



Ministério da
Ciência e Tecnologia



INPE-16682-TDI/1627

GERAÇÃO DE CASOS DE TESTE PARA SISTEMAS DA ÁREA ESPACIAL USANDO CRITÉRIOS DE TESTE PARA MÁQUINAS DE ESTADOS FINITOS

Érica Ferreira de Souza

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada,
orientada pelo Dr. Nandamudi Lankalapalli Vijaykumar, aprovada em 19 de
fevereiro de 2010

Registro do documento original:

<<http://urlib.net/sid.inpe.br/mtc-m19@80/2010/02.06.20.39>>

INPE
São José dos Campos
2010

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE

Gabinete do Diretor (GB)

Serviço de Informação e Documentação (SID)

Caixa Postal 515 - CEP 12.245-970

São José dos Campos - SP - Brasil

Tel.:(012) 3945-6911/6923

Fax: (012) 3945-6919

E-mail: pubtc@sid.inpe.br

CONSELHO DE EDITORAÇÃO:

Presidente:

Dr. Gerald Jean Francis Banon - Coordenação Observação da Terra (OBT)

Membros:

Dr^a Maria do Carmo de Andrade Nono - Conselho de Pós-Graduação

Dr. Haroldo Fraga de Campos Velho - Centro de Tecnologias Especiais (CTE)

Dr^a Inez Staciarini Batista - Coordenação Ciências Espaciais e Atmosféricas (CEA)

Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)

Dr. Ralf Gielow - Centro de Previsão de Tempo e Estudos Climáticos (CPT)

Dr. Wilson Yamaguti - Coordenação Engenharia e Tecnologia Espacial (ETE)

BIBLIOTECA DIGITAL:

Dr. Gerald Jean Francis Banon - Coordenação de Observação da Terra (OBT)

Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)

Jefferson Andrade Ancelmo - Serviço de Informação e Documentação (SID)

Simone A. Del-Ducca Barbedo - Serviço de Informação e Documentação (SID)

REVISÃO E NORMALIZAÇÃO DOCUMENTÁRIA:

Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)

Marilúcia Santos Melo Cid - Serviço de Informação e Documentação (SID)

Yolanda Ribeiro da Silva Souza - Serviço de Informação e Documentação (SID)

EDITORAÇÃO ELETRÔNICA:

Viveca Sant´Ana Lemos - Serviço de Informação e Documentação (SID)



Ministério da
Ciência e Tecnologia



INPE-16682-TDI/1627

GERAÇÃO DE CASOS DE TESTE PARA SISTEMAS DA ÁREA ESPACIAL USANDO CRITÉRIOS DE TESTE PARA MÁQUINAS DE ESTADOS FINITOS

Érica Ferreira de Souza

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada,
orientada pelo Dr. Nandamudi Lankalapalli Vijaykumar, aprovada em 19 de
fevereiro de 2010

Registro do documento original:

<<http://urlib.net/sid.inpe.br/mtc-m19@80/2010/02.06.20.39>>

INPE
São José dos Campos
2010

Dados Internacionais de Catalogação na Publicação (CIP)

Souza, Érica Ferreira de.
So89g Geração de casos de teste para sistemas da área espacial usando
critérios de teste para máquinas de estados finitos / Érica Ferreira
de Souza. – São José dos Campos : INPE, 2010.
xxii + 111 p. ; (INPE-16682-TDI/1627)

Dissertação (Mestrado em Computação Aplicada) – Instituto
Nacional de Pesquisas Espaciais, São José dos Campos, 2010.
Orientador : Dr. Nandamudi Lankalapalli Vijaykumar.

1. Testes de software. 2. Critérios de teste. 3. Automatização.
4. Sistemas críticos. 5. Casos de teste. I. Título.

CDU 004.052.42

Copyright © 2010 do MCT/INPE. Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação, ou transmitida sob qualquer forma ou por qualquer meio, eletrônico, mecânico, fotográfico, reprográfico, de microfilmagem ou outros, sem a permissão escrita do INPE, com exceção de qualquer material fornecido especificamente com o propósito de ser entrado e executado num sistema computacional, para o uso exclusivo do leitor da obra.

Copyright © 2010 by MCT/INPE. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, microfilming, or otherwise, without written permission from INPE, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use of the reader of the work.

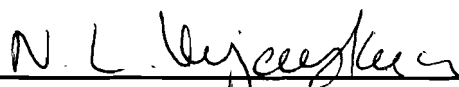
**Aprovado (a) pela Banca Examinadora
em cumprimento ao requisito exigido para
obtenção do Título de Mestre em
Computação Aplicada**

Dr. Edson Luiz França Senne



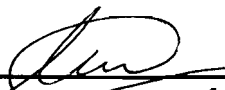
Presidente / UNESP/GUARA / Guaratinguetá - SP

Dr. Nandamudi Lankalapalli Vijaykumar



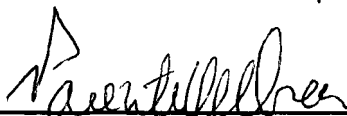
Orientador(a) / INPE / SJCampos - SP

Dr. Maurício Gonçalves Vieira Ferreira



Membro da Banca / INPE / SJCampos - SP

Dr. José Maria Parente de Oliveira



Convidado(a) / ITA / São José dos Campos - SP

Aluno (a): Érica Ferreira de Souza

São José dos Campos, 19 de fevereiro de 2010

A meus pais Osvaldo Dias e Maria Aparecia

AGRADECIMENTOS

Primeiramente agradeço a Deus por todas as oportunidades que me concedeu.

Aos meus orientadores Prof. Nandamudi Vijaykumar e Valdivino Santiago pela condução do trabalho, prontidão e responsabilidade.

A minha família, principalmente aos meus pais **Oswaldo Dias** e **Maria Aparecida**, meus irmãos **Edevaldo** e **Éder**, e minha avó **Ana**.

Ao grupo de pesquisa na área de teste de software da coordenadoria de Ciências Espaciais e Atmosféricas (CEA) e o Laboratório de Computação Aplicada (LAC), pela ajuda na realização das pesquisas deste trabalho, valiosos conhecimentos e experiências adquiridos.

A todas as pessoas especiais que sempre estiveram ao meu lado em todos os momentos, e que de forma direta ou indireta me ajudaram na conquista de mais um sonho.

RESUMO

A geração de casos de teste baseada em Máquinas de Estados Finitos (MEF) tem recebido grande atenção ao longo dos anos. Os Testes de *Software* Baseados em Modelos têm despertado grande interesse de pesquisadores e profissionais na área de teste. Diversos critérios de teste vêm sendo propostos na literatura no intuito de validar sistemas construídos de acordo com alguma técnica formal de modelagem como, por exemplo, as MEF. Por meio de MEF é possível derivar caminhos de execução, ou seja, encontrar casos de teste. Porém, ainda falta uma consolidação desse aspecto no sentido de direcionar, efetivamente, um projetista de teste na escolha do critério mais adequado para gerar casos de teste. Assim, o foco principal deste trabalho consiste no desenvolvimento dos critérios de teste *Unique Input/Output* (UIO), *Distinguishing Sequence* (DS) e *Switch Cover*, visando sua integração a dois ambientes de teste do Instituto Nacional de Pesquisas Espaciais (INPE), bem como uma investigação preliminar do custo e da eficiência desses critérios concebidos com base em uma avaliação empírica, mostrando qual critério apresenta ser mais relevante nos estudos de caso propostos. Os critérios foram integrados à ferramenta Geração Automática de Casos de Teste Baseada em *Statecharts* (GTSC) e à ferramenta *WEB-Performcharts*. *Softwares* embarcados em computadores de satélites científicos foram utilizados como estudos de caso para avaliações de custo e eficiência. Tais *softwares* estão em desenvolvimento na área de Ciências Espaciais e Atmosféricas (CEA) no INPE.

TEST CASE GENERATION FOR SPACE AREA SYSTEMS USING TEST CRITERIA FOR FINITE STATE MACHINES

ABSTRACT

Test case generation based on Finite State Machines (FSM) has been addressed for quite some time. Model-Based Testing has drawn attention from researchers and practitioners in test area. Several test criteria have been proposed in the literature in order to validate systems built according to some modeling formal technique, for example, the FSM. Through MEF it is possible to derive execution paths, that is, to find test cases. However, there is still a lot to do on this aspect in order to effectively direct a test designer to choose a criterion among those most suitable criteria to generate test cases. Thus, the focus of this work is the design and develop Unique Input/Output(UIO), Distinguishing Sequence (DS) and Switch Cover, in order to incorporate them in two test environments (Automatic Generation of Test Cases Based on Statecharts (GTSC) and WEB-PerformCharts within the National Institute for Space Research (INPE). Besides, a preliminary investigation of cost and efficiency of these criteria will be conducted based on an empirical analysis, showing which criteria can be more relevant. Embedded software on computers in scientific satellites will be used as case studies for analysis of cost and efficiency. Such software is under development in the Space and Atmospheric Sciences (CEA) Coordination at INPE.

LISTA DE FIGURAS

	<u>Pág.</u>
2.1 Exemplo de uma Máquinas de Estados Finitos (MEF) que especifica o comportamento de um extrator de comentários (Figura Adaptada). . . .	10
2.2 Exemplo de um <i>Statecharts</i> mostrando ações associadas a eventos.	13
2.3 Máquina de Estados Finito: Exemplo Implementação do Critério <i>Distinguishing Sequence</i> (DS).	15
2.4 Exemplo 01: Árvore de distinção gerada a partir de uma MEF.	17
2.5 Máquina de Estados Finitos para exemplo dos critérios	18
2.6 Exemplo 02: Árvore de distinção gerada a partir de uma MEF.	19
2.7 Árvore de representação a busca em largura do método <i>Unique Input/Output</i> (UIO).	21
2.8 Exemplo de uma MEF simples.	23
2.9 Criação do Grafo dual a partir da MEF original.	23
2.10 Criação das novas transições a partir da MEF original.	24
2.11 Grafo Dual balanceado.	24
2.12 Relação hierárquica de família FCCS.	26
2.13 Critério Análise de Mutantes.	29
2.14 Visão arquitetural do ambiente Geração Automática de Casos de Teste Baseada em <i>Statecharts</i> (GTSC).	32
2.15 Visão arquitetural ao ambiente <i>WEB-PerformCharts</i>	34
3.1 Representação em XML de uma MEF.	40
3.2 <i>Workflow</i> para interpretar uma MEF em XML e gerar casos de teste. . .	40
3.3 Visão geral dos principais pacotes da leitura da MEF em <i>eXtensible Markup Language</i> (XML) e dos critérios de teste.	41
3.4 Pseudo-código para criar a Árvore de Distinção	42
3.5 Pseudo-código para criar a β -sequence	44
3.6 Fluxograma do algoritmo do critério de teste DS.	46
3.7 Pseudo-código para encontrar a Única de Entrada e Saída (UES) de cada estado da MEF	47
3.8 Fluxograma do algoritmo do critério de teste UIO.	49
3.9 Pseudo-código para transformar as transições da MEF em estados e adicionar as novas transições.	50
3.10 Pseudo-código para balacenamento do Grafo Dual.	53

3.11	Pseudo-código para retornar o melhor <i>filho</i>	54
3.12	Pseudo-código para retornar o melhor <i>pai</i>	56
3.13	Exemplo de uma MEF para o critério <i>Switch Cover</i>	57
3.14	MEF inicial e MEF balanceada.	58
3.15	Pseudo-código para geração dos casos de teste.	59
3.16	Fluxograma do algoritmo do critério de teste <i>Switch Cover</i>	61
4.1	Exemplo de uma sequência de casos de teste gerada pelo critério UIO. . .	64
4.2	Canal de comunicação entre o <i>On-Board Data Handling</i> (OBDH) e o <i>EXPEmulator</i>	65
4.3	Formato da mensagem de comando enviada pelo OBDH para o <i>Alpha</i> , <i>Próton and Electron monitoring eXperiment in the magnetosphere</i> (APEX)	66
4.4	Modelo <i>Statecharts</i> do principal componente do <i>software</i> APEX.	68
4.5	Gráficos de resultados sobre custo dos critérios com os operadores de mutação ROR e AOIU	71
4.6	Visão geral do escopo do <i>software Software of Payload Data handling</i> <i>Computer</i> (SWPDC).	73
4.10	Gráfico de custo por cenário dos critérios de teste: Quantidade de casos de teste.	79
4.11	Gráfico de custo para os cenários 01 (um) e 02 (um): Fator tempo.	81
4.7	Modelo <i>Statecharts</i> do cenário 03 (três) do <i>software</i> SWPDC.	82
4.8	Modelo referente a expansão do estado <i>SafeM_ChangingSwPar</i>	83
4.9	Modelo referente a expansão do estado <i>SafeM_LoopingHK</i>	83
A.1	Cenário 01 usado no estudo de caso do SWPDC.	97
B.1	Gráficos de custo para os operadores AOIS, ATRI, JTI, ASRS, COR e EOC.	103
B.2	Gráficos de custo para os operadores LOI, JTD, PRV e IOP	103
B.3	Gráfico de custo para os operadores SSDL e JDC.	104
B.4	Gráfico de custo para o cenários 03 e 04.	104
B.5	Gráfico de custo para os cenários 05 e 06.	104
B.6	Gráfico de custo para os cenários 07 e 08.	105
B.7	Gráfico de custo para os cenários 09 e 10	105
B.8	Gráfico de custo para os cenários 11 e 12	105
B.9	Gráfico de custo para os cenários 13 e 14	106
B.10	Gráfico de custo para os cenários 15 e 16	106
B.11	Gráfico de custo para os cenários 17 e 18	106
B.12	Gráfico de custo para os cenários 19 e 20	107

C.1	Modelo <i>Statecharts</i> do cenário 02 (dois) do <i>software</i> SWPDC.	109
C.2	Modelo <i>Statecharts</i> do cenário 04 (quatro) do <i>software</i> SWPDC.	110
C.3	Modelo <i>Statecharts</i> do cenário 05 (cinco) do <i>software</i> SWPDC.	111

