados no processo de desenvolvimento de software, o produto obtido ainda pode conter erros. Atividades complementares, denominadas de Garantia da Qualidade de Software, têm a finalidade de garantir que tanto o processo de desenvolvimento quanto o produto atingem os níveis de qualidade especificados. Entre essas atividades, Validação, Verificação e Teste (VV&T) visam minimizar a ocorrência de erros e riscos associados. A Verificação permite identificar se o desenvolvimento de software está sendo realizado da maneira correta. Já a Validação avalia se o software produzido está em conformidade com os requisitos do cliente. Teste de software é um elemento crítico da garantia de qualidade de software e aumenta a confiança de que o software apresenta as funções especificadas (Pressman, 2003; Myers et al., 2004), estando entre os processos fundamentais do ciclo de vida de sistemas do padrão ISO/IEC 12.207 (ISO, 2008). O objetivo do processo de teste é executar o sistema para que defeitos sejam encontrados e mostrar que está de acordo com sua especificação (Pressman, 2003; Myers et al., 2004). Teste de software é uma das atividades mais utilizadas devido ao fato de fornecer evidências de confiabilidade do software, juntamente com outras atividades

complementares, como revisões e técnicas formais de especificação e verificação (Maldonado, 1991).

O padrão IEEE 610.12 (IEEE, 1990) diferencia os seguintes termos, no contexto de teste de software: **defeito** (*fault*) – processo, passo ou definição de dados incorretos (comando incorreto); **engano** (*mistake*) – resultado incorreto produzido por ação humana (ação tomada pelo programador); **erro** (*error*) – qualquer estado inconsistente ou inesperado do programa; e **falha** (*failure*) – resultado incorreto produzido pela execução diferente do esperado. O padrão define também o conceito de **caso de teste**, como sendo o conjunto de entradas de teste, condições de execução e saídas esperadas desenvolvido para uma finalidade particular, como por exemplo exercitar um determinado caminho do programa ou verificar determinado requisito. Um caso de teste, ou sequência de teste, pode ser ainda uma sequência de ações de entradas e as saídas esperadas. Um **conjunto de teste** é composto por casos de teste.

A atividade de teste combina várias etapas com uma série de métodos de projeto de casos de teste que ajudam a garantir que erros sejam efetivamente detectados. Para a realização dos testes, técnicas devem ser selecionadas para estabelecer um plano de teste, casos de teste, critérios de teste e a própria realização dos procedimentos de teste, de modo a verificar se os requisitos especificados para o software foram corretamente implementados (Rocha et al., 2001). Nesse sentido, o objetivo do teste é executar casos de teste na implementação sob teste que sejam bons o suficiente para revelar erros (Pressman, 2003; Myers et al., 2004).

Conforme Mathur (2008), existe uma fase de teste compatível com as necessidades e os recursos disponíveis para cada artefato produzido durante o desenvolvimento de software. Deste modo, o teste de software pode ser classificado em três fases complementares que visam testar desde as pequenas unidades do sistema até as alterações provenientes de manutenção do software. Tais fases são descritas a seguir:

- **Teste de unidade:** aplica-se à menor unidade do projeto de software, podendo ser um componente, módulo ou função do software. No paradigma Orientado a Objetos, a unidade pode ser um método (Vincenzi, 2004) ou uma classe (Binder, 1999). Nessa fase são verificadas a lógica interna e estruturas de dados dentro do limite da unidade.

- **Teste de integração:** ocorre juntamente com a fase de integração do software, de modo que testes possam revelar defeitos nas interfaces. Duas estratégias de teste que refletem diferentes abordagem para integração de sistemas são *top-down* e *bottom-up* (Somerville, 2007). Na estratégia *top-down* os componentes de alto nível de um sistema são integrados e testados antes que seus projetos e implementações tenham sido concluídos. Já na integração *bottom-up*, integram-se e testam-se componentes de nível inferior antes que os componentes de nível superior sejam desenvolvidos. *Drivers* e *stubs* precisam ser utilizados, já que o teste de integração é realizado em partes do software que ainda não estão completas (Ammann e Offutt, 2008). O *driver* é um módulo que emula a chamada

para a unidade testada, e o *stub* é um módulo que simula o comportamento da unidade chamada.

- **Teste de sistema:** tem como finalidade exercitar o sistema como um todo por meio de vários tipos de teste, tais como teste de recuperação, de segurança, de estresse e de desempenho (Pressman, 2003). No *teste de recuperação*, falhas no sistema são forçadas de várias formas para verificar se a recuperação é executada corretamente. O *teste de segurança* verifica se os mecanismos de segurança funcionam corretamente quando o testador tenta invadir o sistema. O *teste de estresse* verifica o comportamento do sistema quando há alta demanda de recursos em quantidade, frequência ou volume. O *teste de desempenho* verifica o desempenho do software em tempo de execução.

2.1.1 Técnicas de Teste

Durante o teste, é importante saber se o teste realizado realmente é de boa qualidade, de modo que a probabilidade de encontrar defeitos seja alta. Para isso, critérios de teste são utilizados para definir quais propriedades ou requisitos devem ser testados para avaliar a qualidade dos testes (Zhu et al., 1997). Dados um programa P que está sendo testado, um conjunto de teste T que contém um subconjunto das entradas de P, e um critério de teste C, denomina-se que o critério é C-adequado para o teste de P somente se o conjunto de casos de teste T satisfaz os requisitos de teste estabelecidos pelo critério C.

Critérios de teste são utilizados para que o testador defina se a fase de teste está completa ou para guiar a construção do conjunto de teste (Frankl e Weyuker, 2000). Tais critérios podem ainda auxiliar o testador no momento da seleção dos casos de teste durante o processo de geração, sendo então denominados de *critérios de seleção de teste*. Dado um conjunto de critérios de teste, identificar quais entradas de teste são necessárias para satisfazer esses critérios é um dos meios pelos quais é abordado o problema de geração automática. A definição de critérios é útil para apoiar a geração automática de casos de teste, minimizando erros humanos e facilitando testes de regressão (Ammann e Offutt, 2008).

Diferentes técnicas de teste foram estabelecidas para classificar os critérios de teste, de acordo com a origem das informações para derivar os casos de teste (Maldonado, 1991). As principais técnicas são: *funcional*, *estrutural*, *baseada em defeitos* e *baseada em modelos*.

Os critérios e requisitos da técnica de teste *funcional* (também conhecido como teste caixa-preta) são formados a partir da especificação do software, visando testar as entradas e saídas do programa, sem considerar a implementação (Delamaro et al., 2007). Os casos de teste para o teste funcional são gerados a partir da especificação, considerando-se apenas as funcionalidades do programa. Exemplos de critérios dessa técnica são o *particionamento em classes de equivalência* (que divide o domínio de entrada do programa em classes de equivalência, dos quais são originados os casos de teste) e *análise do valor limite* (utilizado juntamente com o

anterior, com ênfase nos limites relacionados às condições de entrada) (Myers et al., 2004).

Na técnica *estrutural*, ou caixa-branca, os critérios são organizados a partir das características internas da implementação em teste. O objetivo é caracterizar um conjunto de elementos básicos do software que deve ser executado (Myers et al., 2004). A maioria dos critérios utilizados por essa técnica utiliza o Grafo de Fluxo de Controle (GFC) que estabelece uma correspondência entre nós e blocos e indicando possíveis fluxos de controle entre os blocos por meio de arcos (Maldonado et al., 1998). Exemplos de critérios baseados no fluxo de execução do GFC são *todos-nós* (executa pelo menos uma vez cada vértice do GFC), *todos-arcos* (executa pelo menos uma vez cada aresta do GFC) e *todos-caminhos* (executa todos os caminhos possíveis do GFC pelo menos uma vez) (Delamaro et al., 2007).

Já na técnica *baseada em defeitos*, os critérios e requisitos procedem do conhecimento sobre defeitos típicos no processo de desenvolvimento (Maldonado, 1991). O teste baseado em defeitos emprega o conhecimento sobre defeitos mais comuns no processo de desenvolvimento de software para a derivação dos requisitos de teste. Dois exemplos dessa técnica são a Semeadura de Defeitos (*Error Seeding*), que insere determinada quantia de erros no programa a fim de identificar os erros naturais (Ramamoorthy e Bastani, 1982); e Análise de Mutantes (*Mutation Analysis)* avalia a adequação de um conjunto de casos de teste em revelar defeitos específicos (DeMillo, 1978).

Finalmente, os critérios e requisitos da técnica *baseada em modelos* são originados a partir de modelos comportamentais do sistema (Tretmans, 2008). Essa técnica será detalhada na seção a seguir.

2.2 Teste Baseado em Modelos

Abordagens que tornam o teste de software automatizado pela geração de casos de teste baseado em um modelo comportamental do sistema, chamado modelo de teste, são conhecidas como TBM (Sinha e Smidts, 2006). A abordagem de TBM envolve o desenvolvimento e o uso de modelos para gerar testes. Apesar de alguns autores afirmarem que todo teste é baseado em modelos (Binder, 1999), uma vez que modelos mentais explícitos são utilizados para guiar os testes, a ideia em TBM é utilizar modelos explícitos (Pretschner e Philipps, 2005). O modelo de teste, derivado de alguma especificação de requisitos ou fonte de conhecimento sobre o sistema, pode ser construído manualmente, de modo que o comportamento esperado de uma implementação, denominada de sistema sob teste (do inglês, *System Under Test* - SUT), seja codificado.

Com o modelo construído, o TBM tem grande potencial para automatização por meio do uso de ferramentas que tornam o processo de geração de casos de teste mais rápido e menos propenso a erros humanos, uma vez que tarefas rotineiras são automatizadas (Utting e Legeard, 2007). Outra vantagem é que a criação do modelo amplia o entendimento do sistema e também

habilita a geração semiautomática de cenários de teste antes mesmo que o desenvolvimento de software tenha sido concluído, possibilitando que defeitos sejam encontrados nas fases iniciais do processo de desenvolvimento (Barnett et al., 2003). A obtenção de resposta rápida a mudanças durante possíveis evoluções do software é outra vantagem, sendo necessário apenas alterar o modelo e gerar os testes novamente. Como o modelo deve ter as entradas e saídas esperadas, não é necessário preocupar-se com o problema do oráculo, isto é, com o problema de prover um veredicto para os testes.

TBM é considerada uma abordagem formal, pois emprega formalismos para a representação do software em alguma linguagem formal, um algoritmo bem definido para a geração dos casos de teste e emprega uma prova de corretude de que os testes gerados são completos, isto é, detectam todos os defeitos e nenhuma detecção falsa é realizada (Tretmans, 2008). Além disso, a presença de modelos formais pode levar a um teste mais eficiente e efetivo (Hierons et al., 2009). Um modelo é formal se possui um significado preciso e não ambíguo, representando o comportamento de uma forma compreensível e manipulável por ferramentas. Como existe a necessidade de validar o modelo, esse deve ser mais simples que o SUT, ou ainda mais fácil de verificar, manter e modificar (Utting e Legeard, 2007). TBM utiliza a especificação (dada por um modelo de comportamento) como ponto de partida, e assume-se que esse modelo é correto e válido. TBM é também uma abordagem ativa, pois o testador controla e observa a implementação sob teste (do inglês, *Implementation Under Test* - IUT). Além disso, TBM não considera a forma como foi desenvolvida a implementação sob teste, sendo, então, teste caixa-preta. TBM é também um teste de funcionalidade, pois verifica se o sistema fez corretamente o que deveria fazer após um estímulo (Tretmans, 2008).

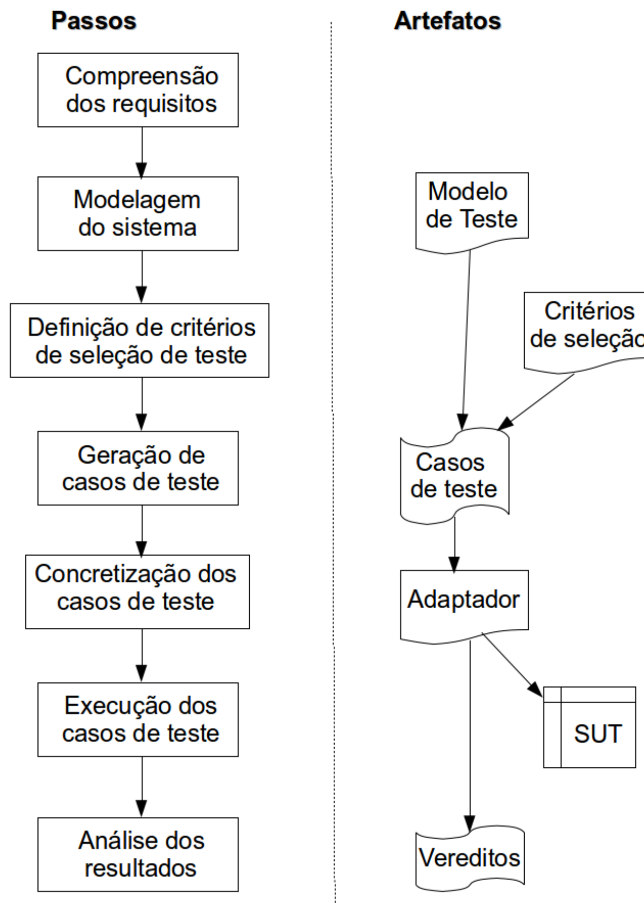
O TBM pode ser aplicado em qualquer fase de teste (unidade, integração e sistema) (Utting et al., 2012). Para ser aplicado durante as fases de teste de unidade e de integração, artefatos de *design* e arquitetura podem auxiliar como fontes de informação para a elaboração de modelos.

Os termos *técnica de modelagem* e *modelo de teste* foram diferenciados neste trabalho. A técnica de modelagem está relacionada à notação empregada para expressar um modelo de teste de forma bem definida. Já o modelo de teste é o artefato gerado pela modelagem durante o processo de TBM.

O TBM apresenta um conjunto de passos para serem realizados durante sua aplicação. Com base na literatura encontrada sobre TBM, um conjunto de passos para composição de um processo genérico de TBM foi abstraído de (El-Far e Whittaker, 2001; Pretschner e Philipps, 2005; Utting e Legeard, 2007; Bouquet et al., 2007). Uma visão geral dos passos e artefatos do processo de teste é apresentada na Figura 2 (adaptada de (Bouquet et al., 2007)). Os passos para o processo genérico de TBM são descritos a seguir.

1. Compreensão dos requisitos: Os testadores podem utilizar os documentos disponíveis bem como aplicar técnicas exploratórias para obter as informações necessárias para a

Figura 2 – Processo de TBM



construção de modelos adequados (Kaner et al., 1999).

2. Modelagem do sistema: Durante a criação do modelo de teste, os modelos gerados devem ser pequenos e mais simples do que o SUT e estarem em um nível mais alto de abstração para facilitar a sua construção. Porém, deve haver detalhes suficientes que descrevam as características a serem testadas.

3. Definição de critérios de seleção de teste. Entre as inúmeras possibilidades de teste, é necessário definir critérios de seleção de teste de modo que um número finito de casos de teste seja gerado. Os critérios selecionados podem ser implementados por ferramentas de apoio.

4. Geração de casos de teste abstratos: Nessa fase, são gerados os casos de teste a partir do modelo criado e de acordo com os critérios estabelecidos. O resultado é um conjunto de sequência de operações executáveis no modelo.

5. Concretização dos casos de teste abstratos: Os casos de teste abstratos são transformados em casos de teste concretos e executáveis no SUT, tendo como resultado um conjunto de *scripts* de teste. Esse passo é importante em TBM para garantir que todo o processo será automatizado (Utting e Legeard, 2007).

6. Execução dos casos de teste: é basicamente realizada a execução dos casos de

teste transformados em *scripts* no SUT. Para apoiar esse passo e alcançar um alto nível de automatização, é essencial que existam ferramentas de apoio. As saídas obtidas na execução são confrontadas com as saídas esperadas no modelo. O resultado é um relatório de execução dos testes.

7. Análise do resultado do teste: os resultados obtidos na execução dos testes são analisados e ações corretivas são definidas no sistema ou no modelo. Devido ao fato do modelo de teste possuir a especificação de entradas e saídas, é possível fornecer veredictos automáticos (auxiliando no problema do oráculo). Os veredictos possíveis são **passou**, **falhou**, e **inconclusivo** (Tretmans, 2008; Jard e Jéron, 2005). *Passou* indica que o teste foi executado com sucesso e o objetivo do caso de teste foi alcançado. *Falhou* define que o resultado obtido não está em conformidade com o objetivo do caso de teste. *Inconclusivo* indica que não foi encontrada evidência de não conformidade e o objetivo de teste não foi alcançado.

A adoção de TBM durante o processo de desenvolvimento pode acarretar em uma série de benefícios como a geração automática de casos de teste, descoberta de defeitos com maior eficiência que o teste tradicional, redução do tempo e custo do teste, melhoria da qualidade dos testes já que estes podem ser reexecutados de forma simples e rastreabilidade (Utting e Legeard, 2007; Blackburn et al., 2004). No entanto, não é possível garantir que todas as diferenças entre o modelo e o SUT serão encontradas (Utting e Legeard, 2007).

Apesar dos vários benefícios adicionados com o emprego de TBM, possíveis desvantagens são (El-Far e Whittaker, 2001; Utting e Legeard, 2007):

- o uso inapropriado em casos em que é difícil modelar o SUT;
- um caso de teste pode falhar por defeitos no SUT, no modelo ou na ferramenta, e pode ser difícil identificar onde se encontra o defeito;
- o testador precisa ter habilidades com modelagem e manuseio da técnica de modelagem de teste;
- pode ocorrer o problema de explosão de estados durante a modelagem de certos casos, levando a modelos complexos de difícil manutenção.

Mesmo que tais desvantagens dificultem ou impeçam a adoção de TBM, soluções práticas existem para a resolução desses problemas. A adoção de ferramentas de apoio, o estabelecimento de um processo bem definido de TBM além de um treinamento adequado, por exemplo, tendem a fornecer soluções para a maior parte dos problemas apresentados.

2.2.1 Técnicas de Modelagem

Uma técnica de modelagem ideal deve ser de fácil entendimento para o testador, deve ser capaz de descrever tanto problemas simples quanto problemas complexos, e ainda possuir uma

forma fácil de ser entendida por uma ferramenta de geração de teste (Dalal et al., 1999). Porém, não existem técnicas de modelagem adequadas para todos os propósitos e intenções (El-Far e Whittaker, 2001). Portanto, é necessário decidir qual técnica é mais adequada para cada situação.

Ao se escolher uma técnica de modelagem, além das características do modelo em si, outros fatores devem ser considerados, como fatores gerenciais, que incluem: a habilidade dos testadores e outras pessoas envolvidas, e a disponibilidade e custo de ferramentas de apoio para determinada técnica. Também deve-se considerar que a técnica escolhida seja a mais adequada para o SUT (Utting e Legeard, 2007).

Entre as técnicas formais mais utilizadas em TBM destacam-se as notações baseadas em transições (Hierons et al., 2009; Utting et al., 2012), já que as ferramentas de automatização de testes geralmente traduzem a especificação para modelos baseados em transições, como Máquinas de Estado Finito (MEFs) ou *Input/Output Transition Systems* (IOTSs) (Bochmann e Petrenko, 1994; Hierons et al., 2009; Tretmans, 2008; Petrenko e Yevtushenko, 2002; Hierons, 2013; Utting et al., 2012). Geralmente, estes modelos são representados por gráficos com nós e arcos, que representam estados e transições/operações do sistema, respectivamente. Por serem modelos amplamente utilizados no TBM, (Hierons et al., 2009; Petrenko e Yevtushenko, 2002; Hierons, 2013), os conceitos do teste baseado em MEFs e do teste baseado em IOTSs serão detalhados neste capítulo.

2.3 Avaliação em Teste de Software

O método primário utilizado pela indústria para avaliar o desenvolvimento de software é o teste de software (Ammann e Offutt, 2008). Devido ao grande número de critérios e abordagens de teste existentes, é importante saber qual deles deve ser empregado e como utiliza-lo para que o melhor resultado seja alcançado. Escolher a abordagem mais adequada para testar determinada característica de um sistema é ainda uma questão em aberto (Bertolino, 2004). Para auxiliar nessa escolha, estudos teóricos e experimentais tem sido realizados para avaliar os critérios de teste.

2.3.1 Estudos Teóricos

A comunidade de teste vem realizando esforços no sentido de estudar a fundamentação teórica de teste (Goodenough e Gerhart, 1975; Hierons et al., 2009; Gaudel, 2010b). Com esses estudos, propriedades que fornecem o embasamento teórico para a adoção de determinado critério de teste são apresentadas. Goodenough e Gerhart (1975) formalizam a noção de que um critério de teste é válido se para todo programa com defeito, o critério de teste é capaz de produzir um conjunto de teste que mostre que o programa possui defeitos. Também formalizam a noção de que um critério de teste é confiável se todo conjunto de teste que pode ser produzido utilizando tal critério conduza ao mesmo veredicto. Hierons et al. (2009) afirmam que o equilíbrio entre o teste

exaustivo e a prova completa do sistema pode ser alcançado pela definição de suposições sobre o artefato em teste que apoiem a seleção de conjuntos de teste menores e eventualmente finitos. Tais suposições são referenciadas como *hipóteses de seleção de teste*, e são formuladas com o conhecimento do programa, critério de cobertura e considerações de custo (Gaudel, 2010b). Gaudel (2010b) identificou dois tipos de hipóteses: uniformes e regulares. Uma hipótese uniforme é aquela que está relacionada à uniformidade do comportamento do programa para determinados intervalos de dados. Uma hipótese regular é relacionada à regularidade do comportamento do programa à medida que o tamanho dos dados aumenta.

Comparações teóricas realizadas entre os critérios de teste apoiam uma relação de inclusão e o estudo da complexidade dos critérios (Rapps e Weyuker, 1985). Na relação de inclusão, há uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Define-se a complexidade como o número máximo de casos de teste requeridos por um critério de teste, no pior caso. No caso dos critérios baseados em fluxo de dados, que tem complexidade exponencial, estudos experimentais têm sido conduzidos para determinar o custo de aplicação do ponto de vista prático (Maldonado et al., 2004).

Alguns estudos abordaram a questão da eficácia de critérios de teste do ponto de vista teórico, e definiram outras relações que captam a capacidade dos critérios em relevar defeitos (Frankl e Weyuker, 1993; Zhu, 1996). O fato de um critério incluir outro, não garante que o primeiro é melhor em revelar defeitos que o segundo, conforme os trabalhos de Frankl e Weyuker (1993). Porém, Zhu (1996) contesta esses resultados, justificando que esse fato é válido apenas em um cenário em que os casos de teste são construídos com base no critério utilizado. Se for considerado um cenário em que os casos de teste são construídos sem o conhecimento do critério, mostrou-se que a relação de inclusão implica também em maior capacidade de descobrir defeitos (Zhu, 1996).

2.3.2 Estudos Experimentais

O tipo de conhecimento utilizado em teste, e de forma mais ampla, em Engenharia de Software, pode ser considerado relativamente de baixa maturidade (Juristo et al., 2004), já que disciplinas de engenharia são descritas pelo emprego de conhecimento maduro, para que os resultados previstos sejam alcançados. Nesse sentido, a realização de estudos experimentais e a pesquisa em como realizá-los em Engenharia de Software tem sido uma preocupação em crescimento nos últimos anos (Kitchenham, 2004; Briand, 2007).

A execução e análise de estudos experimentais trazem evidências sobre uma determinada hipótese. Os dados coletados durante um estudo podem ser quantitativos ou qualitativos (Wohlin et al., 2000). Dados quantitativos envolvem números e classes e são analisados com o uso de estatística. Já dados qualitativos envolvem descrições, figuras ou diagramas e são analisados com o uso de categorização e ordenação. Tal coleta de dados efetuada durante a realização da pesquisa envolve o uso de alguma estratégia de pesquisa. As estratégias de pesquisa são classificadas em

survey, estudo de caso e experimento controlado (Wohlin et al., 2000).

Três aspectos geralmente são avaliados nos estudos experimentais no contexto de teste de software: custo, eficácia e *strength* (dificuldade de satisfação) (Wong et al., 1995; Rocha et al., 2001). O custo reflete o esforço necessário para que o critério seja utilizado; a eficácia refere-se à capacidade que um critério possui de detectar a presença de defeitos; e o *strength* refere-se à probabilidade de satisfazer um critério tendo satisfeito um outro critério (Mathur, 2008). O custo e a eficácia são os aspectos mais presentes em estudos experimentais em teste, principalmente pela necessidade de medir a relação entre ambos (Briand, 2007).

Geralmente, o custo é medido pela quantidade de casos de teste, embora nem sempre essa seja uma medida adequada. Duas dimensões são envolvidas no custo de teste: esforço humano e custo computacional, apesar do primeiro ser mais importante (Briand, 2007). Outras medidas que podem ser consideradas são aquelas relacionadas ao tempo, tais como o tempo de projeto e de execução do conjunto de teste. Também devem ser consideradas a geração e a codificação de toda a infraestrutura necessária para executar os testes.

A eficácia geralmente é medida pelo número de defeitos que o teste encontra (Briand, 2007). Pode ser medida também relacionando o número de defeitos encontrados com o número total de defeitos, e nesse caso, o número total de defeitos é conhecido, já que os defeitos foram manualmente introduzidos no programa. Nos estudos realizados, os defeitos podem ser obtidos de uma base histórica (defeitos reais) ou inseridos propositadamente (defeitos semeados) (Briand, 2007). Defeitos reais são mais complicados de serem obtidos e são encontrados em menor número, limitando a análise estatística de um estudo com tais defeitos, porém, possuindo maior representatividade do mundo real. Defeitos semeados podem ser inseridos manualmente pelo testador, ou automaticamente, por meio da adoção da análise de mutantes como um mecanismo para gerar inúmeras versões defeituosas. O emprego do teste de mutação pode ser uma estratégia interessante, já que o estudo de Andrews et al. (2005) mostra um experimento em que os mutantes são similares a defeitos reais. Outro resultado obtido desse estudo é que defeitos semeados manualmente são mais difíceis de detectar do que defeitos reais e mutantes.

Os estudos experimentais mostram-se mais importantes no contexto de teste de software, uma vez que obter uma prova teórica da eficácia é uma tarefa muito difícil, ou em alguns casos, impossível. Pesquisas relatam o estado atual da experimentação em teste de software (Juristo et al., 2004; Do et al., 2005; González et al., 2014). Uma análise do nível de conhecimento na área de teste de software baseada nos estudos experimentais encontrados na literatura é apresentada por (Juristo et al., 2004). Os estudos foram agrupados conforme a técnica de teste avaliada, e a comparação foi realizada entre critérios da mesma técnica ou de técnicas diferentes. Para cada grupo, os estudos foram brevemente descritos, apresentando os critérios comparados, uma recomendação prática, o nível de maturidade dos resultados e quais conhecimentos precisam de mais avaliação. Esse trabalho conclui que o conhecimento da área de teste é limitado, e que até o momento da realização do estudo, não existia conhecimento formalmente testado, e mais da

metade do conhecimento existente é sempre baseado em impressões ou percepções. González et al. (2014) atualizou tal estudo recentemente e seus resultados apontam que esse panorama não havia mudado, além de evidenciar que os experimentos realizados muitas vezes não são realísticos e não relatam análise estatística.

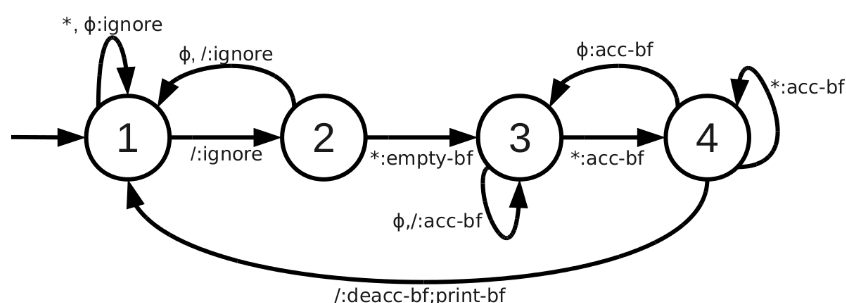
Do et al. (2005) realizaram uma análise de 107 artigos sobre estudos experimentais na área de teste, selecionados nas principais conferências e revistas de Engenharia de Software. Cerca de 34% dos estudos relataram experimentos controlados e menos da metade dos estudos utilizaram vários programas, versões, dados sobre defeitos e compartilhamento limitado de artefatos. Porém, notou-se um crescente interesse por parte dos pesquisadores na realização de experimentos controlados.

2.4 Teste baseado em MEFs

Uma Máquina de Estado Finita (MEF) (Gill, 1962) é um modelo simples e preciso utilizado em diversas áreas como circuitos, análise de programas e protocolos de comunicação (Lee e Yannakakis, 1996). O modelo de MEFs é bem estabelecido e tem sido intensivamente investigado como fundamento para a geração automática de casos de testes (Petrenko e Yevtushenko, 2005; Dorofeeva et al., 2005; Simao et al., 2009b; Hwang et al., 2012; Petrenko e Simao, 2015).

Uma MEF é uma máquina hipotética composta por estados e transições. Cada transição liga um estado ao outro (podendo ligar o mesmo estado). Em resposta a um evento de entrada, a máquina gera um evento de saída e executa uma transição. A cada instante, a máquina pode estar apenas em um dos seus estados, caracterizando uma máquina determinística, caso contrário é uma máquina não-determinística. Tanto o evento de saída quanto o novo estado são determinados unicamente em função do estado atual e do evento de entrada (Petrenko e Yevtushenko, 2005). Um exemplo de MEF é apresentado na Figura 3 (extraída de (Chow, 1978)).

Figura 3 – Exemplo de uma MEF



O teste baseado em MEFs visa a geração de um conjunto de sequências de testes (casos de teste) cujo objetivo é encontrar o máximo de defeitos em uma implementação, verificando se a implementação está de acordo com sua especificação (Cutigi, 2010). Nesse contexto, os defeitos mostrados na implementação podem ser (Chow, 1978): *defeito de transferência* (atinge

um estado incorreto); *defeito de saída* (gera uma saída incorreta); *estados ausentes* (deve-se aumentar os estados para torná-la equivalente à especificação); e *estados extras* (deve-se reduzir os estados para torná-la equivalente à especificação).

Para a geração de um conjunto de sequências de testes a partir de MEFs, são exigidas algumas propriedades das MEFs para que os métodos de geração sejam aplicados, tais como determinismo, minimalidade e completude. Cada método pode requerer diferentes propriedades (Simao, 2007). Os métodos oferecem diferentes resultados, tal como a cobertura de defeitos e o tamanho do conjunto de casos de teste. Existem métodos que garantem cobrir todos os defeitos de um determinado conjunto, chamado domínio de defeitos. Tais domínios são definidos em função do número máximo de estados que a implementação pode ter. Assim, dado um número n de estados da especificação, o domínio de defeitos é o conjunto de todas as MEFs com no máximo n estados. Nesse caso, o número de MEFs é finito, mas não é conhecido. Essa é a *hipótese de teste*, na qual o número de MEFs é calculado a partir de heurísticas (Chow, 1978; Simao, 2007; Hierons e Ural, 2006; Ural et al., 1997).

Existem sequências ou conjuntos de sequências de teste que geralmente são utilizadas para gerar resultados parciais importantes no processo de geração dos casos de teste (Simao, 2007). Algumas sequências básicas que podem ser utilizadas pelos métodos de teste são:

- *state cover*: capaz de conduzir a MEF de seu estado inicial para cada um dos seus estados, incluindo a sequência vazia (representada por ε). Considerando a MEF apresentada na Figura 3, pode-se gerar o conjunto *state cover*: $Q = \{\varepsilon, /, /*, /**\}$.

- *transition cover*: faz com que a MEF exercite todas as suas transições pelo menos uma vez, incluindo a sequência vazia. O conjunto *transition cover* $P = \{\varepsilon, *, /, //, /*, /*/, /**, /**\phi, /**/, /***\}$ cobre todas as transições da MEF representada na Figura 3.

- sequência de distinção: produz saídas diferentes em cada estado da MEF. Se for aplicada a sequência $/*$ em cada estado da MEF da Figura 3, saídas diferentes serão produzidas.

- sequência de sincronização: capaz de conduzir a MEF de um estado qualquer para determinado estado. Geralmente, essa sequência, quando existe, não é única.

- conjunto de caracterização (conjunto W): conjunto formado por sequências de entrada que ao serem aplicadas realizam uma identificação única para cada estado. A sequência $*/$ forma o conjunto de caracterização para a MEF da Figura 3.

- conjuntos de identificação: subconjunto do conjunto de caracterização, é utilizado para identificar um determinado estado. A MEF da Figura 3 possui os conjuntos $W_1 = \{*\}$, $W_2 = \{*\}$, $W_3 = \{*/\}$ e $W_4 = \{*/\}$.

- famílias de separação: um conjunto H_i, para um estado s_i, é uma família de separação se as saídas geradas pelas sequências de entrada desse conjunto tornam possível distinguir o estado s_i de qualquer outro estado (Luo et al., 1995).

Diversos métodos para geração de sequências de teste foram desenvolvidos, automatizando e otimizando esse processo. Entre os métodos existentes, os mais relevantes para a geração de sequências de teste são:

- *Método W* (Chow, 1978): Método que utiliza a sequência *transition cover* para alcançar estados e transições e utiliza o conjunto de caracterização para identificação dos estados. Os casos de teste são resultado da concatenação destes dois conjuntos de sequências. Assume-se que a implementação pode ter o número de estados maior ou igual ao número de estados da especificação.

- *Método Wp* (Fujiwara et al., 1991): Método proposto como uma melhoria ao método W, possibilitando a geração de casos de teste de menor comprimento. Além de utilizar o conjunto de caracterização, utiliza-se os conjuntos de identificação para cada estado.

- *Método HSI* (Luo et al., 1995): O HSI (*Harmonized State Identification*) utiliza famílias de separação para verificar os estados tanto na fase de identificação de estados como na fase de teste de transição. Pode ser aplicado em casos em que a especificação das MEFs são parciais.

- *H* (Dorofeeva et al., 2005): Pode ser visto como uma melhoria do método HSI, onde identificadores de estados apropriados são selecionados em tempo de execução para diminuir o comprimento do conjunto de teste gerado.

- *SPY* (Simao et al., 2009b): Método que tem o objetivo de reduzir o comprimento do conjunto de teste gerado no domínio de defeitos das implementações que têm mais estados do que a especificação. Para isso, é necessário incluir todas as sequências do chamado conjunto de travessia (*traversal set*) que contém todas as sequências com $m-n+1$ símbolos de entrada para obtenção de um conjunto de teste m-completo, onde m é o número de estados da implementação, e n é o número de estados da especificação.

- *P* (Simao e Petrenko, 2010b): Este método foi definido para obter um conjunto n-completo incrementando um conjunto de teste já existente. Entretanto, se o conjunto for vazio, o método pode ser utilizado para gerar conjuntos de teste n-completos da maneira tradicional. Este método escolhe sequências em tempo de execução para distinguir os n estados e verifica as transições não cobertas pelas condições de suficiência. Para cada novo caso de teste gerado, regras podem ser engatilhadas e outras transições podem ser verificadas.

Apesar dos métodos de geração terem como objetivo a verificação da conformidade, eles diferem no custo de execução das sequências de teste, que é determinado pelo tamanho dos conjuntos gerados. Por isso, o conjunto gerado deve apresentar algumas características, viabilizando sua aplicação: ser relativamente pequeno, facilmente executável e ser capaz de identificar todos os defeitos do domínio de defeitos definido (Fujiwara et al., 1991). Estudos recentes compararam os métodos citados acima com a utilização de MEFs geradas aleatoriamente (Simao et al., 2009a; Dorofeeva et al., 2010; Endo e Simao, 2013). Foram comparadas as características dos conjuntos de teste e o tamanho dos conjuntos de teste gerados por cada método e foi mostrado que os métodos mais recentes produzem conjuntos de teste menores e são mais efetivos em revelar defeitos (Endo e Simao, 2013).

Outro tópico que está fortemente relacionado com os métodos de geração é a forma de avaliar a qualidade dos conjuntos gerados. Uma alternativa é utilizar condições de suficiência, que têm o objetivo de garantir a completude dos testes, indicando sob quais condições um conjunto de teste é capaz de revelar todos os defeitos de um dado domínio. Ou seja, se um conjunto de teste satisfaz as condições de suficiência, garante-se que esse conjunto é completo; caso contrário, não se pode afirmar nada a respeito da completude do conjunto. Os principais métodos de geração de sequências de teste, apresentados anteriormente, geram conjuntos de sequências que fazem uso de condições de suficiência implícita (W, Wp e HSI) ou explicitamente (H, SPY, P), de modo que seja garantida a completude do conjunto gerado.

Petrenko et al. (1996) definiram um conjunto de condições de suficiência que permite provar a completude de diversos métodos de geração já existentes, tais como o W, o Wp e o HSI. Com esse estudo, foi possível observar que esses métodos já satisfaziam tais condições de suficiência, porém de forma implícita. No método H são utilizadas as condições de suficiência definidas por Dorofeeva et al. (2005), de forma que as condições são satisfeitas explicitamente. As condições definidas por Petrenko et al. (1996) estão contidas nas condições definidas por (Dorofeeva et al., 2005), e essas generalizam as condições de Petrenko et al. (1996).

Porém, as condições definidas por Dorofeeva et al. (2005) não são aplicáveis a todos os métodos de geração. Nesse sentido, as condições de suficiência definidas por Ural et al. (1997) podem provar a completude de outros métodos de geração, como os métodos de Hierons e Ural (2006); Simao e Petrenko (2008); Ural et al. (1997) e Ural e Zhang (2006).

As condições definidas por Dorofeeva et al. (2005) e Ural et al. (1997) são ortogonais, não sendo possível reduzir umas às outras. Entretanto, Simao e Petrenko (2010b) definiram um conjunto de condições de suficiência que generalizam os dois trabalhos anteriores. Além de serem mais abrangentes em sua aplicação, as condições propostas por Simao e Petrenko (2010b) permitem a geração de casos de teste menores e são menos exigentes que as condições de suficiência propostas anteriormente. Um dos motivos para investigar condições de suficiência é que novas condições podem aprimorar os métodos de geração de teste, reduzindo o tamanho do conjunto sem perder a eficiência na detecção de defeitos, e sendo uma maneira menos complexa

de determinar a completude de um conjunto de casos de teste (Simao e Petrenko, 2010b).

2.5 Teste baseado em IOTS

Sistemas de Transição com Entrada/Saída (do inglês, *Input/Output Transition Systems* - IOTSs) são modelos de especificação mais genéricos do que MEFs especialmente por lidarem com não-determinismo. Esse modelo tem recebido atenção especial da comunidade de teste recentemente (Tretmans, 2008; Huo e Petrenko, 2009; Simao e Petrenko, 2011; Hierons, 2012a, 2013; Noroozi et al., 2013; Simao e Petrenko, 2014).

2.5.1 Definição de IOTS

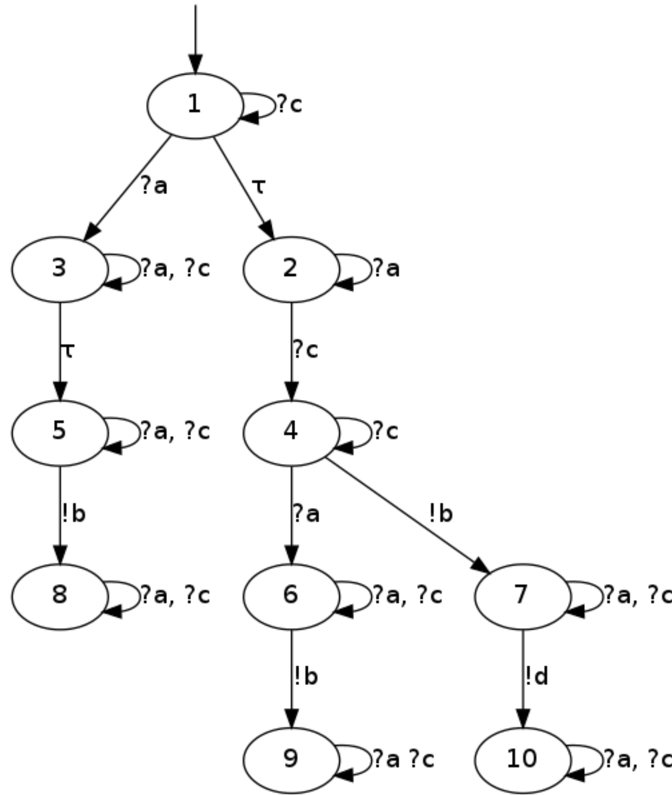
Um IOTS é uma estrutura de estados com transições rotuladas por ações. As transições rotuladas modelam as ações que o sistema pode realizar. O conjunto de ações é subdividido em entradas e saídas, e cada transição modela apenas um tipo de ação. Em IOTSs, sequências de interações são possíveis entre um sistema e seu ambiente, na qual ações de saídas são iniciadas pelo sistema e nunca são recusadas pelo ambiente, e ações de entradas são iniciadas pelo ambiente do sistema e nunca são recusadas pelo sistema. Isso significa que o sistema está sempre preparado para realizar qualquer ação de entrada em qualquer estado alcançável (Tretmans, 2008; Petrenko et al., 2003).

Existe ainda um tipo especial de ação: ações internas (que não pertencem ao conjunto de rótulos e são representadas por τ), que são ações que não podem ser observadas pelo ambiente. Assume-se que os estados também não podem ser observados pelo ambiente. A Figura 4 apresenta um exemplo de IOTS. Os rótulos iniciados por "**?**"representam *ações de entrada*, e aqueles iniciados por "**!**"representam *ações de saída*. Pode-se notar também a presença de ações internas nas transições entre os estados 1–2 e 3–5. Além disso, observa-se em cada estado que todas as ações de entrada estão habilitadas. Uma definição formal de IOTSs bem como suas características são apresentadas a seguir.