LISTA DE TABELAS

	<u>Pág.</u>
2.1 Tabela de transição de estados.	10
2.2 Saídas geradas a partir de cada estado da MEF quando aplicado a sequência de distinção (SD).	18
2.3 Descrição do critério DS.	19
2.4 Sequências de teste UES de cada estado da MEF.	21
2.5 Tabela das UES encontradas.	22
2.6 Descrição do critério UIO.	22
2.7 Comparação entre os critérios UIO, DS e <i>Switch Cover</i>	25
 4.1 Descrição dos comandos enviados pelo OBDH.	 66
4.3 Resultados sobre eficiência dos critérios de teste no <i>software</i> APEX. . . .	69
4.5 Operadores de Mutação para linguagem C usados no <i>software</i> SWPDC. .	74
4.6 Resultados sobre a eficiência dos 20 (vinte) cenários de teste no <i>software</i> SWPDC.	75
4.7 Resultados de custo dos critérios de teste no <i>software</i> SWPDC.	79
4.8 Quantidade de eventos dos casos de teste dos critérios DS, UIO e <i>H-Switch Cover</i>	80

LISTA DE ABREVIATURAS E SIGLAS

APEX *Alpha, Próton and Electron monitoring eXperiment in the magnetosphere*

CEA Ciências Espaciais e Atmosféricas

CEU *Central Electronics Unit*

DS *Distinguishing Sequence*

EPP *Event Pre-Processor*

FCCS Família de Critérios de Cobertura para *Statecharts*

GTSC Geração Automática de Casos de Teste Baseada em *Statecharts*

HXI *Hard X-Ray Imager*

IEEE *Institute of Electrical and Electronics Engineers*

INPE Instituto Nacional de Pesquisas Espaciais

LAC Laboratório de Computação Aplicada

MEF Máquinas de Estados Finitos

OBDH *On-Board Data Handling*

PcML *PerformCharts Markup Language*

PDC *Payload Data handling Computer*

QSEE Qualidade do *Software* Embarcado em Aplicações Espaciais

QSEE-TAS Qualidade do *Software* Embarcado em Aplicações Espaciais - Teste Automatizado de *Software*

RPC Redes de Petri Colorida

SD sequência de distinção

SDL *Specification and Description Language*

SWPDC *Software of Payload Data handling Computer*

TBD Teste Baseado em Defeitos

TBM Testes Baseados em Modelos

TT *Transition Tour*

UES Única de Entrada e Saída

UIO *Unique Input/Output*

USB *Universal Serial Bus*

XML *eXtensible Markup Language*

W3C *Word Wide Web Consortium*

SUMÁRIO

Pág.

LISTA DE ABREVIATURAS E SIGLAS

1 INTRODUÇÃO	1
1.1 Motivação	3
1.2 Objetivo do trabalho	3
1.3 Organização do texto	4
2 FUNDAMENTAÇÃO TEÓRICA	5
2.1 Terminologia e conceitos	5
2.1.1 Defeito, Erro e Falha	5
2.1.2 Teste de <i>Software</i>	5
2.1.3 Técnicas de Teste	6
2.2 Teste Baseado em Modelos (TBM)	8
2.2.1 Máquinas de Estados Finitos (MEF)	9
2.2.2 <i>Statecharts</i>	12
2.3 Critérios de Testes para MEF	13
2.3.1 Critério <i>Distinguishing Sequence</i> (DS)	14
2.3.2 Critério <i>Unique Input/Output</i> (UIO)	18
2.3.3 Critério <i>Switch Cover</i>	21
2.3.4 Comparação entre os Critérios	25
2.4 Critérios de Testes para <i>Statecharts</i>	25
2.5 Avaliação de Critérios de Teste para MEF	27
2.5.1 Eficiência	27
2.5.2 Custo	30
2.6 Ferramentas de apoio ao Teste de <i>Software</i>	31
2.6.1 Ambiente de Teste: Geração Automática de Casos de Teste Baseada em <i>Statecharts</i> (GTSC)	31
2.6.2 Ambiente de Teste: <i>WEB-PerformCharts</i>	33
2.7 Trabalhos Relacionados	34
3 IMPLEMENTAÇÃO DOS CRITÉRIOS DE TESTE PARA MEF	39
3.1 Leitura da Máquina de Estados Finitos	39

3.2	Implementação do Critério DS	41
3.3	Implementação do Critério UIO	46
3.4	Implementação do Critério <i>H-Switch Cover</i>	48
4	AVALIAÇÃO DOS CRITÉRIOS DE TESTE	63
4.1	Planejamento e Operação	63
4.2	Estudo de Caso I: <i>Alpha, Proton and Electron monitoring eXperiment in the magnetosphere</i> (APEX)	64
4.2.1	Avaliação de eficiência	67
4.2.2	Avaliação de Custo	70
4.3	Estudo de Caso II: <i>Software of the Payload Data Handling Computer</i> (SWPDC)	72
4.3.1	Avaliação de eficiência	74
4.3.2	Avaliação de Custo	78
5	CONCLUSÃO	85
5.1	Considerações Gerais	85
5.2	Principais contribuições	87
5.3	Limitações e dificuldades	87
5.4	Sugestões para trabalhos futuros	88
	REFERÊNCIAS BIBLIOGRÁFICAS	91
	APÊNDICE A - EXEMPLO DE PcML E MEF EM XML.	97
A.1	Estrutura em XML do PcML	97
A.2	Estrutura em XML da MEF	99
	APÊNDICE B - GRÁFICOS DAS ANÁLISES	103
B.1	Gráficos de custo dos critérios de teste para o <i>software</i> APEX	103
B.2	Gráficos de custo dos critérios de teste para o <i>software</i> SWPDC	103
	APÊNDICE C - CENÁRIOS DO <i>SOFTWARE</i> SWPDC	109

1 INTRODUÇÃO

Durante as últimas décadas, tem-se visto o desenvolvimento de *softwares* cada vez mais modernos, desde *softwares* mais comuns, presentes em dispositivos, que fazem parte do nosso cotidiano como, aparelhos de celulares e microondas, até *softwares* mais críticos como aplicações que estão relacionadas à ambientes espaciais. Dessa forma, é fundamental aos ambientes de desenvolvimento de *software* a utilização de práticas de teste de uma forma sistemática para que a garantia de qualidade do *software* seja considerada. Diante desse fato, atividades relacionadas à Verificação, Validação e Teste (VV&T), vêm sendo introduzidas ao longo de todo o processo de desenvolvimento de *software*. Essas atividades têm como finalidade verificar a conformidade do *software* construído com o que foi especificado, e garantir que ele alcance um alto nível de qualidade.

O desenvolvimento de *softwares* complexos implica numa estratégia de divisão em etapas de produção. Erros podem ser encontrados em qualquer etapa do desenvolvimento do *software*, desde a fase de análise até a fase de desenvolvimento e implantação. Um *software* com erros pode acarretar sérios problemas futuros, não só o prejuízo financeiro, mas também as consequências que poderiam suceder. Por meio das atividades de teste é possível detectar defeitos inseridos em qualquer etapa do desenvolvimento do *software*.

É difícil garantir que um *software* funcionará corretamente mesmo sem a presença de erros (MYERS, 2004), principalmente nos casos em que este possui características complexas, como é o caso de *softwares* da área espacial. Projetos de *software* embarcados, em geral, são considerados como sendo altamente críticos e todas as agências espaciais mundiais tem dado muita atenção nas últimas décadas à qualidade do processo de teste de tais *softwares* (MATIELLO et al., 2006).

Segundo Levenson (2001), a qualidade de um *software* depende muito dos testes realizados ao nível de conformidade, de forma a garantir a compatibilidade dos requisitos com a implementação. A verificação do que foi implementado com o que foi especificado é chamado de teste de conformidade (BINDER, 2005). Os testes de conformidade permitem não somente revelar erros presentes no escopo do *software*, como também, demonstrar que o *software* em teste implementa todas as funcionalidades requisitadas. Esse tipo de teste tem motivado pesquisas de novas técnicas de teste na fase de especificação do *software*, desde que a especificação seja mode-

lada por meio de alguma técnica formal. Dentre as técnicas formais usadas para especificar o comportamento de um *software*, têm-se a MEF (SIDHU; LEUNG, 1989), *Statecharts* (HAREL, 1987), Redes de Petri (PETERSON, 1981), Estelle (BUDKOWSKI; DEMBINSKI, 1987), *Specification and Description Language* (SDL) (ELLSBERGER et al., 1997), dentre outras.

De acordo com Pressman (2006), um sistema pode ser dividido em várias categorias, tais como: Sistemas Básicos, Sistemas de Informação, Sistemas Científicos, Sistemas de Inteligência Artificial e Sistemas Reativos. Os Sistemas Reativos, contexto no qual se insere este trabalho, são sistemas que reagem a estímulos internos ou do ambiente, de forma a produzir resultados corretos dentro de intervalos de tempo previamente especificados, podendo apresentar um comportamento bastante complexo (HAREL, 1987). Dentre as técnicas citadas anteriormente, MEF é a técnica mais adequada para sistemas reativos, pois o comportamento de um sistema reativo é fortemente baseado em estados.

Considerando os testes de conformidade baseados em especificações de *softwares*, utilizando a técnica MEF, existem vários critérios¹ capazes de gerar sequências de teste a partir de uma MEF. Diversos estudos vem sendo conduzidos para identificar critérios capazes de melhorar o processo da atividade de teste. Critérios são utilizados para gerar sequências de teste (ou casos de teste) com base nos respectivos modelos. Dentre os diversos critérios capazes de gerar casos de teste a partir de MEF, podem-se citar os critérios UIO, DS (SIDHU; LEUNG, 1989), *State Counting* (PETRENKO; YEVTUSHENKO, 2005), *Switch Cover* (PIMONT; RAULT, 1976), dentre outros.

A aplicação de tais critérios torna-se improdutiva quando realizada manualmente e, por isso, existe a necessidade da criação de ferramentas de apoio para utilização desses critérios de teste. São poucas as ferramentas de teste disponíveis que incorporam critérios para geração de casos de teste de forma automática. Da mesma forma, existe uma falta de consolidação para dizer qual critério é o mais relevante a ser utilizado. A escolha sobre o critério a ser utilizado depende de características da MEF e da relação de alguns fatores para avaliação de critérios, tais como custo e eficiência das sequências de teste geradas.

Considerando o contexto acima, o trabalho desta dissertação está relacionado com os

¹Na comunidade científica os critérios de teste também são chamados de métodos, porém em todo contexto deste trabalho será utilizado o termo critério.

testes gerados de forma automática a partir de implementação de critérios de teste para MEF. Outro aspecto importante deste trabalho diz respeito à realização de uma avaliação preliminar, por meio de experimentos, dos critérios em termos de custo e eficiência. Os estudos de caso para avaliação preliminar de critérios inserem-se na categoria dos sistemas reativos, mais precisamente nas dos sistemas embarcados.

1.1 Motivação

Ultimamente tem-se dado muita importância aos estudos voltados para geração automática de casos de Testes Baseados em Modelos (TBM). Tem-se observado que a própria atividade de geração de casos de teste é bastante efetiva em evidenciar a presença de defeitos de *software* (MYERS, 2004). Em virtude disso, aumentam cada vez mais os estudos relacionados a TBM e a geração de casos de teste de forma automática.

É importante que o teste seja realizado de forma automatizada para que este seja eficiente. No entanto, torna-se cada vez mais necessário o desenvolvimento de ferramentas de suporte e automatização da atividade de teste, principalmente por viabilizar a aplicação prática de critérios de teste, bem como propiciar a realização desta atividade em *softwares* maiores e mais complexos (SILVEIRA, 2001).

No Instituto Nacional de Pesquisas Espaciais (INPE), a coordenadoria de Ciências Espaciais e Atmosféricas (CEA) e o Laboratório de Computação Aplicada (LAC), mantêm um grupo de pesquisa na área de teste que trabalha com modelos formais para a especificação de *software*. Nesse sentido foram criados dois ambientes baseados na ferramenta *PerformCharts* (VIJAYKUMAR, 1999; ARANTES et al., 2002; FERREIRA et al., 2008), que geram casos de teste a partir de MEF e *Statecharts*. Considerando esses ambientes, surgiu a necessidade de aperfeiçoar as ferramentas, incorporando novos critérios de geração de casos de teste, tornando-os mais completos e objetivando também validar os critérios para poder demonstrar qual critério tende a ser mais relevante mesmo sendo uma avaliação preliminar.

1.2 Objetivo do trabalho

A concepção deste trabalho de pesquisa de mestrado divide-se em dois objetivos principais:

- a) Desenvolvimento de três critérios de teste para MEF: DS, UIO e *Switch Co-*

ver, com correspondentes incorporações aos ambientes GTSC (SANTIAGO et al., 2008) e *WEB-PerformCharts* (ARANTES et al., 2002).

- b) Avaliação preliminar dos três critérios de teste em termos de custo e eficiência. Para propiciar a avaliação dos respectivos critérios, estudos de caso empíricos são conduzidos. *Softwares* embarcados em computadores de satélites científicos em desenvolvimento no CEA/INPE são utilizados como estudos de caso. Sendo eles o *software* APEX (SANTIAGO et al., 2006) e o SWPDC (SANTIAGO et al., 2007; SANTIAGO et al., 2008).

1.3 Organização do texto

A divisão dos capítulos deste trabalho está descrita a seguir:

- *CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA*: Este capítulo mostra de forma sucinta os principais conceitos relacionados à área de pesquisa deste trabalho em termos de: atividade de testes de *software*; TBM; Ferramentas de apoio ao teste, bem como as ferramentas de teste desenvolvidas no INPE; Critérios de Teste pra MEF; métricas de avaliação de conjunto de casos de teste; e por fim, alguns trabalhos relacionados a essa pesquisa de mestrado.
- *CAPÍTULO 3 - IMPLEMENTAÇÃO DOS CRITÉRIOS DE TESTE PARA MEF*: O Capítulo 3 apresenta como foram desenvolvidos os critérios de teste para MEF e como eles foram incorporados a duas ferramentas de teste em desenvolvimento no INPE.
- *CAPÍTULO 4 - ANÁLISE DOS CRITÉRIOS DE TESTE*: De acordo com as considerações apresentadas nos Capítulos anteriores, serão apresentados nesse Capítulo os principais resultados alcançados com a concepção dos critérios e a avaliação dos mesmos nos estudos de caso utilizados.
- *CAPÍTULO 5 - CONCLUSÃO*: Sintetiza as principais conclusões dessa pesquisa, bem como as principais contribuições e trabalhos futuros. Subseqüentes a este capítulo encontram-se as referências bibliográficas utilizadas na elaboração deste documento e os apêndices produzidos.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta sucintamente os principais conceitos relacionados às atividades de teste de *software* que auxiliarão no entendimento deste trabalho. A Seção 2.1 introduz alguns conceitos e definições da temática de teste, na Seção 2.2 são apresentadas as duas técnicas para especificação de sistemas utilizadas neste trabalho: *MEF* e *Statecharts*. A Seção 2.3 mostra alguns critérios de geração de casos de teste para *MEF*, enquanto que a Seção 2.4 mostra alguns critérios para *Statecharts*. A Seção 2.5 apresenta algumas métricas de medição de casos de teste. Na Seção 2.6 são abordados dois ambientes de geração automática de casos de testes a partir de especificações em *MEF* e *Statecharts*. Tais ambientes foram desenvolvidas por grupos de pesquisa formado por alunos e professores do *LAC* e do *CEA*. E finalmente na Seção 2.7 são mostrados alguns trabalhos relacionados a esta pesquisa.

2.1 Terminologia e conceitos

2.1.1 Defeito, Erro e Falha

Entre os pesquisadores da área de teste de *software*, existem divergências quanto à terminologia adotada referente aos termos *defeito*, *erro* e *falha*. O *Institute of Electrical and Electronics Engineers (IEEE)* 610.12-1990 (*IEEE*, 1990) tem realizado vários esforços com o intuito de padronizar a terminologia utilizada em vários campos do conhecimento, dentre eles o da Engenharia de *Software*. Nesta dissertação, é utilizada a definição estabelecida pelo *IEEE*.

A norma *IEEE* distingue os termos da seguinte forma: defeito (*fault*) - é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto; erro (*error*) - manifestação concreta de um defeito num artefato do *software*, ou seja, é a diferença entre valor obtido e o valor esperado; e falha (*failure*) - o comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

2.1.2 Teste de *Software*

Existem atualmente várias definições para o conceito de teste de *software*. De uma forma simples, testar um *software* significa verificar, através de uma execução contro-

lada, se o seu comportamento ocorre de acordo com o que foi especificado (MYERS, 2004).

A atividade de teste é um elemento crítico de garantia de qualidade do produto. Deve ser aplicada desde o início do processo de desenvolvimento, sendo a última, revisão das fases de especificação, projeto e codificação, pois a descoberta de um erro evita a sua propagação nas fases subsequentes, facilitando a sua remoção. O principal objetivo do teste é revelar a presença de erros no produto. Neste sentido, uma atividade de teste pode ser considerada como bem sucedida quando o programa em teste falha (PRESSMAN, 2006).

Assim como as atividades de desenvolvimento, as atividades de teste também são constituídas por processos. De acordo com Bastos et al. (2007), o processo de teste é composto por diversas etapas ou fases, sendo que as principais delas são: *planejamento, especificação, execução e entrega*. A etapa de planejamento caracteriza-se pela definição de uma proposta de teste (Plano de Teste), apresentando prazos e esforços de teste a serem realizados. A especificação é uma etapa que é caracterizada pela identificação dos casos de teste que deverão ser construídos. Na etapa de execução, os casos de testes gerados na fase de especificação são exercitados na implementação. Finalmente, na última etapa tem-se a entrega de artefatos com a avaliação de todo o processo de teste do *software*, comparando os resultados alcançados em relação ao que foi inicialmente planejado.

A atividade de teste geralmente é realizada em três fases distintas: *teste de unidade, teste de integração e teste de sistema* (PRESSMAN, 2006). O teste de unidade consiste em testar a menor unidade do projeto, sendo a unidade uma classe, um método ou apenas um procedimento. O teste de integração visa garantir que os componentes da aplicação, ou daquele módulo da aplicação, possam ser integrados com sucesso para executar determinada funcionalidade. Já o teste de sistema consiste em garantir que os requisitos do *software* foram cumpridos e implementados corretamente.

2.1.3 Técnicas de Teste

A única maneira de se garantir a correção de um *software* seria através de um teste exaustivo, executando-se o *software* com todas as combinações de valores de entrada. Esta prática é inviável, pois o domínio de entrada pode ser infinito ou, no mínimo, muito grande (MYERS, 2004). Portanto, devem ser utilizadas técnicas para

selecionar um subconjunto de dados de teste. Para se atingir os objetivos de maneira satisfatória, os testes devem ser bem planejados e conduzidos de maneira organizada. Dessa forma, cada etapa de teste é realizada mediante uma série de técnicas de teste e diversos critérios pertencentes a cada uma delas. As três principais técnicas de teste utilizadas são: Teste Estrutural (ou caixa-branca), Teste Funcional (ou caixa-preta) e Teste Baseado em Defeitos.

Teste Estrutural

A Técnica de teste Estrutural, também conhecida como teste de caixa-branca, tem como objetivo testar a estrutura de controle do programa ([PRESSMAN, 2006](#)). Os caminhos lógicos do programa são testados, fornecendo casos de teste que põem à prova tanto conjuntos específicos de condições e laços, bem como definições e usos de variáveis. Os critérios pertencentes a Técnica Estrutural são classificados com base na complexidade do programa, no fluxo de controle e no fluxo de dados ([DELAMARO et al., 2007](#)).

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios para determinar quais estruturas são necessárias. Dentre os critérios baseados em fluxo de controle, os mais conhecidos são: critério todos-nós, critério todos-arcos, e o critério todos-caminhos ([DELAMARO et al., 2007](#)).

Já os critérios baseados no fluxo de dados, utilizam a análise de fluxo de dados como fonte de informação para derivar os casos de teste ([HECHT, 1977](#)). Os critérios pertencentes a essa categoria baseiam-se nas associações entre as definições de variáveis e seus possíveis usos subsequentes. Dentre os critérios baseados em fluxo de dados, destacam-se: critério todas-definições e critério todos-os-usos proposto por ??) e a família de critérios Potenciais-Usos proposto por [Maldonado \(1991\)](#).

Teste Funcional

A Técnica de teste Funcional é uma técnica utilizada para se projetarem casos de teste na qual o programa ou sistema é considerado uma caixa-preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados ([MALDONADO, 1991](#)). A técnica de Teste Funcional não considera a estrutura interna do programa, sendo aplicada quando o sistema já se encontra em fase final ou completamente construído. Segundo

Pressman (2006), os defeitos mais evidenciados neste tipo de teste são: defeitos de desempenho, defeitos de interface, defeitos nas estruturas dos dados ou no acesso a banco de dados externos. Os critérios de teste mais conhecidos referentes à Técnica de Teste Funcional são Particionamento em Classes de Equivalência, Análise do Valor Limite, Grafo de Causa-Efeito e TBM (PRESSMAN, 2006).

O TBM também condiz com a Técnica de Teste Funcional, pois ela utiliza informações sobre o comportamento funcional do sistema, representado em um modelo formal. Este modelo é construído a partir dos requisitos funcionais e determina as entradas possíveis e as saídas esperadas. O foco deste trabalho é o TBM, e a sua descrição é apresentada na Seção 2.2.

Teste Baseado em Defeitos (TBD)

Esta técnica utiliza-se de informações sobre os defeitos mais frequentes que podem estar presentes em um produto para derivar casos de teste. Estas informações podem variar em função da linguagem, técnica ou método de desenvolvimento do *software*. A Análise de Mutantes é um critério de teste da Técnica de Teste Baseado em Defeitos (TBD) (DELAMARO et al., 2007). No critério Análise de Mutantes é gerado um conjunto de programas semelhantes, denominados mutantes, a partir de um programa P , o qual assume-se como correto. Casos de teste são gerados com o intuito de provocar diferenças de comportamento em P e seus mutantes P' , P'' , P''' , etc. Dessa forma, é possível analisar a capacidade que um conjunto de casos de teste tem para distinguir um mutante do programa P . Tais mutantes são escolhidos de forma criteriosa, e que representem boa parte dos defeitos mais comuns encontrados no código. Os detalhes do critério Análise de Mutantes será melhor descrito na Seção 2.5.1, pois esse critério do TBD foi utilizado neste trabalho para avaliar os critérios de teste para MEF em termos de eficiência, ou seja, analisar a capacidade que o critério possui em detectar um maior número de defeitos em relação aos outros critérios.

2.2 Teste Baseado em Modelos (TBM)

A especificação é um documento que representa o comportamento que um determinado sistema deve possuir. Esse documento, também chamado de modelo, permite que o conhecimento sobre o sistema seja capturado e reutilizado durante diversas fases do desenvolvimento. Uma especificação clara é importante para definir o es-

copo do trabalho de desenvolvimento e, de forma especial, o de teste. A modelagem é muito importante para a atividade de teste, pois a partir dela é possível identificar o que um sistema deveria fazer e qual o resultado esperado, e boa parte da atividade de teste é gasta buscando tais informações. Mas, para isso deve-se considerar que o modelo esteja correto. Dessa forma o modelo torna-se um artefato que deve ser testado tanto quanto o próprio sistema. Considerando que o modelo esteja correto, ele é extremamente valioso para o teste como, por exemplo, na geração de casos de teste (DELAMARO et al., 2007).

O TBM consiste em uma técnica em que os casos de teste são derivados automaticamente de um modelo que descreve aspectos (geralmente funcionais) do sistema (BINDER, 2005). Por serem baseados em modelos e não no código fonte propriamente dito, TBM é visto como uma forma de teste de caixa-preta. Um modelo pode ser definido de diversas formas, por exemplo: MEF (SIDHU; LEUNG, 1989), *Statecharts* (HAREL, 1987), Redes de Petri (PETERSON, 1981), Estelle (BUDKOWSKI; DEMBINSKI, 1987), SDL (ELLSBERGER et al., 1997), dentre outros. No contexto deste trabalho são utilizadas MEF para geração de casos de teste, porém a especificação de alguns sistemas pode se tornar mais clara se utilizada a técnica de *Statecharts*, por exemplo. Neste trabalho alguns modelos utilizados como estudo de caso foram modelados em *Statecharts*. Embora alguns modelos sejam especificados em *Statecharts*, estes passam por uma conversão para MEF (VIJAYKUMAR, 1999), ou seja, passam a ter uma representação plana, sem características próprias do *Statecharts* como: paralelismo, profundidade, dentre outros. Em seguida, por estar diretamente relacionada com este trabalho, a técnica de MEF é apresentada com mais detalhes.

2.2.1 Máquinas de Estados Finitos (MEF)

MEF são modelos utilizados para representar o comportamento de um sistema por um número finito de estados e transições (GILL, 1962). Uma MEF tem um estado ativado sempre em resposta a um evento de entrada, e esse evento pode ocasionar a mudança de estado e a produção de uma ação de saída. Uma MEF pode ser representada tanto por um diagrama de transição de estados como por uma tabela de transição. Um diagrama de transição de estados é representado por um grafo, em que os nós são os estados e os arcos são as transições entre dois estados. Em uma tabela de transição, os estados são representados por linhas e as entradas, por colunas.

A Figura 2.1, apresenta um exemplo de uma MEF por meio de um diagrama de transição de estados. Na Tabela 2.1, a mesma MEF é representada por meio de uma tabela de transição. A máquina em exemplo especifica o comportamento de um extrator de comentários. O conjunto de entrada consiste em uma sequência composta por símbolos “*”, “/”, e “v”, sendo “v” qualquer caracter diferente de “*” e “/”. A entrada é uma cadeia de caracteres e os comentários são impressos. Um comentário é uma sequência de caracteres iniciada por “/*” e terminada por “*/”.

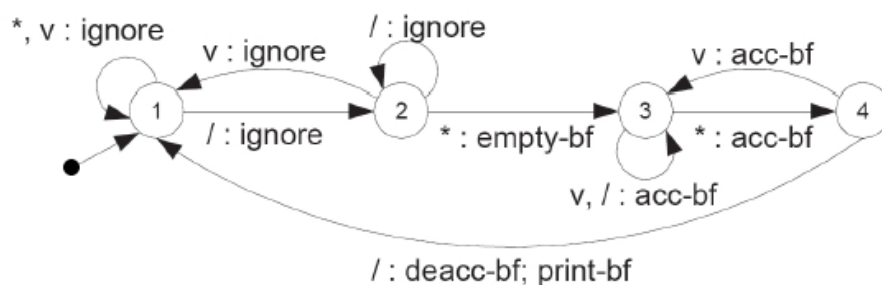


Figura 2.1 - Exemplo de uma MEF que especifica o comportamento de um extrator de comentários (Figura Adaptada).

Fonte: Chow (1978)

Tabela 2.1 - Tabela de transição de estados.

	*	/	v
1	1/ignore	2/ignore	1/ignore
2	3/empty-bf	2/ignore	1/ignore
3	4/acc-bf	3/acc-bf	3/acc-bf
4	4/acc-bf	1/deacc-bf	3/acc-bf

De acordo com Petrenko e Yevtushenko (2005), formalmente uma MEF pode ser representada por uma tupla $M=(X, Z, S, s_0, f_z, f_s)$, sendo que:

- X - é um conjunto finito não-vazio de símbolos de entrada;
- Z - é um conjunto finito de símbolos de saída;
- S - é um conjunto finito não-vazio de estados;
- $s_0 \in S$ é o estado inicial;

- $f_z - (S \times X) \rightarrow Z$ é a função de saída; e
- $f_s - (S \times X) \rightarrow S$ é a função de próximo estado.

Em relação às propriedades de uma [MEF](#), tem-se:

- Completamente especificada - uma [MEF](#) é completamente especificada se ela trata todas as entradas em todos os estados, ou seja, é aquela em que existem transições definidas para todos os símbolos de entrada de cada estado da [MEF](#). Caso contrário, ela é parcialmente especificada;
- Fortemente conectada - uma [MEF](#) é fortemente conectada se para cada par de estados (s_i, s_j) existe um caminho por transições que vai de s_i a s_j . Ela é inicialmente conectada se a partir do estado inicial é possível atingir todos os demais estados da [MEF](#);
- Minimal - uma [MEF](#) é minimal (ou reduzida) se na [MEF](#) não existe quaisquer dois estados equivalentes. Dois estados são ditos equivalentes quando possuem as mesmas entradas e produzem as mesmas saídas;
- Determinística - uma [MEF](#) é determinística quando, para qualquer estado e para uma dada entrada, a [MEF](#) permite uma única transição para um próximo estado; e
- Máquinas de Mealy - uma [MEF](#) é de Mealy quando gera uma palavra de saída para cada transição entre os estados. Neste tipo de [MEF](#) estas palavras de saída dependem do estado atual e do valor das entradas.