Definição 1. Um IOTS é uma 5-tupla $\langle S, I, O, h, s_0 \rangle$ na qual (Simao e Petrenko, 2011):

- S é o conjunto de estados;
- I e O são conjuntos disjuntos de ações de entrada e saída, respectivamente, e a união de I com O resulta no conjunto de rótulos L;
- $h \subseteq S \times (I \cup O \cup \tau) \times S$ é a relação de transição, com o símbolo $\tau \notin (I \cup O)$ denotando ações internas;
- s_0 é o estado inicial.

Figura 4 – Exemplo de um Sistema de Transição com Entrada e Saída (IOTS)



A ocorrência de uma transição rotulada quando o sistema está em um estado q, realiza uma ação μ e é conduzido ao estado q' é descrita como: $q \xrightarrow{\mu} q'$. Uma transição pode ser composta por várias sequências de ações concatenadas, de modo que exista um conjunto de estados de q_0 a q_n, e cada uma dessas sequências faz a transição entre os estados: $q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q'$. É possível que a mesma composição de ações conduza a diferentes estados não-deterministicamente. Assim, podemos escrever: $q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$. Um exemplo de transição que pode ocorrer no IOTS representado na Figura 4 é: $1 \xrightarrow{?a \cdot \tau \cdot !b} 8$. No entanto, se uma composição de ações não conduz a nenhum estado, então escrevemos $q \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$.

O comportamento observável de um IOTS é percebido pela habilidade de realizar sequências de ações observáveis. Isso quer dizer que essa sequência de ações observáveis abstrai as ações internas. Uma sequência desse tipo é chamada de *trace*, pois possui uma sequência de ações observáveis, mas pode conter ações internas que foram abstraídas (Tretmans, 2008; Petrenko e Yevtushenko, 2002). Desse modo, um *trace* cuja transição é $q \xrightarrow{a \cdot b \cdot \tau \cdot c} q'$ pode ser escrito como: $q \xRightarrow{a \cdot b \cdot c} q'$, sendo que $(a, b, c \in I \cup O)$. É possível haver uma transição com uma sequência observável vazia (ε), indicando que podem ocorrer transições internas: $q \xRightarrow{\varepsilon} q' \iff q = q'\ ou\ q \xrightarrow{\tau \cdot \ldots \cdot \tau} q'$. Para o IOTS representado na Figura 4, um possível conjunto de *traces* é: $traces(s) = \{\varepsilon, ?a!b, ?c?a!b, ?c!b!d\}$.

Um operador usual, chamado **after**, denota o conjunto de estados que é alcançável a

partir de um estado t quando um *trace* u é executado: t **after** u (Simao e Petrenko, 2011). Deve-se observar que t **after** ε inclui todos os estados alcançáveis por transições internas bem como os *self-loops* em t. Considerando o IOTS representado na Figura 4, a operação 2 **after** $?c{\cdot}?c{\cdot}!b$ conduz ao estado 7.

Um estado pode atingir o ponto de não produzir nenhuma saída, de modo que o sistema não pode avançar autonomamente até que o ambiente forneça estímulos para o sistema, como podemos observar nos estados 2, 8, 9 e 10 do IOTS da Figura 4. Tal estado é chamado de *quiescente* ou *suspenso* (Tretmans, 2008). A ocorrência de um estado quiescente q é chamado de quietude e denotado por $\delta(q)$. Um estado quiescente é escrito como:

$QTraces(p) =_{def} \{\sigma \in L^* \mid \exists p' \in (p\,\mathbf{after}\,\sigma) : \delta(p')\}$, onde L^* é o conjunto $I \cup O \cup \tau$.

Um sistema em quietude não gera nenhuma saída. Assim, a ausência de saídas pode ser expressa como um evento observável, ou seja, a quietude é considerada como uma ação, denotada por δ. Logo, o conjunto de rótulos que engloba essa nova ação é: L_δ. Uma vez que a ação de quietude não conduz a nenhuma transição, um estado que realize a ação δ é conduzido para o mesmo estado em que estava antes da transição: $q \xrightarrow{\delta} q$ se $\delta(q)$.

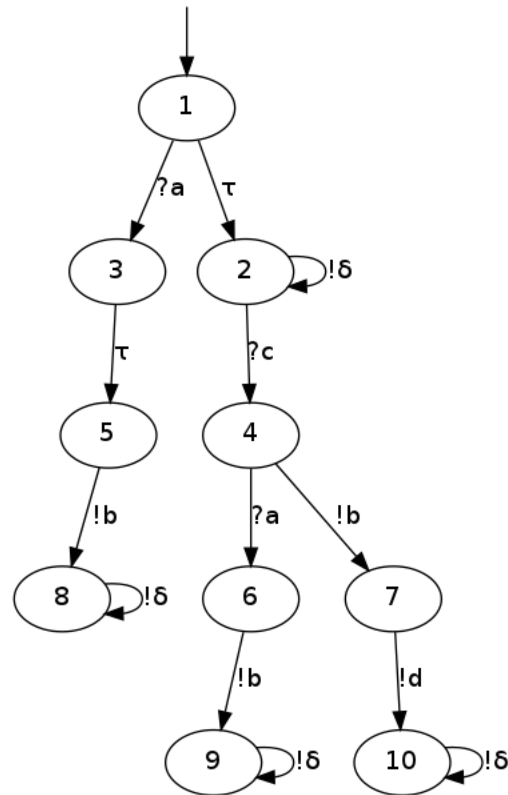
Desse modo, *traces* contendo a ação de quietude são denominados de *Suspension Traces* (Tretmans, 2008; Petrenko et al., 2003; Jard e Jéron, 2005): $STraces(p) =_{def} \{\sigma \in L{*}_\delta \mid p_\delta \stackrel{\sigma}{\Longrightarrow}\}$. O registro da quietude na especificação, descrita como IOTS, resulta na definição de *Suspension Automata* ($\triangle(S)$): torna a quietude explícita em IOTSs por uma ação observável δ. Para gerar um *Suspension automata* a partir de um IOTS, basta acrescentar transições tendo a quietude como saída. O *Suspension automata* obtido a partir do IOTS da Figura 4 é apresentado na Figura 5, que omite as entradas habilitadas e acrescenta as transições que possuem a quietude como saída.

Geralmente, IOTSs são representados por grafos. Entretanto, nem sempre isso é viável. Sistemas realísticos possuem facilmente bilhões de estados, e desenhá-los ou enumerá-los não é uma boa opção. Nesse caso, outra forma de representar IOTSs é definindo uma linguagem com IOTSs como sua semântica operacional. Cada expressão define sua semântica, e expressões podem ser combinadas com operadores da linguagem, de modo que IOTSs complexos possam ser compostos de forma simples. Tal linguagem é chamada de *linguagem de processo* (Tretmans, 2008). A linguagem *Lotos* (*Language Of Temporal Ordering Specification*) (Bolognesi e Brinksma, 1987; ISO/IEC, 1989) é um exemplo de linguagem de processo, sendo uma das mais utilizadas.

2.5.2 Framework para Teste de conformidade

Para compreender como é realizado o teste formal de conformidade, foi desenvolvido um framework para testar uma implementação com respeito à uma especificação formal de seu comportamento funcional (Tretmans, 2008). Para isso, deve-se ter uma implementação i a ser

Figura 5 – $\triangle(S)$ - *Suspension Automata* obtido a partir do IOTS da Figura 4



testada como um objeto real, uma especificação s como elemento de uma linguagem $SPEC$, que é o conjunto de especificações possíveis. Deste modo, é verificado se o comportamento de i está em conformidade com a especificação s.

Parte-se da hipótese de que qualquer implementação real pode ser modelada como um objeto formal i em um conjunto de modelos MOD, pressupondo um domínio particular de modelos. Essa é a *hipótese de teste* (Tretmans, 2008; Gaudel, 2010b). Nesse sentido, assume-se apenas que esse modelo existe, mas não é conhecido *a priori* (Tretmans, 2008). Desse modo, a conformidade pode ser expressa por uma relação formal entre modelos de implementação e especificação, chamada *relação de implementação* (**imp**) (Heerink e Tretmans, 1998; Tretmans, 2008), que é definida sobre um conjunto de observações feitas quando realizam-se testes em determinada implementação:

$\mathbf{imp} \subseteq MOD \times SPEC$, onde i **imp** s

As observações sobre o comportamento de uma implementação são investigadas por meio de experimentos, ou seja, casos de teste que especificam um estímulo e sua resposta esperada, expressos formalmente como elemento de algum domínio de casos de teste $TEST$. Aplicar um teste em uma implementação é chamado de *execução do teste* (Jard e Jéron, 2005; Tretmans, 2008). O sucesso da execução do teste é expresso como i **passes** t, e de não-sucesso como i **fails** t, sendo então que: $T \subseteq TEST : i \text{ } \mathbf{passes} \text{ } T \Leftrightarrow \forall t \in T : i \text{ } \mathbf{passes} \text{ } t$.

Portanto, o teste de conformidade avalia se uma implementação está em conformidade com sua especificação considerando a relação de implementação **imp**, procurando um conjunto de teste T tal que:

$$T : \forall i \in MOD : i \ \mathbf{imp} \ s \Leftrightarrow i \ \mathbf{passes} \ T \qquad (1)$$

Um grupo de testes com tal propriedade é dito completo. Porém, por questões práticas, duas outras propriedades são utilizadas: **consistência** (do inglês, *soundness*), que significa que nenhuma detecção falsa de erros pode ser realizada; e **exaustividade**, que significa que todas as deduções corretas podem ser feitas, ou seja, todos os erros podem ser detectados. Consistência corresponde à implicação da esquerda para direita em (1), e a implicação da direita para esquerda refere-se à exaustividade.

Para realizar os experimentos, é necessário ter casos de testes que revelem defeitos na implementação. Nesse sentido, um algoritmo que gera tal conjunto de casos de testes a partir de uma especificação, chamado de *geração de testes*, é definido como: $gen_{imp} : SPEC \rightarrow P(TEST)$, onde $P(TEST)$ denota o poder do conjunto $TEST$. Tal algoritmo é completo (consistente e exaustivo) se ele gera conjuntos de testes completos para todas as especificações. A geração de testes é o aspecto de maior benefício e visibilidade em TBM, pois permite a produção automática.

Portanto, para aplicação de TBM é necessária uma linguagem de especificação formal $SPEC$, um domínio de modelos de implementação MOD, uma relação de implementação $\mathbf{imp} \subseteq MOD \times SPEC$ que expresse corretude, um procedimento de execução de teste $\mathbf{passes} \subseteq MOD \times TEST$, um algoritmo de geração de testes $gen_{imp} : SPEC \rightarrow P(TEST)$, e uma prova de que um modelo de uma implementação passa um grupo de testes gerado se e somente se é **imp**-correta. Esses conceitos serão utilizados para formalizar o teste em IOTSs.

2.5.2.1 Teste de conformidade com IOTSs

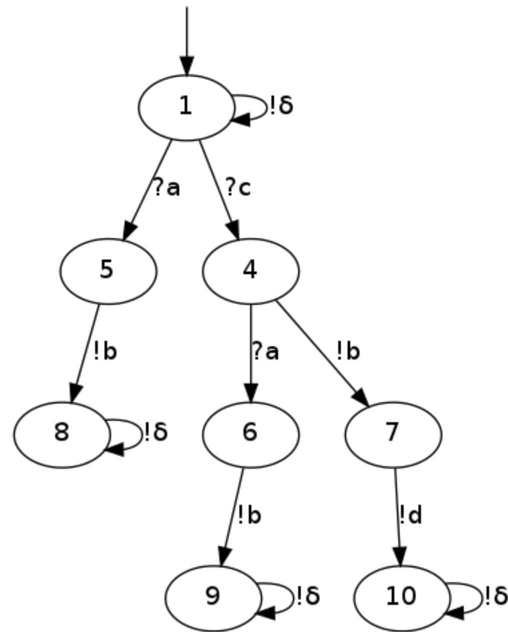
Para realizar o teste de conformidade a partir de IOTSs, que ocorre pela observação de comportamentos observáveis, é necessário que a especificação descrita em um modelo IOTS seja determinística. Nesse sentido, foi definido um procedimento para transformar um IOTS em um IOTS determinístico, denominado determinização (do inglês, *determinization*) (Jard e Jéron, 2005; Tretmans, 2008): duas sequências com os mesmos *traces* não podem ser distinguidas, mas seus respectivos sufixos devem ser considerados como possíveis evoluções do sistema. Além disso, a informação sobre quietude contida na especificação deve ser preservada. Isso é possível apenas se a quietude é computada na especificação, ou seja, se o IOTS foi definido como *Suspension Automata*. Desse modo, tornar um IOTS determinístico consiste em transformá-lo em *Suspension Automata* e realizar o procedimento que mantém no IOTS apenas os estados alcançáveis com comportamento observável. A determinização é escrita como:

$$det(M) = (2^S, (I \cup O), \rightarrow_{det}, s_o \, \mathbf{after} \, \varepsilon) \ \ onde:$$

$$para \ P, P' \in 2^S, a \in (I \cup O), P \xrightarrow{a}_{det} P' \iff P \, \mathbf{after} \, a.$$

Para exemplificar, foi realizada a determinização do IOTS representado na Figura 4. Considerando que esse IOTS foi transformado em um *Suspension Automata* (Figura 5), foi obtido o comportamento visível, representado na Figura 6.

Figura 6 – $det(\triangle(S))$ obtido pela determinização de $\triangle(S)$ da Figura 4

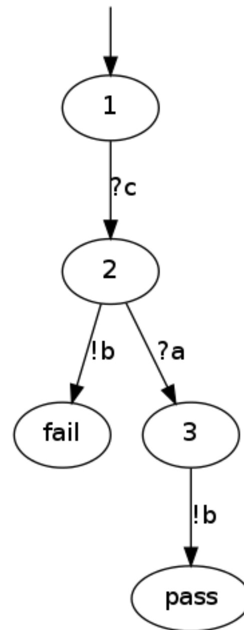


O próximo passo é obter casos de teste para realização do teste. Um caso de teste é a especificação do comportamento de um testador em um experimento. O testador serve como um tipo de ambiente artificial de uma implementação, modelado como um IOTS, porém, com as entradas e saídas trocadas. Além disso, casos de teste devem possuir um mecanismo para atribuir veredictos: **passes**, quando o teste ocorre conforme o previsto; **fail**, quando ocorre uma falha; e **inc**, quando ocorre uma exceção que pára a execução do teste (Petrenko et al., 2003). Esse mecanismo para atribuição de veredictos é definido pela adição de um estado para cada veredicto. Uma vez que um de tais estados for alcançado, o caso de teste termina. Para que seja possível atribuir o veredicto em tempo finito, casos de teste devem sempre permitir que um dos estados de veredicto sejam atingidos dentro de finitas transições (Tretmans, 2008; Jard e Jéron, 2005).

Um exemplo de caso de teste para o IOTS da Figura 4 foi representado na Figura 7. O caso de teste fornece a entrada $?c$ para a implementação. Se ocorrer a saída $!b$, o caso de teste é conduzido ao estado de falha e o teste termina com o veredicto de falha. Se nenhuma saída ocorrer, o caso de teste fornece a entrada $?a$ para a implementação, e se em seguida ocorrer a saída $!b$ o caso de teste vai para o estado passar e é conduzido ao veredicto passar, alcançando o sucesso esperado.

Assume-se que a comunicação entre a implementação e o caso de teste é síncrona. Como resultado, essa comunicação pode ser modelada com o operador de sincronização paralela, de

Figura 7 – Exemplo de um Caso de Teste para o IOTS da Figura 4



modo que as ações com visibilidade comum são sincronizadas. Isso implica que a interação pode ser bloqueada tanto pelo testador quanto pelo sistema. Além disso, a comunicação entre um sistema e seu ambiente é simétrica (Tretmans, 2008; Petrenko et al., 2003; Jard e Jéron, 2005).

Para verificar a conformidade no teste, é utilizada uma relação de implementação, que verifica se uma implementação é correta com respeito à sua especificação. A principal relação de implementação investigada no contexto de IOTSs é a relação *ioco* (do inglês, *input-output conformance*) (Tretmans, 1996; Heerink e Tretmans, 1998; van der Bijl et al., 2004; Jard e Jéron, 2005; Tretmans, 2008; Weiglhofer e Wotawa, 2009; Krichen, 2010). A relação *ioco* é uma técnica para verificar se uma implementação, representada como um IOTS, está em conformidade de entrada/saída com sua especificação, representada como um Sistema de Transição Rotulado (do inglês, *Labeled Transition Systems* - LTS). Uma implementação i está em *ioco*-conformidade com sua especificação s se qualquer experimento (caso de teste) derivado da especificação e executado na implementação conduz a uma saída da implementação que é prevista pela especificação. Ou seja, se após cada *Suspension trace* da especificação, a implementação exibir apenas saídas e quietudes permitidas na especificação (Jard e Jéron, 2005). O conjunto de saídas permitidas pela especificação é denominado $out(s\,\textbf{after}\,\sigma)$. Logo, uma implementação pode ter menos saídas do que a especificação permite, incluindo quietude, mas não pode ter saídas não especificadas. Desse modo, a definição formal de *ioco* é dada por:

$$i\,\textbf{ioco}\,s \Leftrightarrow \forall\sigma \in Straces(s) : out(i\,\textbf{after}\,\sigma) \subseteq out(s\,\textbf{after}\,\sigma)$$

No teste *ioco*, um testador e uma implementação se comunicam sincronamente. A execução de testes com o uso de um algoritmo de geração de casos de teste deve indicar se

uma implementação está em *ioco*-conformidade com sua especificação. Executar um caso de teste com uma implementação é um experimento em que o caso de teste fornece entradas para a implementação, enquanto observa as saídas e a ausência delas na implementação (Tretmans, 2008; Jard e Jéron, 2005; Gaudel, 2010a).

Para a geração de casos de teste, o algoritmo amplamente utilizado foi proposto por Tretmans (1996, 2008). Um caso de teste é obtido por infinitas aplicações de uma das três opções não-determinísticas de forma *on-the-fly* (*online*):

1. o teste passa e a recursão pára. Qualquer saída vinda da implementação é aceita;
2. fornece a próxima entrada para a implementação, mas está preparado para aceitar saídas;
3. verifica a próxima saída da implementação, ou conclui que a implementação está em quietude.

O algoritmo apresentado é recursivo. A primeira transição do caso de teste é derivada a partir dos estados que a especificação pode estar inicialmente. As partes restantes do caso de teste são recursivamente derivadas a partir dos estados alcancáveis na especificação, dado o estado inicial pela primeira transição do caso de teste. O algoritmo é não-determinístico no sentido que cada passo recursivo pode continuar de diferentes maneiras: o caso de teste pode terminar com o veredicto *passar* (escolha 1); o caso de teste pode continuar com qualquer entrada permitida pela especificação e pode ser interrompido por uma saída chegando (escolha 2); ou o caso de teste pode esperar por uma saída e verificá-la, ou concluir que a implementação está em quietude (escolha 3). O caso de teste é construído a cada passo da recursão, e concatenando todos os passos obtém-se um caso de teste (Tretmans, 2008). Infinitos casos de teste podem ser gerados a partir da especificação por meio desse algoritmo.

Considerando todos os componentes da teoria *ioco* que foram apresentados, pode-se definir o que é completude no contexto de teste *ioco* (Tretmans, 2008):

- Um conjunto de teste T é completo $\Leftrightarrow \forall i \in IOTS(L_I, L_U) : i \textbf{ ioco } s \text{ iff } i \textbf{ passes } T$
  - Um conjunto de teste T é consistente $\Leftrightarrow \forall i \in IOTS(L_I, L_U) : i \textbf{ ioco } s$ implica $i \textbf{ passes } T$
  - Um conjunto de teste T é exaustivo $\Leftrightarrow \forall i \in IOTS(L_I, L_U) : i \textbf{ ioco } s \text{ if } i \textbf{ passes } T$

Quando se diz que um conjunto de testes gerado pelo algoritmo é consistente, significa que só devem falhar os casos de teste que não forem *ioco*-corretos. Já um conjunto de testes é exaustivo se todas as falhas existentes forem reveladas pelos casos de teste. A exaustividade é uma questão mais teórica do que prática, uma vez que um conjunto de testes exaustivo deve conter uma infinidade de casos de teste que nunca poderão ser executados em tempo finito. Nesse

sentido, emerge a seleção de testes, na qual casos de teste tem de ser gerados e executados a partir de um conjunto de testes exaustivo. A seleção de testes deve minimizar os custos, em termos de tempo de execução, enquanto maximiza a possibilidade de revelar defeitos (Tretmans, 2008).

Desse modo, como o algoritmo dado anteriormente é não-determinístico, não é possível ter um controle sobre o custo de sua execução, já que são necessárias inúmeras iterações para obter um caso de teste completo. Além disso, apesar de ser mostrado que os conjuntos de teste gerados são completos, não é possível controlar se determinado critério de teste é satisfeito, ou ainda garantir que o conjunto gerado é efetivo em revelar defeitos por meio de condições de suficiência, como no contexto de MEFs. Logo, a questão de seleção de testes é uma das mais importantes questões em aberto nessa teoria (Tretmans, 2008).

2.6 Considerações Finais

Este capítulo apresenta uma visão geral do Teste de Software, especialmente da técnica de Teste Baseado em Modelos. A complexidade e a disponibilidade de mecanismos para automatização do processo de geração são fatores determinantes na escolha da técnica de modelagem de teste. Entre as diversas técnicas de modelagem, as MEFs e IOTSs têm sido amplamente utilizadas nas investigações recentes.

O teste a partir de MEFs possui uma teoria bem estabelecida e estudada há décadas (Chow, 1978), porém com contribuições recentes. Já o teste a partir de IOTSs é um tópico mais recente e que ainda possui lacunas a serem investigadas, já que não fornece o mesmo apoio que o teste baseado em MEFs. Para compreender o estado da arte e fornecer uma visão geral dos métodos existentes, um mapeamento sistemático da literatura sobre os métodos de geração de casos de teste a partir de IOTSs será apresentado no próximo capítulo.

CAPÍTULO

3

UM MAPEAMENTO SISTEMÁTICO SOBRE GERAÇÃO DE TESTES COM IOTS

Um mapeamento sistemático (MS) é um passo importante para identificar estudos relevantes bem como lacunas em um tópico de pesquisa (Petersen et al., 2008). Um mapeamento sistemático fornece uma visão geral sobre determinada área por meio de um processo rigoroso de avaliação e interpretação de todos os estudos disponíveis, visando obter evidências sobre o atual estado da arte de pesquisa em determinado tópico.

Recentemente, alguns trabalhos relataram estudos sistemáticos a respeito de TBM. Dias Neto et al. (2007) apresentaram uma revisão sistemática de literatura, na qual abordagens de TBM com diferentes modelos de comportamento foram identificadas e categorizadas. Todavia, abordagens a partir de IOTS não foram incluídas na revisão. Uma revisão sistemática de ferramentas de teste baseadas em estados foi apresentada por Shafique e Labiche (2013), resultando em 12 ferramentas. Entretanto, nenhuma delas emprega o formalismo IOTS. Um estudo recente apresenta uma revisão sistemática de literatura sobre o uso de modelos de ambiente em TBM (Siavashi e Truscan, 2015), mas não incluiu o formalismo IOTS.

Este capítulo descreve os resultados de um MS sobre métodos para geração de casos de teste a partir de IOTS. De acordo com o conhecimento obtido até aqui, este é o primeiro estudo que apresenta um MS sobre geração de teste a partir de IOTSs. O estudo aqui relatado pode ser encontrado em (Paiva e Simao, 2015). Uma atualização deste estudo foi realizada em Dezembro de 2015. Foram selecionados e analisados 97 estudos que foram categorizados conforme a taxonomia de abordagens TBM de Utting et al. (2012). Tal taxonomia envolve as características dos modelos utilizados, o método de geração de teste (critérios de seleção de teste e tecnologia de geração) e execução dos testes. Além disso, uma visão geral sobre os trabalhos selecionados, o tipo de evidência e o apoio computacional oferecido pelos métodos foi relatada. Este MS é uma fonte de informação útil para guiar investigações neste tópico por fornecer uma visão geral da área de pesquisa e identificar os resultados disponíveis.

Este capítulo está organizado da forma a seguir. As etapas de planejamento e condução realizadas no MS estão detalhadas na Seção 3.1. A Seção 3.2 apresenta a taxonomia para abordagens TBM estendida para o contexto deste trabalho e utilizada para classificação dos estudos. A Seção 3.3 sumariza as informações gerais dos trabalhos selecionados no MS. A Seção 3.4 apresenta a classificação dos trabalhos selecionados em relação à taxonomia de abordagens TBM (Utting et al., 2012).

3.1 Planejamento e Condução do Estudo

Este MS teve como objetivo identificar métodos de **geração de casos de teste a partir de IOTSs** através do processo sugerido por Petersen et al. (2008). Os principais pontos do processo são apresentados a seguir.

3.1.1 Questões de Pesquisa

As questões de pesquisa foram delineadas pela taxonomia de abordagens TBM (Utting et al., 2012):

- *QP1.* Quais são as principais características dos modelos IOTS aplicadas nos estudos?
- *QP2.* Quais são os critérios de seleção de casos de teste utilizados nos estudos?
- *QP3.* Quais são as tecnologias utilizadas na geração de teste a partir de IOTS?
- *QP4.* Quais são as características da execução de testes implementados nos estudos?

3.1.2 Estratégia de Pesquisa

A estratégia de pesquisa foi definida de acordo com uma lista de controle, apresentada na Tabela 1, que reúne os principais estudos já conhecidos pelos revisores. Os estudos desta lista serviram para calibrar a *string* de busca, já que estes deveriam estar entre os estudos relevantes. Foram escolhidas as palavras-chave "*geração de casos de teste*" e "*Sistemas de Transição com Entrada/Saída*" a partir destes estudos, e a *string* de busca foi definida da seguinte forma (em inglês):

*(test case generation **OR** test generation) **AND** (iots **OR** input/output transition systems **OR** labeled transition systems **OR** labelled transition systems **OR** transition system **OR** input/output labeled transition systems **OR** input/output labelled transition systems)*

Foram analisadas e consideradas as fontes que indexam as principais conferências e periódicos de Engenharia de Software, e foram então selecionados os seguintes mecanismos

Tabela 1 – Lista de Controle

Tretmans, J. **Model Based Testing with Labelled Transition Systems**. Formal Methods and Testing, 2008, 1-38 (Tretmans, 2008)
Hierons, R. M. **Implementation Relations for Testing through Asynchronous Channels**, The Computer Journal Advance Access, August 2012, 1-15. (Hierons, 2013)
Hierons, R. M. **The complexity of asynchronous model based testing**. Theoretical Computer Science, v. 451, 2012,70-82. (Hierons, 2012a)
Simao, A. S.; Petrenko, A. **Generating asynchronous test cases from test purposes**. Information and Software Technology, v. 53, 2011, 1252-1262. (Simao e Petrenko, 2011)
Fraser, G.; Weiglhofer, M.; Wotawa, F. **Coverage Based Testing with Test Purposes**. Quality Software, 2008. QSIC '08. The Eighth International Conference on, 2008, 199 -208. (Fraser et al., 2008)

de busca das bibliotecas digitais *web* (Dyba et al., 2007): *IEEE*[1], *Springer*[2], *Scopus*[3], *Web of Science-ISI*[4], *ACM*[5], *Elsevier-Science Direct*[6], *Compendex-Engineering Village*[7] *e Oxford Journals*[8] (incluída por causa de um artigo da lista de controle).

3.1.3 Processo de seleção e extração de dados dos estudos

A *string* de busca identificou os estudos candidatos em cada base, das quais retornaram 1371 estudos. Cerca de 600 estudos recuperados eram indexados pela IEEE e quase 500 estudos pela Springer, já que elas indexam muitas conferências na área de TBM. Após exclusão dos estudos duplicados, restaram 1321 estudos para o processo de seleção, como pode ser visto na Tabela 2.

Na atualização do MS, os estudos inicialmente foram excluídos com base no título, resumo e palavras-chave, resultando em 192 estudos. Tal número é grande devido a vários estudos não fornecerem informações suficientes no resumo, como qual a técnica de modelagem utilizada. Estes estudos candidatos passaram para a segunda fase onde foram lidos por completo e os critérios foram aplicados novamente. Um total de 59 estudos foram selecionados para a fase de extração de dados, como mostra a Tabela 2. Vários estudos que foram excluídos na segunda fase destinavam-se ao teste a partir de MEFs. Isso ocorreu porque este modelo possui uma teoria de teste bem estabelecida e alguns estudos tem relacionado os métodos para MEFs com os métodos para IOTS (Hierons, 2012a, 2013). As ferramentas JabRef [9] e Google Spreadsheet

[1] http://ieeexplore.ieee.org
[2] http://www.springerlink.com/
[3] http://www.scopus.com
[4] http://www.webofknowledge.com/
[5] http://portal.acm.org/dl.cfm
[6] http://www.sciencedirect.com/
[7] http://www.engineeringvillage2.org
[8] http://www.oxfordjournals.org/
[9] http://jabref.sourceforge.net/

foram utilizadas no processo de extração de dados e um especialista validou os resultados. 38 estudos selecionados na primeira execução não foram retornados pelas buscas e foram incluídos manualmente. A Tabela 2 mostra os resultados dos estudos selecionados nesta atualização e o resultado final após a inclusão manual.

Tabela 2 – Estudos retornados e incluídos para cada fonte

Fontes	Retornados	Selecionados	Incluídos
IEEE	609	15	19
ACM	100	0	1
Scopus	30	12	16
Springer Link	477	10	39
ScienceDirect	84	12	12
Web of Science	7	3	3
Engineering Village	10	5	5
Oxford Journals	4	2	2
Total	**1321**	**59**	**97**

O processo de extração de dados foi guiado pelas questões de pesquisa, que seguiram a taxonomia para abordagens TBM (Utting et al., 2012) estendida para o contexto deste trabalho na Seção 3.2. Os estudos foram também classificados de acordo com o *tipo de evidência* (avaliação dos resultados) e *apoio computacional* (ferramentas utilizadas ou propostas nos estudos).

3.2 Taxonomia para abordagens TBM

A taxonomia para abordagens TBM (Utting et al., 2012) caracteriza seis dimensões essenciais que foram divididas em três categorias: Especificação do Modelo, Geração dos Testes e Execução dos Testes. Essa taxonomia foi estendida para o contexto deste trabalho, conforme apresentada na Figura 8 (itens em vermelho foram adicionados) e detalhada a seguir:

1) **Especificação do Modelo:** Esta categoria é refletida por três dimensões:

- *Escopo:* O *escopo* dos estudos é o *comportamento de entrada/saída*, já que considera-se apenas o modelo IOTS.
- *Paradigma:* O *paradigma* é *notação baseada em transições*, uma vez que IOTSs descrevem as transições entre os diferentes estados. Logo, apenas esse paradigma foi considerado.
- *Características:* Os estudos foram agrupados de acordo com as *características* do modelo IOTS, ou seja, temporal/atemporal, determinístico/não-determinístico, discreto/híbrido/contínuo além das diferentes classes de IOTS identificadas.

2) **Geração dos Testes:** Duas dimensões refletem esta categoria:

- *Critérios de seleção de testes:* Os estudos foram agrupados em: cobertura estrutural do modelo, cobertura dos requisitos (divididos em *propósitos de teste* e *teste baseado em propriedades*), especificações de casos de teste que descreve um caso de teste em alto nível (a teoria *ioco* foi considerada separadamente) e baseada em defeitos.
- *Tecnologia:* Os estudos foram agrupados em: geração aleatória/não-determinística, baseada em buscas, verificação de modelos, execução simbólica e resolução de restrições; regras de inferência e abstração de dados foram adicionadas.

3) **Execução dos Testes:** Esta categoria reflete como os conjuntos de teste serão executados. As seguintes dimensões foram consideradas:

- *Offline:* Os casos de teste são gerados estritamente antes de serem executados.
- *Online* ou *on-the-fly:* geração de testes e execução ocorrem simultaneamente.
- *Communicação:* Esta dimensão foi adicionada, e consiste na comunicação entre o testador e a implementação. Esta dimensão é dividida em síncrona (como na teoria *ioco*), assíncrona (utilizando canais de comunicação) ou distribuída (com portas diferentes).

Como os estudos podem adotar várias características, as categorias podem se sobrepor.

3.3 Informações gerais sobre os estudos selecionados

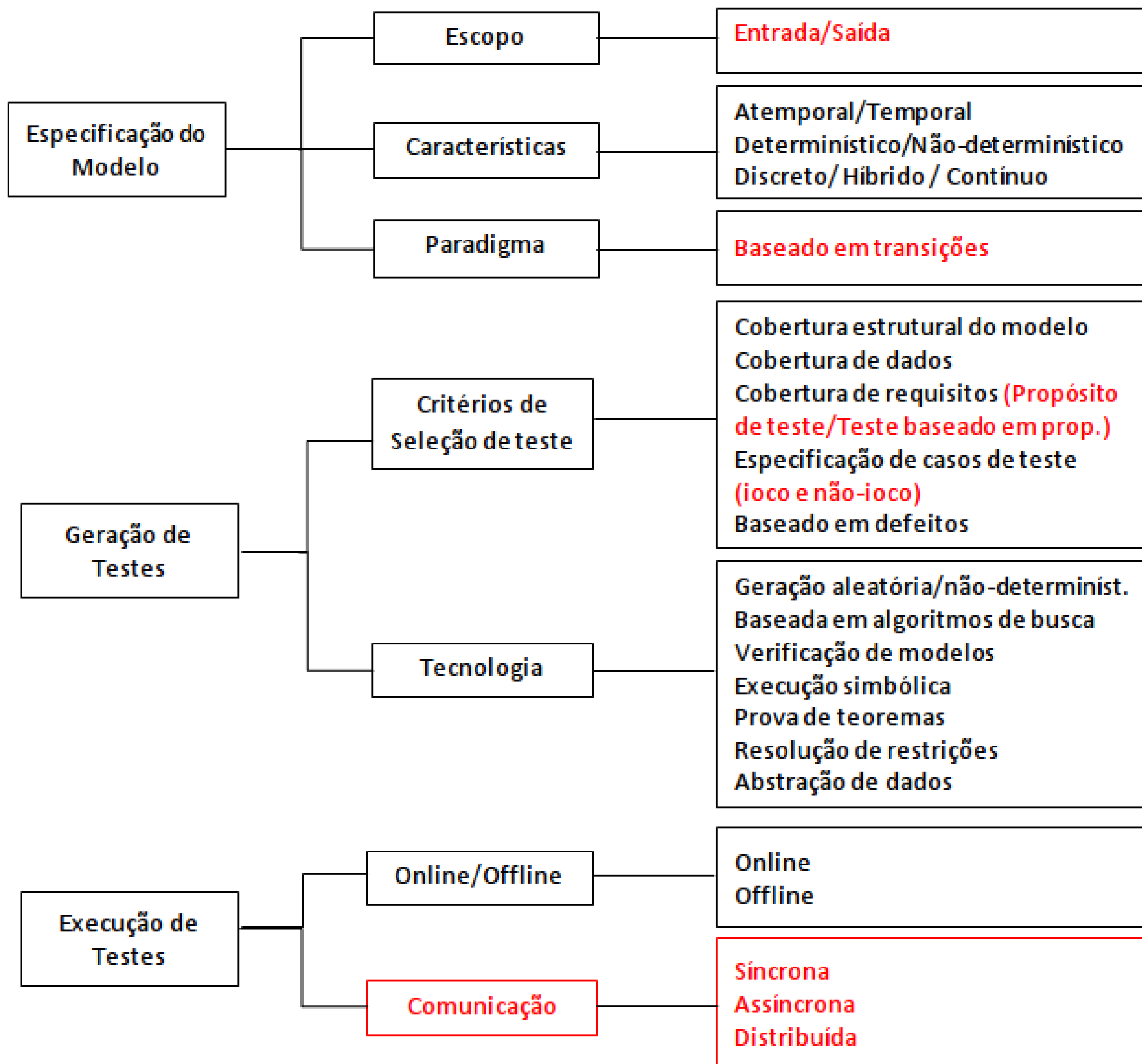
3.3.1 Distribuição dos estudos ao longo dos anos

A Figura 9 mostra a distribuição dos estudos ao longo dos anos, evidenciando o aumento de interesse na geração de testes a partir de IOTS recentemente. Houve notável crescimento de publicações neste tópico a partir de 2004 evidenciando que apesar de ser um tópico investigado há muito tempo, ainda é um tópico muito ativo. Em 2015 não há um número expressivo de estudos, já que esta atualização foi conduzida ainda em 2015.

3.3.2 Distribuição dos estudos nos fóruns

A Figura 10 mostra os tipos de publicação dos estudos. Cerca de 45% dos estudos foram publicados em periódicos. Entretanto, LNCS (*Lecture Notes in Computer Science*, da Springer) foi considerado separadamente, pois corresponde à Anais de conferências indexados pela Springer e sua política de publicação é diferente da política de um periódico. Assim, aproximadamente 55% dos estudos foram publicados em conferências. Os estudos estão bem distribuídos entre os diferentes fóruns, enfatizando-se a conferência *International Conference on Testing Software and Systems (ICTSS)* e o periódico *International Journal on Software Tools for Technology Transfer* como os fóruns mais utilizados.

Figura 8 – Taxonomia estendida para o contexto de geração de testes a partir de IOTSs



3.3.3 Grupos de pesquisa e autoria

A maioria dos estudos foram realizados por grupos de pesquisa localizados na Europa. Entre os pesquisadores que destacam-se estão Robert M. Hierons, autor dos estudos mais recentes, e Jan Tretmans, autor do estudo clássico de teste a partir de IOTS. Além disso, existem trabalhos em colaboração com grupos de pesquisadores brasileiros (Machado et al., 2007; Damasceno et al., 2015; Sampaio et al., 2009; Carvalho et al., 2014; Simao e Petrenko, 2011, 2014), da Universidade Federal de Campina Grande, Universidade Federal de Pernambuco e da Universidade de São Paulo.

Figura 9 – Distribuição dos estudos ao longo dos anos

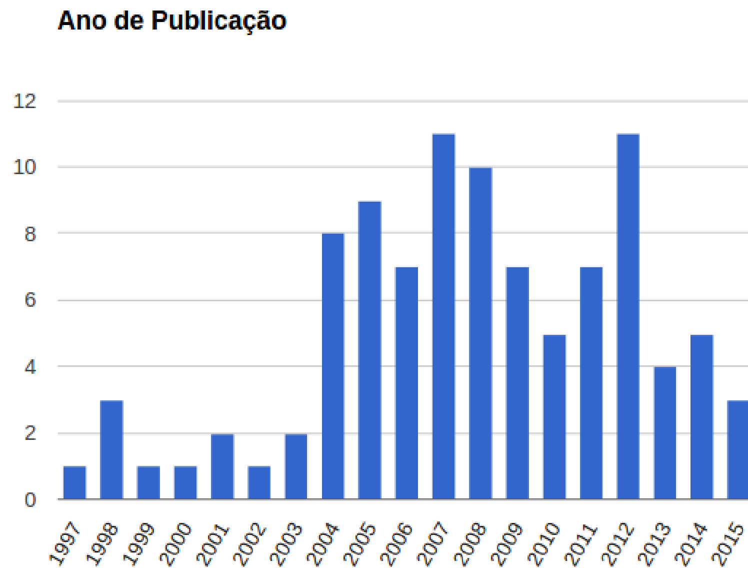
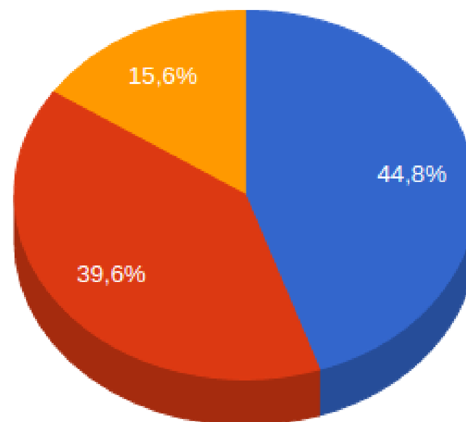


Figura 10 – Tipo de publicação dos estudos



3.3.4 Tipo de evidência

Foram consideradas as classes de estudos empíricos da área de teste de software (Do et al., 2005). 7,3% dos estudos relatam resultados experimentais; 15,6% relatam um estudo de caso; 64,6% apenas fornecem exemplos da abordagem proposta; e 12,5% apenas discutem a aplicação da abordagem. Um grande número de estudos foi validado apenas com exemplos, o que implica em um pequeno número de *experimentação* e sugere que mais estudos empíricos são necessários para validar os resultados teóricos das abordagens de teste a partir de IOTS.

3.3.5 Apoio computacional

O uso de ferramentas em TBM é essencial para a automatização do processo de teste. Algumas das ferramentas identificadas que automatizam a geração de teste a partir de IOTS são:

- *TGV* (*Test Generation with Verification technology*) (Jard e Jéron, 2005; Jéron e Morel, 1999; Aichernig et al., 2007; Clarke et al., 2001; Scollo e Zecchini, 2005; Calamé et al., 2007; Pickin et al., 2007), utilizada tanto em pesquisa quanto na indústria ela possibilita a geração de casos de testes a partir da exploração parcial de gráficos de estados;

- *TorX*, baseada na teoria *ioco* (Tretmans, 2008);

- *UPPAAL* (Hessel et al., 2008), que implementa o teste *offline* como um problema de verificação de modelos e o teste *online* implementa o algoritmo de Tretmans (2008);

- *LTS-BT* (Andrade e Machado, 2012), um seletor de casos de teste para aplicações embarcadas.