O [TBM](#) consiste na geração de um conjunto de casos de teste que consiga encontrar o máximo de defeitos possível em uma implementação. Podem ser encontrados os seguintes defeitos ([BINDER, 2005](#)): *i*) transições incorretas ou não implementadas; *ii*) ações incorretas ou não implementadas; e *iii*) estados extras, não implementados ou corrompidos.

Segundo [VijayKumar \(1999\)](#), *softwares* complexos utilizam recursos como paralelismo, difíceis de serem representados por [MEF](#), acarretando a necessidade de se buscar uma técnica de nível mais alto. Mesmo nos casos em que a modelagem é possível, o modelo resultante pode ser grande, tornando complexa a sua representação.

Com o propósito de aumentar o poder de modelagem, foram propostas várias extensões às MEF, tais como *Statecharts*, Estelle, dentre outras. Como já mencionado anteriormente, também é utilizada no contexto deste trabalho, especificações modeladas em *Statecharts*, porém a especificação deve ser convertida em MEF a fim de que os casos de teste possam ser obtidos através de critérios que dependem da MEF. Critérios de geração de casos de teste a partir de MEF podem ser utilizados. Os critérios concebidos para o desenvolvimento desta pesquisa, são descritos na Seção 2.3.

2.2.2 *Statecharts*

Statecharts é uma técnica formal para especificar comportamento de sistemas complexos como os sistemas reativos (HAREL, 1987). Os *Statecharts* são na verdade uma evolução dos clássicos diagramas de estados, baseados no formalismo de MEF e conseguem reproduzir representações mais enxutas e claras com a adição de alguns recursos como:

- Hierarquia de Estados - permitem representação explícita de encapsulamento de estados através de superestados e subestados;
- Ortogonalidade - possibilita descrever, explicitamente, atividades paralelas entre os componentes;
- Entrada por História - possibilita lembrar o último estado ativo ignorando o estado inicial. Quando um estado é ativado, seu estado *default* é ativado, a menos que a transição possua o símbolo história. Se isso ocorrer, será ativado o último estado visitado; e
- *Broadcasting* - permite descrever a sincronização entre os componentes ortogonais do modelo. *Broadcasting* ocorre quando uma ação (evento interno), associada a uma transição, é disparada; esse evento interno possivelmente irá disparar transições em outros componentes do modelo, caracterizando assim a reação em cadeia.

Os elementos fundamentais para se especificar um sistema em *Statecharts* são: Estados, Eventos, Condições, Ações, Expressões, Variáveis, Rótulos e Transições. Estados são usados para descrever componentes de um determinado *software* e podem ser classificados em dois grupos: básicos e não-básicos. Os estados básicos são aqueles

que não possuem subestados. Já os não-básicos são decompostos em subestados. Essa decomposição pode ser de dois tipos: *XOR* ou *AND*. Se a decomposição é do tipo *XOR*, então esse estado do sistema não poderá estar em mais de um sub-estado simultaneamente. Entretanto, se a decomposição é do tipo *AND*, esse estado deverá encontrar-se em mais de um sub-estado simultaneamente.

Na Figura 2.2 é ilustrado um exemplo de um sistema representado por *Statecharts*. Os estados A, D e H representam superestados que abrangem mais de um estado. Para o superestado A, o estado B será o estado inicial. Para o superestado S será o estado F e para o estado H será o estado J.

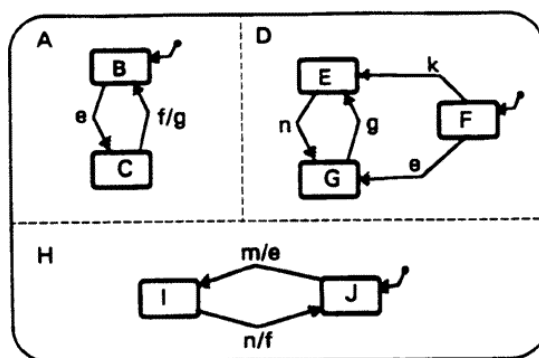


Figura 2.2 - Exemplo de um *Statecharts* mostrando ações associadas a eventos.

Fonte: Harel (1987)

2.3 Critérios de Testes para MEF

A especificação de um sistema por meio de *MEF* viabiliza a geração automática de casos de teste. Ao longo dos anos diversos critérios de teste foram propostos para selecionar conjuntos de casos de teste a partir de uma *MEF*. Um conjunto de caso de teste são sequências de transições que podem ser geradas a partir de uma *MEF*. Um caso de teste corresponde a cada sequência de transição numa mesma direção (BINDER, 2005).

Segundo Myers (2004), é impraticável utilizar todos os elementos de um determinado domínio de entrada para verificar as características funcionais e operacionais de um programa em teste. Como normalmente o conjunto de elementos do domínio é infinito ou extremamente grande, a obtenção de conjuntos significativos e representativos de

casos de teste torna-se fundamental. É necessário criar casos de teste que tenham uma probabilidade de encontrar a maior quantidade de defeitos no menor tempo e com um menor esforço. Dessa forma, os critérios de teste desempenham um papel de extrema relevância, pois podem derivar casos de teste a partir de MEF.

Na literatura, encontram-se muitos trabalhos que definem critérios de geração de casos de teste, ou melhoria nos critérios já existentes. O critério W, criado por Chow (1978), por exemplo, é considerado um dos critérios mais clássicos no contexto de geração de casos de teste baseado em MEF. Chow (1978) afirmou que este critério seria o mais eficiente para detectar as seguintes classes de erros: erros de operação, erros de transferência, erros de estados extras e erros de estados ausentes. Ao longo dos anos, diversos outros critérios foram criados em melhoria ao critério W.

Este trabalho apresenta o desenvolvimento e a análise de três critérios de teste para MEF, o critério DS, critério UIO e o critério *Switch Cover*. O desenvolvimento desses critérios para a integração aos ambientes de teste desenvolvidos no INPE serão explorados com mais detalhes no Capítulo 3. Neste Capítulo serão apresentadas apenas algumas características dos critérios.

2.3.1 Critério *Distinguishing Sequence* (DS)

O critério DS, proposto por Gonenc (1970), utiliza uma sequência de distinção (SD) para gerar os casos de teste. SD é uma sequência de símbolos de entrada que, quando aplicada aos estados da MEF, produz saídas distintas para cada um dos estados. Ou seja, observando-se a saída produzida pela MEF como resposta à SD, pode-se determinar em qual estado a MEF estava originalmente. No entanto, segundo Gill (1962), tal sequência pode não existir. Nesse critério é importante selecionar a menor sequência de distinção para que, conseqüentemente, se obtenha um conjunto menor de casos de teste. De acordo com Gonenc (1970), além da existência da SD, o critério DS só poderá ser aplicado em MEF determinísticas, completas, fortemente conectadas e mínimas.

Nesse trabalho, o critério DS desenvolvido está descrito no trabalho de Sidhu e Leung (1989) e no trabalho de Delamaro et al. (2007). Segundo Sidhu e Leung (1989), o conjunto de testes é gerado a partir de duas etapas. A primeira etapa é encontrar a(s) SD, se esta existir e a segunda etapa é encontrando uma β -sequence.

Primeira etapa

A *SD* pode ser encontrada, construindo-se uma árvore a partir da *MEF*, essa árvore é chamada de Árvore de Distinção (KOHAVI, 1978), que utiliza o conceito de grupos de incerteza, sendo que um grupo de incerteza é um determinado grupo de estados que ainda não foram distinguidos e para os quais é necessário que alguma entrada seja apresentada à sequência para ela se tornar uma *SD*. A árvore também é formada por grupos chamados triviais, sendo este um grupo em que todos os conjuntos de estados são unitários ou vazios. Como exemplo da implementação do critério, será utilizada a Figura 2.3 que mostra todos os exemplos de grupos existentes.

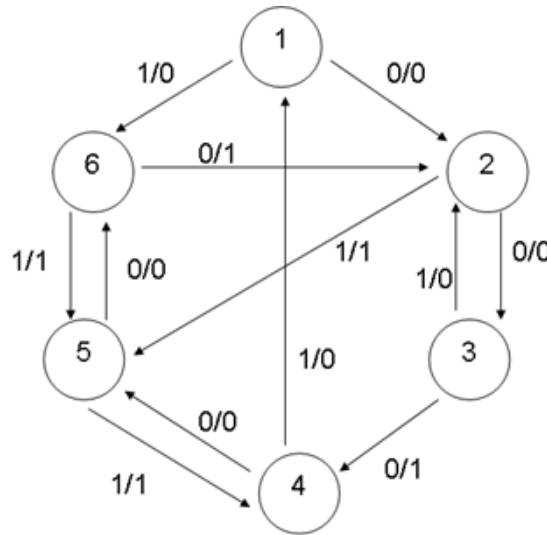


Figura 2.3 - Máquina de Estados Finito: Exemplo Implementação do Critério DS.
Fonte: Delamaro et al. (2007)

Um grupo g pode ser particionado em relação ao valor de entrada x , criando um grupo g' no qual todos os conjuntos de estados são subconjuntos dos conjuntos de g . É importante destacar que no particionamento, são mantidos juntos os estados que produzem a mesma saída em relação a x . Considerando a árvore de distinção criada a partir da *MEF* (Figura 2.4), um exemplo de particionamento seria: o grupo $\{\{0\}, \{1, 3, 4\}, \{2\}\}$ pode ser particionado em relação a 0, criando o grupo $\{\{2, 3, 0\}\}$. Após particionado, e considerando-se a saída, deve-se calcular os próximos estados correspondentes do grupo obtido em relação à entrada em questão.

Quando se atualiza um determinado grupo, este grupo pode-se tornar inconsistente. Um grupo é inconsistente quando, ao calcular os próximos estados, se percebe que dois estados que ainda não foram distinguidos levam ao mesmo estado. Dessa forma, não poderão ser mais distinguidos. Observe na árvore o que acontece com os estados 2 (dois) e 6 (seis). Ao particionar o grupo $\{\{2,3,5,6\}, \{2,4\}\}$ em relação a entrada 1(um), obtém-se o grupo $\{\{2,5,6\}, \{3\}, \{4,2\}\}$, que é inconsistente, pois tanto o estado 2 como o estado 6 levam ao estado 5 (cinco) quando a entrada 1 é aplicada.

Terminado o particionamento, é possível observar uma estrutura de grupos que representa uma árvore de distinção com as seguintes características:

- Em uma árvore de distinção cada nó é formado por um grupo de incerteza;
- O nó raiz é formado pelo grupo no qual todos os estados ainda não foram distinguidos;
- Cada nó possui um filho para cada símbolo de entrada;
- O grupo do nó filho é calculado a partir do grupo do nó pai, particionando-o e, caso o grupo seja inconsistente, atualizando-o;
- A árvore é construída por nível a partir da raiz;
- Um nó é folha se for inconsistente ou se for trivial; e
- Uma **SD** é a sequência de símbolos de entrada presentes no caminho que vai da raiz da árvore a um nó, cujo grupo seja trivial.

A Figura 2.4 apresenta a árvore de distinção para a **MEF** (Figura 2.3). Perceba que os grupos triviais são marcados com uma elipse. Os grupos inconsistentes são marcados com retângulos. Depois de criada a árvore, deve ser realizada busca em profundidade com o propósito de atingir um grupo que seja trivial. Cada sequência gerada é dita como **SD**. Depois de geradas as sequências **SD**, é importante utilizar a menor delas para se obter, no final, os menores casos de teste. As sequências **SD** geradas para a árvore em questão são: 0 0 1, 0 0 0 1, 0 0 0 0 0 e 0 0 0 0 1. E a **SD** a ser utilizada seria a 0 0 1, por ser a menor delas.

Tomando como exemplo a **MEF** da Figura 2.5, é gerando sua respectiva árvore de distinção, nota-se que não foram encontrados grupos inconsistentes. Todos os

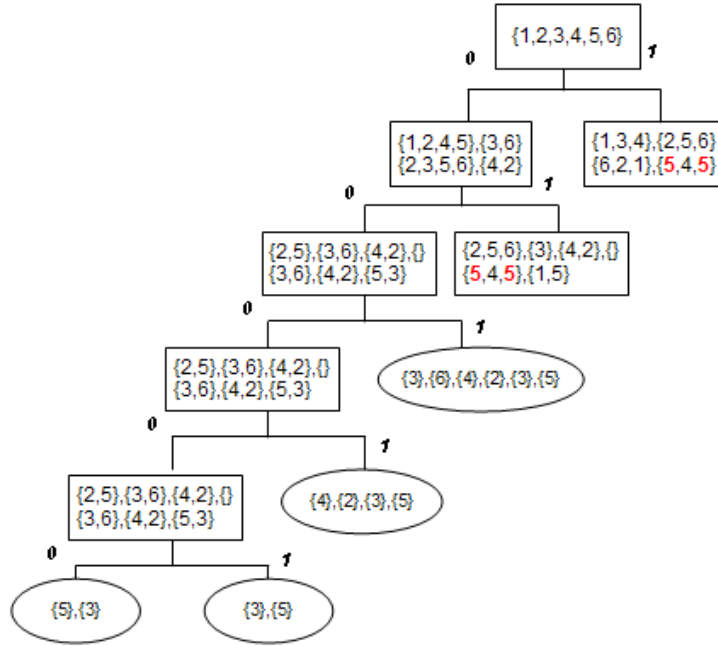


Figura 2.4 - Exemplo 01: Árvore de distinção gerada a partir de uma MEF.
Fonte: Delamaro et al. (2007)

grupos finais da árvore são grupos triviais. A árvore gerada para essa MEF pode ser observada na Figura 2.6 na qual a menor sequência SD gerada é B B. Devido ao tamanho da imagem, foi colocado na Figura 2.6 apenas uma parte da árvore gerada.

Exercitando a sequência BB no modelo, tem-se as seguintes saídas apresentadas na Tabela 2.2. Note que nenhuma saída é igual. Seguindo a segunda etapa do algoritmo e aplicando a menor SD encontrada em cada *subsequence* da β -sequence, tem-se os seguintes casos de teste DS: ABB, BBB, AAAAABB, AAAABBB, AAAABB, AAABBB, AABB, AB BB, AAABB e AABBB.

Segunda etapa

A segunda etapa é a construção de uma β -sequence. Uma β -sequence é uma sequência de símbolos de entrada que leva a MEF a um estado s . A β -sequence, é formada por um conjunto de subsequências (ou *subsequences*), ou seja, β -sequence = Σ *subsequences*, onde cada *subsequence* é formada por um conjunto de símbolos de entrada, que parte do estado inicial até chegar ao estado s_j de uma transição (s_i, s_j) .