Apenas 38% dos estudos mencionam o uso ou implementação de ferramentas, o que mostra que muitos estudos estão interessados apenas em contribuições teóricas para a geração de teste a partir de IOTS. Além disso, a maioria dos estudos aplica ferramentas já existentes, como geradores de casos de teste, geradores de propósitos de teste, provadores de teoremas e verificadores de modelos. 8,3% dos estudos relataram uma extensão à alguma ferramenta. Este cenário mostra a complexidade em construir uma nova ferramenta que apoie a geração de testes.

3.4 Classificação dos estudos

3.4.1 QP1 - Características do modelo IOTS

Embora o foco deste estudo seja o modelo IOTS, uma ampla variedade de termos e propriedades do modelo foram encontradas. Foram agrupadas as principais classes de IOTS encontradas:

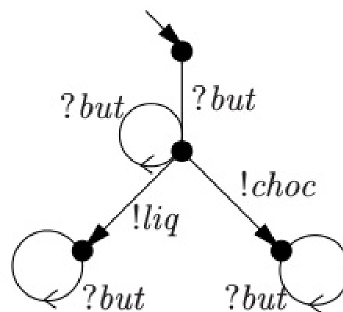
- **IOTS** (28 estudos): entradas do ambiente nunca são rejeitadas pelo sistema e saídas do sistema nunca são rejeitadas pelo ambiente, o que caracteriza a habilidade para entrada, conforme o exemplo apresentado na Figura 11 (extraído de (Tretmans, 2008)). Vários estudos tem considerado que IOTS pode alcançar um estado quiescente (estado na qual nenhuma ação de saída é executada) (Tretmans, 2008; Jard e Jéron, 2005; Jéron e Morel, 1999; Weiglhofer e Wotawa, 2009; Hierons, 2012a; Huo e Petrenko, 2004; Hierons, 2015; Simao e Petrenko, 2014; Hierons et al., 2014; Noroozi et al., 2013).

- **IOLTS** (27 estudos): Um IOLTS (*Input/Output Labeled Transition System*) é um IOTS que não considera a característica de habilidade para entrada. Alguns estudos consideram estados com quietude (Jard e Jéron, 2005; Jéron e Morel, 1999), como pode ser visto na Figura 12 (extraído de (Jard e Jéron, 2005)). Foram encontrados estudos que aplicam o modelo LTS (*Labeled Transition System*), mas consideram as mesmas propriedades de

IOLTS (Gromov e Willemse, 2007; Bourdonov et al., 2006; Kervinen e Virolainen, 2005; Xie e Dang, 2006; Helovuo e Leppanen, 2001).

- **IOSTS/STS** (10 estudos): Um STS (*Symbolic Transition Systems*) é um modelo que inclui uma noção explícita de dados e fluxo de controle dependente de dados. Um IOSTS (*Input/Output STS*) é um STS que distingue ações de entrada e saída (Clarke et al., 2001; Faivre et al., 2007; Frantzen et al., 2006; Gaston et al., 2006; Jeannet et al., 2007; Le Gall et al., 2007; Rusu et al., 2005; Frantzen et al., 2005; Jéron, 2009; Damasceno et al., 2015). Um exemplo é dado na Figura 13 (extraído de (Gaston et al., 2006)).
- **TIOTS** (6 estudos): Timed IOTS (Briones e Brinksma, 2005; Hessel et al., 2008; Schmaltz e Tretmans, 2008; Pardo et al., 2010; Bannour et al., 2013; Carvalho et al., 2014), incorpora a noção de tempo com rótulos que indicam o progresso de tempo, conforme mostrado na Figura 13 (extraído de (Hessel et al., 2008)).
- **MIOTS** (6 estudos): Multi IOTS é um IOTS cujas entradas e saídas são particionadas em diferentes canais, para refletir a localização das ações, como pode ser visto na Figura 15 (extraído de (Li et al., 2004a)). Todo canal de ação de entrada deve ser habilitado para entrada simultaneamente.(Brinksma et al., 1998; Li et al., 2004a, 2003, 2004b; Hierons, 2012b; Hierons et al., 2008). Porém, há trabalhos que utilizam o modelo IOTS mas com arquitetura distribuída (Hierons et al., 2014; Hierons, 2015).

Figura 11 – Exemplo de IOTS



Algumas classes de IOTS têm sido definidas com as mesmas características. Embora a habilidade para entrada seja uma característica específica de IOTSs, alguns estudos tem considerado esta característica em IOLTSs (Jard e Jéron, 2005; Aichernig e Delgado, 2006; Aichernig et al., 2007, 2008; Calamé et al., 2007; Gnesi et al., 2004). Também foram encontrados estudos na qual IOTSs não possuem a habilidade para entrada (Weiglhofer e Wotawa, 2009; Weiglhofer e Aichernig, 2010). Esta variação mostra a falta de padrão nas definições. Por outro lado, cada propriedade do modelo pode ser um fator determinante no contexto de geração de testes. O trabalho de Noroozi et al. (2013) faz uma clara definição entre as propriedades de IOTS e IOLTS e pode ser tomado como referência. Apesar das diferentes classes de IOTSs encontradas neste estudo, o interesse é com as questões relacionadas ao modelo geral IOTS.

Figura 12 – Exemplo de IOLTS

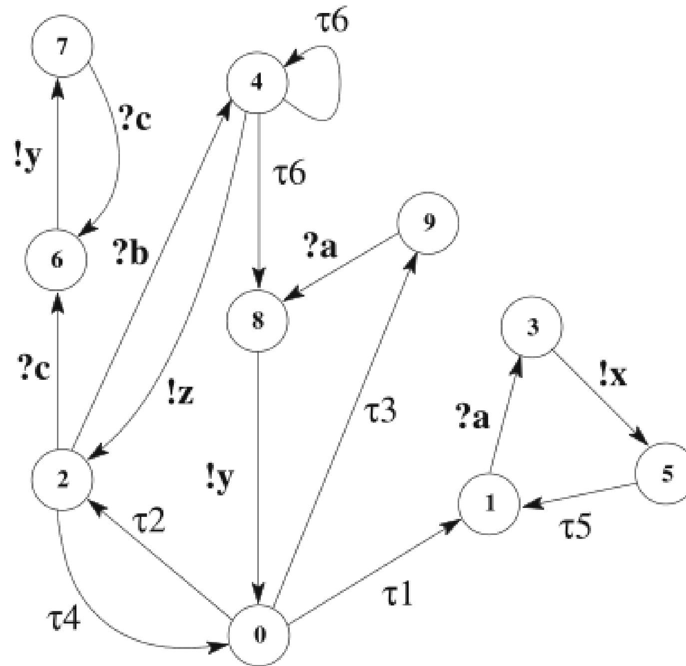
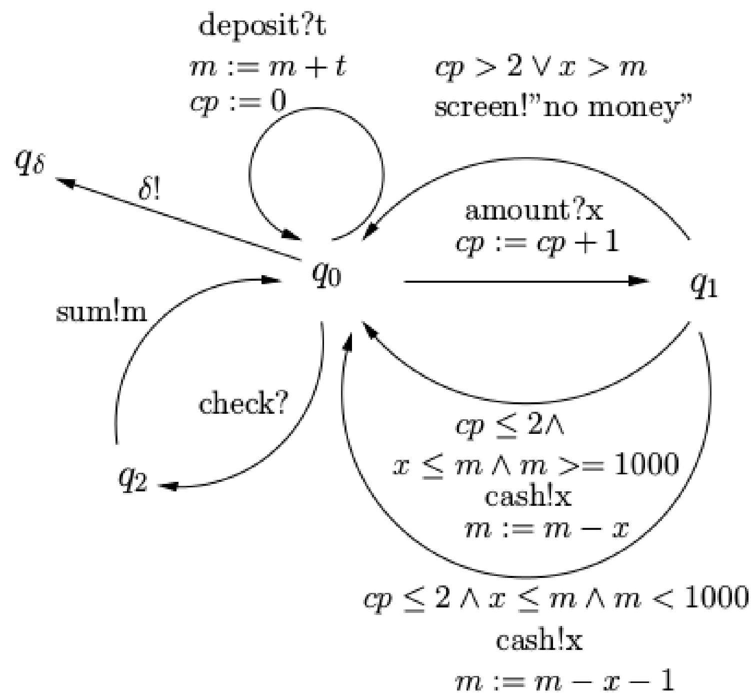


Figura 13 – Exemplo de IOSTS



Os estudos foram categorizados de acordo com as características de modelos (Utting et al., 2012):

- *Atemporal:* 92% e *Temporal:* 8% (Briones e Brinksma, 2005; Hessel et al., 2008; Schmaltz e Tretmans, 2008; Pardo et al., 2010; Bannour et al., 2012; Hierons et al., 2012a; Rollet e Saad-Khorchef, 2007; Carvalho et al., 2014; Bannour et al., 2013; Damasceno et al.,

Figura 14 – Exemplo de TIOTS

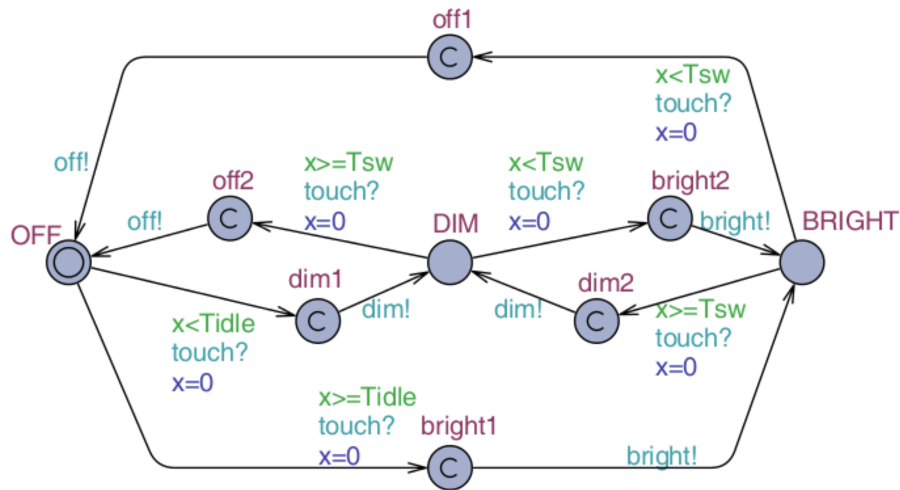
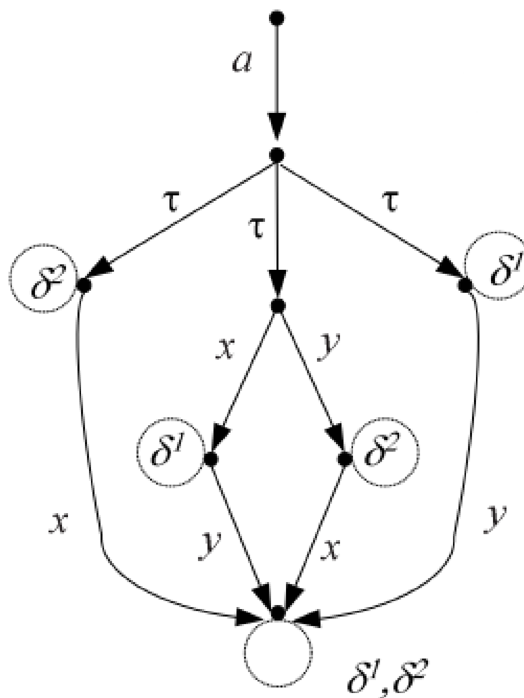


Figura 15 – Exemplo de MIOTS



2015));

- *Não-determinístico:* 88% e *Determinístico:* 12% (Andrade e Machado, 2012; Bensalem et al., 2008) (Briones et al., 2006; Chédor et al., 2012; Hessel et al., 2008; Hierons, 2008, 2012a; Li et al., 2004a, 2003; Muccini et al., 2004);

- *Discreto:* 98%, *Contínuo:* 1% (Briones e Brinksma, 2005) e *Híbrido:* 1% (Brandl et al., 2010).

3.4.2 QP2 - Critérios de Seleção de Testes

A Tabela 3 apresenta a classificação dos estudos conforme as categorias de geração de teste exibidas na Seção 3.1.3. A cobertura de requisitos utilizada na teoria *ioco* e propósitos de teste são os critérios de seleção de testes mais utilizados. Um propósito de teste, que é citado por 31% dos estudos, representa um conjunto finito de *traces* relacionados a uma dada funcionalidade e podem ser considerados como cenários que devem ser seguidos durante o teste. No entanto, o veredicto de teste não é especificado nos propósitos de teste (Simao e Petrenko, 2011).

Outra maneira de definir propriedades a serem testadas é cobrindo estados e transições, que é um critério de cobertura estrutural que pode ser aplicado por um modelo de defeitos. Apesar de ser um critério amplamente utilizado em outras técnicas de modelagem, apenas 4 estudos recentes tem utilizado este critério (Huo e Petrenko, 2005, 2004; Hierons, 2012a; Simao e Petrenko, 2014). O conceito de hipótese de teste, que é similar aquele utilizado em modelos de defeitos, tem desempenhado um papel relativamente pequeno no teste a partir de IOTS (Petrenko et al., 2003; Hierons et al., 2009).

Tabela 3 – Classificação quanto à geração de testes

Classificação	**Referências**	**#**
	Critérios de Seleção de Testes	
Cobertura estrutural do modelo	(Andrade e Machado, 2012; Bensalem et al., 2008; Briones et al., 2006; Frantzen et al., 2006; Fraser et al., 2008; Gaston et al., 2006; Hierons, 2012a; Huo e Petrenko, 2005, 2004; Jard e Jéron, 2005; Le Gall et al., 2007; Muccini et al., 2004; Tretmans, 2011; van der Bijl et al., 2005)	14
Cobertura de requisitos: propósitos de teste	(Aichernig e Delgado, 2006; Aichernig et al., 2007, 2008; Andrade e Machado, 2012; Bannour et al., 2012; Brinksma et al., 1998; Calamé et al., 2007; Chédor et al., 2012; Clarke et al., 2001; Desmoulin e Viho, 2009; Faivre et al., 2007; Fraser et al., 2008; Gaston et al., 2006; Hao e Wu, 1997; Jard e Jéron, 2005; Jeannet et al., 2007; Jéron e Morel, 1999; Koné e Castanet, 2000; Lestiennes e Gaudel, 2002; Li et al., 2004a; Machado et al., 2007; Rollet e Salva, 2009; Sampaio et al., 2009; Scollo e Zecchini, 2005; Simao e Petrenko, 2011; Weiglhofer et al., 2009; Jéron, 2009; Damasceno et al., 2015; Bhateja, 2014)	29
Cobertura de requisitos: propriedades	(Falcone et al., 2012; Fernandez et al., 2004; Machado et al., 2007; Rusu et al., 2005; Weiglhofer et al., 2009)	5
Especificação dos casos de teste	(Hierons, 2012b; Hierons et al., 2011, 2012a; Kervinen e Virolainen, 2005; Xie e Dang, 2006; Fu e Koné, 2012; Hao e Wu, 1998; Pickin et al., 2007; Helovuo e Leppanen, 2001)	9

Especificação dos casos de teste: teoria *ioco*	(Bourdonov et al., 2006; Brandl et al., 2010; Brinksma et al., 1998; Briones et al., 2006; Briones e Brinksma, 2005; Cavalcanti et al., 2011; Faivre et al., 2007; Frantzen et al., 2006; Frantzen e Tretmans, 2007; Frantzen et al., 2005; Gaston et al., 2006; Gnesi et al., 2004; Hierons, 2008; Hierons et al., 2008, 2012b, 2011, 2012a; Jeannet et al., 2007; Lestiennes e Gaudel, 2002; Li et al., 2003, 2004b; Pardo et al., 2010; Rollet e Saad-Khorchef, 2007; Rollet e Salva, 2009, 2008; Sampaio et al., 2009; Schmaltz e Tretmans, 2008; van Beek e Mauw, 2004; van der Bijl et al., 2004; Weiglhofer e Wotawa, 2009; Willemse, 2007; Noroozi et al., 2013; Hierons et al., 2014; Bannour et al., 2013; Carvalho et al., 2014; Hierons, 2015)	36
Baseado em defeitos	(Aichernig e Delgado, 2006; Aichernig et al., 2007, 2008; Briones et al., 2006; Faivre et al., 2007; Fernandez et al., 2005; Gromov e Willemse, 2007; Hierons et al., 2012b; Jeannet et al., 2007; Petrenko et al., 2003; Rollet e Saad-Khorchef, 2007; Rollet e Salva, 2008; Simao e Petrenko, 2014)	13
	Tecnologia	
Algoritmo não-determinístico/ aleatório	(Bourdonov et al., 2006; Brandl et al., 2010; Brinksma et al., 1998; Briones e Brinksma, 2005; Chédor et al., 2012; Faivre et al., 2007; Frantzen et al., 2006; Frantzen e Tretmans, 2007; Gaston et al., 2006; Gaudel, 2010b; Gnesi et al., 2004; Gromov e Willemse, 2007; Hierons et al., 2008, 2012b; Jeannet et al., 2007; Lestiennes e Gaudel, 2002; Li et al., 2004a, 2003; Hessel et al., 2008; Kervinen e Virolainen, 2005; Li et al., 2004b; Pardo et al., 2010; Rollet e Salva, 2009, 2008; Tretmans, 2008; van Beek e Mauw, 2004; van der Bijl et al., 2005, 2004; Weiglhofer e Wotawa, 2009; Bhateja, 2014)	27
Algoritmo baseado em buscas	(Andrade e Machado, 2012; Bi et al., 1998; Briones et al., 2006; Chen, 2011; Desmoulin e Viho, 2009; Fernandez et al., 2004, 2005; Fraser et al., 2008; Fu e Koné, 2012; Hao e Wu, 1997; Hierons, 2012b; Hierons et al., 2012b; Huo e Petrenko, 2005, 2004; Jard e Jéron, 2005; Jéron e Morel, 1999; Kervinen e Virolainen, 2005; Koné e Castanet, 2000; Le Gall et al., 2007; Muccini et al., 2004; Petrenko et al., 2003; Hao e Wu, 1998; Scollo e Zecchini, 2005; Simao e Petrenko, 2011; Weiglhofer et al., 2009; Xie e Dang, 2006; Simao e Petrenko, 2014; Fu e Koné, 2015; De León et al., 2013; Pickin et al., 2007)	30
Verificação de modelos	(Aichernig e Delgado, 2006; Gromov e Willemse, 2007; Hessel et al., 2008; Jard e Jéron, 2005; Jéron e Morel, 1999; Sampaio et al., 2009; Aichernig et al., 2007, 2008; Helovuo e Leppanen, 2001)	9

Execução simbólica	(Bannour et al., 2012; Clarke et al., 2001; Frantzen et al., 2005; Gaston et al., 2006; Hessel et al., 2008; Jeannet et al., 2007; Le Gall et al., 2007; Lestiennes e Gaudel, 2002; Machado et al., 2007; Rusu et al., 2005; Jéron, 2009; Damasceno et al., 2015)	12
Resolução de restrições	(Calamé et al., 2007; Falcone et al., 2012; Fernandez et al., 2005; Frantzen et al., 2006, 2005; Le Gall et al., 2007)	6
Regras de inferência	(Aiguier et al., 2012; Bannour et al., 2012; Bhateja, 2011; Bi et al., 1998; Gaston et al., 2006)	5
Abstração de dados	(Calamé et al., 2007)	1

O framework de teste de conformidade proposto por Tretmans (2008) expõe a necessidade de uma relação de implementação. 39% dos estudos utilizam ou estendem a relação *ioco* para diferentes contextos, como por exemplo *mioco (multi-ioco)* (Brinksma et al., 1998), *sioco (symbolic-ioco)* (Frantzen et al., 2006), *tioco (timed-ioco)* (Briones e Brinksma, 2005), *dioco (distributed-ioco)* (Hierons, 2015) e *co-ioco (concurrent ioco)* (De León et al., 2013).

3.4.3 QP3 - Tecnologia

A geração aleatória/não-determinística (Hessel et al., 2008; Jard e Jéron, 2005; Jeannet et al., 2007) é a tecnologia de geração de testes mais utilizada, por ser a tecnologia do algoritmo clássico de (Tretmans, 2008). 27,05% dos estudos estendem tal algoritmo clássico considerando diferentes aspectos de teste, como por exemplo robustez (Rollet e Salva, 2009; Fernandez et al., 2005).

A execução simbólica visa a uma representação mais concisa e abstrata do sistema, evitando o problema de explosão de estados, embora os algoritmos de geração de teste existentes sejam não-determinísticos. Outras abordagens, como baseada em busca e regras de inferência utilizam a geração não-determinística para obter conjuntos de testes, já que não existe um padrão de modelo de defeitos que conduza a algoritmos determinísticos (Hierons et al., 2009). Apenas alguns estudos tem efetivamente lidado com estratégias de teste bem estabelecidas, como o uso de critérios de cobertura.

As abordagens para teste a partir de IOTSs podem demandar algoritmos de geração de casos de teste mais simples, mas eles tem que lidar com o problema de explosão de estados devido à representação explícita dos valores de dados quando estruturas de dados complexas são envolvidas. Por outro lado, abordagens simbólicas (Clarke et al., 2001; Faivre et al., 2007; Frantzen et al., 2005; Gaston et al., 2006; Frantzen et al., 2006; Jeannet et al., 2007; Le Gall et al., 2007; Rusu et al., 2005; Damasceno et al., 2015; Jéron, 2009) representam os sistemas de forma mais abstrata e concisa e destinam-se ao problema de explosão de estados. No entanto, a

decidibilidade e computabilidade de especificações de dados são de interesse para a geração de casos de teste. A maioria dos estudos existentes estende o framework de teste de conformidade de Tretmans (2008).

Dos estudos analisados, 22,35% tem se destinado a características particulares de certos tipos de sistemas, como sistemas de protocolo de comunicação (Aichernig et al., 2007, 2008; Hao e Wu, 1997), sistemas de tempo real (Bannour et al., 2012; Briones e Brinksma, 2005; Hessel et al., 2008), aplicações *Web* (van Beek e Mauw, 2004), sistemas concorrentes (Bi et al., 1998; Aichernig e Delgado, 2006; Xie e Dang, 2006), sistemas reativos (Andrade e Machado, 2012; Boroday et al., 2008; Bhateja, 2009; Jard e Jéron, 2005; Jeannet et al., 2007; Machado et al., 2007) e sistemas orientados a componentes (Faivre et al., 2007; Frantzen e Tretmans, 2007).

3.4.4 QP4 - Execução dos Testes

A Tabela 4 exibe a categorização dos estudos com relação à execução dos testes, conforme a taxonomia apresentada na Seção 3.1.3. A geração de teste online e síncrona tem sido amplamente empregada, por serem estas as características do método de Tretmans (2008). Geradores de teste online tipicamente utilizam técnicas de escolha aleatória, que fornecem apenas uma garantia probabilística de cobertura para um (irrealístico) longo tempo de execução.

Tabela 4 – Classificação das propriedades para Execução dos Testes

Classificação	Referências	#
	Execução dos Testes	
Offline	(Aichernig e Delgado, 2006; Aichernig et al., 2007, 2008; Andrade e Machado, 2012; Bannour et al., 2012; Bhateja, 2011; Bi et al., 1998; Briones et al., 2006; Chédor et al., 2012; Chen, 2011; Clarke et al., 2001; Desmoulin e Viho, 2009; Fernandez et al., 2004, 2005; Fu e Koné, 2012; Hessel et al., 2008; Hierons et al., 2012b; Huo e Petrenko, 2005, 2004; Jeannet et al., 2007; Jéron e Morel, 1999; Koné e Castanet, 2000; Muccini et al., 2004; Petrenko et al., 2003; Hao e Wu, 1998; Simao e Petrenko, 2014; Jéron, 2009; Helovuo e Leppanen, 2001; Fu e Koné, 2015)	29

Online	(Bhateja, 2011; Bourdonov et al., 2006; Brandl et al., 2010; Brinksma et al., 1998; Chédor et al., 2012; Frantzen et al., 2006; Frantzen e Tretmans, 2007; Frantzen et al., 2005; Fraser et al., 2008; Gaston et al., 2006; Gnesi et al., 2004; Gromov e Willemse, 2007; Hessel et al., 2008; Hierons, 2012a; Hierons et al., 2008, 2012b; Jard e Jéron, 2005; Kervinen e Virolainen, 2005; Koné e Castanet, 2000; Le Gall et al., 2007; Lestiennes e Gaudel, 2002; Li et al., 2004a, 2003, 2004b; Pardo et al., 2010; Rollet e Salva, 2009, 2008; Rusu et al., 2005; Sampaio et al., 2009; Scollo e Zecchini, 2005; Tretmans, 2008; van Beek e Mauw, 2004; van der Bijl et al., 2005; Weiglhofer et al., 2009; Weiglhofer e Wotawa, 2009; Fu e Koné, 2015; Pickin et al., 2007; Bannour et al., 2013; Damasceno et al., 2015; De León et al., 2013)	39
	Comunicação dos Testes	
Síncrona	(Bourdonov et al., 2006; Brandl et al., 2010; Brinksma et al., 1998; Briones e Brinksma, 2005; Chédor et al., 2012; Faivre et al., 2007; Frantzen et al., 2006; Frantzen e Tretmans, 2007; Gaston et al., 2006; Gaudel, 2010b; Gnesi et al., 2004; Gromov e Willemse, 2007; Hierons et al., 2008, 2012b; Jeannet et al., 2007; Lestiennes e Gaudel, 2002; Li et al., 2004a, 2003; Hessel et al., 2008; Kervinen e Virolainen, 2005; Li et al., 2004b; Pardo et al., 2010; Rollet e Salva, 2009, 2008; Tretmans, 2008; van Beek e Mauw, 2004; van der Bijl et al., 2005, 2004; Weiglhofer e Wotawa, 2009; Simao e Petrenko, 2014; Noroozi et al., 2013; Carvalho et al., 2014; Bhateja, 2014; Helovuo e Leppanen, 2001; Fu e Koné, 2015; Bannour et al., 2013; Jéron, 2009; Pickin et al., 2007; Damasceno et al., 2015; De León et al., 2013)	37
Assíncrona	(Bhateja et al., 2006; Bhateja, 2009; Bi et al., 1998; Hierons, 2012a; Hierons et al., 2012b; Huo e Petrenko, 2005, 2004; Petrenko et al., 2003; Simao e Petrenko, 2011; Weiglhofer e Wotawa, 2009; Noroozi et al., 2013)	11
Distribuída	(Bi et al., 1998; Cavalcanti et al., 2011; Hierons, 2008, 2012b; Hierons et al., 2008, 2012b, 2011, 2012a; Hierons e Núñez, 2012, 2010; Li et al., 2004a; Simao e Petrenko, 2011; Hierons et al., 2014; Hierons, 2015)	14

Com relação à comunicação dos testes, a interação entre o testador e o caso de teste é síncrona no teste *ioco*. No entanto, na prática muitas interações são baseadas em comunicação assíncrona ou na troca de mensagens usando *buffers* e podem ser modeladas como filas, tal como em sistemas que interagem através de redes: protocolos de comunicação, sistemas de telecomunicação e serviços *Web* (Simao e Petrenko, 2011; Hierons, 2012a, 2013). Por outro

lado, a composição com filas envolve problemas como a explosão de estados. Alguns estudos têm investigado a derivação de casos de testes diretamente a partir da especificação para evitar a composição de filas (Simao e Petrenko, 2011; Hierons, 2013). Critérios de teste amplamente conhecidos em outras abordagens tem sido utilizados em tais estudos, como alcançar todos os estados e executar todas as transições, de forma semelhante ao que ocorre na geração de testes a partir de MEFs. Estes estudos têm também utilizado outros conceitos do teste a partir de MEFs no teste a partir de IOTS, como condições de suficiência e modelo de defeitos com domínio de defeitos (Hierons, 2012a, 2013; Noroozi et al., 2011). Segundo Hierons (2013), domínios de defeitos não podem ser aplicados para o teste assíncrono com IOTSs porque o problema de utilizar uma relação de implementação no teste assíncrono é indecidível. Isso quer dizer que não é possível decidir se um elemento do domínio de defeitos está em conformidade com a especificação. Porém, o autor aponta domínios específicos com condições sob as quais a relação de implementação pode ser estendida para o teste assíncrono, como a classe *Alternating* IOTS, que tem o comportamento semelhante ao de uma MEF.

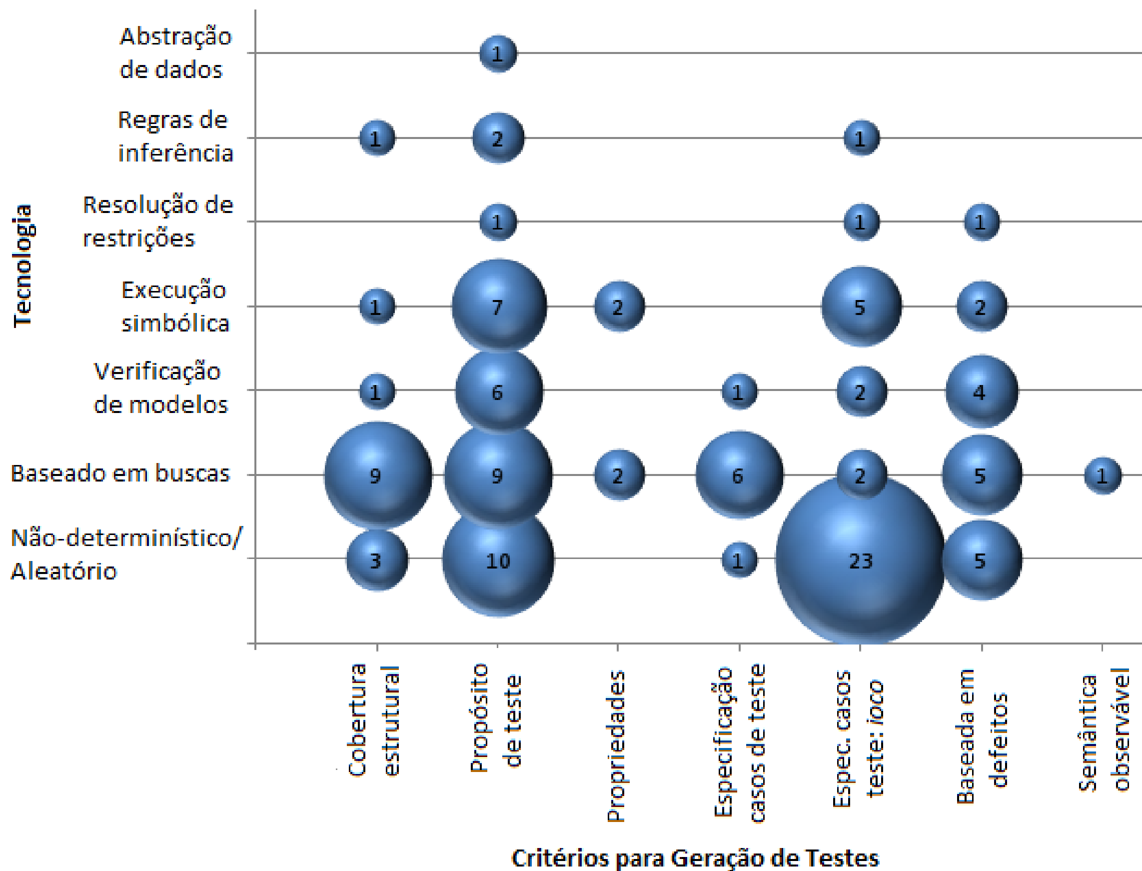
Alguns pressupostos utilizados no teste baseado em MEFs, tais como restrições sobre o modelo da especificação e a geração de certas sequências, poderiam ser importantes para a geração de conjuntos de teste mais eficientes a partir de IOTS, mas eles não são utilizados na maioria dos algoritmos existentes. Alguns estudos recentes, que utilizam critérios de cobertura de estados e transições visam a geração de certas sequências (Huo e Petrenko, 2005, 2004; Simao e Petrenko, 2014), mas somente o método de Simao e Petrenko (2014) visa fornecer o mesmo apoio oferecido pelos métodos de geração a partir de MEFs.

Vários estudos recentes têm se destinado ao teste distribuído, que é aplicado a sistemas que possuem uma série de portas/interfaces fisicamente distribuídas e cada porta contém um testador local. Esse tipo de comunicação pode causar problemas para a geração de casos de teste, uma vez que a teoria existente explora casos de teste determinísticos e o modelo IOTS possui natureza não-determinística. Hierons et al. (2012b) exploram o problema de geração de casos de teste controláveis para a arquitetura de teste distribuída e também utilizam modelos probabilísticos para isso (Hierons e Núñez, 2012, 2010). Porém, é necessário explorar um relógio global para sincronizar os testadores locais.

3.4.5 Mapa da geração de testes a partir de IOTS

A Figura 16 exibe o mapa resultante a partir das questões de pesquisa 2 (critérios de seleção de testes, representado pelo eixo x) em comparação com a questão de pesquisa 3 (tecnologia de geração de testes, representado pelo eixo y), fornecendo uma visão holística deste tópico. Embora a maioria dos estudos apliquem propósitos de teste e especificação de casos de teste com algoritmos não-determinísticos, estas abordagens fornecem garantia probabilística de cobertura após um longo tempo de execução. Por outro lado, os estudos mais recentes tem aplicado critérios baseados em defeitos e algoritmos baseados em buscas que fornecem total

Figura 16 – Mapa dos estudos selecionados



cobertura aplicando algoritmos determinísticos. No entanto, estes estudos são apenas um primeiro passo em direção a algoritmos determinísticos efetivos para o teste a partir de IOTS.

3.5 Considerações Finais

Esta seção relatou os resultados de um mapeamento sistemático que sintetiza informações a respeito de métodos de geração de testes a partir de IOTSs e proporciona informações úteis para guiar investigações neste tópico. A geração de testes a partir de IOTS é considerada uma estratégia promissora em TBM, já que IOTSs são modelos mais expressivos que outros formalismos e são adequados para modelar o processo de teste. A taxonomia de abordagens TBM auxiliou na construção das questões de pesquisa e classificação dos estudos.

A maioria dos métodos encontrados relata extensões da teoria de teste estabelecida por Tretmans (2008), que estabelece um framework para geração automatizada de casos de teste a partir de IOTSs. Vários contextos de aplicação de teste foram identificados, tais como o tipo de sistema (sistemas de tempo-real, sistemas de protocolo de comunicação, aplicações *Web*) e o tipo de teste realizado (teste síncrono, teste assíncrono com a utilização de canais de comunicação ou teste distribuído).

Os resultados indicaram a falta de padrão nas características do modelo IOTS que podem

restringir a adoção de cada método. Critérios de cobertura amplamente utilizados, como a cobertura de estados e transições, tem sido aplicados em IOTSs apenas por poucos estudos recentes. Além disso, a maioria das tecnologias de geração de testes é não-determinística, já que não existe um modelo de defeitos padrão, de forma similar ao que existe nos métodos a partir de MEFs (Hierons et al., 2009). O conceito de hipótese de teste, que é similar ao conceito de modelo de defeitos, tem sido utilizado minimamente no teste a partir de IOTS com a relação *ioco*, considerando apenas que o sistema pode ser modelado utilizando o mesmo formalismo que a especificação. Os resultados dos estudos com IOTS indicam que a completude é garantida na teoria pela repetição do processo um número não limitado de vezes e sem satisfazer um determinado critério de seleção de testes. O teste a partir de MEFs compreende uma teoria bem estabelecida que utiliza modelos de defeitos e algoritmos determinísticos, mas as relações de implementação são mais restritas do que aquelas aplicadas em IOTSs.

Certas subclasses do modelo IOTS foram identificadas. É importante compreender sob quais circunstâncias tais modelos são aplicados, uma vez que possuem diferentes características. Apesar disso, algumas definições acabam tornando-se similares por considerar ou não certas propriedades.

Conclui-se que existe um número considerável de estudos na área de teste formal baseado em IOTSs, e alguns pontos merecem atenção. A etapa de seleção de teste ainda é um desafio. Apesar dos algoritmos existentes, eles são não-determinísticos e não garantem a cobertura de critérios amplamente utilizados em teste. Além disso, ferramentas que apoiem tal atividade são importantes, automatizando a fase de teste e tornando-a menos cara. O próximo capítulo apresenta um método determinístico de geração de testes a partir de IOTSs com a utilização de domínio de defeitos.

CAPÍTULO

4

MÉTODO W PARA MEALY IOTS

A geração de testes a partir de MEFs avançou consideravelmente devido à sua fundação bem estabelecida em hipóteses de teste e modelos de defeitos com garantia de completude dos conjuntos de teste. O modelo de defeitos em MEFs emprega um domínio de defeitos que é coberto pelo conjunto de testes. Estes conceitos têm sido utilizados de maneira mínima no teste baseado em IOTS (Hierons et al., 2009). Em IOTSs, a única hipótese de teste considerada é que o sistema pode ser modelado com o mesmo formalismo da especificação. De fato, o teste baseado em IOTSs não fornece um modelo de defeitos padrão como nos métodos para geração de teste a partir de MEF.

A tecnologia de geração de teste mais utilizada no teste a partir de IOTS é a geração não-determinística (Paiva e Simao, 2015), sendo o algoritmo de Tretmans um dos mais utilizados Jard e Jéron (2005); Tretmans e Brinksma (2003); Tretmans (2008); Belinfante (2010). Entretanto, não se pode garantir que tal algoritmo satisfaz um critério de cobertura e a completude é uma garantia probabilística (irrealista) após um longo tempo de execução. Trabalhos recentes têm desenvolvido métodos de geração de teste determinísticos a partir de IOTSs, garantindo a cobertura de estados e transições em um número finito de passos (Huo e Petrenko, 2009; Hierons, 2012a; Simao e Petrenko, 2014). Há estudos aplicáveis a sistemas baseados em comunicação assíncrona (Huo e Petrenko, 2009; Hierons, 2012a). O problema de construir um conjunto finito de testes para uma dada especificação que é completa em um domínio de defeitos pré-definido para a clássica relação *ioco* mesmo na presença de conflitos de entrada/saída foi investigado por (Simao e Petrenko, 2014), porém, esta abordagem trata de um problema diferente do que o que será aqui proposto. Outra abordagem para definição de propriedades ou funcionalidades a serem testadas é o uso de propósitos de teste (Aichernig e Delgado, 2006; Simao e Petrenko, 2011; Weiglhofer et al., 2009), mas a cobertura de um dado critério de teste é garantido apenas manualmente pelo testador.

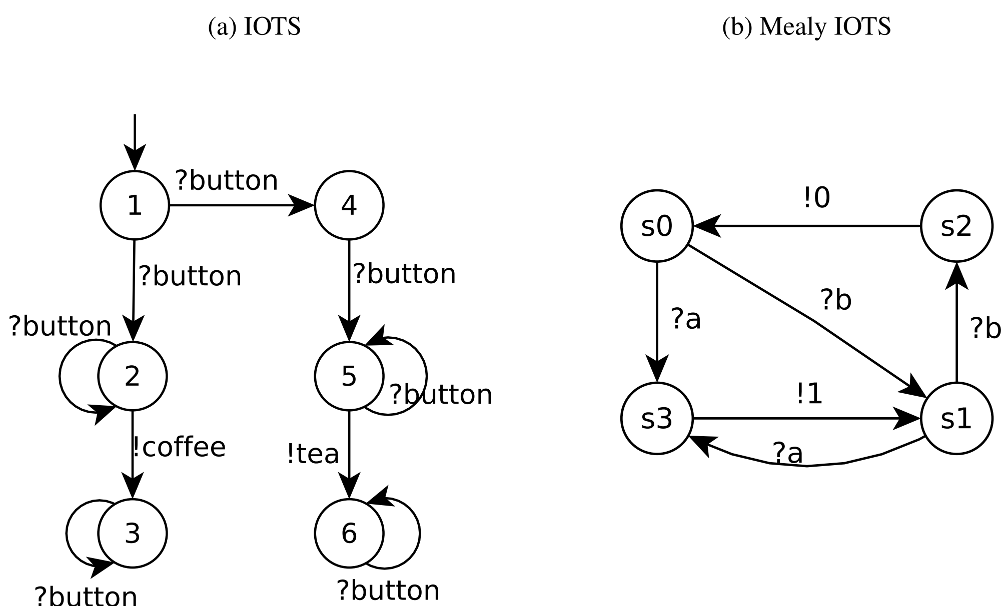
Apesar de existirem trabalhos que buscam solucionar os problemas do teste para IOTSs a

partir de soluções propostas para o teste a partir de MEFs (Hierons, 2012a, 2013; Huo e Petrenko, 2005; Simao e Petrenko, 2014), ainda não há um método que aplique sistematicamente tais conceitos para o processo de geração de casos de teste a partir de IOTSs. Consequentemente, a completude dos conjuntos de testes gerados pelos métodos para IOTSs é garantida apenas em teoria e a reexecução dos testes é requerida um número não limitado de vezes, já que a exaustividade da execução do teste é que garante a completude. Porém, a execução aleatória não garante que determinada transição que é difícil de alcançar será coberta em tempo finito.

Este capítulo descreve como derivar casos de teste a partir de IOTSs com a aplicação de um modelo de defeitos. Como Hierons (2013) mostrou que não é possível aplicar domínios de defeitos para a classe geral de IOTSs, foi considerada uma classe que é mais genérica do que a exemplificada por esse autor. A classe de IOTSs considerada, denominada Mealy IOTS por Simao e Petrenko (2011), comporta-se similarmente a uma máquina de Mealy determinística e recebe entradas apenas em estados estáveis, isto é, não existem estados mistos com entradas e saídas e nem ações internas. A Figura 17b mostra um exemplo de Mealy IOTS, em contraste com o modelo IOTS geral na Figura 17a (adaptado de (van der Bijl e Peureux, 2005)). Vários resultados da teoria de teste a partir de IOTS e MEF, tais como o uso de domínio de defeitos, convergem para esta classe de IOTS (Simao e Petrenko, 2011). Essa classe requer que a quietude seja alcançada antes das entradas serem fornecidas; portanto, a distorção causada pela comunicação assíncrona pode ser ignorada. Em especial, será mostrado que a geração de testes completos é viável para Mealy IOTS através do método de geração de teste aqui proposto, que utiliza conceitos do teste baseado em MEF com cobertura de defeitos de um domínio de defeitos em tempo finito.

Figura 17 – Modelos utilizados em TBM

(a) IOTS (b) Mealy IOTS



O método introduzido, denominado W_{IOTS}, cobre defeitos em um domínio de defeitos

que contém todos as implementações de IOTSs com tantos estados quanto a especificação. Este é um primeiro passo para a definição de métodos de geração de testes com domínio de defeitos similares à teoria de teste baseado em MEFs. A noção de completude em MEFs (Simao e Petrenko, 2010a), geralmente denominada de *n*-completude, foi reformulada para o modelo IOTS. A partir deste conceito, o método W desenvolvido para MEFs (Chow, 1978) foi adotado para gerar conjuntos de teste *n*-completos a partir de IOTSs em um número limitado de passos, ou seja, de forma determinística. O algoritmo para geração de testes a partir de MEFs foi adaptado para o contexto de Mealy IOTSs. A ideia básica é gerar o conjunto de cobertura de transições e o conjunto de caracterização e concatena-los.

O exemplo e o estudo de caso realizados mostram a viabilidade do método proposto, uma vez que o conjunto de testes gerado é finito e com garantia de cobertura de defeitos. O método proposto foi comparado com o método clássico (Tretmans, 2008) através da especificação do Protocolo de Controle de Sessão (do inglês, *Session Control Protocol - SCP*). Os resultados sugerem que o método proposto tem maior qualidade e pode gerar conjuntos de teste muito menores do que o método clássico.

Este capítulo está organizado da seguinte forma: a Seção 4.1 apresenta os conceitos utilizados pelo método proposto; a Seção 4.2 apresenta o algoritmo para geração de casos de teste a partir de IOTSs; a Seção 4.3 apresenta os aspectos de implementação do algoritmo proposto; e por fim, a Seção 4.4 mostra uma comparação entre o conjunto de teste gerado pelo método proposto e pelo método clássico para IOTSs (Tretmans, 2008).

4.1 Definições

Um IOTS M é uma 5-tupla $\langle S, I, O, h, s_0 \rangle$, como definido na Seção 2.5.1, diferindo apenas na relação de transição $h \subseteq S \times (I \cup O \cup \{\delta\}) \times S$, com o símbolo $\delta \notin (I \cup O)$ denotando quietude. A Figura 17a apresenta um exemplo de IOTS, onde *I={?button}*, *O= {!coffee, !tea}* e *1* é o estado inicial.

Para o IOTS M, um *caminho* a partir do estado s_1 para o estado s_{n+1} é uma sequência de transições (s_1, a_1, s_2) $(s_2, a_2, s_3)...(s_n, a_n,\ s_{n+1})$, onde $(s_i, a_i, s_{i+1}) \in h$ para $i = 1, ..., n$. Um caminho *atravessa* uma transição se a transição está no caminho. Um *trace* definido é uma sequência de ações $u \in (I \cup O \cup \{\delta\})^*$ de um IOTS M a partir do estado $s_1 \in S$ se existe um caminho $(s_1, a_1, s_2)(s_2, a_2, s_3)...\ (s_n, a_n, s_{n+1})$, tal que $u = (a_1, ..., a_n)$. O conjunto de todos os *traces* definidos para o estado s é denotado por *tr(s)* e $tr(M)$ é uma forma abreviada para $tr(s_0)$, isto é, para os *traces* definidos a partir do estado inicial de M e para o próprio M. É utilizado *tr(T)* para denotar o conjunto de *traces* a partir dos estados de $T \subseteq S$. Portanto, $tr(M) = tr(s_0)$ e $tr(s) = tr(\{s\})$. A sequência *?button ?button !tea* é um *trace* definido para o IOTS da Figura 17a.

Seja ε o *trace* vazio. T-**after**-U denota o conjunto de estados alcançados após a execução dos *traces* em U a partir de $T \subseteq S$. Na Figura 17a, temos que 1-**after**-$?button = \{2, 4\}$. O

conjunto *init(s)* denota o conjunto de ações habilitadas no estado s, isto é, $init(s) = \{a \in (I \cup O) | \exists s' \in S, (s,a,s') \in h\}$. Sejam *inp(s)* e *out(s)* o conjunto de entradas e saídas, respectivamente, habilidados no estado s.

Um IOTS M é *observável* se h é uma função, isto é, se $(s,x,s') \in h$ e $(s,x,s'') \in h$, então $s' = s''$. Um IOTS é observável se $|M\text{-}\mathbf{after}\text{-}\alpha| = 1$, para cada $\alpha \in tr_M$, isto é, apenas um único estado pode ser alcançado depois de um *trace* definido. M é *saída-determinístico* (Hierons, 2012a) se $|out(s)| \leq 1$ para cada estado s, significando que no máximo uma saída é habilitada em cada estado. A Figura 17b tem os estados *s2* e *s3* com apenas uma saída habilitada.

Um IOTS M é *não-oscilante* se não existe um ciclo rotulado apenas com saídas, isto é, não há um *trace* com uma subsequência infinita apenas de saídas. Assume-se que existe uma operação *reset r* confiável e que indica o recomeço de um IOTS, ou seja, a implementação é movida para o seu estado inicial.

Um estado $s \in S$ é estável (quiescente) se nenhuma saída está habilitada em s. S_{stable} denota o conjunto de todos os estados estáveis do IOTS M. Na Figura 17b, os estados estáveis são *s0* e *s1*. Um IOTS é *completamente especificado* (ou *completo*) se todas as entradas estão habilitadas nos estados estáveis, isto é, $inp(s) \neq \varnothing$ implica que $inp(s) = I$, para cada estado s. UM IOTS é *Mealy* se $inp(s) \neq \varnothing \Leftrightarrow out(s) = \varnothing$, ou seja, as entradas estão habilitadas apenas nos estados estáveis (Simao e Petrenko, 2011). A Figura 17b apresenta um exemplo de Mealy IOTS, que mostra completude nos estados estáveis *s0* e *s1*.

Dados os *traces* $\alpha, \beta, \gamma \in I^*$, se $\beta = \alpha\gamma$, então α é um prefixo de β. $pref(\beta)$ é o conjunto de prefixos de β. Se α não é vazio, então α é um prefixo próprio de β. Para o conjunto de *traces* A, $pref(A)$ é a união de $pref(\beta)$ para todo $\beta \in A$. Se $A = pref(A)$, então dizemos que A é *prefixo-fechado*, isto é, A contém todos os prefixos de todos os *traces* de A.

Um *trace* definido u do IOTS M é um *bridge trace do estado s* se $s\text{-}\mathbf{after}\text{-}u \in S_{stable}$ e para cada prefixo próprio w de u $s\text{-}\mathbf{after}\text{-}w \notin S_{stable}$. Portanto, θ é uma função que retorna o *bridge trace* depois da ocorrência de uma ação de entrada em um estado estável: $\theta : S_{stable} \times I \rightarrow O^* \cdot \{\delta\}$. Dado $s \in S_{stable}$ e $x \in I$, $\theta(s,x) = \gamma$ para $\gamma \in O^*$ se $x\gamma$ é um *bridge trace* de s. Um *bridge trace* do IOTS da Figura 17b é $\theta(s0, ?a) = !1$.

Um IOTS M é *inicialmente conexo* se cada estado é alcançável a partir do estado inicial, isto é, existe um *trace* definido $\alpha \in tr(M)$, chamado *trace* de *transferência* para o estado s, tal que $s_0\text{-}\mathbf{after}\text{-}\alpha = s$.

Um conjunto A para o IOTS M é um *state cover* se, para cada $s \in S_{stable}$, existe $\alpha \in tr(M)$ tal que $s_0\text{-}\mathbf{after}\text{-}\alpha = s$. Um *state cover* minimal é um conjunto que possui um único *trace* de transferência para cada estado estável.

Um conjunto de *traces* é *inicializado* se ele contém o *trace* vazio. Um conjunto de *traces* A é um *transition cover* para o IOTS M se, para cada transição $(s,x,s') \in h$ de M, existe um *trace* $\alpha \in A$, tal que $M\text{-}\mathbf{after}\text{-}\alpha = s$ e $\alpha x \in A$.

Seja S_1 e S_2 conjuntos. A concatenação de S_1 e S_2, denotada por $S_1 \cdot S_2$, consiste em todos os *traces* $s_1 \cdot s_2$ obtidos após acrescentar s_2 no final de s_1, para todo $s_1 \in S_1$ e $s_2 \in S_2$.

Seja M um *IOTS(I,O)* com n estados estáveis, um *domínio de defeitos* $\Im(M)$ para M é o conjunto de todas as possíveis implementações de M com no máximo n estados estáveis e com o mesmo alfabeto de entrada que M.

As noções de equivalência e distinguibilidade foram adaptadas de Tretmans (2008) e Simao e Petrenko (2014). Dados dois IOTSs A e $B \in$ *IOTS(I,O)*, $A = \langle A, I, O, h_A, a_0 \rangle$ e $B = \langle B, I, O, h_B, b_0 \rangle$, escreve-se A **ioco** B se, para cada *trace* $\alpha \in tr(A)$ temos que $out(A\text{-}\mathbf{after}\text{-}\alpha) \subseteq out(B\text{-}\mathbf{after}\text{-}\alpha)$. Caso contrário, isto é, se não A **ioco** B, escreve-se A **io̸co** B.

Dado um conjunto $C \subseteq tr(s) \cap tr(s')$, os estados s e s' são C-distinguíveis se existe $\gamma \in C$, tal que $tr(s\text{-}\mathbf{after}\text{-}\gamma) \neq tr(s'\text{-}\mathbf{after}\text{-}\gamma)$. Caso contrário, isto é, se $tr(s\text{-}\mathbf{after}\text{-}\gamma) = tr(s'\text{-}\mathbf{after}\text{-}\gamma)$ para todo $\gamma \in C$, os estados s e s' são C-equivalentes. Dois IOTSs com o mesmo alfabeto de entrada são distinguíveis se seus respectivos estados iniciais são distinguíveis. Um IOTS é *minimal* se quaisquer dois estados são distinguíveis. Um *conjunto de caracterização* do IOTS M é um conjunto de *traces* W, tal que todo par de estados são W-distinguíveis. O conjunto $W_s \subseteq W$ é um identificador de estados para o estado s se qualquer outro estado é W_s-distinguível de s (Simao e Petrenko, 2010b).

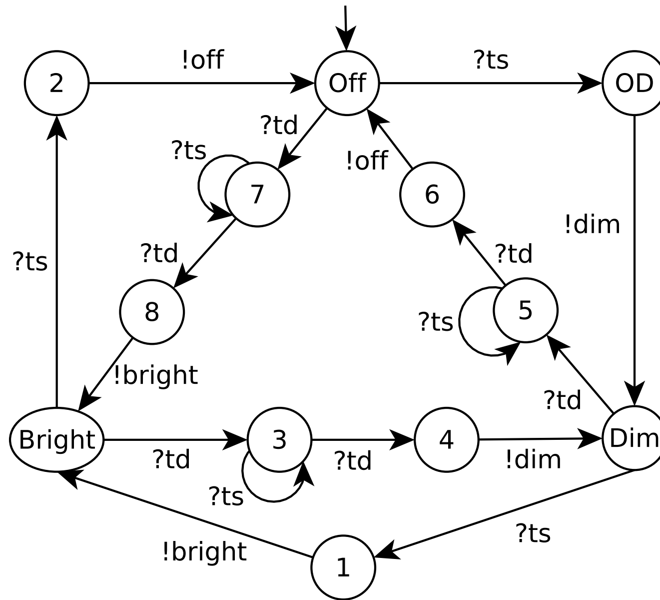
Neste capítulo assume-se que as especificações e implementações são IOTS *não-oscilantes observáveis saída-determinísticos minimais inicialmente conexos* e *completos* e utiliza-se *IOTS(I,O)* para denotar o conjunto de tais Mealy IOTSs com o conjunto de entradas I e conjunto de saídas O.

Exemplo 1: A Figura 18 mostra um *IOTS(I,O)* que modela um dispositivo de luz. Este modelo é uma adaptação do exemplo apresentado em (Bensalem et al., 2008), onde *I={?td,?ts}*, *O= {!dim,!off,!bright}* e *Off* é o estado inicial. O dispositivo acende uma lâmpada com os níveis de intensidade *Dim* ou *Bright*, ou desliga a lâmpada (*Off*). O IOTS comporta-se da seguinte maneira: a entrada *ts* (um toque simples) significa *um nível acima de intensidade*, enquanto a entrada *td* (toque duplo) duas vezes ou mais significa *um nível abaixo de intensidade*. O conjunto de estados estáveis é S_{stable} = *{Off,7,Dim,5,Bright,3}*. O rótulo x em uma aresta (transição) a partir do estado s para o estado s' indica $(s,x,s') \in h$, isto é, quando o IOTS está no estado s e ocorre a ação x, o IOTS move-se para o estado s'. Exemplos de *bridge traces* para o IOTS M_1 são $\theta(Off, ?td) = \delta$ and $\theta(Dim, ?ts) = !bright \cdot \delta$.

4.1.1 Definições sobre Teste

Assume-se que dois IOTSs $M = (S, I, O, h, s_0)$ e $N = (Q, I, O', f, q_0) \in IOTS(I, O)$ representam uma especificação e uma implementação, respectivamente. θ e Δ são funções que retornam um *bridge trace* para M e N, respectivamente. O número de estados estáveis de M é representado por n e $\Im(M)$ é o conjunto de todos os IOTSs pertencentes à *IOTS(I,O)* com o

Figura 18 – Um *IOTS(I,O)* M_1



mesmo alfabeto de entrada de M e com até n estados estáveis representando todos os defeitos que uma implementação de M com no máximo n estados estáveis pode ter. Casos de teste são *traces* definidos sobre a especificação de M que podem detectar defeitos. Quando um conjunto de teste é executado, todos os seus prefixos próprios são executados. Logo, apenas testes maximais devem ser considerados. As definições abaixo formalizam estes conceitos de MEF (Simao e Petrenko, 2010a) para Mealy IOTSs.

Definição 2. Um caso de teste para um IOTS M é um *trace* definido de M. Um conjunto de casos de teste finito e prefixo-fechado de M é um conjunto de teste de M. Um teste $\alpha \in T$ é *maximal* (com respeito a T) se ele não é um prefixo próprio de um outro teste pertencente à T.

Definição 3. Um conjunto de teste T do IOTS M é n-completo se, para cada IOTS $N \in \Im(M)$ distinguível de M, existe um teste pertencente à T que distingue-os.

4.2 Geração de conjuntos de teste completos para Mealy IOTS

Nesta seção será introduzido o método para geração de conjuntos de teste n-completos. Adotou-se o conceito de conjuntos de teste n-completos com respeito a um domínio de defeitos a partir do método W para MEFs (Chow, 1978) para o contexto de Mealy IOTSs. Entretanto, muitas propriedades dos testes são requeridas pelo método proposto para a geração de conjuntos de teste completos. A existência de um conjunto *transition cover* inicializado para M deve ser garantido a fim de verificar se todas as transições de uma implementação correspondem à

transições de sua especificação. A ideia principal é gerar *traces* divididos em dois conjuntos, isto é, conjuntos *transition cover* e de caracterização e concatena-los como ocorre no método W para MEFs (Chow, 1978). Este método consiste em dois passos principais (Chow, 1978):

1. Geração de *traces* de teste para cada transição da especificação;

2. Verificação das respostas para os *traces* de teste executados na implementação.

Primeiramente, casos de teste de um conjunto de teste são gerados através da concatenação do conjunto *transition cover* P e do conjunto de caracterização W. Cada caso de teste, ou *trace* de teste, pertencente ao conjunto de teste é então executado na implementação e a conformidade das respostas da implementação em relação às respostas esperadas na especificação são verificadas através da relação *ioco*.

4.2.1 Condições para conjuntos de teste completos

Foram reformuladas as condições definidas para a teoria de teste a partir de MEFs (Simao e Petrenko, 2010a) que são suficientes para garantir que um dado conjunto de teste é n-completo. Essas condições são semelhantes àquelas requeridas pelo método W para a geração de sequências de teste a partir de MEFs (Chow, 1978). Como o domínio de defeitos aqui definido é semelhante ao proposto por (Chow, 1978), tais condições cobrem os mesmos tipos de defeitos do método W para MEFs (Chow, 1978): defeitos de transferência, defeitos de saída e defeitos de estados ausentes. Primeiramente, será introduzido o conceito de um conjunto de *traces* definidos que alcançam todos os estados estáveis de um IOTS, na qual é semelhante ao conceito de conjuntos confirmados utilizado no teste a partir de MEFs (Simao e Petrenko, 2010a). Utilizou-se N como um elemento arbitrário de $\Im(M)$. Um dado domínio de defeitos $\Im_T(M)$ para um dado conjunto de teste T é o conjunto de todo $N \in \Im(M)$ tal que N e M são T-equivalentes.

Definição 4. Dado um conjunto de teste T do IOTS M, e $K \subseteq T$. O conjunto K cobre todos os estados estáveis se s_0-**after**-$K = S_{stable}$ e, para cada $N \in \Im_T(M)$, segue que para qualquer $\alpha, \beta \in K$, q_0-**after**-$\alpha = q_0$-**after**-β se e somente se s_0-**after**-$\alpha = s_0$-**after**-β. Um *trace* definido cobre um estado estável se existe um conjunto de *traces* definidos que o contém.

De acordo com a Definição 3, um conjunto de *traces* definidos garante a cobertura de todos os estados estáveis de M por meio dos *traces* de transferência. Deste modo, para todo IOTS que tem o mesmo número de estados de M qualquer *trace* conduz ao mesmo estado. Portanto, um conjunto de teste T converge em qualquer IOTS que se comporta de maneira semelhante ao IOTS M quando dois *traces* de um conjunto confirmado de um conjunto de teste T são executados na implementação. Os métodos para construção de conjuntos completos para MEFs têm explorado essa propriedade-chave de uma maneira ou de outra (Dorofeeva et al., 2010; Fujiwara et al., 1991; Simao e Petrenko, 2010b,a).

Exemplo 2: Para o IOTS da Figura 18, temos o conjunto $K = \{\varepsilon$, *?td, ?td ?td !bright, ?td ?td !bright ?td, ?ts !dim, ?ts !dim ?td* $\}$, na qual cada *trace* pertencente ao conjunto K alcança um respectivo estado estável *Off, 7, Bright, 3, Dim* e *5*.

O lema a seguir destina-se a construção de tal conjunto que contém *traces* definidos cobrindo todos os estados estáveis. No Lema 1, uma condição de suficiência é fornecida por um conjunto *state cover* que é minimal (um conjunto que tem um único *trace* de transferência para cada estado estável) para formar tal conjunto. Dado um conjunto de teste T do IOTS M, dois *traces* $\alpha, \beta \in T$ são T-distinguíveis se existe $\alpha\gamma; \beta\gamma \in T$, tal que $\theta(s_0\textbf{-after-}\alpha, \gamma) \neq \theta(s_0\textbf{-after-}\beta, \gamma)$.

Lema 1. Dado um conjunto de teste T para o IOTS M e K um conjunto *state cover* minimal. Se cada dois *traces* de K são T-distinguíveis, então K cobre todos os estados estáveis pertencentes à S_{stable}.

Demonstração. Seja $N \in \Im_T(M)$. Como o conjunto K é um conjunto *state cover* minimal, ele contém exatamente n *traces* de transferência para todo $s \in S_{stable}$ de M. Então, existe apenas um *trace* de K que conduz M à s. Para qualquer $\alpha, \beta \in K$, temos $s_0\textbf{-after-}\alpha \neq s_0\textbf{-after-}\beta$ e $q_0\textbf{-after-}\alpha \neq q_0\textbf{-after-}\beta$. Portanto, como M não tem mais estados estáveis que M, temos $|q_0\textbf{-after-}K| = n$. Assim, K cobre todos os estados estáveis. □

Exemplo 3: O conjunto K dado no Exemplo 2 é um conjunto *state cover* minimal para o IOTS M_1 porque contém apenas uma sequência que alcança cada um dos estados estáveis.

Um domínio de defeitos $\Im(M)$ contém todas as implementações com no máximo n estados estáveis e os alfabetos de entrada e de saída iguais aos da especificação. Cada caso de teste deve conduzir a especificação e cada implementação do domínio de defeitos para o mesmo estado, isto é, a especificação e a implementação devem produzir o mesmo trace. Para eliminar todos os IOTSs defeituosos do domínio de defeitos, basta que todos os *traces* da especificação estejam no conjunto de teste, de modo que apenas a implementação que possui um comportamento similar ao da especificação não deve ser distinguido. Assim, o conjunto de teste deve cobrir todas as transições da especificação. Um conjunto K de *traces* definidos que atravessam todos os estados estáveis cobre uma transição se o conjunto inclui um *trace* de transferência a partir do estado inicial até tal estado estável e esse *trace* é estendido com uma entrada que rotula a transição em questão. O Teorema 1 formaliza este conceito. A prova segue a ideia do teorema correspondente para MEFs (Simao e Petrenko, 2010b).

Teorema 1. Seja T um conjunto de teste de um Mealy IOTS(I,O) M com n estados estáveis. T é n-completo em relação à M se existe um conjunto *state cover* para os estados estáveis $K \subseteq T$ tal que possui as seguintes propriedades:

1. $\varepsilon \in K$.

2. Para cada $s \in S_{stable}$ e $x \in I$, existe $\alpha, \alpha x \in K$ e $s' \in S$, tal que s_0-**after**-$\alpha = s$ e $(s,x,s') \in h$.

Demonstração. Suponha que existe um conjunto *state cover* K para os estados estáveis do IOTS M e T é n-completo. Pela Definição 2, existe um IOTS $N \in \Im(M)$, tal que $N \in \Im_T(M)$ e $tr(M) = tr(N)$. Como M é inicialmente conexo, para cada $s \in S_{stable}$, existe $\alpha \in K$, tal que s_0-**after**-$\alpha = s$. Para cada $\beta \in K$, se s_0-**after**-$\beta = s_0$-**after**-α, então q_0-**after**-$\beta = q_0$-**after**-α. Portanto, $|Q_{stable}| = n$ e existe uma função bijetora $f : S_{stable} \longrightarrow Q_{stable}$, tal que para cada $\alpha \in K$, $f(s_0$-**after**-$\alpha) = q_0$-**after**-α. Como $\varepsilon \in K$, $f(s_0) = q_0$.

Primeiro será mostrado, por indução no comprimento de u, que para cada $u \in tr(M)$, $f(s_0$-**after**-$u) = q_0$-**after**-u e $\theta(s,x) = \Delta(f(s_0),x)$, para cada $(s,x,s') \in h$. No caso base, se $u = \varepsilon$, então $u \in K$ e, por definição, $f(s_0$-**after**-$u) = q_0$-**after**-u. Como K inclui este trace, o resultado é mantido. Seja $u = \varphi x$ e assume-se que $f(s_0$-**after**-$\varphi) = q_0$-**after**-φ. Existe $\alpha \in K$, tal que s_0-**after**-$\alpha = s_0$-**after**-φ e $\alpha x \in K$. Portanto, $f(s_0$-**after**-$\alpha x) = q_0$-**after**-αx e q_0-**after**-$\alpha = f(s_0$-**after**-$\alpha) = f(s_0$-**after**-$\varphi) = q_0$-**after**-φ. Segue que $f(s_0$-**after**-$\varphi x) = f((s_0$-**after**-$\varphi)$-**after**-$x) = f((s_0$-**after**-$\alpha)$-**after**-$x) = f(s_0$-**after**-$\alpha x)$ e $f(s_0$-**after**-$\alpha x) = q_0$-**after**-αx. Assim, q_0-**after**-$\alpha x = (q_0$-**after**-$\alpha)$-**after**-$x = (q_0$-**after**-$\varphi)$-**after**-$x = q_0$-**after**-φx. Portanto, $f(s_0$-**after**-$\varphi x) = q_0$-**after**-φx e, por indução, para qualquer $u \in tr(M)$, $f(s_0$-**after**-$u) = q_0$-**after**-u.

Para cada $x\gamma \in tr(M)$, existe $\alpha x \in T$, s_0-**after**-$\alpha = s$, $\alpha \in K$. Portanto, $\theta(s_0$-**after**-$\alpha, x) = \Delta(q_0$-**after**-$\alpha, x)$. Como $\alpha \in K$, temos q_0-**after**-$\alpha = f(s)$ e, como N é T-equivalente à M, segue que $\theta(s,x) = \Delta(f(s),x)$.

Como $tr(N) = tr(M)$, para cada *trace* definido $ux \in tr(M)$, tal que $\theta(s_0,u) = \Delta(q_0,u)$ e $\theta(s_0,ux) = \Delta(q_0,ux)$, e existe $\alpha \in K$, tal que s_0-**after**-$\alpha = s_0$-**after**-u, e $\alpha x \in K$, tal que $\theta(s_0$-**after**-$\alpha, x) = \Delta(f(s_0$-**after**-$\alpha), x)$. A partir de s_0-**after**-$\alpha = s_0$-**after**-u, segue que $\theta(s_0$-**after**-$u, x) = \Delta(f(s_0$-**after**-$u), x) = \Delta(q_0$-**after**-$u, x)$; a partir de $\theta(s_0,u) = \Delta(q_0,u)$, segue que $\theta(s_0,ux) = \theta(s_0,u)\theta(s_0$-**after**-$u,x) = \Delta(q_0,u)\Delta(q_0$-**after**-$u,x) = \Delta(q_0,ux)$. Assim, T é n-completo.

□

O conjunto K apresentado no Exemplo 2 é um conjunto de *traces* de transferência para cada estado estável do IOTS M_1. Para obter o conjunto com as características do Teorema 1, é necessário adicionar sequências que cubram cada uma das transições que saem dos estados estáveis. Para isso, um algoritmo para geração de tais *traces* será introduzido na seção a seguir.

4.2.2 Algoritmo para geração de conjuntos de teste n-completos

O primeiro passo para a geração de conjuntos de teste que satisfaçam as condições do Teorema 1 é obter o conjunto *transition cover* P, chamado também de árvore de teste (Chow, 1978). O algoritmo descrito a seguir gera conjuntos de teste n-completos em conformidade com o Teorema 1. O Algoritmo 1 descreve o procedimento para a construção do conjunto *transition*

cover, que é uma adaptação do procedimento já existente em MEFs (Chow, 1978; Zurowska e Dingel, 2010). Dado um IOTS M, a ideia é gerar todos os seus *bridge traces*, uma vez que todas as saídas produzidas durante a execução do teste devem ser observadas. O procedimento começa a partir do estado inicial do IOTS, os nós da árvore de teste são rotulados com os nomes dos estados e as arestas são rotuladas com ações de entradas (quando os nós representam estados estáveis) e de saída (quando os nós representam estados não estáveis) usando busca em largura. O conjunto *Term* inclui todos os estados estáveis já explorados durante o procedimento. O procedimento termina quando todos os estados estáveis tiverem sido visitados e todos os caminhos terminarem em um estado estável, já que um *bridge trace* termina sempre em um estado estável. O resultado é o conjunto de todos os *bridge traces* obtidos na árvore de teste.

Algoritmo 1: Construção do conjunto *transition cover*

```
   Entrada: IOTS minimal M = ⟨S, I, O, h, s0⟩ com nó raíz s0 ∈ S
   Saída: Uma árvore de teste com todos os bridge traces do IOTS M
1  início
2      Rotula a raíz da árvore T com s0
3      k ← 1
4      Term ← ∅
5      enquanto S_stable ≠ Term e existe um nó d ∉ S_stable no nível k faça
6          para cada nó d do nível k da árvore T da esquerda para direita faça
7              1. se d ∈ S_stable e d ∉ Term então
8                  Term ← Term ∪ {d} s ← rótulo de d
9                  para cada i ∈ I faça
10                     Adiciona uma transição de d para um novo nó d_c
11                     Rotula a transição com ?i e d_c com s′ se (s, i, s′) ∈ h
12                 fim
13             fim
14             2. senão se d ∉ S_stable então
15                 Adiciona uma transição de d para um novo nó d_c
16                 Rotula uma transição com !o ∈ O e d_c com s′ se (s, o, s′) ∈ h
17             fim
18         fim
19         k ← k + 1
20     fim
21     P ← rótulos de todos os caminhos de T
22 fim
23 retorna P
```

Exemplo 4: O IOTS M_1 é representado na Figura 18. O estado *Off* é o estado inicial. O algoritmo começa com a raíz da árvore sendo rotulada como *Off*, explorado no primeiro nível. Este estado é adicionado ao conjunto *Term*. As arestas são adicionadas para todas as entradas do alfabeto $\{?td, ?ts\}$ e conectadas aos nós com rótulos *7* e *OD*, uma vez que o estado *Off* é estável. No segundo nível, os nós *7* e *OD* são explorados. O nó *7* é um estado estável e tem duas transições, uma para cada entrada do alfabeto. Depois de adicionar à árvore duas transições

rotuladas com *?td* e *?ts* para os estados *8* e *7*, respectivamente, o estado *7* é adicionado ao conjunto *Term*. O próximo nó a ser explorado é *OD* e uma transição com rótulo *!dim* para o estado *Dim* é adicionada à árvore. No terceiro nível, apenas os nós *8* e *Dim* são explorados, uma vez que o nó *7* é estável e já está no conjunto *Term*. O procedimento termina quando todos os estados estáveis *Off, 7, Dim, 5, Bright, 3* tiverem sido explorados e adicionados ao conjunto *Term* e cada nó folha da árvore for um estado estável. Os *bridge traces* são então tomados da árvore e o conjunto *transition cover* obtido para o IOTS M_1 é

$P = \{\ \varepsilon$

?td ?ts

?td ?td !bright ?ts !off

?td ?td !bright ?td ?ts

?td ?td !bright ?td ?td !dim

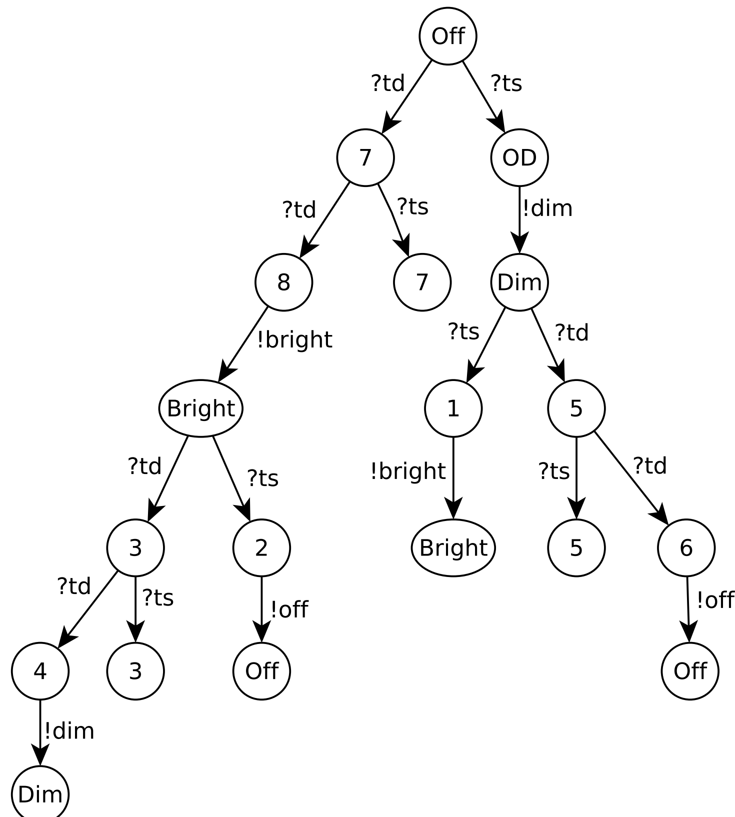
?ts !dim ?ts !bright

?ts !dim ?td ?ts

?ts !dim ?td ?td !off $\}$

A Figura 19 mostra a árvore de teste resultante.

Figura 19 – Árvore de teste para o IOTS M_1



A construção do conjunto de caracterização W é o próximo passo para obtenção do conjunto de teste. O Algoritmo 2 descreve esse procedimento, que é similar à construção do conjunto de caracterização para MEFs (Chow, 1978; Zurowska e Dingel, 2010). Esse procedimento particiona o conjunto de estados estáveis S_{stable} através do *bridge trace* produzido em resposta ao mesmo *trace* de entrada, até que todas as partições possuam apenas um estado. Deve haver uma ação de entrada x que produza diferentes *bridge traces* para cada par de estados, uma vez que a especificação é minimal. A Tabela 5 mostra os estados estáveis do IOTS M_1 da Figura 18 e seus respectivos *bridge traces* após a aplicação de cada ação de entrada. Para cada par de estados, uma tabela pode ser construída, tal que x possa ser encontrado. A saída desse procedimento é um conjunto de entradas que distingue todos os estados estáveis.

Algoritmo 2: Construção do conjunto de caracterização

```
   Entrada: Um IOTS minimal M = ⟨S, I, O, h, s0⟩
   Saída: Conjunto de entradas
 1 início
 2   W ← ∅
 3   B1 ← S_stable
 4   enquanto B_i possuir mais do que um elemento para todo i = 0,...,n faça
 5     se s,t ∈ S_stable para algum j, s ≠ t então
 6       Encontre x ∈ I*, tal que θ(s,x) ≠ θ(t,x)
 7       Particione B_j em B_j1,...,B_jm, tal que r,r' ∈ B_jk ⇔ θ(r,x) = θ(r',x) para todo
         k = 1,...,m
 8       se x ∉ W então
 9         W ← W ∪ {x}
10       fim
11     fim
12   fim
13 fim
14 retorna W
```

Exemplo 5: Em relação ao IOTS M_1 da Figura 18, temos o conjunto $B_1 =$ *{Off, 7, Dim, 5, Bright, 3}*, que é igual ao conjunto S_{stable}. O *trace* de entrada $?td$ distingue os estados *7*, *5* e *3*, porque $\theta(7,?td) =$!bright·δ, $\theta(5,?td) =$!off·δ e $\theta(3,?td) =$!dim·δ, isto é, as saídas produzidas após $?td$ são diferentes, como visto na Tabela 5. Assim, $?td$ é adicionado ao conjunto W. O conjunto B_1 é particionado em $B_{1_1} = \{7\}$, $B_{1_2} = \{5\}$, $B_{1_3} = \{3\}$ e $B_{1_4} = \{Off,Dim,Bright\}$. O conjunto B_{1_4} não possui apenas um elemento, então a próxima iteração explora este conjunto. O *trace* de entrada $?ts$ distingue os estados *{Off,Dim,Bright}*, como visto na Tabela 5, e então $?ts$ é adicionado ao conjunto W. Dessa forma, o conjunto B_{1_4} é particionado em $B_{2_1} = \{Off\}$, $B_{2_2} = \{Dim\}$ e $B_{2_3} = \{Bright\}$. Como todos os conjuntos possuem apenas um elemento, o algoritmo termina retornando $W = \{?td, ?ts\}$, e é possível identificar todos os estados estáveis do IOTS M_1 a partir deste conjunto.

Para finalizar, os conjuntos P e W devem ser concatenados juntamente com a operação

Tabela 5 – Resposta de cada um dos estados para a geração do conjunto W

$\mathbf{S_{stable}}$	**?td**	**?ts**
Off	δ	!dim·δ
7	!bright·δ	δ
Dim	δ	!bright·δ
5	!off·δ	δ
Bright	δ	!off·δ
3	!dim·δ	δ

reset r para a construção do conjunto de teste final. As saídas resultantes após a aplicação da entrada do conjunto W devem ser também concatenadas em cada *trace* de teste, representando assim o *trace* que deve ser realmente observado durante a aplicação dos testes. Este procedimento é formalizado no Algoritmo 3.

Algoritmo 3: Geração de conjuntos de teste n-completos

Entrada: Um IOTS minimal $M = \langle S, I, O, h, s_0 \rangle$ com estado inicial $s_0 \in S$
Saída: Conjunto de casos de teste

```
1  início
2  |  T ← ∅
3  |  Obter o conjunto transition cover P
4  |  Obter o conjunto de caracterização W
5  |  para cada p ∈ P faça
6  |  |  para cada w ∈ W faça
7  |  |  |  se s1-after-(r · p · w) ∉ S_stable então
8  |  |  |  |  t ← {r} · p · w · θ(s, w)
9  |  |  |  fim
10 |  |  |  senão
11 |  |  |  |  t ← {r} · p · w
12 |  |  |  fim
13 |  |  |  T ← T ∪ {t}
14 |  |  fim
15 |  fim
16 fim
17 retorna T
```

Finalmente, será mostrado que o algoritmo proposto pode ser executado em tempo polinomial se os algoritmos que geram o conjunto *transition cover* e o conjunto de caracterização podem ser executados em um número finito de vezes. O algoritmo para geração de conjuntos de teste n-completos contém dois ciclos e estes podem ser executados em um número finito de vezes. No ciclo do algoritmo de geração do conjunto *transition cover*, um nó e no mínimo uma transição para este nó são processadas em cada execução do Passo 1. Como o número de estados do IOTS M é n mais o número de estados não estáveis no pior caso, o Passo 1 pode ser executado no máximo $2(n+k)$ vezes até que um nó que representa um estado estável seja repetido. Após

todos os estados estáveis terem sido visitados e todos os caminhos terminarem em um estado estável, o algoritmo termina.

S_{stable} tem y estados na iteração i do algoritmo de geração do conjunto de caracterização. Como o IOTS processado é minimal, a iteração é executada no máximo $y-1$ vezes (Chow, 1978).

No Algoritmo 3, a iteração w pode ser executada apenas um número finito de passos, uma vez que o conjunto W tem k elementos. Na iteração p, o conjunto P tem um número finito de caminhos. Portanto, este ciclo pode ser executado um número finito de vezes.

O seguinte teorema demonstra que o algoritmo proposto para a geração de casos de teste a partir de IOTSs satistaz as condições do Teorema 1.

Teorema 2. Seja T um conjunto de teste gerado pelo Algoritmo 3 para o IOTS M. Portanto, T satisfaz as condições do Teorema 1.

Demonstração. O algoritmo contém dois ciclos. O Ciclo 1 percorre o conjunto P. P é o conjunto *transition cover* para cada $s \in S_{stable}$ e contém todos os *bridge traces* de M começando no estado inicial s_0, incluindo o *trace* vazio estabelecido na condição 1 do Teorema 1. Para cada $\alpha \in P$, temos s_0-**after**-$\alpha = s$, que está na Condição 2 do Teorema 1. O Ciclo 2 corresponde ao conjunto de caracterização W. Para cada *trace* definido de P, uma entrada de W é concatenada juntamente com seu bridge trace, formando um caso de teste. Portanto, todo *trace* definido de T pode ser distinguido. Para cada dois *traces* $\alpha\gamma, \beta\gamma \in T$, temos que $\theta(s_0\text{-}\mathbf{after}\text{-}\alpha, \gamma) \neq \theta(s_0\text{-}\mathbf{after}\text{-}\beta, \gamma)$, pelo Lema 1. Portanto, como $\varepsilon \in T$ (recordando que T é prefixo-fechado), T satisfaz as condições do Teorema 1. □

Exemplo 6: Considere o IOTS M_1 da Figura 18 e o conjunto P gerado no Exemplo 4 e o conjunto W gerado no Exemplo 5. O conjunto de teste deve ser produzido pela concatenação de P e W e a adição das saídas produzidas após a ação de entrada do conjunto W. O conjunto de teste final tem tamanho 111 e 14 operações reset:

$T = \{\ r\cdot?td\cdot?ts\cdot?ts\cdot\delta$

$r\cdot?td\cdot?ts\cdot?td\cdot!bright\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?td\cdot?ts\cdot?ts\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?td\cdot?ts\cdot?td\cdot!dim\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?td\cdot?td\cdot!dim\cdot?ts\cdot!bright\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?td\cdot?td\cdot!dim\cdot?td\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?ts\cdot!off\cdot?ts\cdot!dim\cdot\delta$

$r\cdot?td\cdot?td\cdot!bright\cdot?ts\cdot!off\cdot?td\cdot\delta$

$r\cdot?ts\cdot!dim\cdot?ts\cdot!bright\cdot?ts\cdot!off\cdot\delta$

$r \cdot ?ts \cdot !dim \cdot ?ts \cdot !bright \cdot ?td \cdot \delta$

$r \cdot ?ts \cdot !dim \cdot ?td \cdot ?ts \cdot ?ts \cdot \delta$

$r \cdot ?ts \cdot !dim \cdot ?td \cdot ?ts \cdot ?td \cdot !off \cdot \delta$

$r \cdot ?ts \cdot !dim \cdot ?td \cdot ?td \cdot !off \cdot ?ts \cdot !dim \cdot \delta$

$r \cdot ?ts \cdot !dim \cdot ?td \cdot ?td \cdot !off \cdot ?td \cdot \delta \quad \}$

Note que o Algoritmo 3 pode gerar conjuntos de teste n-completos com cobertura garantida de defeitos. O conjunto *transition cover P* garante que todas as transições sejam cobertas pelo conjunto de teste, como requerido pelo Teorema 1. Além disso, diferentes casos de teste do conjunto de teste, como por exemplo $r \cdot ?ts \cdot !dim \cdot ?td \cdot ?td \cdot !off \cdot ?td \cdot \delta$ e $r \cdot ?ts \cdot !dim \cdot ?td \cdot ?td \cdot !off \cdot ?ts \cdot !dim \cdot \delta$ são T-distinguíveis, uma vez que eles são estendidos por $?td$ e $?ts$, respectivamente, na qual distinguem os estados estáveis. Como o número máximo de estados estáveis do IOTS M_1 (Figura 18) é conhecido e cada dois testes que são T-distinguíveis alcançam apenas um estado do IOTS M_1, eles devem convergir no mesmo estado de qualquer IOTS do domínio de defeitos que esteja em conformidade com o IOTS M_1. Portanto, de acordo com o Teorema 1, T é n-completo. Da mesma forma, com relação a teoria *ioco*, T é completo com relação à exaustividade já que o conjunto de teste gerado cobre exaustivamente todos os tipos de defeitos do domínio de defeitos definido. Além disso, é também consistente pois apenas as implementações que não são equivalentes a especificação falharão no teste.