Tabela 2.2 - Saídas geradas a partir de cada estado da MEF quando aplicado a sequência SD.

Estado	SD
0	$\lambda \lambda$
1	0 0
2	0 1
3	1 1
4	1 λ

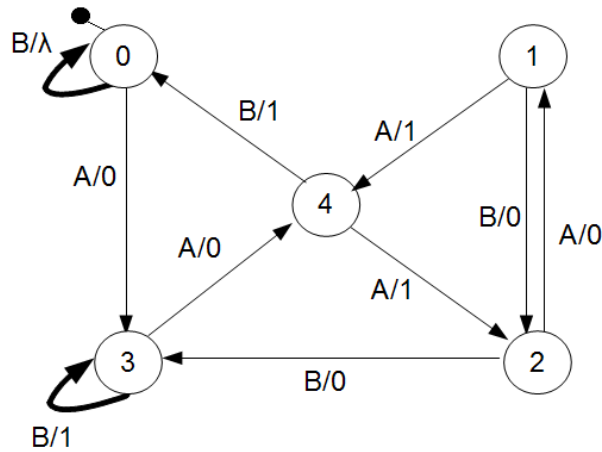


Figura 2.5 - Máquina de Estados Finitos para exemplo dos critérios
Fonte: Sidhu e Leung (1989)

A Tabela 2.3 apresenta as *subsequences* da β -sequence com a SD e as transições alcançadas. Na coluna Transição, são apresentadas as transições que foram alcançadas para gerar as *subsequences* da β -sequence.

2.3.2 Critério *Unique Input/Output* (UIO)

Conforme pôde ser visto na Seção anterior, nem todas as MEF possuem uma sequência SD. Entretanto, ainda é possível fazer uma distinção entre um estado e os demais usando uma sequência Única de Entrada e Saída (UES). O critério UIO, proposto originalmente por Sabnani (1988), produz uma sequência de identificação de estado, ou seja, uma sequência UES é utilizada para verificar se a MEF está em um estado particular. Dessa forma, para cada estado da MEF pode-se ter uma sequência UES distinta, em função de considerar as entradas e as saídas. Assim como o critério DS, o critério UIO só poderá ser aplicado em MEF determinísticas, completas, fortemente

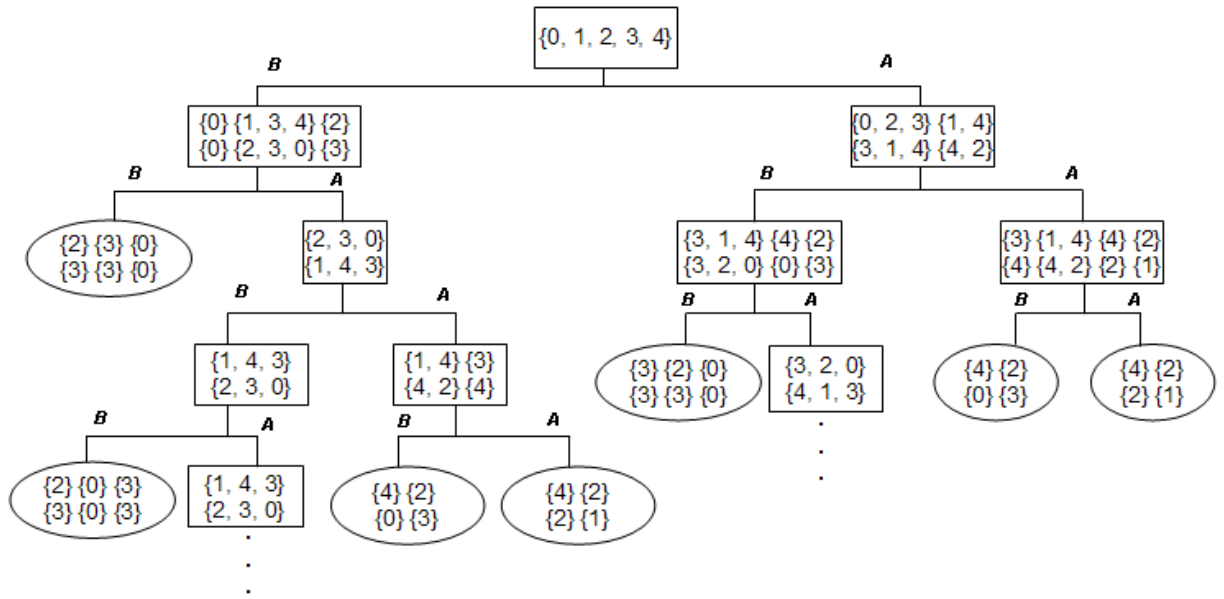


Figura 2.6 - Exemplo 02: Árvore de distinção gerada a partir de uma MEF.

Tabela 2.3 - Descrição do critério DS.

<i>Subsequence da β-sequence</i>	<i>Transição</i>	<i>Subsequence + SD</i>
A	(0, 3)	A B B
B	(0, 0)	B B B
A A A A A	(1, 4)	A A A A A B B
A A A A B	(1, 2)	A A A A B B B
A A A A	(2, 1)	A A A A B B
A A A B	(2, 3)	A A A A B B B
A A	(3, 4)	A A B B
A B	(3, 3)	A B B B
A A A	(4, 2)	A A A B B
A A B	(4, 0)	A A B B B

conectada e mínimas.

Da mesma forma que o critério DS, o critério UIO também foi desenvolvido com base no trabalho descrito por Sidhu e Leung (1989). O conjunto de teste é gerado a partir de duas etapas do algoritmo. A primeira etapa é encontrar a sequência UES para cada estado da MEF. A segunda etapa é a mesma apresentada para o critério DS para encontrar a β -sequence. A diferença encontra-se no terceiro passo, onde, ao

contrário do critério **DS** que aplica a menor **SD** encontrada, o critério **UIO** aplica-se à sequência **UES** do estado s_i .

Primeira etapa

Para encontrar a sequência **UES** de cada estado de uma determinada **MEF**, são realizadas as seguintes verificações: partindo de um estado s_i da **MEF**, é realizada uma busca em largura, sendo que em cada estado alcançado é verificado se a entrada e saída correspondente desse estado é única em relação aos outros estados da **MEF**. Para isso deve-se percorrer novamente toda a **MEF**, fazendo esta verificação. Caso seja constatado que a entrada e a saída são únicas, a busca em largura é encerrada e essa será a sequência **UES** para o estado s_i . Por outro lado, se a entrada e a saída não forem únicas, a busca em largura pela **MEF** continua concatenando as próximas entradas e saídas dos próximos estados e verificando se são únicas. Esse processo é realizado para cada um dos estados da **MEF**.

A Figura 2.7 ajudará a compreender como a busca ocorre. Note que a busca está iniciando pelo estado 1 (um) procurando a **UES** para esse estado. Primeiro será analisada a entrada e saída 1/0 (transição do estado 1 para o estado 6 (seis)). Se essa não for uma sequência **UES**, será então analisada a entrada e saída 0/0 (transição do estado 1 para estado 2 (dois)). Se mesmo assim não encontrar uma sequência **UES**, começa então a descer a árvore, concatenando os valores das transições que já foram visitados, ou seja, 1/0 e 0/1, depois 1/0 e 1/1. Assim, sucessivamente, até encontrar a sequência **UES** para o estado 1.

O uso da busca em largura no critério **UIO** garante que as sequências **UES** encontradas serão as menores para cada estado da **MEF**. Usando o mesmo exemplo de **MEF** da Seção anterior, aplicando a primeira fase do algoritmo é possível encontrar a sequência **UES** para cada estado da **MEF** em questão (Tabela 2.4), ou seja, apenas aquele estado em questão tem aquele conjunto de entrada e saída dentro da **MEF**.

Segunda etapa

A segunda etapa do critério **UIO** é similar ao critério **DS**, pois também é necessário encontrar a β -sequence da **MEF**, porém, para fazer a junção das **UES** e da β -sequence, leva-se em conta a qual estado a **UES** pertence. Para saber em qual *subsequence* da β -sequence colocar a **UES** correspondente, deve-se considerar o estado de destino da transição que está sendo alcançado a partir do estado inicial.

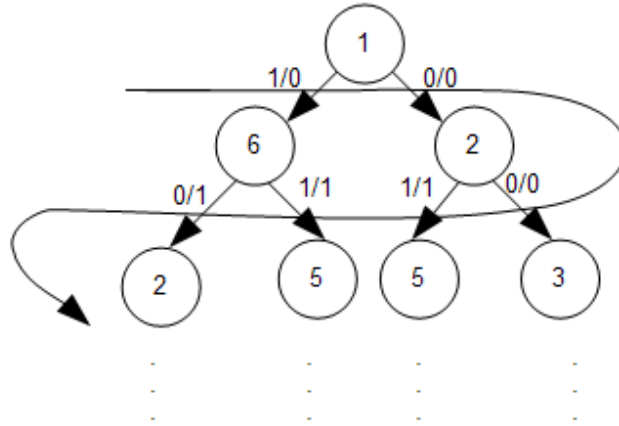


Figura 2.7 - Árvore de representação a busca em largura do método UIO.

Tabela 2.4 - Sequências de teste UES de cada estado da MEF.

Estado	UES
0	B/ λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

Por exemplo, seguindo o mesmo exemplo de MEF (Figura 2.5), foram geradas as sequências UES apresentadas na Tabela 2.5.

A Tabela 2.6 apresenta as *subsequences* da β -sequence com seus respectivos UES e as transições alcançadas. Not, na coluna Transição, as transições que foram alcançadas para gerar as *subsequence* da β -sequence. A primeira transição, por exemplo, parte do estado 0 e vai para o estado 3. Sendo o estado 3, o estado destino, então deve ser concatenada à *subsequence* da β -sequence a UES do estado 3, que é B B.

2.3.3 Critério *Switch Cover*

O critério *Switch Cover* é um critério de teste conhecido como “todas as combinações” (PIMONT; RAULT, 1976). Segundo Pimont e Rault (1976), este critério é um dos mais rigorosos em relação a cobertura da MEF, pois define que todos os pares de transições devem ser executados, ou seja, todos os pares de transições adjacentes

Tabela 2.5 - Tabela das UES encontradas.

Estado	UES
0	B
1	A A
2	B
3	B B
4	A A

Tabela 2.6 - Descrição do critério UIO.

<i>Subsequence da β-sequence</i>	Transição	<i>Subsequence + UES</i>
A	(0, 3)	A B B
B	(0, 0)	B B
A A A A A	(1, 4)	A A A A A A A
A A A A B	(1, 2)	A A A A B B
A A A A	(2, 1)	A A A A A A
A A A B	(2, 3)	A A A A B B B
A A	(3, 4)	A A A A
A B	(3, 3)	A B B B
A A A	(4, 2)	A A A B
A A B	(4, 0)	A A B B

devem ser cobertos. No entanto, dependendo do tamanho da MEF, a execução de todas as combinações das transições pode ser considerada inviável. Na teoria de grafos, o algoritmo usado para implementar o critério *Switch Cover* é conhecido como sequências de “*de Bruijn*” (ROBINSON, 1999).

Pelo fato de ser um pouco mais complexo, seguindo de diversas etapas para a implementação deste critério, será utilizado uma máquina com menos estados (Figura 2.8) para representar seu algoritmo e facilitar o entendimento do mesmo.

Os passos para a implementação do algoritmo do critério *Switch Cover* são:

a) Criação do Grafo Dual a partir do original

Nessa primeira etapa do algoritmo todas as transições da MEF devem ser transformadas em estados (Figura 2.9). De acordo com Pimont e Rault (1976), transformar as transições em estados torna o algoritmo mais fácil para garantir a cobertura de todos os pares de transições.

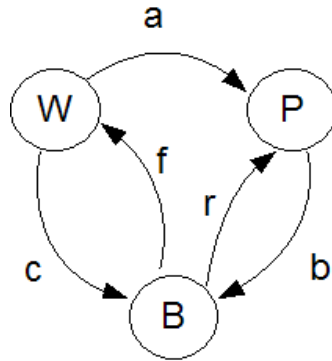


Figura 2.8 - Exemplo de uma MEF simples.
Fonte: Amaral (2005)

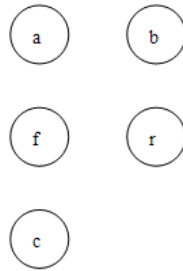


Figura 2.9 - Criação do Grafo dual a partir da MEF original.

b) Criação das novas transições

Considerando o comportamento das transições que existem na MEF original, devem ser criadas as novas transições no grafo dual. Para criação das novas transições leva-se em conta os pares de transições existentes no grafo dual. Por exemplo, na MEF original tem-se saindo do estado W uma transição *a*, que está indo para o estado P (Figura 2.10(a)). A transição adjacente à transição *a* que sai do estado P é a transição *b*. Logo, no grafo dual existirá uma transição que liga os estados *a* e *b* (Figura 2.10(b)). Isso ocorre para todos os outros pares de transição da MEF original.

c) Balanceamento do Grafo Dual

A partir do grafo dual gerado, as polaridades dos nós deverão ser balanceadas, construindo um Grafo Euleriano. Como já mencionado no item anterior, o algoritmo deve percorrer todos os pares de transição. Para isso

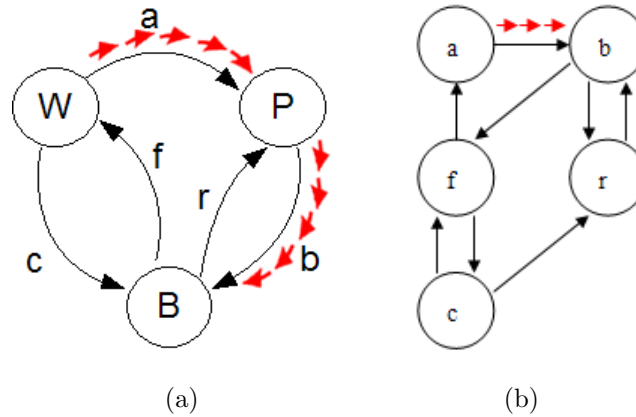


Figura 2.10 - Criação das novas transições a partir da MEF original.