4.3 Aspectos de Implementação

O algoritmo 3 do método W_{IOTS} foi implementado como uma prova de conceito para demonstrar a viabilidade do método proposto. Essa implementação feita em Java automatiza dois passos-chave: geração de conjuntos de teste para uma dada implementação e execução de um conjunto de teste em relação à um modelo que simula a implementação. A ferramenta pode ser utilizada via interface de linha de comando e um menu permite escolher uma das duas funcionalidades.

4.3.1 Geração de conjuntos de teste

Para obter um conjunto de teste para determinada especificação através da ferramenta, é necessário ter um arquivo em que a especificação do IOTS está registrado. Deve, então, ser informado o local do arquivo para a ferramenta e este é lido e transformado na devida estrutura de IOTS definida pela ferramenta. O formato empregado para IOTSs é simples e semelhante ao formato já utilizado em outras ferramentas de teste a partir de MEFs. Um exemplo de arquivo com o IOTS da Figura 17b é apresentado na Tabela 6. Cada linha representa uma transição, e em cada linha há quatro *tokens*: o primeiro indica o estado que parte a transição, o segundo representa o tipo de ação (**?** para ações de entrada e **!** para ações de saída), o terceiro indica o

rótulo da ação e o quarto *token* indica o estado para qual a transição está se movendo. O primeiro estado indicado na primeira linha é considerado o estado inicial.

Tabela 6 – Conteúdo de arquivo texto com especificação de IOTS

```
s0 ? a s3
s0 ? b s1
s1 ? a s3
s1 ? b s2
s2 ! 0 s0
s3 ! 1 s1
```

A partir da especificação estruturada como um objeto IOTS, a ferramenta identifica quais são os estados estáveis. Também verifica se a especificação contém os pressupostos exigidos para a aplicação do método, a saber: estados estáveis completamente especificados, IOTS determinístico, IOTS inicialmente conexo (todos os estados são alcançáveis a partir do estado inicial), IOTS progressivo e IOTS minimal nos estados estáveis. Considerando que a especificação contém os pressupostos requisitados, pode-se partir para a geração do conjunto de testes. O processo de geração consiste em quatro passos:

1. Criação do conjunto *transition cover*: é gerada a árvore de teste para a especificação.

2. Criação do conjunto de caracterização: obtém-se o conjunto de entradas que identificam cada um dos estados estáveis.

3. Concatenação dos conjuntos *transition cover* e de caracterização: são gerados os casos de teste através da concatenação dos dois conjuntos e adição do *bridge trace* no final de cada caso de teste.

4. Registro do conjunto de testes em um arquivo: o conjunto de teste é gravado em um arquivo, na qual cada linha representa um caso de teste, conforme exemplo na Tabela 7 para o IOTS da Tabela 6.

Tabela 7 – Conteúdo de arquivo texto com o conjunto de teste gerado pela ferramenta

```
r ?a !1 ?b !0 DELTA
r ?b ?a !1 ?b !0 DELTA
r ?b ?b !0 ?b DELTA
```

4.3.2 Execução de um conjunto de teste

Para executar um conjunto de teste em relação à uma simulação do modelo da implementação é necessário informar o local do arquivo do modelo da implementação e do arquivo que contém o conjunto de teste. O modelo é lido e transformado na devida estrutura de IOTS da

ferramenta e o conjunto de teste também é lido a partir do arquivo. Antes de executar o teste, a ferramenta escolhe aleatoriamente a ordem de execução dos casos de teste. Desse modo, é possível fazer várias execuções com uma ordem diferente dos casos de teste.

A execução dos testes é realizada de forma paralela. Os casos de teste são executados na implementação com a ordem invertida de entradas e saídas. Uma vez que se inicia a execução do teste, para cada caso de teste, a ferramenta irá caminhar em paralelo na implementação e no caso de teste, de modo que, se a execução atingir o final do caso de teste, a implementação passa no caso de teste. Caso contrário, se houver alguma não-conformidade antes de atingir o final, a implementação falha. Se a implementação passar em todos os casos de teste do conjunto de teste, então a implementação passou no teste e a ferramenta exibe o veredicto. Caso contrário, se a implementação não passar em todos os testes, a ferramenta exibe que o teste falhou e indica quantos passos (ações) foram executados até que o teste falhasse.

4.4 Estudo de caso: Protocolo de Controle de Sessão

Esta seção ilustra a aplicação do Algoritmo 3 para geração de conjuntos de teste *n*-completos. A especificação considerada é a do Protocolo de Controle de Sessão (do inglês - *Session Control Protocol* -SCP) (Spero, 2007).

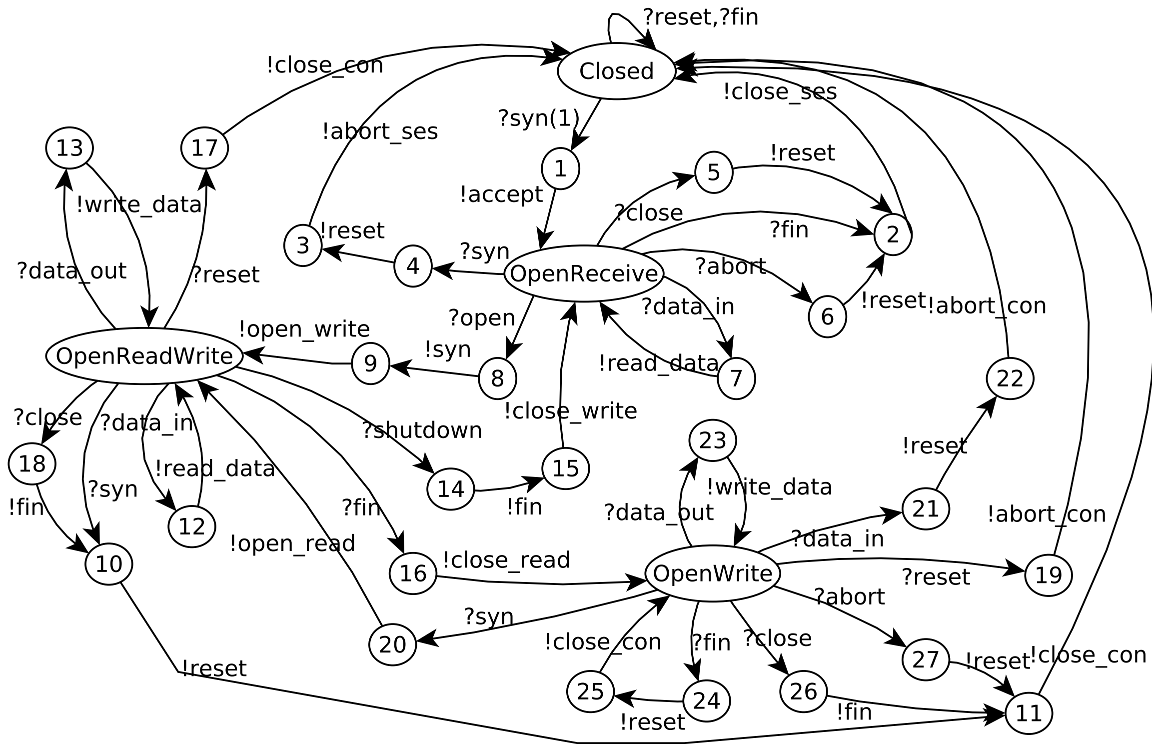
SCP é um protocolo simples que é executado no topo do TCP e pode ser utilizado para a criação de múltiplas conexões leves sobre uma única conexão TCP. O cliente é a parte que inicia a conexão e o servidor é a parte que a aceita. Sessões unidirecionais e bidirecionais são permitidas e sessões bidirecionais podem ser fechadas separadamente. Mensagens podem ser enviadas após uma sessão ter sido aberta. A aplicação pode sinalizar o final de uma sessão ou indicar o lançamento abortivo pela operação *reset*.

Em resposta ao recebimento de um pacote com uma ação, a sessão pode mudar seu *status*. Ações produzidas em resposta à um dado evento são dadas em ordem de prioridade. A especificação do SCP (Spero, 2007) fornece uma tabela que descreve as ações de resposta para cada evento especificado. A partir desta tabela, foi gerado um Mealy $IOTS(I,O)$ que representa o comportamento do SCP. Os eventos foram modelados como entradas e ações como saídas do IOTS. O modelo resultante pode ser visto na Figura 20 e contém 31 estados, dos quais 4 são estáveis: *closed, openReceive, openReadWrite* and *openWrite*. O restante dos estados dispara ações de saída.

O modelo obtido foi utilizado para derivar o conjunto de testes através do Algoritmo 3. A geração ocorre concatenando-se o conjunto *transition cover* P e o conjunto de caracterização W. O conjunto P resultou em 21 caminhos, cobrindo todas as transições da especificação vista da Figura 20:

$P = \{ \varepsilon$

Figura 20 – SCP representado como um IOTS



?reset

?fin

?syn !accept ?fin !close-s

?syn !accept ?reset !abort-s

?syn !accept ?syn !reset !abort-s

?syn !accept ?close !reset !close-s

?syn !accept ?abort !reset !close-s

?syn !accept ?data-in !read-data

?syn !accept ?open !syn !open-write ?syn !reset !close-con

?syn !accept ?open !syn !open-write ?data-in !read-data

?syn !accept ?open !syn !open-write ?data-out !write-data

?syn !accept ?open !syn !open-write ?shutdown !fin !close-write

?syn !accept ?open !syn !open-write ?fin !close-read ?reset !abort-con

?syn !accept ?open !syn !open-write ?fin !close-read ?syn !open-read

?syn !accept ?open !syn !open-write ?fin !close-read ?data-in !reset !abort-con

?syn !accept ?open !syn !open-write ?fin !close-read ?data-out !write-data

?syn !accept ?open !syn !open-write ?fin !close-read ?fin !reset !close-con

?syn !accept ?open !syn !open-write ?fin !close-read ?close !fin !close-con

?syn !accept ?open !syn !open-write ?fin !close-read ?abort !reset !close-con

?syn !accept ?open !syn !open-write ?reset !abort-con

?syn !accept ?open !syn !open-write ?close !fin !reset !close-con }

O conjunto W gerado pelo algoritmo foi $\{?syn\}$, sendo essa única entrada suficiente para distinguir cada um dos estados estáveis da especificação na Figura 20. A partir destes conjuntos, foi gerado o conjunto de teste final, com 21 *traces* de teste, 21 operações reset r e com comprimento 230:

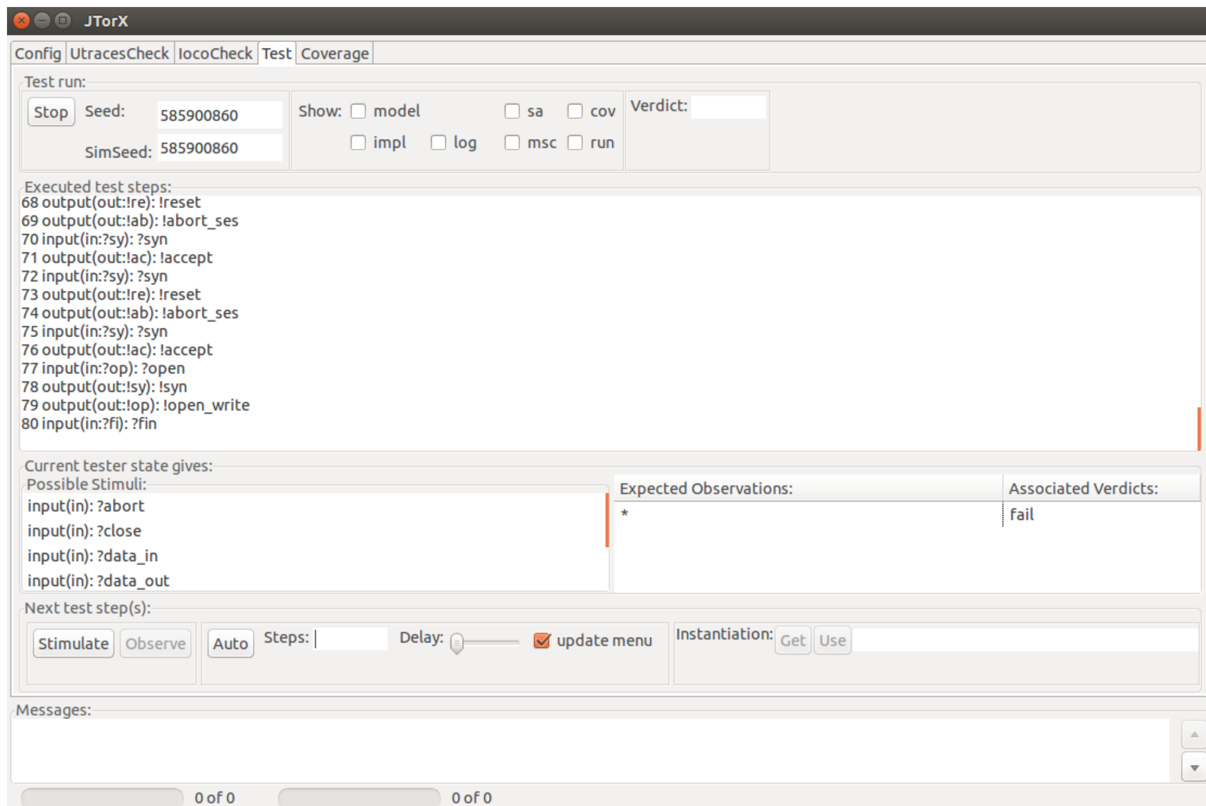
$T = \{$ *r ?reset ?syn !accept* δ
r ?fin ?syn !accept δ
r ?syn !accept ?fin !close-s ?syn !accept δ
r ?syn !accept ?reset !abort-s ?syn !accept δ
r ?syn !accept ?syn !reset !abort-s ?syn !accept δ
r ?syn !accept ?close !reset !close-s ?syn !accept δ
r ?syn !accept ?abort !reset !close-s ?syn !accept δ
r ?syn !accept ?data-in !read-data ?syn !reset !abort-s δ
r ?syn !accept ?open !syn !open-write ?syn !reset !close-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?data-in !close-read ?syn !open-read δ
r ?syn !accept ?open !syn !open-write ?data-out !write-data ?syn !reset !close-con δ
r ?syn !accept ?open !syn !open-write ?shutdown !fin !close-write ?syn !reset !abort-s δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?reset !abort-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?syn !open-read ?syn !reset !close-con δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?data-in !reset !abort-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?data-out !write-data ?syn !open-read δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?fin !reset !close-con ?syn !open-read δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?close !fin !close-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?fin !close-read ?abort !reset !close-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?reset !abort-con ?syn !accept δ
r ?syn !accept ?open !syn !open-write ?close !fin !reset !close-con ?syn !accept δ $\}$

Através desse exemplo, a viabilidade da abordagem proposta é ilustrada em um caso de tamanho real e o conjunto de teste gerado é finito e garante a cobertura de transições. Essa abordagem habilita a detecção de todas as implementações do domínio de defeitos que não estão em conformidade (defeituosas) com a especificação.

Foi realizada uma comparação da abordagem proposta com o método implementado na ferramenta JTorX (Belinfante, 2010), exibida na Figura 21. Esta ferramenta é uma melhoria da

ferramenta TorX (Tretmans e Brinksma, 2003), na qual implementa a teoria mais utilizada e bem-estabelecida para IOTSs em cenários acadêmicos e industriais. A JTorX utiliza a teoria *ioco* (Tretmans, 2008), na qual casos de teste são gerados de forma não-determinística. O principal objetivo dessa comparação é demonstrar as vantagens de um método determinístico (repetível, com garantia de cobertura e realizada em um número finito de passos) em contraste com a JTorX, que é um método não-determinístico e *online*.

Figura 21 – Tela de geração de testes da ferramenta JTorX



O modelo SCP foi representado no formato GRAPHML e o mesmo modelo foi utilizado como especificação e como implementação para gerar 10 conjuntos de teste na JTorX. Começando a partir de um conjunto de teste vazio, foi executado iterativamente 100 passos do algoritmo da JTorX e verificado manualmente se a sequência gerada cobria todas as transições. Esse processo foi repetido até que o conjunto de teste tivesse coberto todas as transições do modelo. O número total de passos é o tamanho do conjunto de teste. Por exemplo, a primeira execução tem tamanho 7300, o que significa que foi executado iterativamente 73 vezes (100 passos para cada execução) até que todas as transições tivessem sido cobertas.

A Tabela 8 mostra os resultados do comprimento do conjunto de testes para 10 execuções na JTorX. No final da tabela estão os resultados para o conjunto de teste gerado pelo Algoritmo 3. Os resultados apontam que o Algoritmo 3 é muito melhor que a média de comprimento dos 10 conjuntos de teste gerados pela JTorX. A média de passos para a cobertura de todas as transições nas 10 execuções na JTorX foi 5810, que é 25 vezes maior do que o comprimento do conjunto de

Método	#	*Comprimento do conjunto de teste/passos*
JTorX	1	7300
	2	7300
	3	8500
	4	4800
	5	4200
	6	5400
	7	5200
	8	5700
	9	6400
	10	3300
	Média	**5810**
Algoritmo 3	1	**230**

Tabela 8 – Comparação de resultados entre a JTorX e o Algoritmo 3

teste gerado pelo método proposto. Esses resultados indicam que o método clássico de (Tretmans, 2008) cobre todas as transições após um grande número de passos. Entretanto, não é possível afirmar em que momento da execução todas as transições serão cobertas. O método proposto, que estende o método W para MEFs, reduz significativamente o comprimento dos conjuntos de teste e ainda garante a cobertura de transições em um tempo finito e a completude dos conjuntos gerados em relação a um domínio de defeitos.

4.5 Considerações Finais

Este capítulo tratou do problema de construção de conjuntos de teste n-completos para uma dada especificação e fornece uma abordagem para resolvê-lo. Foi reformulado o conceito de completude de conjuntos de teste a partir de IOTSs para garantir o uso do conceito de domínio de defeitos para Mealy IOTSs. Um domínio de defeitos habilita a detecção de todas as implementações que não estão em conformidade com a especificação (defeituosas) em um número limitado de passos.

As condições aqui propostas são requeridas para a geração de conjuntos de teste com o Algoritmo 3. A ideia era reformular o método W do teste a partir de MEF para IOTS, uma vez que existe apoio para a geração de conjuntos de teste completos nos métodos para MEFs. O algoritmo proposto satisfaz o Teorema 1, já que este método apoia a cobertura de todas as transições da especificação e, simultaneamente, detecta todos os defeitos do domínio de defeitos em tempo finito. O método W para Mealy IOTSs é determinístico e o processo é repetível em tempo finito, ao contrário dos métodos existentes. Entretanto, domínios de defeitos podem ser utilizados apenas sob um número de pressupostos sobre a especificação. Assim, foi utilizado Mealy IOTSs que exigem o alcance da quietude antes de fornecer entradas, de modo que o problema relacionado à comunicação entre testadores e implementação é eliminado.

O exemplo e estudo de caso ilustraram a viabilidade da abordagem, já que o conjunto de teste gerado para o SCP é finito, garante a cobertura de defeitos e é produzido em um número limitado de passos, ao contrário do método clássico. Os conjuntos de teste gerados tem mostrado o compromisso entre a cobertura de defeitos e o tamanho do conjunto de teste, como nos métodos para MEFs aqui estendidos. Este é o primeiro passo em direção a métodos com total cobertura de defeitos garantida a partir de IOTS. Apesar desse exemplo, mais resultados são necessários para avaliar o método aqui proposto. O próximo capítulo apresenta outro estudo de caso com vários modelos reais e resultados experimentais sobre o método aqui proposto.

CAPÍTULO

5

AVALIAÇÃO DO MÉTODO PROPOSTO

A condução de estudos empíricos tem sido um tópico de crescente interesse nos últimos anos (Briand, 2007). Entretanto, existem poucos estudos empíricos na área de teste de software e muitos deles não são realísticos (González et al., 2014). Há a necessidade de estudos empíricos mais detalhados e experiências com sistemas reais no tópico de teste a partir de IOTSs, conforme discutido na Seção 3.3.4.

Para validar a contribuição que o método W_{IOTS} representa foram realizados estudos empíricos. A Seção 5.1 avalia o custo de geração de conjuntos de teste para IOTSs obtidos através de geração aleatória. A Seção 5.2 apresenta um estudo de caso com várias especificações reais comparando os resultados do método W com o método tradicional de Tretmans (2008) para IOTSs. Na Seção 5.3 é apresentada uma comparação dos resultados obtidos com as especificações reais apresentados e de IOTSs gerados aleatoriamente.

5.1 Resultados Experimentais

Apesar de dados experimentais serem úteis para evidenciar características e limitações de métodos de geração de casos de teste, existem poucos estudos na literatura que relatam experimentos e estudos de caso empregando o teste baseado em IOTSs, conforme mostrado na Seção 3.3.4. Existem relatos de experimentos com modelos gerados aleatoriamente na área de teste a partir de MEFs (Simao et al., 2009a; Dorofeeva et al., 2010; Endo e Simao, 2013), como discutido na Seção 2.4, que avaliam o custo de geração dos principais métodos. No entanto, nenhum estudo com esse mesmo objetivo foi realizado para o teste a partir de IOTSs.

Nesse sentido, foi realizado um experimento com conjuntos de teste produzidos pelo método W para Mealy IOTSs gerados aleatoriamente com diferentes configurações. Esse estudo teve o objetivo de *analisar o custo de geração de conjuntos de teste* a partir do método proposto no Capítulo 4. As seguintes questões foram investigadas:

- Como o número de casos de teste e o tamanho do conjunto de teste mudam quando o número de estados estáveis e não-estáveis varia?
- Como o número de casos de teste e o tamanho do conjunto de teste mudam quando o número de estados estáveis varia?
- Como o número de casos de teste e o tamanho do conjunto de teste mudam quando o número de estados não-estáveis varia?
- Como o número de casos de teste e o tamanho do conjunto de teste mudam quando o número de entradas varia?
- Como o número de casos de teste e o tamanho do conjunto de teste mudam quando o número de saídas varia?

Diferentes configurações foram utilizadas para avaliar a quantidade de casos de teste e o tamanho total dos conjuntos de teste (somatório do tamanho de cada caso de teste). Cinco configurações de Mealy IOTSs foram utilizadas, variando o número de estados estáveis (#est), estados não-estáveis (#n-est), entradas (#e) e saídas (#s):

1. *Número de estados:* 4 entradas, 4 saídas e o número de estados estáveis e não-estáveis variando de 4 a 30.
2. *Número de estados estáveis:* 4 entradas, 4 saídas, 10 estados não estáveis e o número de estados estáveis variando de 4 a 28.
3. *Número de estados não-estáveis:* 4 entradas, 4 saídas, 10 estados estáveis e o número de estados não-estáveis variando de 4 a 30.
4. *Número de entradas:* 4 saídas, 10 estados estáveis, 10 estados não-estáveis e o número de entradas variando de 2 a 10.
5. *Número de saídas:* 4 entradas, 10 estados estáveis, 10 estados não-estáveis e o número de saídas variando de 2 a 10.

A variação no número de estados e de ações de entrada e saída foram embasadas nos experimentos existentes para MEFs (Endo e Simao, 2013) A metodologia adotada nesse experimento foi a seguinte:

1. **Geração de Mealy IOTSs aleatoriamente:** para cada uma das configurações citadas acima, foram gerados aleatoriamente 100 Mealy IOTSs.
2. **Geração dos conjuntos de teste:** para cada Mealy IOTS gerado no passo anterior, foi gerado um conjunto de teste com o auxílio da ferramenta descrita na Seção 4.3.

3. **Análise dos resultados:** foram extraídas duas medidas dos conjuntos de teste gerados no passo anterior: o número de casos de teste de cada conjunto de teste e o tamanho (número de ações) dos conjuntos de teste. Para cada configuração, dois gráficos foram gerados com o número médio de casos de teste e com o tamanho médio dos conjuntos de teste. Foram identificadas as correlações/dependências entre as dimensões analisadas. Para isso, foi empregado o coeficiente de correlação de Spearman (ρ). O cálculo deste coeficiente foi realizado através da ferramenta de Wessa (2012), que é uma ferramenta de cálculos estatísticos para uso acadêmico que emprega a linguagem R[1].

Nesse sentido, foi implementada uma ferramenta para geração aleatória de Mealy IOTSs. O algoritmo de geração aleatória é descrito a seguir.

5.1.1 Geração aleatória de Mealy IOTSs

Foi implementada uma ferramenta para geração aleatória de Mealy IOTSs determinísticos, inicialmente conexos, saída-determinísticos e completos, permitindo a variação do número de estados estáveis, estados não-estáveis, entradas e saídas.

A ideia básica da geração aleatória é semelhante àquela apresentada em (Simao et al., 2009a): a ferramenta primeiro gera o conjunto de estados estáveis e não-estáveis, entradas e saídas, conforme o número determinado para cada um deles. O processo de geração ocorre em três fases: Na primeira fase, um estado estável é selecionado como estado inicial e marcado como "alcançado". Em seguida, para cada estado *s* não "alcançado", o gerador aleatório seleciona um estado *s'* qualquer e uma transição é adicionada de acordo com o caso: se o estado *s'* é um estado estável e que não foi visitado ainda, o gerador adiciona uma transição para cada entrada do alfabeto de entrada de *s'* para um estado *s* qualquer, garantindo que aquele estado é completamente especificado, e o estado *s'* é marcado como "visitado"; se o estado *s'* é um estado não-estável (estado de saída) e está marcado como "alcançado", o gerador escolhe uma saída aleatoriamente, adiciona uma transição rotulada com esta saída a partir de *s'* para *s* e marca *s'* como "visitado". Na terceira fase, se houver algum estado que não tiver transições partindo dele, então é adicionada uma transição com um rótulo de saída (se for um estado não-estável) ou uma transição para cada entrada do alfabeto de entrada para um estado qualquer (se for um estado estável). O Algoritmo 4 formaliza o processo descrito acima. Se o IOTS gerado não for progressivo, minimal e determinístico, ele é descartado e o processo é repetido.

5.1.2 Análise dos Resultados

Nessa seção serão analisados os resultados obtidos pela geração aleatória em relação ao número médio de casos de teste dos conjuntos de teste e o tamanho médio dos conjuntos de teste.

[1] https://www.r-project.org/

Algoritmo 4: Algoritmo para geração aleatória de Mealy IOTSs

Entrada: Uma configuração (#est, #n-est, #e, #s)
Saída: Um Mealy IOTS

```
1  início
2      Gerar os conjuntos de estados estáveis S_stable com #est estados, estados não-estáveis
       S\S_stable com #n-est estados, entradas I com #e e saídas O com #s saídas.
3      Escolher um estado estável s ∈ S e marca como estado inicial e como alcançado
4      enquanto existe um estado s não alcançado e um estado estável s' não visitado faça
5          Selecionar aleatoriamente um estado alcançado s'
6          se s' é um estado estável e está marcado como alcançado então
7              para cada entrada i do alfabeto de entrada faça
8                  Adicionar uma transição de s' para s com rótulo i e marcar s' como visitado
9              fim
10         fim
11         senão se s' é um estado não-estável e está marcado como alcançado então
12             Selecionar uma saída o do alfabeto de saída
13             Adicionar uma transição de s' para s com rótulo o e marcar s' como visitado
14         fim
15     fim
16     se existe um estado estável s' marcado como alcançado então
17         para cada entrada i do alfabeto de entrada faça
18             Selecionar um estado s
19             Adicionar uma transição de s' para s com rótulo i e marcar s como alcançado
20         fim
21     fim
22     se existe um estado não-estável s' marcado como alcançado então
23         Selecionar um estado estável uma saída o do alfabeto de saída
24         Adicionar uma transição de s' para s com rótulo o
25     fim
26 fim
27 retorna IOTS M
```

5.1.2.1 Número de casos de teste

As Figuras 22, 23 e 24 mostram o número médio de casos de teste para os conjuntos de teste gerados com o método W_{IOTS} quando o número de estados estáveis e de estados não estáveis varia (Configuração 1), quando o número de estados estáveis varia (Configuração 2), e quando o número de estados não-estáveis varia (Configuração 3), respectivamente.

Há uma forte correlação positiva entre o número de estados estáveis e o número médio de casos de teste por conjunto ($\rho = 1.0$), como pode ser visto nas Figuras 22 e 23. Porém, há uma forte correlação negativa entre o número de estados de saída e o número médio de casos de teste por conjunto ($\rho = -0.99$), já que o número de casos de teste tende a cair, como pode ser visto na Figura 24. Portanto, o método tende a gerar conjuntos com maior número de casos de teste quando são adicionados mais estados estáveis a um IOTS.

Figura 22 – Resultados quando os estados estáveis e não-estáveis aumentam

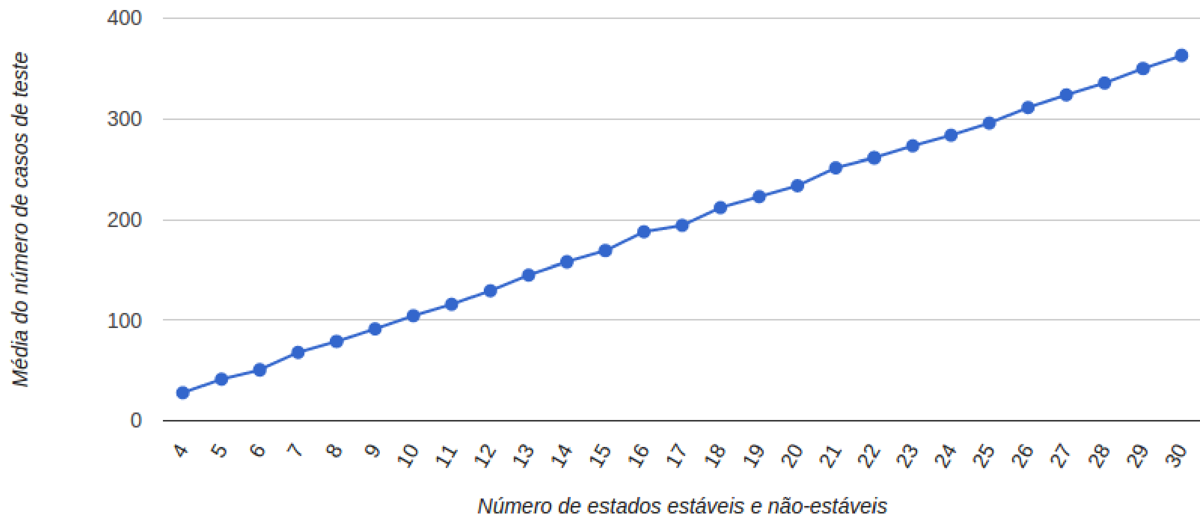
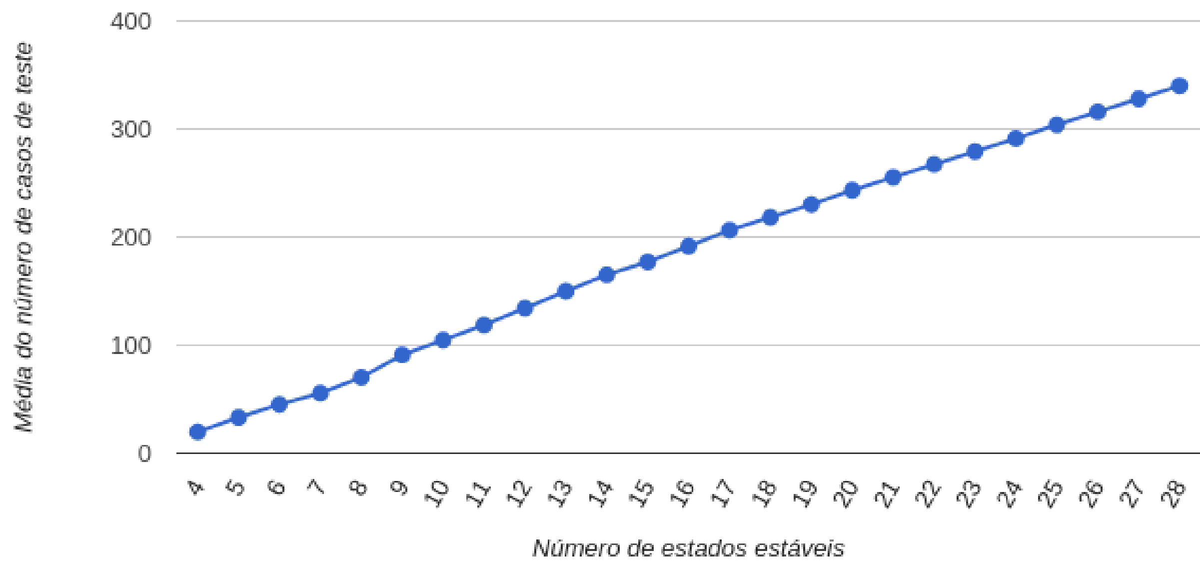


Figura 23 – Resultados quando os estados estáveis aumentam



As Figuras 25 e 26 mostram os resultados dos conjuntos de teste gerados com o método W quando o número de entradas varia (Configuração 4) e quando o número de saídas varia (Configuração 5), respectivamente. Observa-se uma forte correlação positiva entre o número de entradas e o número médio de casos de teste ($\rho = 1.0$), como visto na Figura 25. Já em relação ao número de saídas e o número médio de casos de teste, há uma forte correlação negativa ($\rho = -0.81$), como pode ser observado na Figura 26. Logo, o método tende a gerar conjuntos com maior número de casos de teste quando são adicionadas mais entradas a um IOTS.

Figura 24 – Resultados quando os estados não-estáveis aumentam

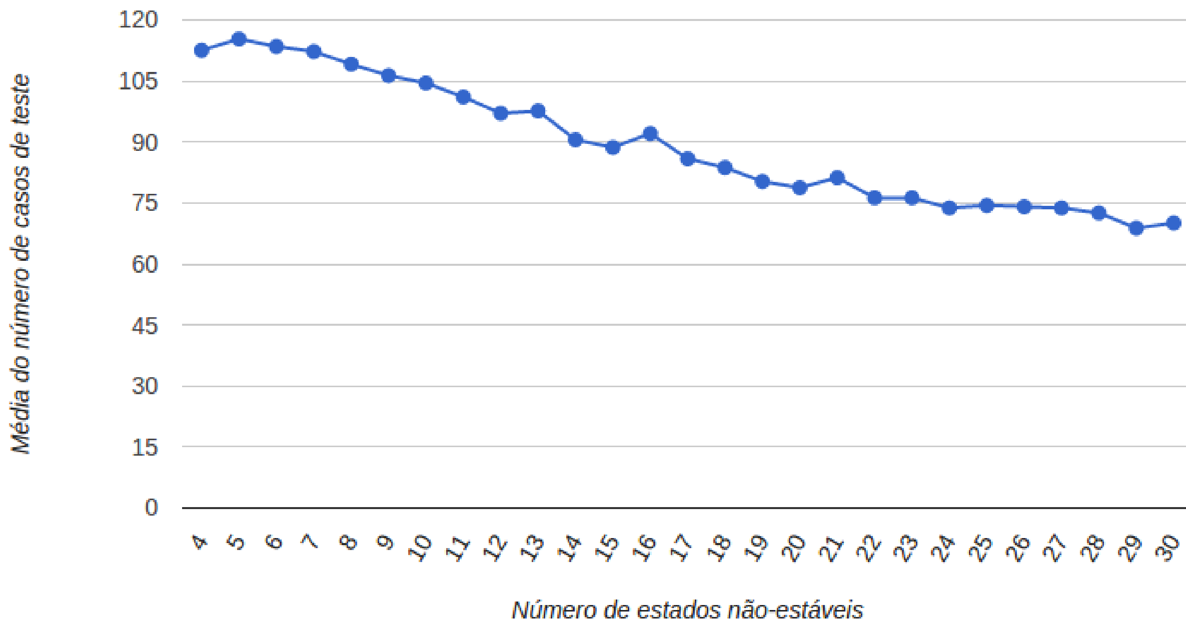
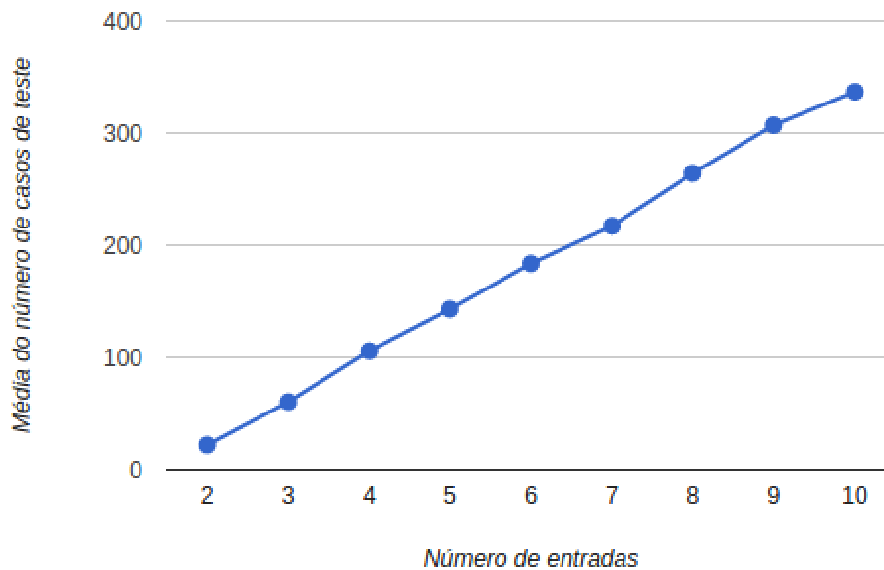


Figura 25 – Resultados quando as entradas aumentam



5.1.2.2 Tamanho total dos conjuntos de teste

Foi realizada a mesma análise de correlação em relação ao tamanho total dos conjuntos de teste. Nota-se uma forte correlação positiva entre o número de estados estáveis e o tamanho médio dos conjuntos de teste ($\rho = 1.0$), como visto nas Figuras 27 e 28. Em relação aos estados não-estáveis (Figura 29), nota-se uma correlação positiva ($\rho = 0.81$). Isto significa que tanto o número de estados estáveis como de não-estáveis influencia no tamanho total dos conjuntos de teste.

Em relação ao número de entradas, nota-se uma forte correlação positiva entre o número

Figura 26 – Resultados quando as saídas aumentam

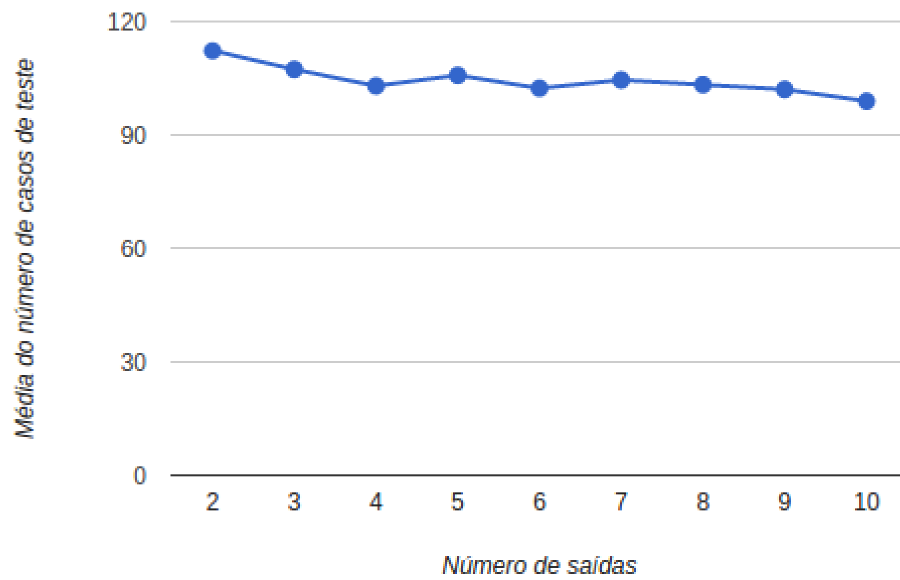
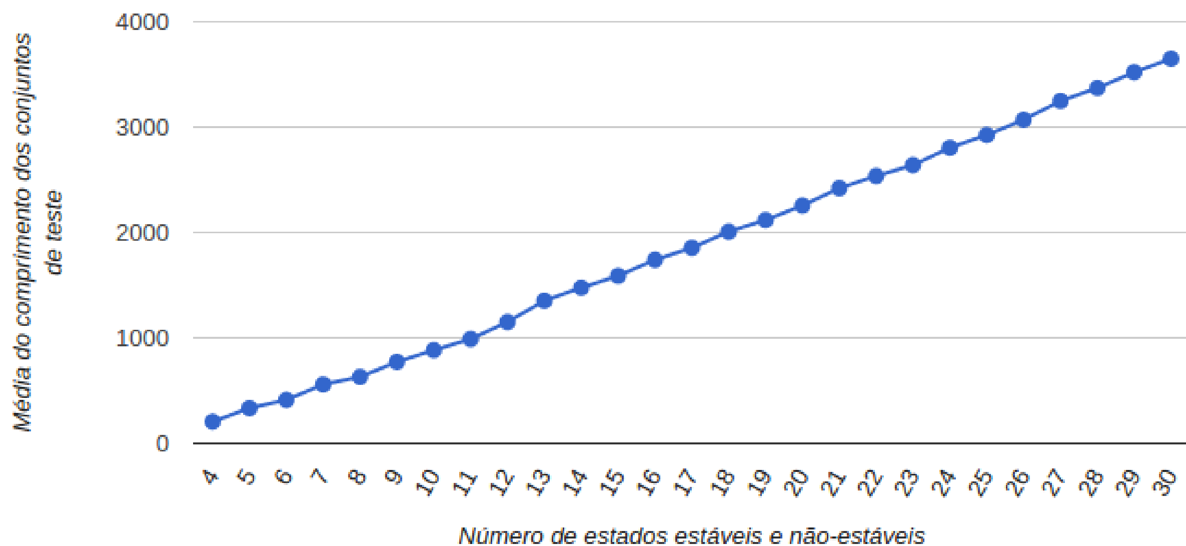


Figura 27 – Resultados quando os estados estáveis e não-estáveis aumentam



de entradas e o tamanho médio dos conjuntos de teste ($\rho = 1.0$), como visto na Figura 30. Já em relação ao número de saídas, há correlação negativa ($\rho = -0.66$) comparada ao tamanho médio dos conjuntos de teste, como visto na Figura 31. Portanto, o método tende a produzir conjuntos de teste de tamanho maior quando são adicionadas entradas no IOTS.

Assim, o número de entradas e de estados estáveis tem forte correlação com o número de casos de teste e o tamanho total do conjunto de teste gerado pelo método W_{IOTS}. O mesmo ocorre com o método W para MEFs, já que a concatenação dos conjuntos *transition cover* e conjunto de caracterização produzem um número maior de casos de teste e consequentemente um conjunto maior em relação a outros métodos de teste baseado em MEFs mais recentes (Endo e Simao, 2013).

Figura 28 – Resultados quando os estados estáveis aumentam

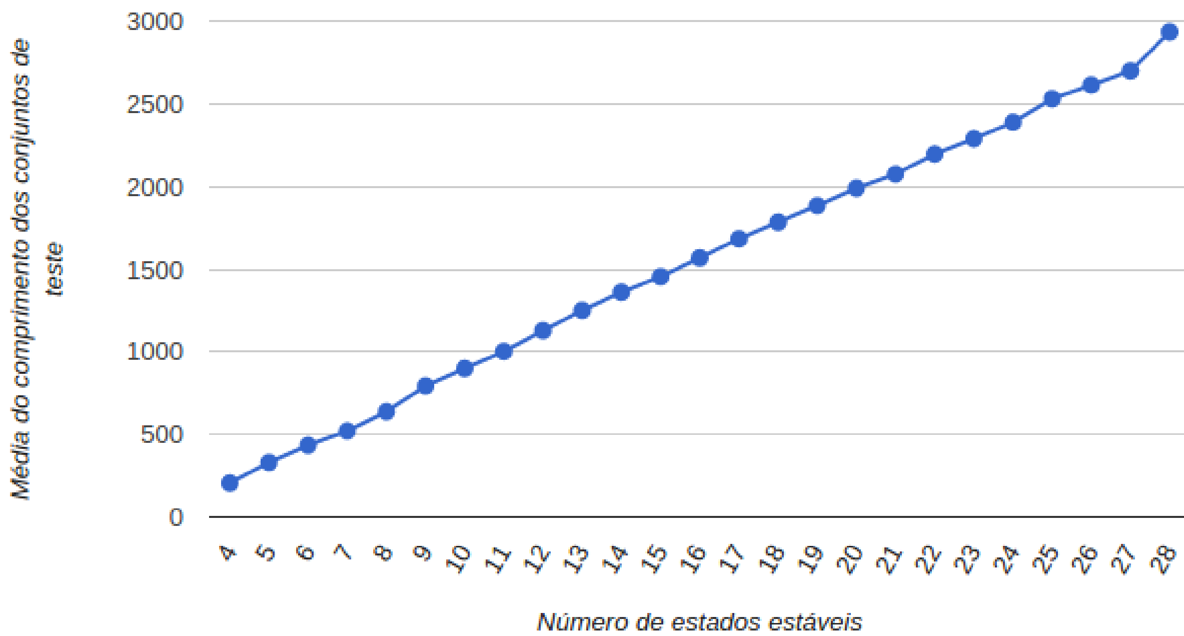
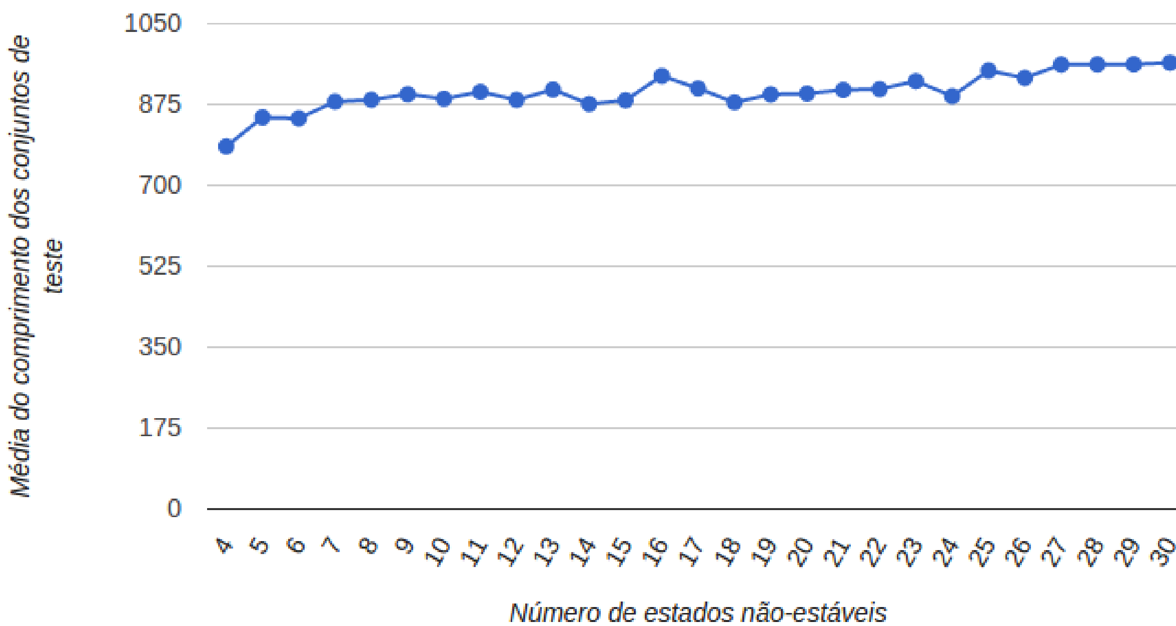


Figura 29 – Resultados quando os estados não-estáveis aumentam



5.2 Estudo de Caso

Embora os resultados experimentais sejam importantes para avaliar o custo de geração do método proposto, o emprego do método em casos reais evidencia sua aplicabilidade. Sistemas ciber-físicos (do inglês, *Cyber Physical Systems*) têm sido alvo de interesse da comunidade científica, por integrarem o controle de entidades físicas com elementos de software e comunicação (Shi et al., 2011). Desse modo, um estudo de caso com especificações da indústria para

Figura 30 – Resultados quando as entradas aumentam

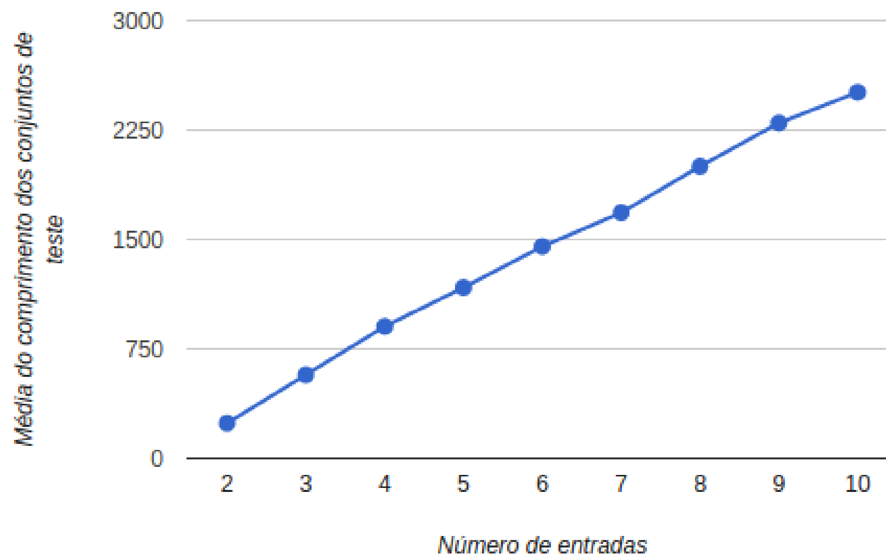
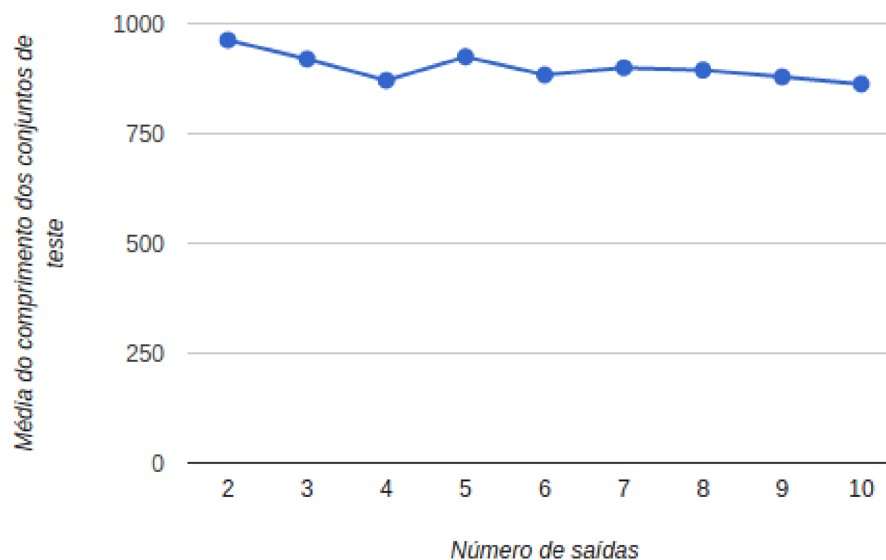


Figura 31 – Resultados quando as saídas aumentam



Sistemas ciber-físicos foi realizado com o objetivo de comparar a efetividade dos conjuntos de teste produzido pelo método W_{IOTS} proposto no Capítulo 4 com os conjuntos produzido pelo método tradicional de Tretmans (2008) por meio da ferramenta JTorX (Belinfante, 2010) que utiliza a refinada teoria *ioco*. Esse estudo foi realizado em colaboração com o pesquisador M. R. Mousavi, da Universidade de Halmstad, Suécia. Foram selecionadas especificações que possuiam as propriedades requeridas pelo método W_{IOTS} para Mealy IOTSs.

Os passos a seguir resumem a metodologia adotada neste estudo:

1. *Preparação de versões defeituosas da especificação*: Para cada especificação, foi produzido um grupo de 20 versões defeituosas (mutantes) de modo que em cada uma foi semeado um defeito de transição ou um defeito de saída.

2. *Geração de conjuntos de teste com a JTorX:* Cada mutante e suas respectivas especificações foram representados no formato GRAPHML. Cada um dos mutantes foi executado na JTorX 50 vezes contra a especificação até que o mutante fosse morto, isto é, o defeito fosse descoberto. O número total de passos até atingir esse ponto foi registrado.

3. *Geração de conjuntos de teste com o método proposto:* Foi produzido um conjunto de teste com o método W_{IOTS} para cada uma das especificações com o auxílio da ferramenta descrita na Seção 4.3. Para cada especificação, foi executado o conjunto de teste 50 vezes em comparação a cada mutante com o auxílio da mesma ferramenta (Seção 4.3). Em cada execução, a sequência dos casos de teste foi escolhida aleatoriamente. O número total de passos (ações de entrada ou saída) executados até que o mutante fosse morto foi registrado.

4. *Análise dos resultados:* duas medidas foram utilizadas para comparar a eficiência dos métodos em detectar defeitos: o nível de profundidade do defeito em cada mutante e a média de passos até o defeito ser descoberto.

Para a geração das versões defeituosas de cada especificação, foi adotada a seguinte metodologia *ad hoc*: a partir do estado inicial da árvore de teste, em cada nível da árvore foi selecionada pelo menos uma transição para inserir um defeito de transferência (mudar o estado que a transição atinge) e também pelo menos uma transição em que foi inserido um defeito de saída (alterando a saída/entrada da transição). Deste modo, os defeitos inseridos foram igualmente distribuídos entre defeitos de saída e defeitos de transferência.

5.2.1 Especificações

As especificações utilizadas neste estudo são do domínio de sistemas ciber-físicos: *Ceiling Speed Monitoring* (CSM) com a Intervenção do serviço de freio (*Service Brake Intervention* - SBI) e com a Intervenção do freio de emergência (*Emergency Brake Intervention* - EBI) (Braunstein et al., 2014b); a especificação de luzes indicadoras de direção para veículos (*Turn Indicator Light* - TIL) (Peleska et al., 2011); e a especificação do comportamento do componente padrão de Espelho Exterior (*Exterior Mirror* - EM) e do componente do sistema de alarme (*Alarm System* - AS) para o *Body Comfort System* (Lity et al., 2013) para veículos. A seguir serão detalhadas cada uma delas.

5.2.1.1 Ceiling Speed Monitoring

A especificação CSM trata de um monitor de velocidade máxima para trens. Existem duas configurações possíveis: um trem deve ter o recurso de freio de emergência, ou um trem pode ter o recurso de serviço de freio. A ideia é que o trem sem o recurso de serviço de freio deve utilizar o recurso de freio de emergência para diminuir sua velocidade independentemente da situação. Já o trem que possui o recurso de serviço de freio deve utilizar o recurso de freio de emergência apenas em uma situação de emergência (Braunstein et al., 2014b).

Se o monitor de velocidade máxima detectar um limiar de sobrevelocidade, então o serviço de freio é acionado caso ele esteja disponível. Caso contrário, o freio de emergência é acionado. A partir do serviço de freio é possível retornar ao estado normal se a velocidade diminuir após a intervenção. Se o trem continuar em aceleração, o freio de emergência é acionado.

Desse modo, essa especificação foi separada em dois modelos IOTSs, um para cada configuração: SBI (com recurso de serviço de freio - do inglês, *Service Brake Intervention*) e EBI (apenas com freio de emergência - do inglês, *Emergency Brake Intervention*). As entradas são relacionadas às condições que engatilham alguma ação, conforme definido em (Braunstein et al., 2014a) e apresentado na Tabela 9. As saídas são o resultado das condições. O comportamento do monitor é descrito a seguir:

- Se o trem estiver no estado *normal* e detecta que a velocidade atual (V_{est}) está um pouco acima da velocidade permitida mas dentro do limite de velocidade aplicável (V_{MRSP}), o monitor vai para o estado de *sobrevelocidade* (*overspeed*).
- Se o trem estiver em *sobrevelocidade* mas não atinge o limite aplicável, o monitor volta ao estado normal. Caso contrário, o monitor passa para o estado de *atenção*.
- Se o trem estiver no estado *normal* e detecta que a velocidade atual está acima do limite de velocidade aplicável, então há uma mudança para o status de *atenção* (*warning*).
- Se a velocidade atual continuar aumentando e for maior que o segundo limite definido ($dV_{Warning}(V_{MRSP})$), então o serviço de freio/freio de emergência é acionado.
- Se o trem tiver um serviço de freio já acionado e a velocidade continuar aumentando, então o freio de emergência é acionado.
- Se o trem estiver com o serviço de freio/freio de emergência acionado e verificar que a velocidade atingiu um limite seguro, então o monitor volta para o estado *normal*.

As Figuras 32 e 33 mostram os modelos IOTS produzidos para as configurações SBI e EBI, respectivamente.

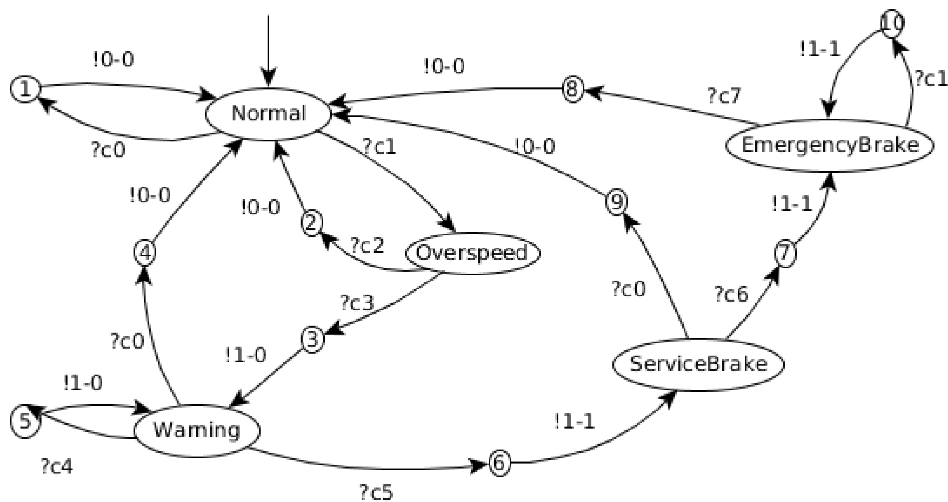
5.2.1.2 Indicadores de direção

Um modelo para funcionalidades de sistemas automotivos relacionada aos indicadores de direção disponíveis nos veículos da Mercedes Benz foi apresentado em Peleska et al. (2011). O modelo de luzes indicadoras de direção (*Turn Indicator Light*) cobre todas as funcionalidades de luzes indicadoras para direita/esquerda, luzes de emergência, luzes de colisão, luzes de furto e luzes de abertura/fechamento do veículo. O modelo comportamental que compreende essas funcionalidades é mostrado na Figura 34.

Tabela 9 – Condições para o funcionamento do monitor de velocidade máxima para trens

#	Condições para SBI	Condições para EBI
c_0	$V_{est} \leq V_{MRSP}$	$V_{est} \leq V_{MRSP}$
c_1	$V_{est} > V_{MRSP}$	$V_{est} > V_{MRSP}$
c_2	$V_{est} \leq V_{MRSP}$	$V_{est} \leq V_{MRSP}$
c_3	$V_{est} > V_{MRSP} + dV_{Warning}(V_{MRSP})$	$V_{est} > V_{MRSP} + dV_{Warning}(V_{MRSP})$
c_4	$V_{est} < V_{MRSP} + dV_{Warning}(V_{MRSP}) \wedge V_{est} < V_{MRSP} + dV_{EBI}(V_{MRSP})$	$V_{est} < V_{MRSP} + dV_{Warning}(V_{MRSP}) \wedge V_{est} < V_{MRSP} + dV_{SBI}(V_{MRSP})$
c_5	$V_{est} > V_{MRSP} + dV_{EBI}(V_{MRSP})$	$V_{est} > V_{MRSP} + dV_{SBI}(V_{MRSP})$
c_6	$(V_{est} \leq V_{MRSP} \wedge a) \vee V_{est} = 0$	$V_{est} > V_{MRSP} + dV_{EBI}(V_{MRSP})$
c_7	-	$(V_{est} \leq V_{MRSP} \wedge a) \vee V_{est} = 0$

Figura 32 – Especificação SBI



5.2.1.3 Body Comfort System

O sistema de conforto (Body Comfort System) (Lity et al., 2013) é um estudo de caso para o domínio automotivo, que compreende um sistema complexo e inerentemente reativo. Os diferentes componentes de software desse sistema implementam a interação de tarefas de controle reativas entre eles e com o ambiente via sinais de entrada fornecidos por sensores e sinais de saída emitidos por atuadores. Foram utilizadas no estudo de caso as especificações de dois componentes desse sistema: componente padrão de espelho externo (*Exterior Mirror* - EM) e do componente padrão do sistema de alarme (*Alarm System* - AS). O componente AS controla a ativação/desativação do sistema de alarme bem como o acionamento do alarme. Já o componente EM controla os movimentos do espelho externo. O comportamento desses componentes é representado pelos IOTSs das Figuras 35 e 36, respectivamente.

5.2.2 Resultados

As Figuras 37, 38, 39, 40, 41, e 42 mostram os resultados desta comparação para as especificações SBI, EBI, TIL, EM , AS e SCP, respectivamente. Os mutantes foram ordenados

Figura 33 – Especificação EBI

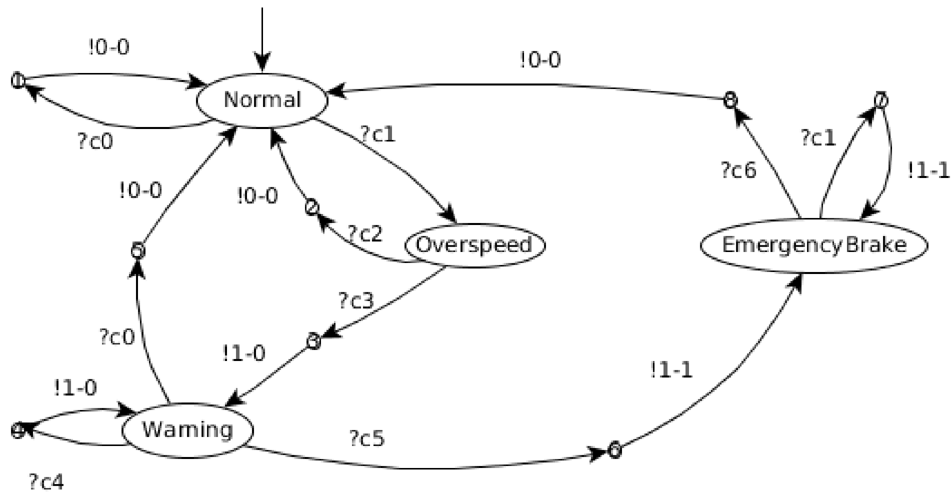
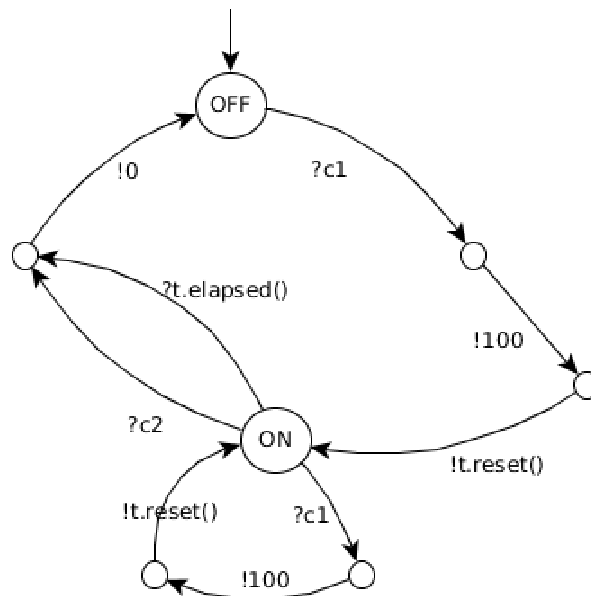


Figura 34 – Especificação TIL



em ordem crescente do nível de profundidade em que o defeito foi inserido.

Os mutantes com defeitos em níveis mais profundos podem ser detectados pelo método W_{IOTS} com mais eficiência, como pode ser visto nas Figuras 37, 38 e 42. A ferramenta JTorX pode detectar defeitos em nível superficial com no mínimo a mesma eficiência do método W_{IOTS} ou melhor. O modelo SCP é profundo e com várias ramificações, e isso tornou bem mais custosa a descoberta de defeitos pela JTorX e o método W_{IOTS} bem mais eficiente. Os modelos SBI e EBI são profundos, por isso os defeitos desses modelos foram detectados de forma mais eficiente pelo método W_{IOTS}, como mostrado nas Figuras 37 e 38. O modelo TIL tem um número de profundidade semelhante aos modelos EBI e SBI, mas possui um pequeno número de *traces*. Isso torna mais fácil a descoberta de defeitos pela JTorX, como visto na Figura 39. Entretanto, o modelo EM é raso, logo os defeitos inseridos são em níveis superficiais. Além disso, possui um grande número de *traces*. Logo, a ferramenta JTorX foi mais eficiente em detectar os defeitos

Figura 35 – Especificação EM

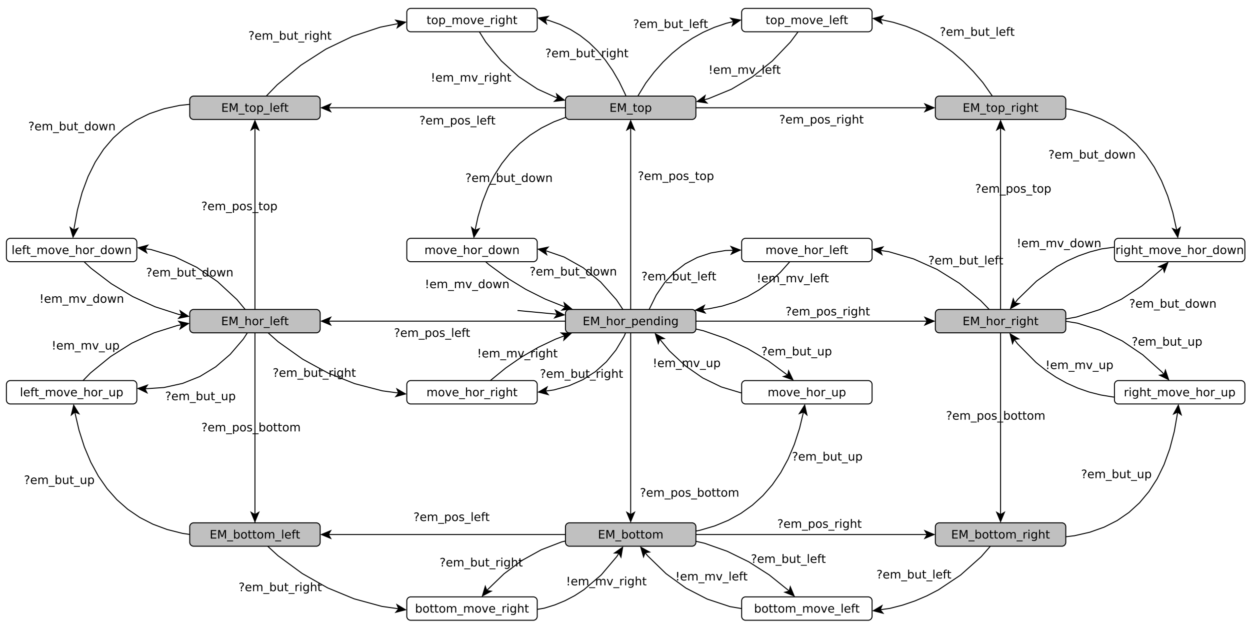
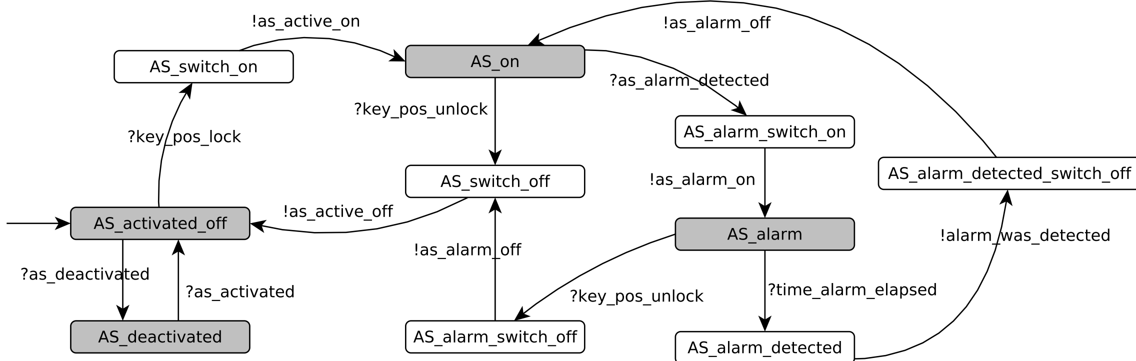


Figura 36 – Especificação AS



desse modelo, como visto na Figura 40. O modelo AS é mais profundo, semelhante aos modelos EBI e SBI, mas possui um número menor de *traces*, fazendo com que os resultados da JTorX sejam melhores do que o método W_{IOTS} (Figura 41).

Estes resultados apontam que o método W_{IOTS} é mais eficiente em detectar defeitos em um nível mais profundo do que a JTorX. Do mesmo modo, JTorX pode detectar defeitos em um nível superficial com mais eficiência. Porém, modelos que possuem um pequeno número de *traces* podem ser percorridos mais rapidamente pela JTorX. Logo, nesse tipo de modelo os defeitos podem ser detectados mais rapidamente pela JTorX.

Figura 37 – Resultados para a especificação SBI

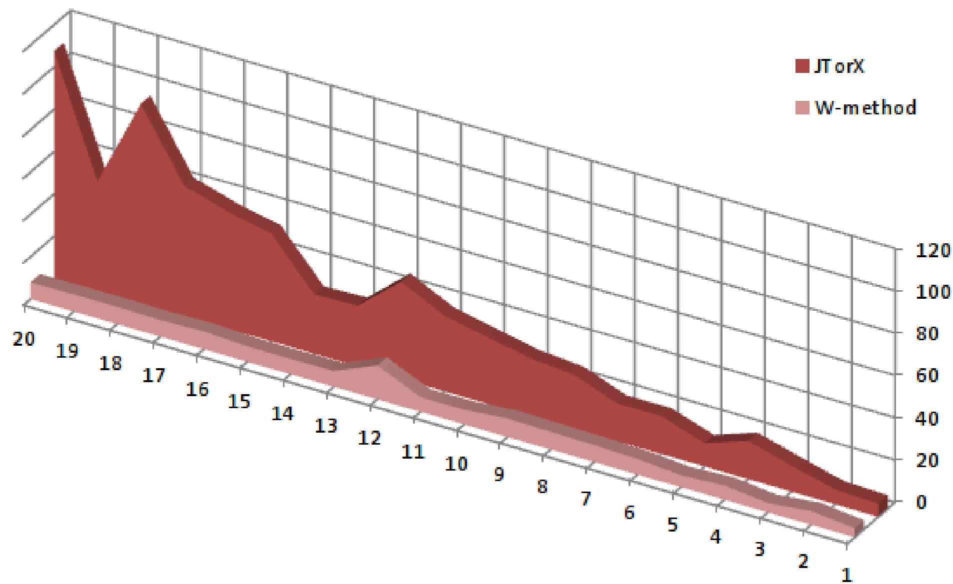
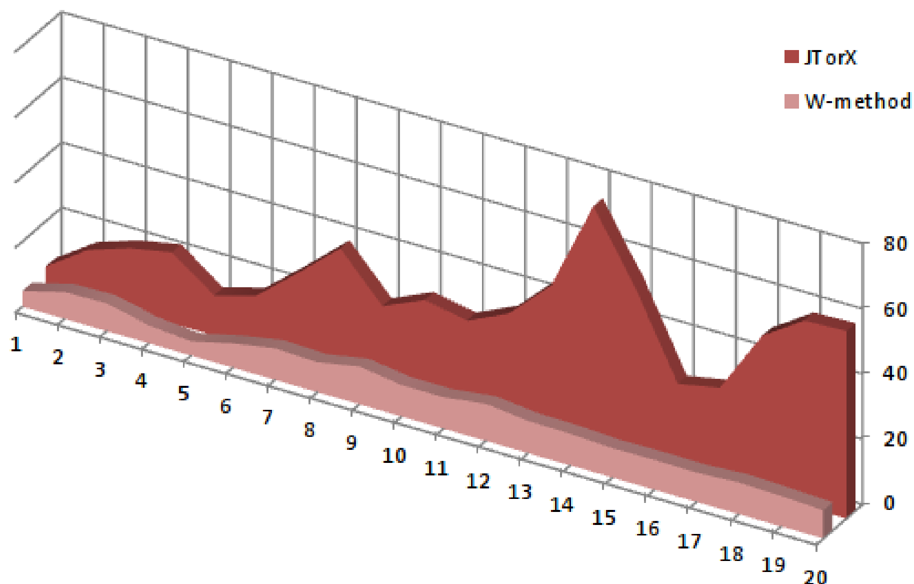


Figura 38 – Resultados para a especificação EBI



5.3 Comparação do Estudo de Caso com Geração Aleatória

Essa seção apresenta uma comparação entre os conjuntos de teste gerados a partir de modelos aleatórios e o conjunto de teste gerado por cada uma das especificações do estudo de caso. Nesse sentido, foram gerados Mealy IOTSs aleatoriamente com a ferramenta apresentada na Seção 5.1. Para cada especificação, foram gerados 100 IOTSs com a mesma configuração da especificação. A configuração de cada um dos modelos, do número de casos de teste e do tamanho do conjunto de teste gerado pelo método W_{IOTS} são apresentados na Tabela 10.

Figura 39 – Resultados para a especificação TIL

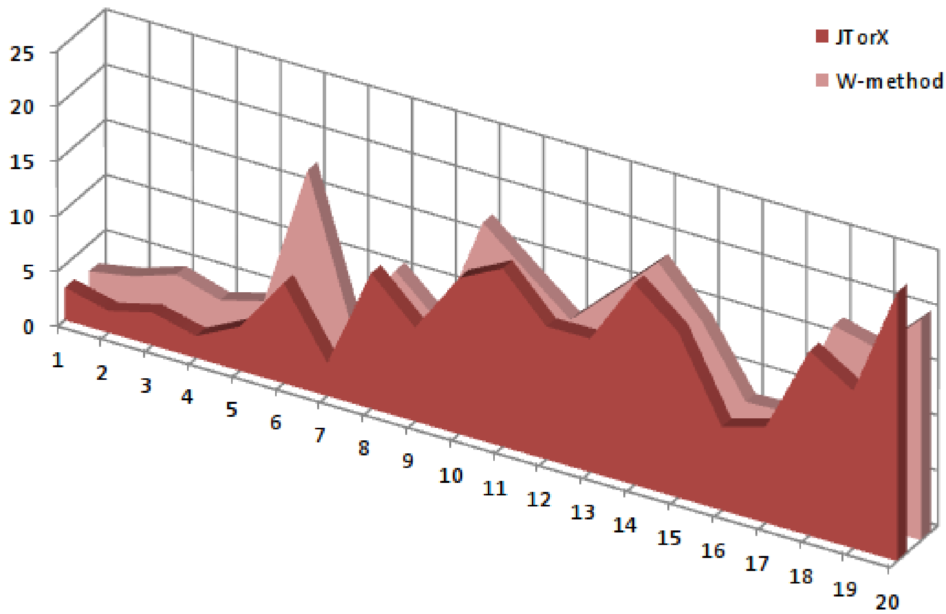
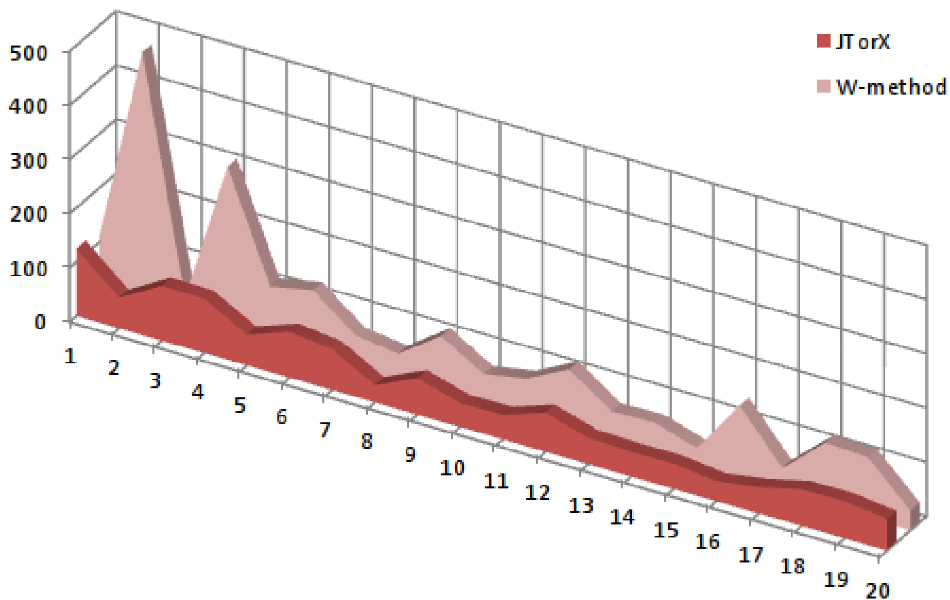


Figura 40 – Resultados para a especificação EM



As Figuras 43, 44, 45, 47 e 46 apresentam os resultados obtidos com a geração aleatória e com o método W_{IOTS} para cada especificação. Como visto nos gráficos, quase todos os resultados das especificações ficaram dentro da margem de resultados produzidos pelos IOTSs gerados aleatoriamente, com exceção do número de casos de teste do modelo SBI (Figura 43) e do modelo EM (Figura 46).

Figura 41 – Resultados para a especficação AS

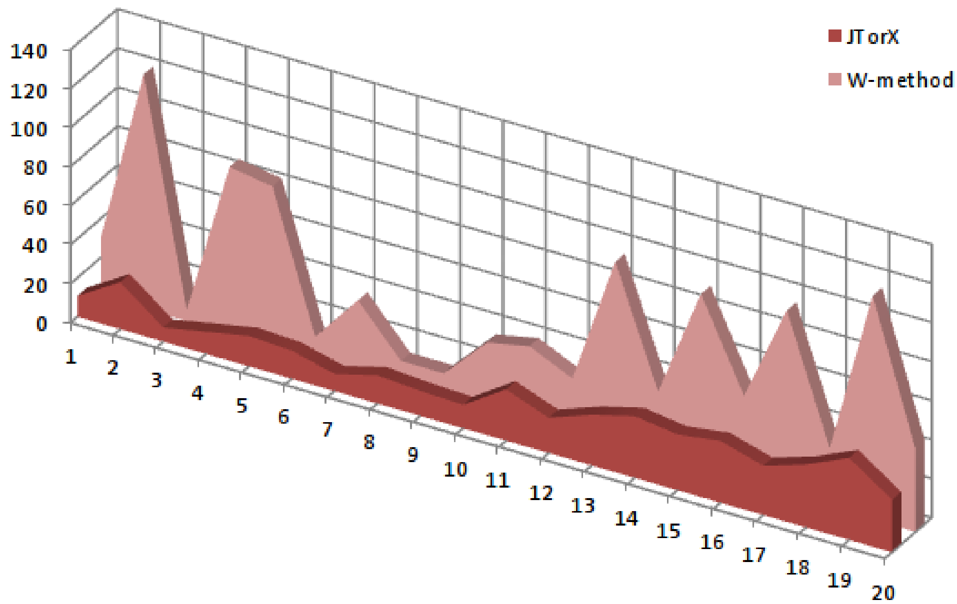
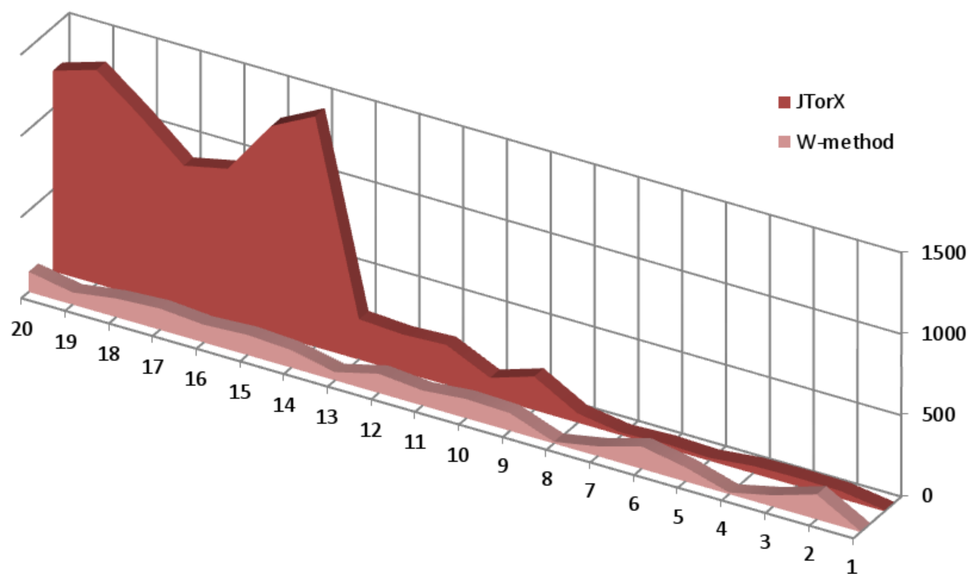


Figura 42 – Resultados para a especificação SCP



5.4 Discussão dos Resultados

Os resultados experimentais apresentados na Seção 5.1 auxiliam na avaliação do custo de aplicação do método W_{IOTS}. Foi mostrado que o número de casos de teste está diretamente relacionado ao número de entradas e estados estáveis. Além disso, o tamanho dos conjuntos de teste também está relacionado ao número de entradas, número de estados estáveis e número de estados não-estáveis. No entanto, um número muito grande de estados estáveis e de entradas pode levar a um alto custo de aplicação desse método.