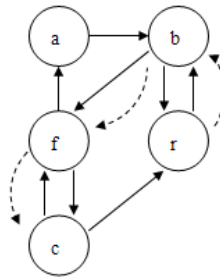


Figura 2.11 - Grafo Dual balanceado.

ocorrer é necessário garantir que, quando uma transição entrar em um estado, seja possível sair desse estado por uma transição de saída. Dessa forma, pode-se encontrar um caminho chamado de caminho euleriano. Um caminho euleriano é um caminho no grafo onde cada aresta é visitada apenas uma vez (LIPSCHUTZ; LIPSON, 1997). A Figura 2.11 representa a MEF da Figura 2.10(b) balanceada. Note que a quantidade de transições de entrada em um estado, é a mesma quantidade de transições de saída.

d) Geração das Sequências de Teste

O grafo deverá ser percorrido gerando as sequências de teste, sempre partindo do estado inicial e retornando para ele. Neste exemplo, foi considerado que tanto o estado *a* como o estado *c* são iniciais, pois eles eram transições que saíam do estado *W*, considerado inicial na MEF original.

Para esse exemplo têm-se as seguintes sequências geradas: *abf*, *abrbf*, *cf*, *crbf*.

2.3.4 Comparação entre os Critérios

Para que um projetista de teste escolha um critério de geração de casos de teste com o objetivo de aplicá-lo em alguma especificação baseada em MEF, é necessário que algumas características sejam observadas. Essas características referem-se a exigência de cada critério para que a MEF possua determinadas propriedades, ao tamanho das sequências geradas e a aplicabilidade de cada uma.

É importante ressaltar também que a escolha do melhor critério depende não só das propriedades da MEF, mas também do custo e da eficiência do critério, pois desejar-se o menor custo e sem prejudicar a eficiência. A Tabela 2.7 apresenta um resumo comparativo de todas as propriedades necessárias à aplicação de cada critério.

Tabela 2.7 - Comparação entre os critérios UIO, DS e *Switch Cover*.

Propriedades	DS	UIO	<i>Switch Cover</i>
Minimilidade	X	X	X
Completamente Especificada	X	X	X
Fortemente Conectada	X	X	X
Determinismo	X	X	X
Sequência de Distinção	X		
Sequência única de entrada e saída		X	
Máquina de Mealy	X	X	

2.4 Critérios de Testes para *Statecharts*

Embora os critérios de teste para *Statecharts* não sejam foco deste trabalho, para conhecimento, são apresentados brevemente nesta Seção alguns critérios de teste para *Statecharts*.

Para validação de especificações baseadas em *Statecharts*, existem alguns critérios de teste baseados em fluxo de controle da especificação que abordam as características específicas da técnica *Statecharts*. Souza (2000) em seu trabalho propôs a Família de Critérios de Cobertura para *Statecharts* (FCCS). Essa família de critérios de teste auxilia o processo de validação de aspectos de fluxo de controle e as características

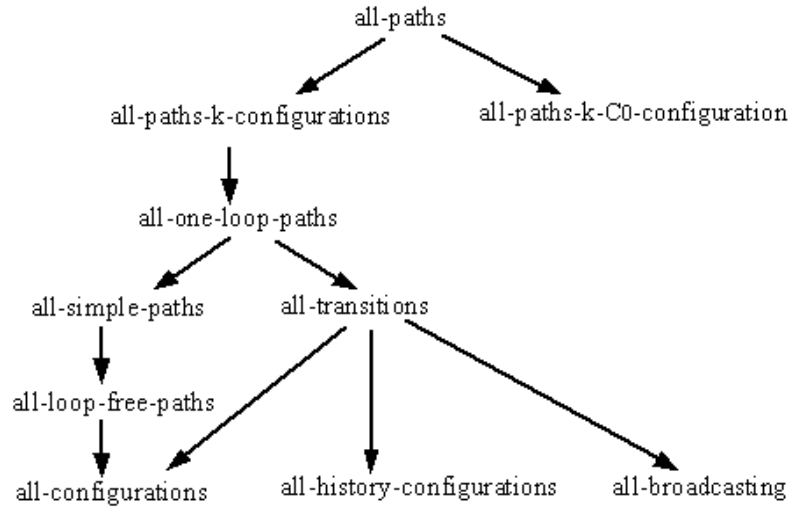


Figura 2.12 - Relação hierárquica de família FCCS.
 Fonte: Souza (2000)

intrínsecas da técnica *Statecharts*, como, por exemplo, história, paralelismo e *broadcasting*, fornecendo mecanismos para analisar a cobertura do teste de sequência de uma especificação. A Figura 2.12 apresenta a relação hierárquica entre esses critérios.

De cima para baixo da hierarquia, seguindo as setas, um critério estritamente inclui outro. Por exemplo, o critério *all-paths* inclui o critério *all-paths-k-configurations*, bem como *all-paths-k-C0-configuration*. Alguns desses critérios são:

- *All-configuration* - requer que todas as configurações do *Statecharts* sejam alcançadas pelo menos uma vez na sequência de teste;
- *All-transitions* - requer que todas as transições do *Statecharts* sejam alcançadas pelo menos uma vez na sequência de teste;
- *All-paths* - requer que todos os caminhos do *Statecharts* sejam alcançados pelo menos uma vez na sequência de teste; e
- *All-simple-paths* - requer que todos os caminhos simples do *Statecharts* sejam alcançados pelo menos uma vez na sequência de teste.

Atualmente encontra-se incorporado ao ambiente de teste GTSC os critérios *All-transitions* e *All-simple-paths*. Porém, estes critérios não estão associados a este

trabalho. As contribuições iniciais foram as implementações e incorporações, aos ambientes *GTSC* e *WEB-PerformCharts*, três critérios de teste para MEF: *DS*, *UIO* e *Switch Cover*.

2.5 Avaliação de Critérios de Teste para MEF

Diversos estudos teóricos e experimentais permitem comparar os critérios de teste existentes, pois decidir qual deles deva ser utilizado não é uma tarefa simples. Embora os critérios possuam um objetivo comum de verificar se uma implementação está em concordância com o que foi especificado, eles diferem em relação ao custo da geração das sequências de teste, tamanho do conjunto de casos de teste e capacidade de detecção de defeitos (efetividade).

De acordo com [Fujiwara et al. \(1991\)](#), um critério deve gerar um conjunto de casos de teste que possua as seguintes características: *i*) ser relativamente pequeno para que se tenha baixo custo de execução durante o teste da implementação; e *ii*) conseguir cobrir o máximo de defeitos possíveis que uma implementação possa ter.

A realização de uma avaliação preliminar dos critérios de teste utilizando-se de métricas, como custo e eficiência dos casos de teste concebidos por cada critério, encontra-se entre os objetivos deste trabalho. Entende-se por custo o esforço necessário para que o critério seja usado. Em relação à eficiência, é avaliada a capacidade que um critério possui em detectar um defeito. Para isso pode ser utilizado o critério Análise de Mutantes da técnica *TBD*. Maiores detalhes desses fatores de análise são encontrados nas próximas Seções.

2.5.1 Eficiência

Através da técnica de *TBD*, é possível definir a eficiência de um determinado critério de teste em encontrar um defeito no código. A *TBD* utiliza informações dos tipos mais comuns de erros no processo de desenvolvimento do *software* para derivar os requisitos de testes. A Análise de Mutantes, é um critério de teste da *TBD*, de forma que o programa que está sendo testado é alterado diversas vezes, criando-se um conjunto de programas alternativos ou mutantes. O objetivo é inserir falhas no programa original, e usando um determinado conjunto de casos de teste, encontrar a diferença de comportamento entre o programa original e o programa mutante ([DELAMARO et al., 2007](#)).

Os principais passos para aplicar esse critério são: geração dos mutantes, execução do programa de teste, execução dos mutantes e a análise dos mutantes. Para a geração dos mutantes, considera-se um conjunto Φ de programas alternativos, que depende de um programa P . Tal conjunto $\Phi(P)$ é chamado de “vizinhança” de P (DELAMARO et al., 2007). A intenção é mostrar que um programa é correto por meio de um processo de eliminação, mostrando que nenhum programa da vizinhança de P , não equivalente a P , é correto. Isso é possível utilizando-se um conjunto de casos de teste T para o qual todos os membros de $\Phi(P)$, não equivalentes a P , falhem em pelo menos um caso de teste.

A geração dos mutantes é o passo chave da Análise de Mutantes. O sucesso da aplicação do critério depende totalmente da escolha da vizinhança $\Phi(P)$. A vizinhança $\Phi(P)$ deve: *i*) ser abrangente, de modo que um conjunto de casos de teste adequado de acordo com a vizinhança gerada, consiga revelar a maior parte dos erros de P ; e *ii*) ter cardinalidade pequena, para verificar se um conjunto de casos de teste é adequado.

Para a criação dos mutantes $\Phi(P)$, utilizam-se operadores de mutação. Um operador de mutação aplicado a um programa P , transforma P em uma versão modificada da versão original, os quais deverão apresentar um comportamento diferente do esperado (MA et al., 2009). A idéia principal é gerar um conjunto $\Phi(P) = \{M_1, M_2, \dots, M_n\}$ de mutantes e aplicar a cada caso de teste T em P e $\forall M_i \in \Phi(P)$. Dentro desse contexto, uma sequência de teste é considerada adequada quando detecta (mata) o mutante. Assim, procura-se obter casos de teste que resultem em mutantes mortos. A Tabela 2.8, apresenta alguns exemplos de operadores de mutação em trechos de código de classes e métodos na linguagem Java.

Tabela 2.8 - Exemplos de operadores de mutação para linguagem Java

Operador de Mutação	Especificação	Original	Mutante
ROR	<i>Relational Operator Replacement</i>	if (x>y) ...	if (x<y) ...
COR	<i>Conditional Operator Replacement</i>	if (x==y) ...	if (x!=y) ...
AMC	<i>Access Modifier Change</i>	public Carro c;	private Carro c; protected Carro c;

A etapa de execução consiste em executar o programa P , usando os casos de teste selecionados e verificar se o seu comportamento é o esperado. Logo em seguida é realizada a execução dos mutantes. Cada mutante é executado usando o conjunto de casos de teste T (Figura 2.13). Se um mutante apresentar resultado diferente de P , isso significa que os casos de teste utilizados são sensíveis e conseguem encontrar a diferença entre P e M . Sendo assim, M é considerado um mutante morto. Se M apresentar um comportamento igual a P , diz-se que M continua vivo. Isso pode indicar uma fraqueza em T , pois não conseguiu diferenciar P de M , ou que P e M são equivalentes (OFFUTT; CRAFT, 1994).

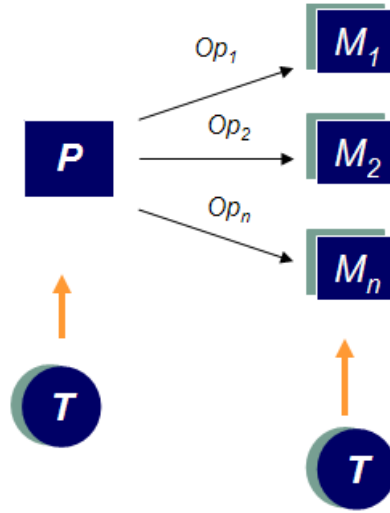


Figura 2.13 - Critério Análise de Mutantes.

Segundo o critério análise de mutantes, a eficiência de um conjunto de casos de teste está relacionada com a habilidade que eles têm para matar um mutante (FERREIRA et al., 2008). O escore de mutação (*mutation score*) pode definir o quão eficiente um critério de teste. O escore pode variar entre 0 (zero) e 1 (um). Dado o programa P e o conjunto de casos de teste T , pode-se calcular o escore de mutação $MS(P, T)$ da seguinte forma:

$$MS(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

onde:

$MS(P, T)$ = escore de mutação;

$DM(P, T)$ = número de mutantes mortos pelo conjunto de casos de teste T ;

$M(P)$ = número de mutantes gerados a partir do programa P ; e

$EM(P)$ = número de mutantes gerados que são equivalentes a P .

Quanto mais próximo de 1 for o escore de mutação, mais adequado será o conjunto de casos de teste gerados para a aplicação em teste. O principal problema da análise de mutantes é o grande esforço no processo de identificação de quais são os mutantes equivalentes e também a quantidade de mutantes gerados. Essa quantidade pode exigir um alto custo computacional na execução desses mutantes, sendo necessária a utilização de ferramentas de teste que apoiem e gerenciem tal tarefa. Entretanto, até o final desta pesquisa, ainda não havia sido encontrada, na literatura, nenhuma ferramenta que apoiasse a geração e execução de mutantes para aplicações de sistemas embarcados. Dessa forma, os mutantes para os experimentos aqui apresentados foram gerados de forma manual.

2.5.2 Custo

O custo pode ser medido pelo número de casos de teste necessários para satisfazer um critério ou por outras métricas dependentes do critério (DELAMARO et al., 2007). Algumas métricas referentes a custo são:

- Tempo necessário para executar todos os mutantes gerados. Normalmente o teste de mutação pode gerar um número muito grande de mutantes, gerando um alto custo para execução;
- Tempo gasto para identificar quais mutantes são equivalentes. É uma tarefa que requer muita intervenção humana, por exemplo, a do testador. É o testador que vai decidir se o teste continua ou não e se o conjunto de casos de teste T é um bom conjunto para um determinado programa P (OFFUTT; CRAFT, 1994);
- Tamanho dos conjuntos de casos de teste e também do tamanho das sequências de eventos referente a cada caso de teste; e
- Tempo necessário para que um critério de teste encontre um defeito no código.

Neste trabalho são utilizadas como métricas para medir o custo dos critérios de teste analisados, o tempo necessário para que um critério encontre um determinado defeito no código fonte. Também são utilizados como medida de custo, o tamanho dos conjuntos de casos de teste e a quantidade de eventos pertencente aos casos de teste.

2.6 Ferramentas de apoio ao Teste de *Software*

É importante que o teste seja realizado de forma automatizada para que seja eficiente. No entanto, torna-se cada vez mais necessário o desenvolvimento de ferramentas de suporte e automatização da atividade de teste, principalmente por tornar viável a aplicação prática de critérios de teste, bem como propiciar a realização desta atividade em sistemas maiores e mais complexos (SILVEIRA, 2001).

Segundo Harrold (2000), o uso de ferramentas de teste de *software* auxilia no processo de transferência de tecnologia para a indústria, constituindo um fator importante para sua contínua evolução. As ferramentas de teste de *software* também auxiliam na condução de estudos empíricos que visem à análise e a comparação entre os critérios e as estratégias de teste existentes.