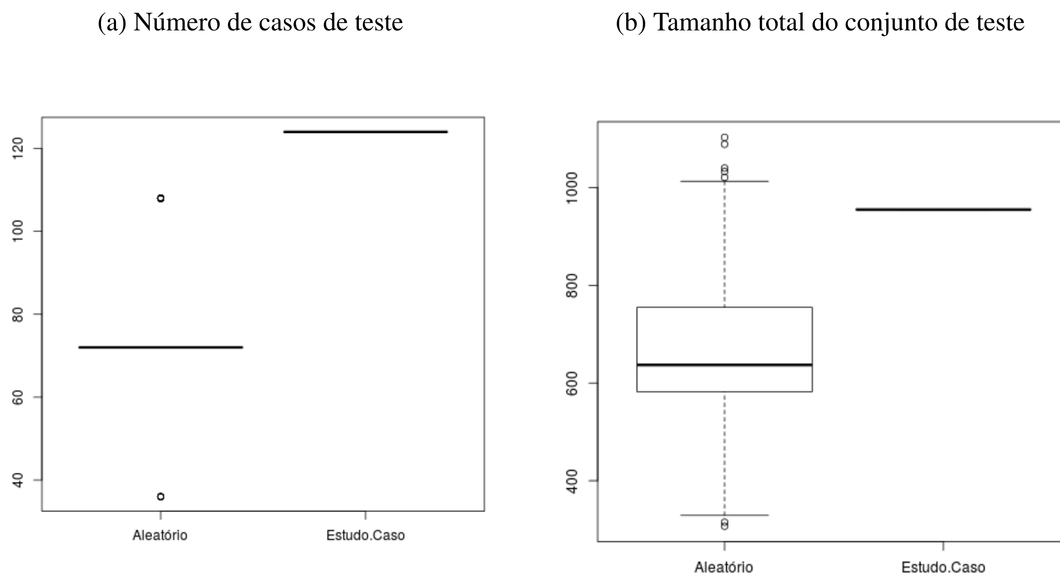
O estudo de caso demonstrou que o método é aplicável a especificações reais de software. Todos os defeitos semeados na especificação, que pertenciam ao domínio de defeitos, foram

Tabela 10 – Características dos modelos utilizados no estudo de caso

Especificação	Número casos de teste	Tamanho conj. teste	Entradas	Saídas	Estados estáveis	Estados não-estáveis
SBI	124	955	8	3	5	10
EBI	75	516	7	3	4	8
TIL	5	34	3	3	2	5
AS	63	407	6	5	4	6
EM	256	1660	6	4	9	12

Figura 43 – Comparação da especificação SBI com a geração aleatória

(a) Número de casos de teste (b) Tamanho total do conjunto de teste

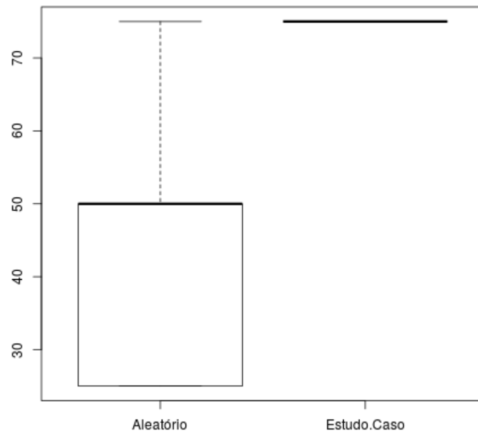


detectados pelo método W_{IOTS}. Os resultados apontaram que os defeitos de modelos inseridos em níveis mais profundos e em modelos com mais ramificações podem ser detectados com mais eficiência pelo método W_{IOTS}. Já os defeitos inseridos em níveis superficiais podem ser detectados com eficiência pela ferramenta JTorX. Portanto, o método W_{IOTS} é eficiente em detectar defeitos, sendo mais eficiente que o método tradicional em alguns casos.

Os resultados obtidos por IOTSs gerados aleatoriamente e pelas especificações reais foram semelhantes. Isto aponta que os resultados obtidos por IOTSs aleatórios são válidos para avaliar a aplicabilidade e custo de um método, já que é mais simples obter um grande número de IOTSs aleatórios do que obter um grande número de especificações reais com as propriedades requeridas pelo método W_{IOTS}. Além disso, a geração aleatória permite obter IOTSs com diversas configurações, possibilitando uma avaliação mais ampla da aplicação de um método de geração.

Figura 44 – Comparação da especificação EBI com a geração aleatória

(a) Número de casos de teste



(b) Tamanho total do conjunto de teste

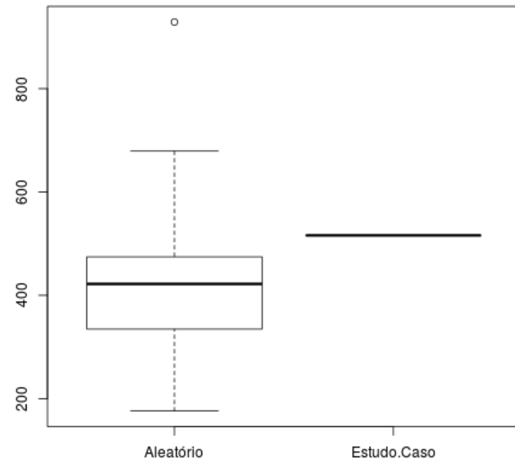
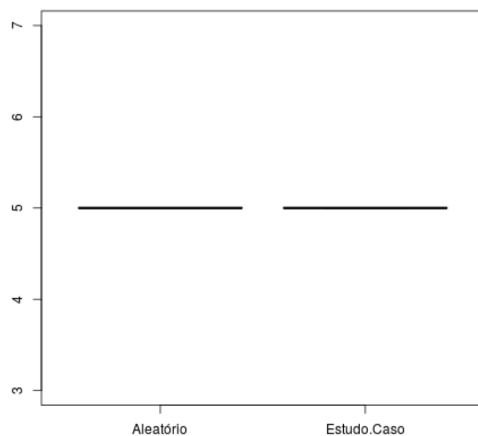
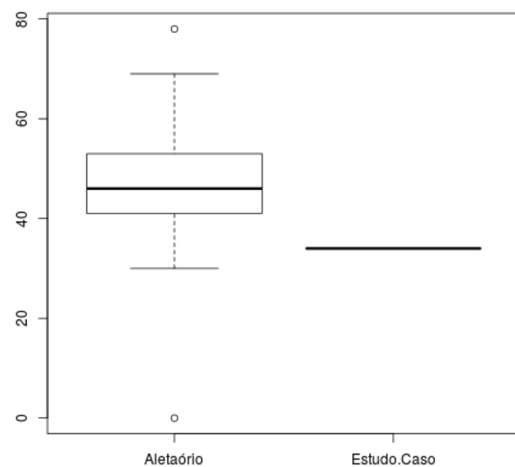


Figura 45 – Comparação da especificação TIL com a geração aleatória

(a) Número de casos de teste



(b) Tamanho total do conjunto de teste



5.5 Considerações Finais

Este capítulo apresentou resultados experimentais sobre os conjuntos de teste gerados pelo método proposto para Mealy IOTSs, denominado W_{IOTS}, e um estudo de caso comparando o método proposto com o método tradicional de Tretmans (2008). Os resultados do experimento com IOTSs gerados aleatoriamente indicam que o número de estados estáveis e o número de entrada tem forte relação com o número de casos de teste e com o tamanho médio dos conjuntos de teste.

Figura 46 – Comparação da especificação EM com a geração aleatória

(a) Número de casos de teste (b) Tamanho total do conjunto de teste

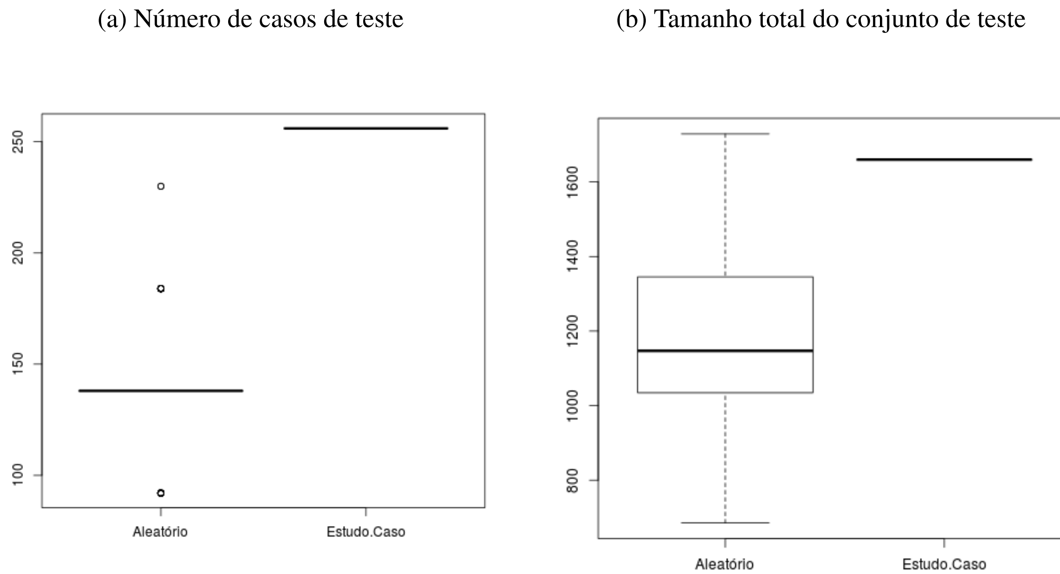
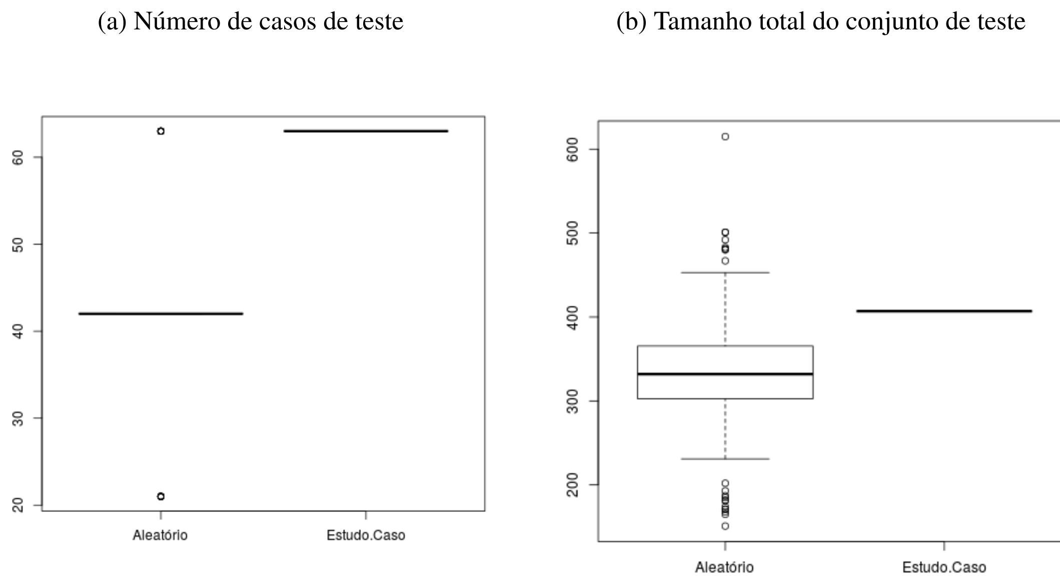


Figura 47 – Comparação da especificação AS com a geração aleatória

(a) Número de casos de teste (b) Tamanho total do conjunto de teste



O estudo de caso apontou que o método proposto é mais eficiente para identificar defeitos em um nível mais profundo da especificação do que o método tradicional da ferramenta JTorX. Além disso, os resultados produzidos pelo método W_{IOTS} para IOTSs gerados aleatoriamente é semelhante aos resultados produzidos pelas especificações reais do estudo de caso em relação ao tamanho dos conjuntos de teste e do número de casos de teste.

O próximo capítulo conclui esta tese, sumarizando as principais contribuições, discutindo as limitações e mencionando direções para trabalhos futuros.

CAPÍTULO

6

CONCLUSÕES

O TBM proporciona a formalização da atividade de teste em aplicações complexas. Porém, não é suficiente utilizar modelos formais se a tecnologia de geração dos casos de teste não garante a satisfação de critérios de cobertura de defeitos em tempo hábil. O TBM a partir de IOTSs tem sido amplamente utilizado pela comunidade científica para formalizar o teste de aplicações complexas, como sistemas reativos. Porém, ainda há desafios relacionados à seleção de casos de teste que detectem defeitos de maneira eficiente e com baixo custo.

Nesse contexto, esta tese contribui com estudos teóricos e empíricos para avanço do tópico de seleção de casos de teste a partir de IOTSs. Os resultados aqui apresentados apontam que é possível responder positivamente à pergunta do Capítulo 1, isto é, *é possível aplicar modelos de defeitos com domínios de defeitos em métodos determinísticos para a geração de conjuntos de teste completos a partir de IOTSs*, sob certos pressupostos.

As contribuições que apoiam a resposta a essa questão são apresentadas na Seção 6.1. A Seção 6.2 aponta as limitações e discute futuras direções deste trabalho.

6.1 Contribuições

Esta seção revisita as contribuições desta tese.

Condições de suficiência: A aplicação de um modelo de defeitos no processo de teste implica que o conjunto de teste obtido deve cobrir todos os defeitos de um dado domínio de defeitos. Condições de suficiência para completude de conjuntos de teste foram definidas na Seção 4.2.1. Tais condições definem que um conjunto de teste será completo se estiver contido nele um conjunto confirmado com *traces* de transferência para cada um dos estados estáveis da especificação. A partir de tal conjunto, uma transição é coberta se existir um *trace* de transferência que é estendido por tal transição. Foi demonstrado que a satisfação de tais condições garante a cobertura de todas as transições da especificação pelo conjunto de teste.

Método para geração de casos de teste: Foi introduzido o método W_{IOTS} para Mealy IOTSs no Capítulo 4. Este é um método determinístico que aplica um modelo de defeitos para a geração de conjuntos de teste completos. Esse método foi adaptado a partir do método W para MEFs (Chow, 1978) para o contexto de IOTSs, de modo que são gerados os conjuntos *transition cover* e de caracterização e são concatenados para a obtenção dos casos de teste. O método garante a cobertura de todas as transições através da satisfação das condições de suficiência propostas na Seção 4.2.1. Este é o primeiro passo em direção a métodos com total cobertura de defeitos de um dado domínio a partir de IOTS. O método foi implementado como uma prova de conceito da proposta e uma comparação com a ferramenta JTorX evidencia a redução dos conjuntos gerados com o método proposto em relação ao método tradicional de Tretmans (2008).

Avaliação do método proposto: Estudos empíricos foram realizados e apresentados no Capítulo 5 para avaliar o custo de geração e viabilidade de aplicação do método proposto. Uma análise de custo de geração com o uso de IOTSs gerados aleatoriamente foi realizada. Este foi o primeiro estudo deste tipo realizado no contexto de IOTSs. Foram analisados o número de casos de teste e o tamanho médio dos conjuntos de teste produzidos pelo método W_{IOTS}. Os resultados indicam que o número de casos de teste e o tamanho do conjunto de teste está diretamente relacionado ao número de entradas e de estados estáveis. Também foi conduzido um estudo de caso com especificações reais para Sistemas ciber-físicos para comparar a efetividade do método W_{IOTS} com o método tradicional implementado na ferramenta JTorX. Os resultados evidenciaram que o método W_{IOTS} detectou todos os defeitos, além de ser mais eficiente em detectar defeitos em níveis mais profundos e em modelos que possuem maior número de ramificações. Uma comparação entre os resultados produzidos pelo método W_{IOTS} para as especificações reais e para IOTSs gerados aleatoriamente com a mesma configuração das especificações reais apontou resultados semelhantes, evidenciando que os modelos aleatórios podem auxiliar na avaliação e comparação de novos métodos de geração de testes.

6.2 Limitações e Trabalhos Futuros

Esta seção apresenta as limitações das contribuições apresentadas e possíveis formas de lidar com essas limitações.

Condições de suficiência: As condições de suficiência propostas abrangem apenas o domínio de defeitos em que as implementações possuem no máximo o mesmo número de estados estáveis da especificação.

Método de geração: O método proposto para Mealy IOTSs é baseado no método W para MEFs, que é um dos métodos mais antigos. Este foi um primeiro passo para a construção de métodos que empregam um modelo de defeitos no teste a partir de IOTSs. Porém, comparado a métodos mais recentes de MEFs, ele produz conjuntos de teste grandes, podendo ser inviável sua aplicação em modelos com grande número de estados estáveis. Além disso, o método é aplicável

apenas em especificações que possuem as propriedades requeridas.

Estudos empíricos: O método proposto foi avaliado por estudos empíricos apresentados no Capítulo 5. As especificações reais limitam-se a sistemas ciber-físicos e um protocolo de comunicação. No entanto, as características dos modelos apresentados podem ter limitado os resultados apresentados. Além disso, outros domínios de sistemas podem ter diferentes tipos de defeitos, que podem não ter sido cobertos pelo domínio de defeitos aqui definido.

6.2.1 Possíveis extensões

Algumas extensões possíveis para a contribuição deste trabalho incluem as condições de suficiência aqui propostas, que podem ser estendidas de modo que inclua um domínio de defeitos mais abrangente. Desse modo, podem ser incluídas implementações que possuem mais estados estáveis do que a especificação (estados extras).

O método de geração aqui proposto é limitado a Mealy IOTSs. Os pressupostos definidos limitam a aplicação deste método. Portanto, uma possível direção é relaxar os pressupostos aqui definidos em direção ao modelo geral de IOTS. Também pode ser investigada a aplicação do método para especificações não-completas nos estados estáveis.

Além disso, métodos do teste a partir de MEFs que geram conjuntos menores e mais eficientes podem ser investigados para delinear a definição de novos métodos de geração de casos de teste a partir de IOTSs.

Outra possível direção é melhorar a ferramenta que implementa o método, definindo uma interface de usuário e integrando com outras ferramentas, podendo agilizar o processo de execução de testes.

6.3 Publicações resultantes

Os artigos publicados resultantes desta tese são:

- PAIVA, S. L. C.; SIMAO, A. A Systematic Mapping Study on Test Generation from Input/Output Transition Systems. In: Software Engineering and Advanced Applications (SEAA), 2015 41th EUROMICRO Conference on, IEEE Computer Society, 2015. (Paiva e Simao, 2015)

- PAIVA, S. C.; SIMAO, A. Generation of Complete Test Suites from Mealy Input/Output Transition Systems. Formal Aspects of Computing, p. 1–14, 2015. (Paiva e Simao, 2016)

Pretende-se publicar os resultados dos estudos empíricos:

- Estudo de caso com especificações reais, em colaboração com o pesquisador M. R. Mousavi, apresentado na Seção 5.2.

- Resultados experimentais com geração aleatória e comparação com especificações reais, apresentado nas Seções 5.1 e 5.3.

REFERÊNCIAS

AICHERNIG, B.; DELGADO, C. From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems. In: BARESI, L.; HECKEL, R., eds. *Fundamental Approaches to Software Engineering*, v. 3922 de *Lecture Notes in Computer Science*, Springer Berlin/ Heidelberg, p. 324–338, 2006. Citado 7 vezes nas páginas 65, 68, 69, 71, 72, 77 e 149.

AICHERNIG, B.; PEISCHL, B.; WEIGLHOFER, M.; WOTAWA, F. Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods. In: *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, IEEE Computer Society, 2007, p. 215–226. Citado 7 vezes nas páginas 64, 65, 68, 69, 71, 72 e 149.

AICHERNIG, B. K.; WEIGLHOFER, M.; WOTAWA, F. Improving Fault-based Conformance Testing. *Electronic Notes in Theoretical Computer Science*, v. 220, n. 1, p. 63–77, MBT'08, 2008. Citado 6 vezes nas páginas 65, 68, 69, 71, 72 e 149.

AIGUIER, M.; BOULANGER, F.; KANSO, B. A Formal Abstract Framework for Modelling and Testing Complex Software Systems. *Theoretical Computer Science*, v. 455, p. 66–97, International Colloquium on Theoretical Aspects of Computing 2010, 2012. Citado 2 vezes nas páginas 69 e 149.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. New York, USA: Cambridge University Press, 2008. Citado 2 vezes nas páginas 37 e 42.

ANDRADE, W.; MACHADO, P. Testing Interruptions in Reactive Systems. *Formal Aspects of Computing*, v. 24, n. 3, p. 331–353, 2012. Citado 7 vezes nas páginas 64, 66, 68, 69, 71, 72 e 149.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: *Proceedings of the 27th International Conference on Software Engineering*, New York, NY, USA: ACM, 2005, p. 402–411. Citado na página 44.

BANNOUR, B.; ESCOBEDO, J.; GASTON, C.; LE GALL, P. Off-Line Test Case Generation for Timed Symbolic Model-Based Conformance Testing. In: NIELSEN, B.; WEISE, C., eds. *Testing Software and Systems*, Lecture Notes in Computer Science, Springer Berlin/Heidelberg, p. 119–135, 2012. Citado 6 vezes nas páginas 66, 68, 69, 71, 72 e 149.

BANNOUR, B.; GASTON, C.; AIGUIER, M.; LAPITRE, A. Results for Compositional Timed Testing. In: *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, IEEE, 2013, p. 559–564. Citado 6 vezes nas páginas 65, 66, 68, 72, 73 e 149.

BARNETT, M.; GRIESKAMP, W.; NACHMANSON, L.; SCHULTE, W.; TILLMANN, N.; VEANES, M. Model-Based Testing with AsmL.NET. In: *1st European Conference on Model-Driven Software Engineering*, 2003. Citado na página 39.

BEEK, H.; MAUW, S. Automatic Conformance Testing of Internet Applications. In: PETRENKO, A.; ULRICH, A., eds. *Formal Approaches to Software Testing*, v. 2931 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1106–1106, 2004. Citado 6 vezes nas páginas 68, 69, 71, 72, 73 e 154.

BELINFANTE, A. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: ESPARZA, J.; MAJUMDAR, R., eds. *Tools and Algorithms for the Construction and Analysis of Systems*, v. 6015 de *Lecture Notes in Computer Science*, Springer Berlin/ Heidelberg, p. 266–270, 2010. Citado 3 vezes nas páginas 77, 96 e 106.

BENSALEM, S.; KRICHEN, M.; TRIPAKIS, S. State Identification Problems for Input/Output Transition Systems. In: *Proceedings of 9th WODES - International Workshop on Discrete Event Systems*, IEEE, 2008, p. 225–230. Citado 4 vezes nas páginas 66, 68, 81 e 150.

BERTOLINO, A. *Software Engineering Body of Knowledge*, v. 1. IEEE, 2004. Citado na página 42.

BHATEJA, P. Grammar Based Asynchronous Testing. In: *Proceedings of the 2nd India Software Engineering Conference*, New York, NY, USA: ACM, 2009, p. 105–110 (*ISEC '09*, v.1). Citado 3 vezes nas páginas 71, 73 e 150.

BHATEJA, P. Test Case Generation using PDA. In: *Proceedings of International Symposium on Theoretical Aspects of Software Engineering*, IEEE, 2011, p. 221–224. Citado 3 vezes nas páginas 69, 72 e 150.

BHATEJA, P. A TGV-like Approach for Asynchronous Testing. In: *ACM International Conference Proceeding Series*, DA-IICT, Gandhinagar, India: ACM, 2014. Citado 4 vezes nas páginas 68, 69, 73 e 150.

BHATEJA, P.; GASTIN, P.; MUKUND, M. A Fresh Look at Testing for Asynchronous Communication. In: GRAF, S.; ZHANG, W., eds. *Automated Technology for Verification and Analysis*, v. 4218 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 369–383, 2006. Citado 2 vezes nas páginas 73 e 150.

BI, J.; WU, J.; CHEN, X. A Concurrent TTCN based Approach to Conformance Testing of Distributed Routing Protocol OSPF v2. In: *Proceedings of International Conference on*

Computer Communications and Networks, IEEE, 1998, p. 760–767. Citado 5 vezes nas páginas 69, 71, 72, 73 e 150.

BIJL, M.; PEUREUX, F. 7 I/O-automata Based Testing. In: BROY, M.; JONSSON, B.; KATOEN, J.-P.; LEUCKER, M.; PRETSCHNER, A., eds. *Model-Based Testing of Reactive Systems*, v. 3472 de *Lecture Notes in Computer Science*, Springer Berlin /Heidelberg, p. 173–200, 2005. Citado 3 vezes nas páginas 29, 30 e 78.

BIJL, M.; RENSINK, A.; TRETMANS, J. Compositional Testing with ioco. In: PETRENKO, A.; ULRICH, A., eds. *Formal Approaches to Software Testing*, v. 2931 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1102–1102, 2004. Citado 6 vezes nas páginas 30, 54, 68, 69, 73 e 154.

BIJL, M.; RENSINK, A.; TRETMANS, J. Action Refinement in Conformance Testing. In: KHENDEK, F.; DSSOULI, R., eds. *Testing of Communicating Systems*, v. 3502 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 363–363, 2005. Citado 5 vezes nas páginas 68, 69, 72, 73 e 154.

BINDER, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. Citado 2 vezes nas páginas 36 e 38.

BLACKBURN, M.; BUSSER, R.; NAUMAN, A. Why Model-Based Test Automation is Different and What you Should Know to Get Started. In: *Proceedings of International Conference on Practical Software Quality and Testing*, 2004. Citado na página 41.

BOCHMANN, G. V.; PETRENKO, A. Protocol testing: Review of methods and relevance for software testing. In: *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, 1994, p. 109–124 (*ISSTA '94*, v.1). Citado 3 vezes nas páginas 28, 29 e 42.

BOLOGNESI, T.; BRINKSMA, E. Introduction to the ISO Specification Language LOTOS. *Computer Network ISDN Systems*, v. 14, n. 1, p. 25–59, 1987. Citado na página 51.

BORODAY, S.; PETRENKO, A.; ULRICH, A. Test Suite Consistency Verification. In: *East-West Design Test Symposium (EWDTS)*, 2008, p. 235–239. Citado 2 vezes nas páginas 71 e 150.

BOUQUET, F.; GRANDPIERRE, C.; LEGEARD, B.; PEUREUX, F.; VACELET, N.; UTTING, M. A Subset of Precise UML for Model-Based Testing. In: *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, New York, NY, USA: ACM, 2007, p. 95–104. Citado na página 39.

BOURDONOV, I. B.; KOSSATCHEV, A. S.; KULIAMIN, V. V. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. *Electronic Notes in Theoretical*

Computer Science, v. 164, n. 4, p. 83–96, MBT 2006, 2006. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 150.

BRANDL, H.; WEIGLHOFER, M.; AICHERNIG, B. Automated Conformance Verification of Hybrid Systems. In: *10th International Conference on Quality Software*, 2010, p. 3–12. Citado 6 vezes nas páginas 66, 68, 69, 72, 73 e 150.

BRAUNSTEIN, C.; HAXTHAUSEN, A. E.; HUANG, W.-L.; HÜBNER, F.; PELESKA, J.; SCHULZE, U.; VU HONG, L. Complete Model-Based Equivalence Class Testing for the ETCS Ceiling Speed Monitor. In: MERZ, S.; PANG, J., eds. *Formal Methods and Software Engineering*, v. 8829 de *Lecture Notes in Computer Science*, Springer International Publishing, p. 380–395, 2014a. Citado na página 110.

BRAUNSTEIN, C.; PELESKA, J.; SCHULZE, U.; HUBNER, F.; HUANG, W.-L.; HAXTHAUSEN, A. E.; VU, L. H. *A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor*. Tech Report 11025, Embedded Systems Testing Benchmarks Site, 2014b. Citado 2 vezes nas páginas 108 e 110.

BRIAND, L. A Critical Analysis of Empirical Research in Software Testing. In: *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, 2007, p. 1–8. Citado 3 vezes nas páginas 43, 44 e 99.

BRINKSMA, E.; HEERINK, L.; TRETMANS, J. Factorized Test Generation for Multi-Input/Output Transition Systems. In: PETRENKO, A. AND YEVTUSHENKO, N., ed. *Testing of Communicating Systems - IWTCS 98*, 1998, p. 67–82. Citado 7 vezes nas páginas 65, 68, 69, 70, 72, 73 e 150.

BRIONES, L.; BRINKSMA, E. A Test Generation Framework for Quiescent Real-Time Systems. In: GRABOWSKI, J.; NIELSEN, B., eds. *Formal Approaches to Software Testing*, v. 3395 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 64–78, 2005. Citado 8 vezes nas páginas 65, 66, 68, 69, 70, 71, 73 e 150.

BRIONES, L.; BRINKSMA, E.; STOELINGA, M. A Semantic Framework for Test Coverage. In: GRAF, S.; ZHANG, W., eds. *Automated Technology for Verification and Analysis*, v. 4218 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 399–414, 2006. Citado 5 vezes nas páginas 66, 68, 69, 72 e 150.

CALAMÉ, J. R.; IOUSTINOVA, N.; POL, J. Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. *Electronic Notes in Theoretical Computer Science*, v. 191, n. 0, p. 25–48, Proceedings of Doctoral Symposium of IFM 2005, 2007. Citado 5 vezes nas páginas 64, 65, 68, 69 e 150.

CARVALHO, G.; CARVALHO, A.; ROCHA, E.; CAVALCANTI, A.; SAMPAIO, A. A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In: *Lecture Notes in*

Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Luxembourg, 2014, p. 43–58. Citado 6 vezes nas páginas 62, 65, 66, 68, 73 e 150.

CAVALCANTI, A.; GAUDEL, M.-C.; HIERONS, R. Conformance Relations for Distributed Testing Based on CSP. In: WOLFF, B.; ZAIDI, F., eds. *Testing Software and Systems*, v. 7019 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 48–63, 2011. Citado 3 vezes nas páginas 68, 73 e 150.

CHÉDOR, S.; JÉRON, T.; MORVAN, C. Test Generation from Recursive Tiles Systems. In: BRUCKER, A.; JULLIAND, J., eds. *Tests and Proofs*, v. 7305 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 99–114, 2012. Citado 6 vezes nas páginas 66, 68, 69, 72, 73 e 150.

CHEN, L. Automatic Test Cases Generation for Statechart Specifications from Semantics to Algorithm. *Journal of Computers*, v. 6, n. 4, p. 769–775, 2011. Citado 3 vezes nas páginas 69, 72 e 150.

CHOW, T. S. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, v. 4, n. 3, p. 178–187, 1978. Citado 14 vezes nas páginas 29, 31, 45, 46, 47, 56, 79, 82, 83, 85, 86, 87, 89 e 126.

CLARKE, D.; JÉRON, T.; RUSU, V.; ZINOVIEVA, E. Automated Test and Oracle Generation for Smart-Card Applications. In: ATTALI, I.; JENSEN, T., eds. *Smart Card Programming and Security*, v. 2140 de *Lecture Notes in Computer Science*, Spring Berlin/Heidelberg, p. 58–70, 2001. Citado 7 vezes nas páginas 64, 65, 68, 69, 71, 72 e 151.

CUTIGI, J. F. *Uma Estratégia para Redução de Conjuntos de Sequências de Teste para Máquinas de Estados Finitos*. Dissertação de Mestrado, ICMC/Universidade de São Paulo, São Carlos, SP, Brazil, 2010. Citado na página 45.

DALAL, S. R.; JAIN, A.; KARUNANITHI, N.; LEATON, J. M.; LOTT, C. M.; PATTON, G. C.; HOROWITZ, B. M. Model-Based Testing in Practice. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, Los Angeles, USA: ACM, 1999, p. 285–294. Citado na página 41.

DAMASCENO, A. C.; MACHADO, P. D. L.; ANDRADE, W. L. Testing real-time systems from compositional symbolic specifications. *International Journal on Software Tools for Technology Transfer*, 2015. Citado 9 vezes nas páginas 62, 65, 66, 68, 69, 71, 72, 73 e 151.

DE LEÓN, H. P.; HAAR, S.; LONGUET, D.; DE LEON, H. P.; HAAR, S.; LONGUET, D. Unfolding-Based Test Selection for Concurrent Conformance. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in*

Bioinformatics), Istanbul, Turkey, 2013, p. 98–113 (*Lecture Notes in Computer Science*, v.8254). Citado 5 vezes nas páginas 69, 70, 72, 73 e 151.

DELAMARO, E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*, cáp. 1 Conceitos Básicos. Elsevier, p. 1–7, 2007. Citado 4 vezes nas páginas 27, 28, 37 e 38.

DEMILLO, R. A. *Software Testing and Evaluation*. The Benjamim/Commings Publishing Company, Inc, 1978. Citado na página 38.

DESMOULIN, A.; VIHO, C. Formalizing Interoperability for Test Case Generation Purpose. *International Journal on Software Tools for Technology Transfer*, v. 11, n. 3, p. 261–267, 2009. Citado 4 vezes nas páginas 68, 69, 72 e 151.

DIAS NETO, A. C.; SUBRAMANYAN, R.; VIEIRA, M.; TRAVASSOS, G. H. A Survey on Model-based Testing Approaches: A Systematic Review. In: *Proceedings of the 1st ACM Intern. Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM ASE 2007*, New York, NY, USA: ACM, 2007, p. 31–36 (*WEASELTech '07*, v.1). Citado na página 57.

DIAS NETO, A. C.; TRAVASSOS, G. H. Model-Based Testing Approaches Selection for Software Projects. *Information and Software Technology*, v. 51, n. 11, p. 1487 – 1504, Third IEEE International Workshop on Automation of Software Test (AST 2008) Eighth International Conference on Quality Software (QSIC 2008), 2009. Citado 2 vezes nas páginas 27 e 28.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, v. 10, n. 4, p. 405–435, 2005. Citado 3 vezes nas páginas 44, 45 e 63.

DOROFEEVA, R.; EL-FAKIH, K.; MAAG, S.; CAVALLI, A. R.; YEVTUSHENKO, N. FSM-based Conformance Testing Methods: A survey Annotated with Experimental Evaluation. *Information and Software Technology*, v. 52, n. 12, p. 1286–1297, 2010. Citado 4 vezes nas páginas 31, 48, 84 e 99.

DOROFEEVA, R.; EL-FAKIH, K.; YEVTUSHENKO, N. An Improved Conformance Testing Method. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, v. 3731 de *Lecture Notes in Computer Science*, Springer Berlin, p. 204–218, 2005. Citado 4 vezes nas páginas 29, 45, 47 e 48.

DYBA, T.; DINGSOYR, T.; HANSSEN, G. K. Applying Systematic Reviews to Diverse Study Types: An Experience Report. In: *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, IEEE, 2007, p. 225–234. Citado na página 59.

EL-FAR, I. K.; WHITTAKER, J. A. Model-Based Software Testing. In: *Encyclopedia on Software Engineering*, Wiley, p. 825–837, 2001. Citado 2 vezes nas páginas 39 e 41.

ENDO, A. T.; SIMAO, A. Evaluating Test Suite Characteristics, Cost, and Effectiveness of FSM-based Testing Methods. *Information and Software Technology*, v. 55, n. 6, p. 1045–1062, 2013. Citado 5 vezes nas páginas 31, 48, 99, 100 e 105.

FAIVRE, A.; GASTON, C.; LE GALL, P. Symbolic Model Based Testing for Component Oriented Systems. In: PETRENKO, A.; VEANES, M.; TRETMANS, J.; GRIESKAMP, W., eds. *Testing of Software and Communication Systems*, v. 4581 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 90–106, 2007. Citado 6 vezes nas páginas 65, 68, 69, 71, 73 e 151.

FALCONE, Y.; FERNANDEZ, J.-C.; JÉRON, T.; MARCHAND, H.; MOUNIER, L. More Testable Properties. *International Journal on Software Tools for Technology Transfer*, v. 14, n. 4, p. 407–437, 2012. Citado 3 vezes nas páginas 68, 69 e 151.

FERNANDEZ, J.-C.; MOUNIER, L.; PACHON, C. Property Oriented Test Case Generation. In: PETRENKO, A.; ULRICH, A., eds. *Formal Approaches to Software Testing*, v. 2931 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1101–1102, 2004. Citado 4 vezes nas páginas 68, 69, 72 e 151.

FERNANDEZ, J.-C.; MOUNIER, L.; PACHON, C. A Model-Based Approach for Robustness Testing. In: KHENDEK, F.; DSSOULI, R., eds. *Testing of Communicating Systems*, v. 3502 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 313–313, 2005. Citado 5 vezes nas páginas 68, 69, 70, 72 e 151.

FRANKL, P. G.; WEYUKER, E. J. A Formal Analysis of the Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering*, v. 19, n. 3, p. 202–213, 1993. Citado na página 43.

FRANKL, P. G.; WEYUKER, E. J. Testing Software to Detect and Reduce Risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000. Citado na página 37.

FRANTZEN, L.; TRETMANS, J. Model-Based Testing of Environmental Conformance of Components. In: BOER, F.; BONSANGUE, M.; GRAF, S.; ROEVER, W.-P., eds. *Formal Methods for Components and Objects*, v. 4709 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1–25, 2007. Citado 6 vezes nas páginas 68, 69, 71, 72, 73 e 151.

FRANTZEN, L.; TRETMANS, J.; WILLEMSE, T. Test Generation Based on Symbolic Specifications. In: GRABOWSKI, J.; NIELSEN, B., eds. *Formal Approaches to Software Testing*, v. 3395 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1–15, 2005. Citado 6 vezes nas páginas 65, 68, 69, 71, 72 e 151.

FRANTZEN, L.; TRETMANS, J.; WILLEMSE, T. A Symbolic Framework for Model-Based Testing. In: HAVELUND, K.; NÚÑEZ, M.; ROSU, G.; WOLFF, B., eds. *Formal Approaches to Software Testing and Runtime Verification*, v. 4262 de *Lecture Notes in Computer Science*,

Springer Berlin/Heidelberg, p. 40–54, 2006. Citado 8 vezes nas páginas 65, 68, 69, 70, 71, 72, 73 e 151.

FRASER, G.; WEIGLHOFER, M.; WOTAWA, F. Coverage Based Testing with Test Purposes. In: *Proceedings of International Conference on Quality Software*, 2008, p. 199–208. Citado 5 vezes nas páginas 59, 68, 69, 72 e 151.

FU, Y.; KONÉ, O. A Robustness Testing Method for Network Security. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, v. 99, p. 38–45, 2012. Citado 4 vezes nas páginas 68, 69, 72 e 151.

FU, Y.; KONÉ, O. Model based security verification of protocol implementation. In: *Journal of Information Security and Applications*, University of PAU, IUT de Mont de Marsan, France, 2015, p. 17–27. Citado 4 vezes nas páginas 69, 72, 73 e 151.

FUJIWARA, S.; BOCHMANN, G.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, v. 17, n. 6, p. 591–603, 1991. Citado 4 vezes nas páginas 29, 47, 48 e 84.

GASTON, C.; LE GALL, P.; RAPIN, N.; TOUIL, A. Symbolic Execution Techniques for Test Purpose Definition. In: UYAR, M.; DUALE, A.; FECKO, M., eds. *Testing of Communicating Systems*, v. 3964 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1–18, 2006. Citado 7 vezes nas páginas 65, 68, 69, 71, 72, 73 e 151.

GAUDEL, M.-C. Testing can be formal, too. In: MOSSES, P. D.; NIELSEN, M.; SCHWARTZBACH, M. I., eds. *TAPSOFT 95: Theory and Practice of Software Development*, v. 915 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 82–96, 1995. Citado na página 29.

GAUDEL, M.-C. Software Testing Based on Formal Specification. In: BORBA, P.; CAVALCANTI, A.; SAMPAIO, A.; WOODCOCK, J., eds. *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE 2007, December 3-7, 2007, Revised Lectures*, v. 6153 de *Lecture Notes in Computer Science*, Springer, p. 215–242, 2010a. Citado 2 vezes nas páginas 28 e 54.

GAUDEL, M.-C. Software Testing Based on Formal Specification. In: BORBA, P.; CAVALCANTI, A.; SAMPAIO, A.; WOODCOOK, J., eds. *Testing Techniques in Software Engineering*, v. 6153 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 215–242, 2010b. Citado 7 vezes nas páginas 30, 42, 43, 51, 69, 73 e 151.

GILL, A. *Introduction to the Theory of Finite-State Machines*. New York: McGraw-Hill, 1962. Citado na página 45.

GNESI, S.; LATELLA, D.; MASSINK, M. Formal Test-Case Generation for UML Statecharts. In: *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, 2004, p. 75–84. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 151.

GONZÁLEZ, J. E.; JURISTO, N.; VEGAS, S. A Systematic Mapping Study on Testing Technique Experiments: Has the Situation Changed Since 2000? In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA: ACM, 2014, p. 3:1–3:4 (*ESEM '14*, v.1). Citado 2 vezes nas páginas 44 e 99.

GOODENOUGH, J. B.; GERHART, S. L. Toward a Theory of Test Data Selection. In: *Proceedings of the International Conference on Reliable Software*, ACM, 1975, p. 493–510. Citado na página 42.

GROMOV, M.; WILLEMSE, T. Testing and Model-Checking Techniques for Diagnosis. In: PETRENKO, A.; VEANES, M.; TRETMANS, J.; GRIESKAMP, W., eds. *Testing of Software and Communication Systems*, v. 4581 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 138–154, 2007. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 151.

HAO, R.; WU, J. Toward formal TTCN-based test execution. In: *INFOCOM '97. 16th Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, 1997, p. 230–235. Citado 4 vezes nas páginas 68, 69, 71 e 152.

HAO, R.; WU, J. A Formal Approach to Protocol Interoperability Testing. *Journal of Computer Science and Technology*, v. 13, n. 1, p. 79–90, 1998. Citado 4 vezes nas páginas 68, 69, 72 e 152.

HEERINK, L.; TRETMANS, J. Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In: *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, London, UK, UK: Chapman & Hall, Ltd., 1998, p. 23–38. Citado 2 vezes nas páginas 52 e 54.

HELOVUO, J.; LEPPANEN, S. Exploration testing. In: *Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on*, IEEE, 2001, p. 201–210. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 152.

HESSEL, A.; LARSEN, K.; MIKUCIONIS, M.; NIELSEN, B.; PETTERSSON, P.; SKOU, A. Testing Real-Time Systems Using UPPAAL. In: HIERONS, R.; BOWEN, J.; HARMAN, M., eds. *Formal Methods and Testing*, v. 4949 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 77–117, 2008. Citado 10 vezes nas páginas 30, 64, 65, 66, 69, 70, 71, 72, 73 e 152.

HIERONS, R. Testing in the Distributed Test Architecture: An Extended Abstract. In: *8th International Conference on Quality Software. QSIC '08.*, 2008, p. 11–14. Citado 4 vezes nas páginas 66, 68, 73 e 152.