Neste trabalho, as ferramentas para geração automática dos casos de teste utilizadas para incorporar os critérios de teste implementados estão em volta a dois ambientes: os ambientes GTSC e WEB-PerformCharts. A base do ambiente GTSC e da WEB-PerformCharts, descritas a seguir, é a ferramenta PerformCharts (VIJAYKUMAR, 1999). A primeira versão da ferramenta consistiu em associar a especificação Statecharts a uma Cadeia de Markov para gerar probabilidades limite para avaliar desempenho de sistemas reativos. Como a Cadeia de Markov é visualizada por um diagrama de estados e transições e o mesmo ocorre com a MEF, a PerformCharts foi adaptada para gerar casos de testes automaticamente (AMARAL, 2005).

2.6.1 Ambiente de Teste: Geração Automática de Casos de Teste Baseada em Statecharts (GTSC)

O ambiente GTSC é um produto do projeto de pesquisa e desenvolvimento no INPE, resultante da cooperação entre o CEA e o LAC. O ambiente GTSC permite a geração automática de casos de teste baseando-se em modelagem comportamental em Statecharts e em MEF (SANTIAGO et al., 2008) e tem sido extensivamente usado em estudos de caso que envolvem *software* embarcado em instrumentos que estão a

bordo de satélites científicos e balões estratosféricos (SANTIAGO et al., 2006) (SANTIAGO et al., 2008).

O Ambiente integra uma série de ferramentas, de modo que o projetista de teste possa ter um ambiente de modelagem comportamental de *software* em *Statecharts* e em MEF para gerar casos de teste. A Figura 2.14 apresenta a arquitetura do GTSC.

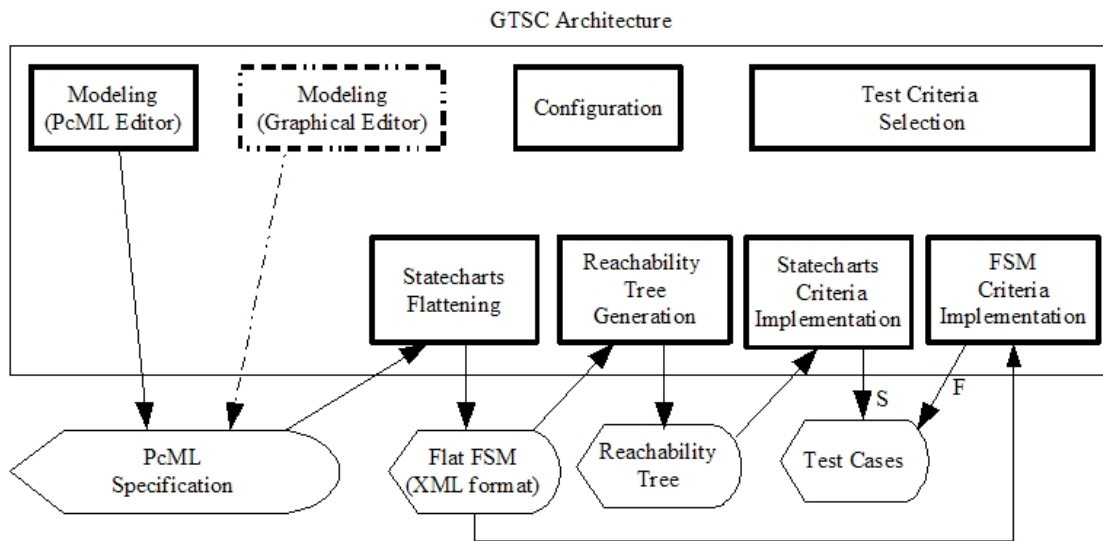


Figura 2.14 - Visão arquitetural do ambiente GTSC.
Fonte: Santiago et al. (2010)

O ambiente GTSC converte um modelo em *Statecharts* para uma MEF plana, ou seja, um modelo em que todas as características de hierarquia e ortogonalidade dos modelos em *Statecharts* são removidas. O projetista de teste deve traduzir a especificação em *Statecharts* para a linguagem *PerformCharts Markup Language* (PcML) (SANTIAGO et al., 2006) para usar o ambiente. Como ilustra a Figura 2.14, o GTSC fornece um editor para escrever as especificações em PcML (*PcML Editor*). As linhas pontilhadas do editor gráfico (*Graphical Editor*) significam que este está sendo desenvolvido e, uma vez desenvolvido, o projetista poderá desenhar o modelo em *Statecharts* e a correspondente especificação em PcML será obtida automaticamente.

Atualmente, três abordagens podem ser adotadas para geração automática de casos de testes por meio do GTSC. As primeiras duas se referem à técnica *Statecharts*. Os

critérios de testes para *Statecharts* implementados no GTSC são baseados na FCCS (SOUZA, 2000). Até o momento, os critérios *All-transitions* e *All-simple-paths* estão implementados no GTSC. A segunda abordagem, também relacionada à *Statecharts*, leva em consideração o fato de que o GTSC gera uma MEF plana no início do processo. Com a MEF, é possível escolher um critério para geração de casos de teste. Na terceira abordagem, algumas características do *software* podem ser modeladas por *Statecharts* (quando, por exemplo, componentes paralelos são necessários de serem representados) e outras por MEF (quando, por exemplo, a reatividade não é tão complexa). Como um *Statecharts* plano é uma MEF (de fato, uma MEF estendida), um projetista de testes pode especificar modelos em MEF, em PcML, e utilizar o GTSC para gerar automaticamente casos de testes.

A fim de tornar o ambiente GTSC um ambiente mais robusto, surgiu a necessidade de melhorar o ambiente. Dessa forma, um dos objetivos deste trabalho de pesquisa foi a implementação dos critérios: DS, UIO e *Switch Cover*.

2.6.2 Ambiente de Teste: *WEB-PerformCharts*

O desenvolvimento de *software* de forma distribuída é uma realidade cada vez mais comum para os profissionais, distantes entre si e espalhados por diversos pontos, poderem trabalhar juntos no desenvolvimento de um produto. Nesse sentido, a utilização da Internet é um recurso que possibilita o trabalho cooperativo entre esses profissionais geograficamente distantes. O ambiente *WEB-PerformCharts* (ARANTES et al., 2002) é acessível pela Internet, que adapta as rotinas da ferramenta *PerformCharts* (VIJAYKUMAR, 1999), possibilitando a geração e armazenamento de casos de teste, remotamente, via internet, pelos testadores de *software*. O funcionamento da ferramenta proposta baseia-se na especificação de sistemas reativos utilizando a técnica *Statecharts* e na geração de casos de teste para a mesma de acordo com alguns critérios disponíveis.

Assim como no ambiente GTSC, a *WEB-PerformCharts* interpreta, inicialmente, uma especificação em PcML, transformando essa especificação em uma cadeia de Markov ou uma MEF no formato XML. A partir da MEF, critérios de teste podem ser aplicados para gerar os casos de teste. Uma versão anterior do *WEB-PerformCharts* dependia de uma ferramenta chamada Condado (MARTINS et al., 1999) para gerar os casos de teste. Atualmente, uma outra versão do critério foi desenvolvido por Arantes et al. (2002), que foi incorporada ao ambiente *WEB-*

PerformCharts. A Figura 2.15 apresenta uma visão arquitetural do ambiente *WEB-PerformCharts*. Note que o ambiente apresenta duas versões para a geração de casos de teste, utilizando o critério *Switch Cover* e a ferramenta Condado. Também encontra-se incorporado ao ambiente o critério *Transition Tour* (TT) (NAITO; TSU-NOYAMA, 1981).

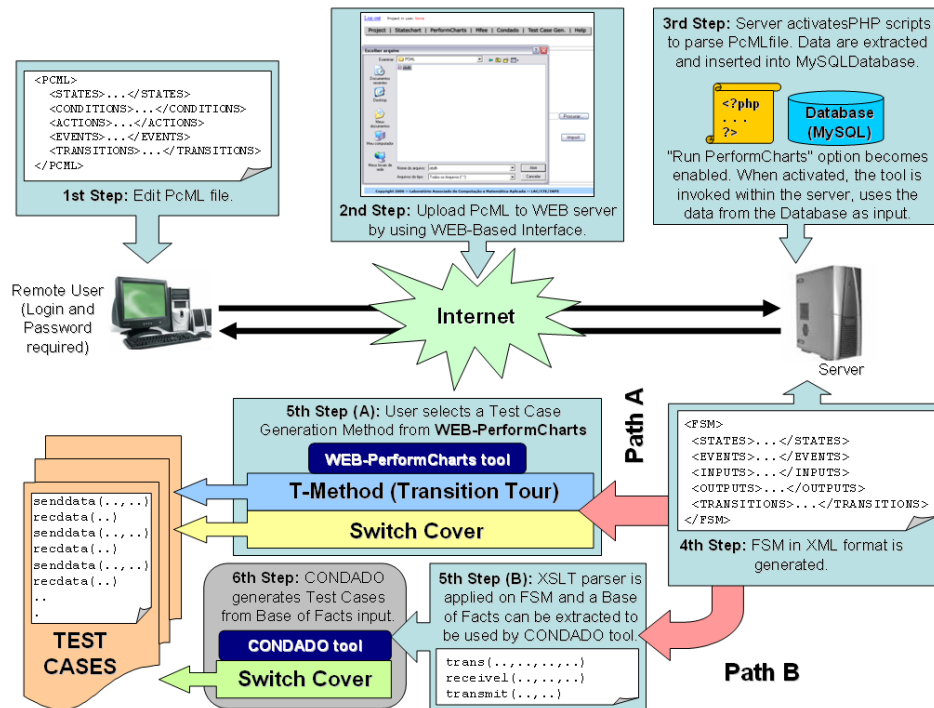


Figura 2.15 - Visão arquitetural ao ambiente *WEB-PerformCharts*.

Fonte: Arantes et al. (2002)

2.7 Trabalhos Relacionados

A geração de casos de teste para sistemas complexos, em especial aplicações críticas como sistemas embarcados em satélites, atualmente, representa um dos principais desafios a serem superados para garantir a qualidade do *software*. Conforme visto anteriormente, os TBM referem-se à geração de casos de teste a partir de um modelo que representa o comportamento do *software*. Diversos critérios (ou métodos) de geração de casos de teste tem sido propostos há décadas para gerar, de forma automática, casos de teste a partir de um modelo.

Em 1978, [Chow \(1978\)](#) apresentou o critério de teste W. Ele afirmava que o critério W era eficiente na detecção de diversas classes de erros como, erros de operação, transferência e estados ausentes ([CHOW, 1978](#)). O critério W verifica a estrutura de controle da [MEF](#). Para a sua aplicação, é necessário que a [MEF](#) possua as seguintes propriedades: minimal, determinística, Máquina de Mealy e completamente especificada. Ao longo dos anos, diversos outros critérios foram surgindo, muitos em melhoria ao critério W.

Em 1981, o critério [TT](#) foi proposto por [Naito e Tsunoyama \(1981\)](#). Esse critério percorre a máquina visitando cada estado e cada transição ao menos uma vez sem que ela precise ser reiniciada após a execução de cada teste. Porém o critério TT obtém somente uma cobertura das transições. Dessa forma, o critério não garante a detecção de defeitos de transferência.

Mais recentemente, em 2001, o critério *Round-Trip Path* foi proposto por [Binder \(2005\)](#) que também compreende uma adaptação do critério W. O critério *Round-Trip Path* utiliza uma estratégia que consiste em percorrer o grafo correspondente à [MEF](#) e gerar então uma árvore correspondente a esse percurso. Esse percurso é realizado, utilizando-se o algoritmo de busca em largura (*breadth-first search*) ([CORMEN et al., 2001](#)). Os casos de teste são derivados das sequências de transições da árvore gerada.

Devido à diversidade de critérios de testes existentes, decidir qual deles é o mais apropriado para ser utilizado não é uma tarefa fácil ([DELAMARO et al., 2007](#)). Segundo [Chow \(1978\)](#), um dos maiores problemas do teste é determinar um critério de seleção de casos de teste que seja válido e confiável. Dessa forma, aumentam cada vez mais os estudos relacionados a testes baseados em modelos, e a comunidade científica tem apresentado diversos trabalhos na análise do custo e da eficiência de critérios de teste para [MEF](#). Diversas métricas podem ser utilizadas como medida para analisar critérios de teste, dentre elas, têm-se o custo e a eficiência de um determinado conjunto de casos de teste. Em 1988, [Ntafos \(\)](#) apresentou alguns ensaios experimentais para avaliação da eficácia das estratégias de teste, utilizando análises de mutantes como medida.

Em 2002, [Antoniol et al. \(2002\)](#) mostraram um estudo de caso objetivando analisar o custo e a eficiência do critério *Round-trip Path* proposto por [Binder \(2005\)](#). [Antoniol et al. \(2002\)](#) utilizaram a técnica de análise de mutantes em seus experimentos e selecionaram um conjunto de operadores de mutação proposto para código orientado

a objeto. A conclusão principal de seu trabalho foi que o critério *Round-trip Path* é razoavelmente eficaz na detecção de falhas, com uma média de 87% de eficiência, mas que um número significativo de falhas inseridas no código só podem ser encontradas aumentando os casos de teste. Com o aumento do número de casos de teste gerados, aumenta o número de execuções a serem realizadas e, portanto, aumenta o custo do teste.

Já em 2004, [Briand et al. \(2004\)](#) mostraram uma simulação e um procedimento de análise de eficiência sob técnicas de testes baseados em *Statecharts*. Através desse procedimento, [Briand et al. \(2004\)](#) também analisaram o custo e a eficiência de conjuntos de casos de teste na detecção de falhas para critérios de cobertura para *Statecharts* em três estudos de caso.

É interessante citar o trabalho realizado por [Herculano e Delamaro \(2007\)](#), que teve grande influência para a realização deste trabalho. [Herculano e Delamaro \(2007\)](#) mostraram estudos comparativos que investigaram o relacionamento de técnicas de teste, analisando como podem ser usadas em conjunto a fim de compor uma estratégia de teste. Em seu trabalho é analisado o custo e a eficácia de critérios de teste estruturais de fluxo de dados e de fluxo de controle, a fim de analisar qual o relacionamento entre eles e como podem ser utilizadas em conjunto. As técnicas de especificação formal utilizadas foram MEF e Redes de Petri Colorida (RPC). Os critérios de geração de casos de teste analisados foram: critério W ([CHOW, 1978](#)), *State-Counting* ([PETRENKO; YEVTUSHENKO, 2005](#)) e UIO ([SIDHU; LEUNG, 1989](#)). Em seu trabalho, [Herculano e Delamaro \(2007\)](#) constatam que o critério W apresentou maior cobertura estrutural do que os outros critérios, porém, em termos de custo, o critério W também apresentou ter um custo bem maior. [Herculano e Delamaro \(2007\)](#) tratam o custo como sendo o número de casos de teste efetivamente utilizados e a eficiência como cobertura estrutural do código.