HIERONS, R.; MERAYO, M.; NÚÑEZ, M. Scenarios-Based Testing of Systems with Distributed Ports. *Software-Practice and Experience*, v. 41, n. 10, p. 999–1026, 2011. Citado 3 vezes nas páginas 68, 73 e 152.

HIERONS, R.; MERAYO, M.; NÚÑEZ, M. Using Time to Add Order to Distributed Testing. In: GIANNAKOPOULOU, D.; MÉRY, D., eds. *FM 2012: Formal Methods*, v. 7436 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 232–246, 2012a. Citado 4 vezes nas páginas 66, 68, 73 e 152.

HIERONS, R.; NÚÑEZ, M. Testing Probabilistic Distributed Systems. In: HATCLIFF, J.; ZUCCA, E., eds. *Formal Techniques for Distributed Systems*, v. 6117 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 63–77, 2010. Citado 3 vezes nas páginas 73, 74 e 152.

HIERONS, R.; NÚÑEZ, M. Using Schedulers to Test Probabilistic Distributed Systems. *Formal Aspects of Computing*, v. 24, n. 4, p. 679–699, 2012. Citado 3 vezes nas páginas 73, 74 e 152.

HIERONS, R. M. The Complexity of Asynchronous Model Based Testing. *Theoretical Computer Science*, v. 451, n. 14, p. 70 – 82, 2012a. Citado 13 vezes nas páginas 31, 49, 59, 64, 66, 67, 68, 72, 73, 77, 78, 80 e 152.

HIERONS, R. M. Overcoming Controllability Problems in Distributed Testing from an Input Output Transition System. *Distributed Computing*, v. 25, n. 1, p. 63–81, 2012b. Citado 5 vezes nas páginas 65, 68, 69, 73 e 152.

HIERONS, R. M. Implementation Relations for Testing Through Asynchronous Channels. *The Computer Journal*, v. 56, n. 11, p. 1305–1319, 2013. Citado 7 vezes nas páginas 31, 42, 49, 59, 73, 78 e 152.

HIERONS, R. M. A More Precise Implementation Relation for Distributed Testing. *The Computer Journal*, p. bxv057, 2015. Citado 6 vezes nas páginas 64, 65, 68, 70, 73 e 152.

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LÜTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using Formal Specifications to Support Testing. *ACM Computer Surveys*, v. 41, n. 2, p. 9:1–9:76, 2009. Citado 11 vezes nas páginas 27, 28, 29, 30, 31, 39, 42, 67, 70, 75 e 77.

HIERONS, R. M.; MERAYO, M.; NÚÑEZ, M. Controllable Test Cases for the Distributed Test Architecture. In: CHA, S.; CHOI, J.-Y.; KIM, M.; LEE, I.; VISWANATHAN, M., eds.

Automated Technology for Verification and Analysis, v. 5311 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 201–215, 2008. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 152.

HIERONS, R. M.; MERAYO, M.; NÚÑEZ, M. Implementation Relations and Test Generation for Systems with Distributed Interfaces. *Distributed Computing*, v. 25, n. 1, p. 35–62, 2012b. Citado 6 vezes nas páginas 68, 69, 72, 73, 74 e 152.

HIERONS, R. M.; MERAYO, M. G.; NÚÑEZ, M. Timed Implementation Relations for the Distributed Test Architecture. *Distributed Computing*, v. 27, n. 3, p. 181–201, 2014. Citado 5 vezes nas páginas 64, 65, 68, 73 e 152.

HIERONS, R. M.; URAL, H. Optimizing the Length of Checking Sequences. *IEEE Transactions on Computers*, v. 55, n. 5, p. 618–629, 2006. Citado 3 vezes nas páginas 29, 46 e 48.

HIERONS, R. M.; URAL, H. Generating a Checking Sequence with a Minimum Number of Reset Transitions. *Automated Software Engineering*, v. 17, n. 3, p. 217–250, 2010. Citado na página 29.

HUO, J.; PETRENKO, A. On Testing Partially Specified IOTS through Lossless Queues. In: GROZ, R.; HIERONS, R., eds. *Testing of Communicating Systems*, v. 2978 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 1–11, 2004. Citado 8 vezes nas páginas 64, 67, 68, 69, 72, 73, 74 e 152.

HUO, J.; PETRENKO, A. Covering Transitions of Concurrent Systems through Queues. In: *16th IEEE International Symposium on Software Reliability Engineering*, IEEE, 2005, p. 334–345. Citado 8 vezes nas páginas 67, 68, 69, 72, 73, 74, 78 e 152.

HUO, J.; PETRENKO, A. Transition Covering Tests for Systems with Queues. *Software Testing, Verification and Reliability*, v. 19, n. 1, p. 55–83, 2009. Citado 3 vezes nas páginas 31, 49 e 77.

HWANG, I.; YEVTUSHENKO, N.; CAVALLI, A. R. Tight Bound on the Length of Distinguishing Sequences for non-observable nondeterministic Finite-State Machines with a Polynomial Number of Inputs and Outputs. *Information Processing Letters*, v. 112, n. 7, p. 298–301, 2012. Citado 2 vezes nas páginas 29 e 45.

IEEE IEEE Standard Glossary of Software Engineering Terminology. In: *Padrão 620.12*, IEEE, p. 1–84, 1990. Citado na página 36.

ISO ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes. *IEEE STD 12207-2008*, p. c1–138, 2008. Citado na página 35.

ISO/IEC *ISO: Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807, Geneve, 1989. Citado na página 51.

JARD, C.; JÉRON, T. TGV: Theory, Principles and Algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, v. 7, n. 4, p. 297–315, 2005. Citado 16 vezes nas páginas 30, 31, 41, 51, 52, 53, 54, 64, 65, 68, 69, 70, 71, 72, 77 e 153.

JEANNET, B.; JÉRON, T.; RUSU, V. Model-Based Test Selection for Infinite-State Reactive Systems. In: BOER, F.; BONSANGUE, M.; GRAF, S.; ROEVER, W.-P., eds. *Formal Methods for Components and Objects*, v. 4709 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 47–69, 2007. Citado 9 vezes nas páginas 30, 65, 68, 69, 70, 71, 72, 73 e 153.

JÉRON, T. Symbolic Model-based Test Selection. *Electronic Notes in Theoretical Computer Science*, v. 240, p. 167–184, 2009. Citado 7 vezes nas páginas 65, 68, 69, 71, 72, 73 e 153.

JÉRON, T.; MOREL, P. Test Generation Derived from Model-Checking. In: HALBWACHS, N.; PELED, D., eds. *Computer Aided Verification*, v. 1633 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 108–122, 1999. Citado 6 vezes nas páginas 64, 65, 68, 69, 72 e 153.

JURISTO, N.; MORENO, A. M.; VEGAS, S. Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering*, v. 9, n. 1-2, p. 7–44, 2004. Citado 2 vezes nas páginas 43 e 44.

KANER, C.; FALK, J. L.; NGUYEN, H. Q. *Testing Computer Software*. 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. Citado na página 40.

KERVINEN, A.; VIROLAINEN, P. Heuristics for Faster Error Detection with Automated Black Box Testing. *Electronic Notes in Theoretical Computer Science*, v. 111, n. 1, p. 53–71, MBT 2004, 2005. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 153.

KITCHENHAM, B. *Procedures for Performing Systematic Reviews*. Technical Report TR/SE-0401, Keele University and NICTA, 2004. Citado na página 43.

KONÉ, O.; CASTANET, R. Test Generation for Interworking Systems. *Computer Communications*, v. 23, n. 7, p. 642–652, 2000. Citado 4 vezes nas páginas 68, 69, 72 e 153.

KRICHEN, M. A Formal Framework for Conformance Testing of Distributed Real-Time Systems. In: *Proceedings of the 14th International Conference on Principles of Distributed Systems*, Berlin, Heidelberg: Springer-Verlag, 2010, p. 139–142. Citado na página 54.

KUSHIK, N.; EL-FAKIH, K.; YEVTUSHENKO, N.; CAVALLI, A. R. On Adaptive Experiments for Nondeterministic Finite State Machines. *International Journal on Software Tools for Technology Transfer*, p. 1–14, 2014. Citado na página 29.

LE GALL, P.; RAPIN, N.; TOUIL, A. Symbolic Execution Techniques for Refinement Testing. In: GUREVICH, Y.; MEYER, B., eds. *Tests and Proofs*, v. 4454 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 131–148, 2007. Citado 6 vezes nas páginas 65, 68, 69, 71, 72 e 153.

LEE, D.; YANNAKAKIS, M. Principles and Methods of Testing Finite State Machines - a Survey. *Proceedings of the IEEE*, v. 84, n. 8, p. 1090–1123, 1996. Citado 2 vezes nas páginas 28 e 45.

LESTIENNES, G.; GAUDEL, M.-C. Testing Processes from Formal Specifications with Inputs, Outputs and Data Types. In: *Proceedings of 13th International Symposium on Software Reliability Engineering*, 2002, p. 3–14. Citado 5 vezes nas páginas 68, 69, 72, 73 e 153.

LI, Z.; WU, J.; YIN, X. Refusal Testing for MIOTS with Nonlockable Output Channels. In: *Proceedings of International Conference on Communications, Networking and Mobile Computing*, Washington, DC, USA: IEEE, 2003, p. 517–522. Citado 7 vezes nas páginas 65, 66, 68, 69, 72, 73 e 153.

LI, Z.; WU, J.; YIN, X. Testing Multi Input/Output Transition System with All-Observer. In: GROZ, R.; HIERONS, R., eds. *Testing of Communicating Systems*, v. 2978 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 2705–2705, 2004a. Citado 7 vezes nas páginas 65, 66, 68, 69, 72, 73 e 153.

LI, Z.; YIN, X.; WU, J. Distributed Testing of Multi Input/Output Transition System. In: *Proceedings of the 2nd Software Engineering and Formal Methods*, 2004b, p. 271–280. Citado 6 vezes nas páginas 65, 68, 69, 72, 73 e 153.

LITY, S.; LACHMANN, R.; LOCHAU, M.; SCHAEFER, I. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. Technical Report 2012-07, TU Braunschweig, 2013. Citado 2 vezes nas páginas 109 e 113.

LUO, G.; PETRENKO, A.; BOCHMANN, G. V. Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines. In: *7th IFIP WG 6.1 International Workshop on Protocol Test Systems*, London, UK: Chapman & Hall, Ltd., 1995, p. 95–110. Citado 2 vezes nas páginas 29 e 47.

MACHADO, P. D.; SILVA, D. A.; MOTA, A. C. Towards Property Oriented Testing. *Electronic Notes in Theoretical Computer Science*, v. 184, n. 12, p. 3–19, SBMF 2005, 2007. Citado 5 vezes nas páginas 62, 68, 69, 71 e 153.

MALDONADO, J. C. *Critérios Potenciais Usos : Uma Contribuição ao Teste Estrutural de Software*. Tese de Doutoramento, FEEC/Unicamp, 1991. Citado 3 vezes nas páginas 36, 37 e 38.

MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. *Introdução ao Teste de Software*. Relatório Técnico 65, Série Computação, ICMC/USP, São Carlos, notas didáticas do ICMC, 2004. Citado 2 vezes nas páginas 28 e 43.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. R. S.; DELAMARO, M. E. *Aspectos Teóricos e Empíricos de Teste de Cobertura de Software*. Technical Report 31, ICMC/USP, 1998. Citado na página 38.

MATHUR, A. P. *Foundations of Software Testing*. 1st ed. Addison-Wesley Professional, 2008. Citado 2 vezes nas páginas 36 e 44.

MUCCINI, H.; BERTOLINO, A.; INVERARDI, P. Using Software Architecture for Code Testing. *IEEE Transactions on Software Engineering*, v. 30, n. 3, p. 160–171, 2004. Citado 5 vezes nas páginas 66, 68, 69, 72 e 153.

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. R. *The Art of Software Testing*. John Wiley & Sons, 2004. Citado 3 vezes nas páginas 35, 36 e 38.

NOROOZI, N.; KHOSRAVI, R.; MOUSAVI, M.; WILLEMSE, T. Synchronizing Asynchronous Conformance Testing. In: BARTHE, G.; PARDO, A.; SCHNEIDER, G., eds. *Software Engineering and Formal Methods*, v. 7041 de *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 334–349, 2011. Citado 3 vezes nas páginas 30, 73 e 153.

NOROOZI, N.; KHOSRAVI, R.; MOUSAVI, M. R.; WILLEMSE, T. A. C. Synchrony and Asynchrony in Conformance Testing. *Software & Systems Modeling*, v. 14, n. 1, p. 149–172, 2013. Citado 6 vezes nas páginas 49, 64, 66, 68, 73 e 153.

PAIVA, S. C.; SIMAO, A. Generation of complete test suites from mealy input/output transition systems. *Formal Aspects of Computing*, v. 28, n. 1, p. 65–78, 2016. Citado 2 vezes nas páginas 33 e 127.

PAIVA, S. L. C.; SIMAO, A. A Systematic Mapping Study on Test Generation from Input/Output Transition Systems. In: *Software Engineering and Advanced Applications (SEAA), 2015 41th EUROMICRO Conference on*, IEEE Computer Society, 2015. Citado 4 vezes nas páginas 33, 57, 77 e 127.

PARDO, J.; NÚÑEZ, M.; RUIZ, M. Specification and Testing of E-Commerce Agents Described by Using UIOLTSs. In: HATCLIFF, J.; ZUCCA, E., eds. *Formal Techniques for Distributed Systems*, v. 6117 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 78–86, 2010. Citado 7 vezes nas páginas 65, 66, 68, 69, 72, 73 e 153.

PETRENKO, A.; YEVTUSHENKO, N. Adaptive Testing of Nondeterministic Systems with FSM. In: *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, 2014, p. 224–228. Citado na página 29.

PELESKA, J.; HONISCH, A.; LAPSCHIES, F.; LODING, H.; SCHMID, H.; SMUDA, P.; VOROBEV, E.; ZAHLTEN, C. A Real-world Benchmark Model for Testing Concurrent Real-time Systems in the Automotive Domain. In: *Proceedings of the 23rd IFIP WG 6.1 International Conference on Testing Software and Systems*, Berlin, Heidelberg: Springer-Verlag, 2011, p. 146–161 (*ICTSS'11*, v.1). Citado 2 vezes nas páginas 109 e 112.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic Mapping Studies in Software Engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, Swinton, UK: British CS., 2008, p. 68–77 (*EASE'08*, v.17). Citado 2 vezes nas páginas 57 e 58.

PETRENKO, A.; BOCHMANN, G.; YAO, M. On Fault Coverage of Tests for Finite State Specifications. *Computer Networks and {ISDN} Systems*, v. 29, n. 1, p. 81–106, protocol Testing, 1996. Citado na página 48.

PETRENKO, A.; SIMAO, A. Generalizing the ds-methods for testing non-deterministic fsms. *The Computer Journal*, v. 58, n. 7, p. 1656–1672, 2015. Citado 2 vezes nas páginas 29 e 45.

PETRENKO, A.; YEVTUSHENKO, N. Queued Testing of Transition Systems with Inputs and Outputs. In: HIERONS, R.; JERON, T., eds. *Proceedings of Workshop on Formal Approaches to Testing of Software, FATES'02, CONCUR'02.*, 2002. Citado 3 vezes nas páginas 29, 42 e 50.

PETRENKO, A.; YEVTUSHENKO, N. Testing from Partial Deterministic FSM Specifications. *IEEE Transactions on Computers*, v. 54, n. 9, p. 1154–1165, 2005. Citado 2 vezes nas páginas 29 e 45.

PETRENKO, A.; YEVTUSHENKO, N. Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs. In: *23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*, Springer Berlin Heidelberg, 2011, p. 162–178 (*Lecture Notes in Computer Science*, v.7019). Citado na página 29.

PETRENKO, A.; YEVTUSHENKO, N.; HUO, J. Testing Transition Systems with Input and Output Testers. In: HOGREFE, D.; WILES, A., eds. *Testing of Communicating Systems*, v. 2644 de *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 609–609, 2003. Citado 11 vezes nas páginas 29, 49, 51, 53, 54, 67, 68, 69, 72, 73 e 153.

PICKIN, S.; JARD, C.; JERON, T.; JEZEQUEL, J.; LE TRAON, Y. Test Synthesis from UML Models of Distributed Software. *Software Engineering, IEEE Transactions on*, v. 33, n. 4, p. 252–269, 2007. Citado 6 vezes nas páginas 64, 68, 69, 72, 73 e 153.

PRESSMAN, R. *Engenharia de Software*. 5 ed. McGraw-Hill, 2003. Citado 4 vezes nas páginas 27, 35, 36 e 37.

PRETSCHNER, A.; PHILIPPS, J. Methodological Issues in Model-Based Testing. In: BROY, M.; JONSSON, B.; KATOEN, J.-P.; LEUCKER, M.; PRETSCHNER, A., eds. *Model-Based Testing of Reactive Systems*, v. 3472 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 281–291, 2005. Citado 2 vezes nas páginas 38 e 39.

RAMAMOORTHY, C.; BASTANI, F. Software Reliability - Status and Perspectives. *IEEE Transactions on Software Engineering*, v. 8, n. 4, p. 354–271, 1982. Citado na página 38.

RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data using Data Flow Information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985. Citado na página 43.

ROCHA, A. R.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de Software: Teoria e Prática*. 1 ed. Prentice-Hall, 2001. Citado 4 vezes nas páginas 27, 28, 36 e 44.

ROLLET, A.; SAAD-KHORCHEF, F. A Formal Approach to Test the Robustness of Embedded Systems using Behaviour Analysis. In: *Proceedings of International Conference on Software Engineering Research, Management and Applications*, IEEE, 2007, p. 667–674. Citado 3 vezes nas páginas 66, 68 e 154.

ROLLET, A.; SALVA, S. Two Complementary Approaches to Test Robustness of Reactive Systems. In: *IEEE International Conference on Automation, Quality and Testing, Robotics*, IEEE, 2008, p. 47–53. Citado 5 vezes nas páginas 68, 69, 72, 73 e 154.

ROLLET, A.; SALVA, S. Testing Robustness of Communicating Systems using ioco-based Approach. In: *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, IEEE Computer Society, 2009, p. 67–72. Citado 6 vezes nas páginas 68, 69, 70, 72, 73 e 154.

RUSU, V.; MARCHAND, H.; JÉRON, T. Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems. In: FITZGERALD, J.; HAYES, I. J.; TARLECKI, A., eds. *FM 2005: Formal Methods*, v. 3582 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 189–204, 2005. Citado 6 vezes nas páginas 65, 68, 69, 71, 72 e 154.

SAMPAIO, A.; NOGUEIRA, S.; MOTA, A. Compositional Verification of Input-Output Conformance via CSP Refinement Checking. In: BREITMAN, K.; CAVALCANTI, A., eds. *Formal Methods and Software Engineering*, v. 5885 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 20–48, 2009. Citado 5 vezes nas páginas 62, 68, 69, 72 e 154.

SCHMALTZ, J.; TRETMANS, J. On Conformance Testing for Timed Systems. In: CASSEZ, F.; JARD, C., eds. *Formal Modeling and Analysis of Timed Systems*, v. 5215 de *Lecture Notes in*

Computer Science, Springer Berlin/Heidelberg, p. 250–264, 2008. Citado 4 vezes nas páginas 65, 66, 68 e 154.

SCOLLO, G.; ZECCHINI, S. Architectural Unit Testing. *Electronic Notes in Theoretical Computer Science*, v. 111, n. 1, p. 27–52, MBT 2004, 2005. Citado 5 vezes nas páginas 64, 68, 69, 72 e 154.

SHAFIQUE, M.; LABICHE, Y. A Systematic Review of State-based Test Tools. *International Journal on Software Tools for Technology Transfer*, v. 17, n. 1, p. 59–76, 2013. Citado na página 57.

SHI, J.; WAN, J.; YAN, H.; SUO, H. A survey of Cyber-Physical Systems. In: *Wireless Communications and Signal Processing (WCSP), 2011 International Conference on*, 2011, p. 1–6. Citado na página 106.

SIAVASHI, F.; TRUSCAN, D. Environment Modeling in Model-based Testing: Concepts, Prospects and Research Challenges: A Systematic Literature Review. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA: ACM, 2015, p. 30:1–30:6 (*EASE '15*, v.1). Citado na página 57.

SIMAO, A. *Introdução ao Teste de Software*, cáp. Teste Baseado em Modelos Elsevier, p. 27–45, 2007. Citado 4 vezes nas páginas 28, 31, 45 e 46.

SIMAO, A.; PETRENKO, A. Generating Checking Sequences for Partial Reduced Finite State Machines. In: *Proceedings of the 20th IFIP International Conference on Testing of Software and Communicating Systems*, Springer-Verlag, 2008, p. 153–168. Citado na página 48.

SIMAO, A.; PETRENKO, A. Checking Completeness of Tests for Finite State Machines. *IEEE Transactions on Computers*, v. 59, n. 8, p. 1023–1032, 2010a. Citado 5 vezes nas páginas 29, 79, 82, 83 e 84.

SIMAO, A.; PETRENKO, A. Fault Coverage-Driven Incremental Test Generation. *The Computer Journal*, v. 53, n. 9, p. 1508–1522, 2010b. Citado 7 vezes nas páginas 28, 29, 31, 48, 49, 81 e 84.

SIMAO, A.; PETRENKO, A. Generating Asynchronous Test Cases from Test Purposes. *Information and Software Technology*, v. 53, n. 11, p. 1252–1262, AMOST 2010, 2011. Citado 12 vezes nas páginas 49, 50, 59, 62, 67, 68, 69, 73, 77, 78, 80 e 154.

SIMAO, A.; PETRENKO, A. Generating Complete and Finite Test Suite for ioco: Is it Possible? In: *Electronic Proceedings in Theoretical Computer Science, EPTCS*, Grenoble, France, 2014, p. 56–70. Citado 14 vezes nas páginas 31, 49, 62, 64, 67, 68, 69, 72, 73, 74, 77, 78, 81 e 154.

SIMAO, A.; PETRENKO, A.; MALDONADO, J. Comparing Finite State Machine Test. *IET Software*, v. 3, n. 2, p. 91–105, 2009a. Citado 3 vezes nas páginas 48, 99 e 101.

SIMAO, A.; PETRENKO, A.; YEVTUSHENKO, N. Generating Reduced Tests for FSMs with Extra States. In: *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, Berlin, Heidelberg: Springer-Verlag, 2009b, p. 129–145. Citado 2 vezes nas páginas 45 e 47.

SINHA, A.; SMIDTS, C. A Model-Based Test Design Technique for Enhanced Testing of Domain-Specific Applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 15, n. 3, p. 242–278, 2006. Citado na página 38.

SOMERVILLE, I. *Engenharia de Software*. 8 ed. Addison-Wesley, 2007. Citado na página 36.

SPERO, S. E. SCP - Session Control Protocol V 1.1, Available in: http://www.ibiblio.org/ses/scp.html, 2007. Citado na página 93.

TRETMANS, J. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, v. 17, n. 3, p. 103–120, 1996. Citado 2 vezes nas páginas 30 e 54.

TRETMANS, J. Model Based Testing with Labelled Transition Systems. In: HIERONS, R. M.; BOWEN, J. P.; HARMAN, M., eds. *Formal Methods and Testing*, v. 4949 de *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 1–38, 2008. Citado 37 vezes nas páginas 27, 28, 29, 30, 31, 32, 33, 38, 39, 41, 42, 49, 50, 51, 52, 53, 54, 55, 56, 59, 64, 69, 70, 71, 72, 73, 75, 77, 79, 81, 96, 97, 99, 106, 119, 126 e 154.

TRETMANS, J. Model-Based Testing and Some Steps towards Test-Based Modelling. In: BERNARDO, M.; ISSARNY, V., eds. *Formal Methods for Eternal Networked Software Systems*, v. 6659 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 297–326, 2011. Citado 2 vezes nas páginas 68 e 154.

TRETMANS, J.; BRINKSMA, H. TorX: Automated Model-Based Testing. In: HARTMAN, A.; DUSSA-ZIEGLER, K., eds. *1st European Conference on Model-Driven Software Engineering*, 2003, p. 31–43. Citado 2 vezes nas páginas 77 e 96.

URAL, H.; WU, X.; ZHANG, F. On Minimizing the Lengths of Checking Sequences. *IEEE Transactions on Computers*, v. 46, n. 1, p. 93–99, 1997. Citado 2 vezes nas páginas 46 e 48.

URAL, H.; ZHANG, F. Reducing the Lengths of Checking Sequences by Overlapping. In: *Proceedings of the 18th IFIP TC6/WG6.1 international conference on Testing of Communicating Systems*, Berlin, Heidelberg: Springer-Verlag, 2006, p. 274–288. Citado 2 vezes nas páginas 29 e 48.

UTTING, M.; LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. Citado 5 vezes nas páginas 38, 39, 40, 41 e 42.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, v. 22, n. 5, p. 297–312, 2012. Citado 9 vezes nas páginas 27, 28, 33, 39, 42, 57, 58, 60 e 66.

VINCENZI, A. M. R. *Orientação a Objeto: Definição e Análise de Recursos de Teste e Validação*. Tese de doutorado, ICMC/USP, São Carlos, SP, 2004. Citado na página 36.

WEIGLHOFER, M.; AICHERNIG, B. Unifying Input Output Conformance. In: BUTTERFIELD, A., ed. *Unifying Theories of Programming*, v. 5713 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 181–201, 2010. Citado 3 vezes nas páginas 30, 66 e 154.

WEIGLHOFER, M.; FRASER, G.; WOTAWA, F. Using Coverage to Automate and Improve Test Purpose Based Testing. *Information and Software Technology*, v. 51, n. 11, p. 1601–1617, AST 2008 - QSIC 2008, 2009. Citado 5 vezes nas páginas 68, 69, 72, 77 e 154.

WEIGLHOFER, M.; WOTAWA, F. Asynchronous Input-Output Conformance Testing. In: *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, 2009, p. 154–159. Citado 9 vezes nas páginas 30, 54, 64, 65, 68, 69, 72, 73 e 154.

WESSA, P. Spearman rank correlation (v1.0.1) in free statistics software (v1.1.23-r7). Acessado em Novembro, 2015, 2012.
Disponivel em: `http://www.wessa.net/rwasp_spearman.wasp/` Citado na página 101.

WILLEMSE, T. Heuristics for ioco-Based Test-Based Modelling. In: BRIM, L.; HAVERKORT, B.; LEUCKER, M.; POL, J., eds. *Formal Methods: Applications and Technology*, v. 4346 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 132–147, 2007. Citado 2 vezes nas páginas 68 e 155.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in Software Engineering: an Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. Citado na página 43.

WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. *Mutation versus All-Uses: An Empirical Evaluation of Cost, Strength and Effectiveness*, cáp. 40 London: Chapmann & Hall, p. 258–265, 1995. Citado na página 44.

XIE, G.; DANG, Z. Testing Systems of Concurrent Black-Boxes – An Automata-Theoretic and Decompositional Approach. In: GRIESKAMP, W.; WEISE, C., eds. *Formal Approaches to Software Testing*, v. 3997 de *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, p. 170–186, 2006. Citado 5 vezes nas páginas 65, 68, 69, 71 e 155.

ZHU, H. A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria. *IEEE Transactions on Software Engineering*, v. 22, n. 4, p. 248–255, 1996. Citado na página 43.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR)*, v. 29, n. 4, p. 366–427, 1997. Citado na página 37.

ZUROWSKA, K.; DINGEL, J. *Model-based Generation of Test Cases for Reactive Systems*. Technical Report 2010- 573, School of Computing/Queen's University, 2010. Citado 2 vezes nas páginas 86 e 87.

APÊNDICE

A

ESTUDOS SELECIONADOS NO MAPEAMENTO SISTEMÁTICO

Esta seção apresenta uma descrição dos 96 estudos selecionados após a condução do mapeamento sistemático. A Tabela 11 apresenta a identificação, título, autores e ano de publicação para cada estudo.

Tabela 11 – Estudos selecionados

ID	Título	Autores	Ano
1	From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems (Aichernig e Delgado, 2006)	Aichernig, B.; Delgado, C.	2006
2	Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods (Aichernig et al., 2007)	Aichernig, B.; Peischl, B.; Weiglhofer, M.; Wotawa, F.	2007
3	Improving Fault-based Conformance Testing (Aichernig et al., 2008)	Aichernig, B. K.; Weiglhofer, M.; Wotawa, F.	2008
4	A formal abstract framework for modelling and testing complex software systems (Aiguier et al., 2012)	Aiguier, M.; Boulanger, F.; Kanso, B.	2012
5	Testing interruptions in reactive systems (Andrade e Machado, 2012)	Andrade, W.; Machado, P.	2012
6	Off-line Test Case Generation for Timed Symbolic Model-based Conformance Testing (Bannour et al., 2012)	Bannour, B.; Escobedo, J.; Gaston, C.; Le Gall, P.	2012
7	Results for Compositional Timed Testing (Bannour et al., 2013)	Bannour, B.; Gaston, C.; Aiguier, M.; Lapitre, A.	2013

8	State identification problems for input/output transition systems (Bensalem et al., 2008)	Bensalem, S.; Krichen, M.; Tripakis, S.	2008
9	A Fresh Look at Testing for Asynchronous Communication (Bhateja et al., 2006)	Bhateja, P.; Gastin, P.; Mukund, M.	2006
10	Grammar based asynchronous testing (Bhateja, 2009)	Bhateja, P.	2009
11	Test case generation using PDA (Bhateja, 2011)	Bhateja, P.	2011
12	A TGV-like Approach for Asynchronous Testing (Bhateja, 2014)	Bhateja, P.	2014
13	A concurrent TTCN based approach to conformance testing of distributed routing protocol OSPF v2 (Bi et al., 1998)	Bi, J.; Wu, J.; Chen, X.	1998
14	Test suite consistency verification (Boroday et al., 2008)	Boroday, S.; Petrenko, A.; Ulrich, A.	2008
15	Formal Conformance Testing of Systems with Refused Inputs and Forbbiden Actions (Bourdonov et al., 2006)	Bourdonov, I. B.; Kossatchev, A. S.; Kuliamin, V. V.	2006
16	Automated Conformance Verification of Hybrid Systems (Brandl et al., 2010)	Brandl, H.; Weiglhofer, M.; Aichernig, B.	2010
17	Factorized test-generation for multi-input/output transition systems (Brinksma et al., 1998)	Brinksma, E.; Heerink, L.; Tretmans, J.	1998
18	A Test Generation Framework for quiescent Real-Time Systems (Briones e Brinksma, 2005)	Briones, L.; Brinksma, E.	2005
19	A Semantic Framework for Test Coverage (Briones et al., 2006)	Briones, L.; Brinksma, E.; Stoelinga, M.	2006
20	Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction (Calamé et al., 2007)	Calamé, J. R.; Ioustinova, N. van der Pol, J.	2007
21	A Formal Model for Natural-Language Timed Requirements of Reactive Systems (Carvalho et al., 2014)	Carvalho, G.; Carvalho, A.; Rocha, E.; Cavalcanti, A.; Sampaio, A.	2014
22	Conformance Relations for Distributed Testing Based on CSP (Cavalcanti et al., 2011)	Cavalcanti, A.; Gaudel, M.-C.; Hierons, R.	2011
23	Test Generation from Recursive Tiles Systems (Chédor et al., 2012)	Chédor, S.; Jéron, T.; Morvan, C.	2012
24	Automatic test cases generation for statechart specifications from semantics to algorithm (Chen, 2011)	Chen, L.	2011

25	Automated Test and Oracle Generation for Smart-Card Applications (Clarke et al., 2001)	Clarke, D.; Jéron, T.; Rusu, V.; Zinovieva, E.	2001
26	Testing real-time systems from compositional symbolic specifications (Damasceno et al., 2015)	Damasceno, A. C.; Machado, P. D. L.; Andrade, W. L.	2015
27	Unfolding-Based Test Selection for Concurrent Conformance (De León et al., 2013)	De León, H. P.; Haar, S.; Longuet, D.; De Leon, H. P.; Haar, S.; Longuet, D.	2013
28	Formalizing interoperability for test case generation purpose (Desmoulin e Viho, 2009)	Desmoulin, A.; Viho, C.	2009
29	More testable properties (Falcone et al., 2012)	Falcone, Y.; Fernandez, J.-C.; Jéron, T.; Marchand, H.; Mounier, L.	2012
30	Symbolic Model Based Testing for Component Oriented Systems (Faivre et al., 2007)	Faivre, A.; Gaston, C.; Le Gall, P.	2007
31	Property Oriented Test Case Generation (Fernandez et al., 2004)	Fernandez, J.-C.; Mounier, L.; Pachon, C.	2004
32	A Model-Based Approach for Robustness Testing (Fernandez et al., 2005)	Fernandez, J.-C.; Mounier, L.; Pachon, C.	2005
33	Test Generation Based on Symbolic Specifications (Frantzen et al., 2005)	Frantzen, L.; Tretmans, J.; Willemse, T.	2005
34	A Symbolic Framework for Model-Based Testing (Frantzen et al., 2006)	Frantzen, L.; Tretmans, J.; Willemse, T.	2006
35	Model-Based Testing of Environmental Conformance of Components (Frantzen e Tretmans, 2007)	Frantzen, L.; Tretmans, J.	2007
36	Coverage based testing with test purposes (Fraser et al., 2008)	Fraser, G.; Weiglhofer, M.; Wotawa, F.	2008
37	A robustness testing method for network security (Fu e Koné, 2012)	Fu, Y.; Kone, O.	2012
38	Model based security verification of protocol implementation (Fu e Koné, 2015)	Fu, Y.; Kone, O.	2015
39	Symbolic Execution Techniques for Test Purpose Definition (Gaston et al., 2006)	Gaston, C.; Le Gall, P.; Rapin, N.; Touil, A.	2006
40	Software Testing Based on Formal Specification (Gaudel, 2010b)	Gaudel, M.-C.	2010
41	Formal test-case generation for UML statecharts (Gnesi et al., 2004)	Gnesi, S.; Latella, D.; Massink, M.	2004
42	Testing and Model-Checking Techniques for Diagnosis (Gromov e Willemse, 2007)	Gromov, M.; Willemse, T.	2007

43	A formal approach to protocol interoperability testing (Hao e Wu, 1998)	Hao, R.; Wu, J.	1998
44	Toward formal TTCN-based test execution (Hao e Wu, 1997)	Hao, R.; Wu, J.	1997
45	Testing Real-Time Systems Using UPPAAL (Hessel et al., 2008)	Hessel, A.; Larsen, K.; Mikucionis, M.; Nielsen, B.; Pettersson, P.; Skou, A.	2008
46	Exploration testing (Helovuo e Leppanen, 2001)	Helovuo, J.; Leppanen, S.	2001
47	Testing in the Distributed Test Architecture: An Extended Abstract (Hierons, 2008)	Hierons, R. M.	2008
48	Controllable Test Cases for the Distributed Test Architecture (Hierons et al., 2008)	Hierons, R.; Merayo, M.; Núñez, M.	2008
49	Testing probabilistic distributed systems (Hierons e Núñez, 2010)	Hierons, R.; Núñez, M.	2010
50	Scenarios-Based testing of systems with distributed ports (Hierons et al., 2011)	Hierons, R.; Merayo, M.; Núñez, M.	2011
51	Using schedulers to test probabilistic distributed systems (Hierons e Núñez, 2012)	Hierons, R.; Núñez, M.	2012
52	Implementation Relations for Testing Through Asynchronous Channels (Hierons, 2013)	Hierons, R. M.	2012
53	Overcoming controllability problems in distributed testing from an input output transition system (Hierons, 2012b)	Hierons, R.	2012
54	The complexity of asynchronous model based testing (Hierons, 2012a)	Hierons, R. M.	2012
55	Using Time to Add Order to Distributed Testing (Hierons et al., 2012a)	Hierons, R.; Merayo, M.; Núñez, M.	2012
56	Implementation relations and test generation for systems with distributed interfaces (Hierons et al., 2012b)	Hierons, R.; Merayo, M.; Núñez, M.	2012
57	Timed Implementation Relations for the Distributed Test Architecture (Hierons et al., 2014)	Hierons, R. M.; Merayo, M. G.; Núñez, M.	2014
58	A More Precise Implementation Relation for Distributed Testing (Hierons, 2015)	Hierons, R. M.	2015
59	On Testing Partially Specified IOTS through Lossless Queues (Huo e Petrenko, 2004)	Huo, J. L.; Petrenko, A.	2004
60	Covering transitions of concurrent systems through queues (Huo e Petrenko, 2005)	Huo, J.; Petrenko, A.	2005

61	TGV: theory, principles and algorithms (Jard e Jéron, 2005)	Jard, C.; Jéron, T.	2005
62	Test Generation Derived from Model-Checking (Jéron e Morel, 1999)	Jéron, T.; Morel, P.	1999
63	Symbolic Model-based Test Selection (Jéron, 2009)	Jéron, T.	2009
64	Model-based Test Selection for Infinite-State Reactive Systems (Jeannet et al., 2007)	Jeannet, B.; Jéron, T.; Rusu, V.	2007
65	Heuristics for Faster Error Detection With Automated Black Box Testing (Kervinen e Virolainen, 2005)	Kervinen, A.; Virolainen, P.	2005
66	Test generation for interworking systems (Koné e Castanet, 2000)	Koné, O.; Castanet, R.	2000
67	Symbolic execution techniques for Refinement Testing (Le Gall et al., 2007)	Le Gall, P.; Rapin, N.; Touil, A.	2007
68	Testing processes from formal specifications with inputs, outputs and data types (Lestiennes e Gaudel, 2002)	Lestiennes, G.; Gaudel, M.-C.	2002
69	Refusal testing for MIOTS with nonlockable output channel (Li et al., 2003)	Li, Z.; Wu, J.; Yin, X.	2003
70	Testing Multi Input/Output Transition System with All-Observer (Li et al., 2004a)	Li, Z.; Wu, J.; Yin, X.	2004
71	Distributed testing of multi input/output transition system (Li et al., 2004b)	Li, Z.; Yin, X.; Wu, J.	2004
72	Towards Property Oriented Testing (Machado et al., 2007)	Machado, P. D.; Silva, D. A.; Mota, A. C.	2007
73	Using software architecture for code testing (Muccini et al., 2004)	Muccini, H.; Bertolino, A.; Inverardi, P.	2004
74	Synchronizing Asynchronous Conformance Testing (Noroozi et al., 2011)	Noroozi, N.; Khosravi, R.; Mousavi, M.; Willemse, T.	2011
75	Synchrony and Asynchrony in Conformance Testing (Noroozi et al., 2013)	Noroozi, N.; Khosravi, R.; Mousavi, M.; Willemse, T.	2013
76	Specification and Testing of E-Commerce Agents Described by Using UIOLTSs (Pardo et al., 2010)	Pardo, J.; Núñez, M.; Ruiz, M.	2010
77	Testing Transition Systems with Input and Output Testers (Petrenko et al., 2003)	Petrenko, A.; Yevtushenko, N.; Huo, J.	2003
78	Test Synthesis from UML Models of Distributed Software (Pickin et al., 2007)	Pickin, S.; Jard, C.; Jeron, T.; Jezequel, J.; Le Traon, Y.	2007

79	A Formal Approach to Test the Robustness of Embedded Systems using Behaviour Analysis (Rollet e Saad-Khorchef, 2007)	Rollet, A.; Saad-Khorchef, F.	2007
80	Two complementary approaches to test robustness of reactive systems (Rollet e Salva, 2008)	Rollet, A.; Salva, S.	2008
81	Testing robustness of communicating systems using ioco-based approach (Rollet e Salva, 2009)	Rollet, A.; Salva, S.	2009
82	Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems (Rusu et al., 2005)	Rusu, V.; Marchand, H.; Jéron, T.	2005
83	Compositional Verification of Input-Output Conformance via CSP Refinement Checking (Sampaio et al., 2009)	Sampaio, A.; Nogueira, S.; Mota, A.	2009
84	On Conformance Testing for Timed Systems (Schmaltz e Tretmans, 2008)	Schmaltz, J.; Tretmans, J.	2008
85	Architectural Unit Testing (Scollo e Zecchini, 2005)	Scollo, G.; Zecchini, S.	2005
86	Generating asynchronous test cases from test purposes (Simao e Petrenko, 2011)	Simao, A.; Petrenko, A.	2011
87	Generating Complete and Finite Test Suite for ioco: Is it Possible? (Simao e Petrenko, 2014)	Simao, A.; Petrenko, A.	2014
88	Model based testing with labelled transition systems (Tretmans, 2008)	Tretmans, J.	2008
89	Model-Based Testing and Some Steps towards Test-Based Modelling (Tretmans, 2011)	Tretmans, J.	2011
90	Automatic Conformance Testing of Internet Applications (van Beek e Mauw, 2004)	van Beek, H.; Mauw, S.	2004
91	Compositional Testing with ioco (van der Bijl et al., 2004)	van der Bijl, M.; Rensink, A.; Tretmans, J.	2004
92	Action Refinement in Conformance Testing (van der Bijl et al., 2005)	van der Bijl, M.; Rensink, A.; Tretmans, J.	2005
93	Asynchronous Input-Output Conformance Testing (Weiglhofer e Wotawa, 2009)	Weiglhofer, M.; Wotawa, F.	2009
94	Using coverage to automate and improve test purpose based testing (Weiglhofer et al., 2009)	Weiglhofer, M.; Fraser, G.; Wotawa, F.	2009
95	Unifying Input Output Conformance (Weiglhofer e Aichernig, 2010)	Weiglhofer, M.; Aichernig, B.	2010
96	Heuristics for ioco-Based Test-Based Modelling (Willemse, 2007)	Willemse, T.	2007

97	Testing Systems of Concurrent Black-Boxes—An Automata-Theoretic and Decompositional Approach (Xie e Dang, 2006)	Xie, G.; Dang, Z.	2006