Ao contrário do trabalho de [Herculano e Delamaro \(2007\)](#), neste trabalho de pesquisa foram analisados o custo como sendo o tamanho dos conjuntos de casos de teste e da média de eventos por casos de teste e a eficiência está relacionada com a capacidade dos casos de testes em encontrarem um defeito no código e a técnica de teste utilizada é a técnica de teste funcional.

Mais recentemente, em 2008, [Ferreira et al. \(2008\)](#) apresentaram uma avaliação empírica, em termos de custo e eficiência, para o critério de teste para MEF, *Switch*

Cover, e dois critérios de testes da FCCS (SOUZA, 2000), *all-transitions* e *all-simple-paths*, em aplicações de sistemas na área espacial. Os resultados mostraram que os três critérios apresentam a mesma eficiência, no entanto, o critério *all-simple-paths* apresentou um melhor custo em relação aos outros critérios. Ferreira et al. (2008), consideraram a eficiência como sendo a capacidade dos casos de testes em encontrarem defeitos no código e custo sendo o tamanho dos conjuntos de casos de teste.

3 IMPLEMENTAÇÃO DOS CRITÉRIOS DE TESTE PARA MEF

Os Capítulos anteriores mostraram os conceitos relacionados com [MEF](#), *Statecharts*, ferramentas de apoio aos testes e critérios de teste. Este Capítulo descreve as primeiras contribuições associadas com a realização deste trabalho: concepção e implementação dos critérios de teste [DS](#), [UIO](#) e *Switch Cover* que deu origem ao um novo critério denominado *H-Switch Cover*, bem como a incorporação dos critérios às duas ferramentas de apoio ao teste desenvolvidas no [INPE](#): os ambientes [GTSC](#) e *WEB-PerformCharts*, já apresentadas no Capítulo anterior.

3.1 Leitura da Máquina de Estados Finitos

Para implementação dos critérios, foi necessário um estudo do comportamento das ferramentas de testes já existentes no [INPE](#). Como já mencionado, as ferramentas aqui apresentadas integram uma série de outras ferramentas se baseando em arquivos no formato [XML](#) para a geração de casos de teste. Dessa forma, a implementação dos critérios de teste deve levar em consideração esse tipo de formato.

O [XML](#) é uma linguagem de descrição de dados derivada do *Standard Generalized Markup Language* (SGML) e padronizada pela *World Wide Web Consortium* ([W3C](#)), capaz de descrever tipos de dados ([W3C, 1996](#)). A troca de informações entre diferentes ferramentas, possibilitando a comunicação e a interoperabilidade dos diversos artefatos gerados, é uma preocupação constante dos desenvolvedores, como é o caso do ambiente [GTSC](#) e *WEB-PerformCharts*, onde grande parte das entradas e saídas geradas pelas ferramentas é em formato [XML](#). Dessa forma, os critérios de teste desenvolvidos recebem como entrada uma [MEF](#) no formato de [XML](#).

A entrada para as ferramentas é a modelagem comportamental transformada para um arquivo em formato [PcML](#). Como mencionado anteriormente, essa modelagem é transformada, pelas ferramentas, em uma [MEF](#) plana no formato [XML](#). O arquivo [XML](#) que descreve uma [MEF](#), como pode ser observado na Figura 3.1, é formado por elementos (*tags*) que representam o início da [MEF](#) (<MFEE>), os estados (<STATES>), os eventos (<EVENTS>), as entradas (<INPUTS>), as saídas (<OUTPUTS>) e as transições (<TRANSITIONS>). Os atributos de estado descrevem, respectivamente: o nome do estado (*name*) e o tipo (*type*). O tipo se refere se o estado é inicial, final ou normal. Já os atributos de transição descrevem: a origem da transição (*source*) e o destino da transição (*destination*). O Apêndice A representa,

respectivamente, um exemplo do PcML e da MEF completa, no formato XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="mfeeTesteX.xsl"?>
<MFEE>
  <STATES>
    <STATE NAME="S1" TYPE="inicial"/>
    <STATE NAME="S2" TYPE="normal"/>
  </STATES>

  <EVENTS>
    <EVENT VALUE="1" NAME="A"/>
    <EVENT VALUE="1" NAME="B"/>
    ...
  </EVENTS>

  <INPUTS>
    ...
  </INPUTS>
  <OUTPUTS>
    ...
  </OUTPUTS>

  <TRANSITIONS>
    <TRANSITION SOURCE="S0" DESTINATION="S0">
      <INPUT INTERFACE="L">B</INPUT>
      <OUTPUT>Y</OUTPUT>
    </TRANSITION>

    <TRANSITION SOURCE="S0" DESTINATION="S3">
      <INPUT INTERFACE="L">A</INPUT>
      <OUTPUT>0</OUTPUT>
    </TRANSITION>
    ...
  </TRANSITIONS>
</MFEE>
```

Figura 3.1 - Representação em XML de uma MEF.

Todos os critérios devem ler e interpretar um XML que representa a MEF. Esta etapa de leitura e interpretação do arquivo XML é idêntica para qualquer critério; por esse motivo foi elaborada um *workflow* comum que representa esta etapa. O *workflow* para leitura do XML é representado na Figura 3.2.



Figura 3.2 - *Workflow* para interpretar uma MEF em XML e gerar casos de teste.

Os critérios foram implementados usando conceitos de Orientação a Objetos (OO). O processo de leitura do arquivo XML é formado por três pacotes principais: *principal*, *parser* e *maquina*. O primeiro pacote, como o próprio nome já diz, é o pacote *principal*. Nele se encontram as classes responsáveis pelas chamadas dos métodos que

irão ler os arquivos XML. O segundo pacote, *parser*, representa as classes que fazem a leitura do arquivo XML. Para isso é utilizada a *Application Programming Interface* (API) do *Simple API for XML* (SAX), da linguagem Java (DEITEL; DEITEL, 2005). O terceiro pacote, *maquina*, é formado por classes responsáveis em montar um objeto que represente todas as características da MEF em Java. Além dos pacotes mencionados, outros adicionais são utilizados, visando atender aos requisitos estabelecidos para construção dos critérios de teste. A Figura 3.3 representa uma visão geral dos principais pacotes e as dependências entre eles. Note que cada critério de teste a ser implementado faz uso do objeto de saída dos pacotes de leitura do XML, ou seja, os critérios têm como entrada um objeto, que representa a MEF, gerado pela leitura do XML.

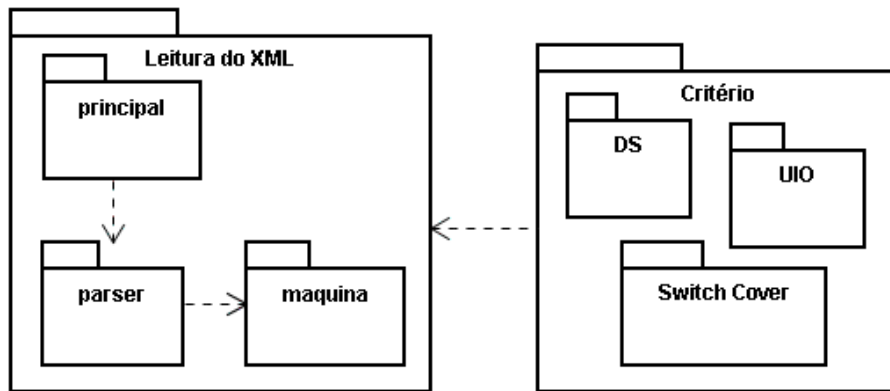


Figura 3.3 - Visão geral dos principais pacotes da leitura da MEF em XML e dos critérios de teste.

3.2 Implementação do Critério DS

O critério DS, conforme foi levantado no Capítulo 2, possui duas etapas a serem seguidas em sua implementação: *i*) encontrar a menor SD, se essa existir; e *ii*) encontrar a β -sequence.

Para encontrar uma SD, referente à primeira etapa do algoritmo, é necessário criar uma árvore de distinção a partir de uma MEF. Um pseudo-código da geração da árvore de distinção é representado na Figura 3.4.

```

input :  $m$ 
output: arvore

1 begin {
2 for  $i \leftarrow 0$  to  $m$  do
3    $grupoIncerteza \leftarrow estado$ ;
4    $noGrupo \leftarrow grupoIncerteza$ ;
5  $listaGruposCriados \leftarrow noGrupo$ ;
6  $distinguirGrupos(listaGruposCriados)$  ;
7 return arvore;
8 };
9  $distinguirGrupos (listaGruposCriados)\{$ 
10 for  $j \leftarrow noGrupo$  to  $listaGruposCriados$  do
11   for  $y \leftarrow entrada$  to  $listaValoresEntrada$  do
12      $listaDistinguidos \leftarrow distinguirGruposIncerteza(gruposIncerteza)$ ;
13      $criarEstadosDestinos(grupoIncerteza)$ ;
14      $novoNoGrupo \leftarrow listaDistinguidos$ ;
15      $arvore \leftarrow novoNoGrupo$ ;
16      $listaNovoNoGrupo \leftarrow novoNoGrupo$ ;
17  $novaListaDistinguir \leftarrow validarNoGrupo(listaNovoNoGrupo)$  ;
18 if  $novaListaDistinguir > 0$  then
19    $distinguirGrupos(novaListaDistinguir)$ ;
20 };

```

Figura 3.4 - Pseudo-código para criar a Árvore de Distinção

Em um primeiro momento do pseudo-código acima é criado o grupo de incerteza que recebe todos os estados da Máquina. Logo depois é chamado o método *distinguirGrupos()* que é responsável em criar, adicionar na árvore e distinguir cada novo grupo formado, isto de forma recursiva até que todos os grupos sejam triviais, conforme destacado na Seção 2.3.1.

As Tabelas 3.1 e 3.2, apresentam respectivamente as principais variáveis e métodos do código.

Tabela 3.1 - Variáveis utilizadas no código da Árvore de Distinção.

Variável	Descrição
m	Máquina de estados, onde $m = (X, Z, S, s_0, f_z, f_s)$. X é um conjunto finito não-vazio de símbolos de entrada; Z é um conjunto finito de símbolos de saída; S é um conjunto finito não-vazio de estados; $s_0 \in S$ é o estado inicial; $f_z: (S \times X) \rightarrow Z$ é a função de saída; e $f_s: (S \times X) \rightarrow S$ é a função de próximo estado.
gruposIncerteza	Lista de estados que não foram distinguidos.
noGrupo	Lista de grupos de incerteza.
arvore	Árvore de distinção montada com noGrupo e as respectivas transições.
e	Estado pertencente a MEF .
listaValoresEntrada	Todas as entradas existentes na MEF .

Tabela 3.2 - Principais métodos utilizados no código da Árvore de Distinção.

Método	Descrição
iniciar()	Responsável por inicializar a criação da árvore de distinção.
distinguirGrupos (listaGruposCriados)	Responsável por percorrer a lista de grupos que ainda não foram distintos e chamar outros métodos relacionados para fazer formando a nova lista de grupos não distintos.
distinguirGruposIncerteza (gruposIncerteza)	Um noGrupo é formado por gruposIncerteas e são exatamente estes que serão distinguidos pelo método distinguirGruposIncerteza(gruposIncerteza).
criarEstadosDestinos (grupoIncerteza)	Depois de ser distinguido um grupo, deverá criar os grupos de estados destinos, pois são esses a serem usados para gerar os novos noGrupos da árvore de distinção.
validarNoGrupo (listaNovoNoGrupo)	Responsável por verificar se o noGrupo é trivial ou inconsistente.

Para implementar a segunda etapa do critério é necessário encontrar as *subsequence* da β -sequence da MEF. Para encontrar as *subsequences*, devem ser seguidos os seguintes passos: *i*) sempre começar o algoritmo partindo do estado inicial da MEF; *ii*) para cada transição da MEF encontrar a menor sequência que leva a MEF do estado inicial até o estado s_j de uma transição (s_i, s_j) ; e *iii*) aplicar a menor SD encontrada através da Árvore de Distinção em cada *subsequence*. Dessa forma, para todas as *subsequence* geradas, deve-se fazer: $DS = subsequence + SD$.

O pseudo-código implementado para encontrar a β -sequence é apresentado na Figura 3.5, e as Tabelas 3.3 e 3.4 referem-se às principais variáveis e métodos do pseudo-código para encontrar a β -sequence.

```

input :  $m$ 
output:  $\beta$ -sequences
1  $listaCaminhos \leftarrow null$ ;
2 begin {
3  $estadoInicial \leftarrow retornarEstadoIncial()$ ;
4  $listaFilhos \leftarrow criarListaFilhos(estadoInicial)$ ;
5  $percorrer(listaFilhos, listaCaminhos)$  ;
6 return  $listaCaminhos$ ;
7 };
8  $percorrer (listaFilhos, listaCaminhos)\{$ 
9 for  $j \leftarrow noBean$  to  $listaFilhos$  do
10 |  $listaCaminhos \leftarrow gerarCaminhos(listaFilhos)$ ;
11 | if  $noVisitado(noBean)$  then
12 | |  $novaListaFilhos \leftarrow criarListaFilhos(noBean)$ ;
13 | |  $listaAuxiliar = returnAll(novaListaFilhos)$ ;
14 | | else  $listaAuxiliar.remove(noBean)$ ;
15 if  $listaAuxiliar > 0$  then
16 | |  $percorrer(listaAuxiliar, listaCaminhos)$ ;
17 | };

```

Figura 3.5 - Pseudo-código para criar a β -sequence

Tabela 3.3 - Variáveis utilizadas no algoritmo da β -sequence.

Variável	Descrição
listaCaminhos	Caminho do estado s_i até o estado inicial.
estadoInicial	Estado inicial da MEF.
listaFilhos	Lista de estados considerados estados destinos (filhos) de um determinado estado s_i .
noBean	Estado corrente.
novaListaFilhos	Nova lista de estados considerados estados destinos (filhos) de um determinado estado s_i .
listaAuxiliar	Lista de controle para verificar quando todas as UES já foram encontradas.

Tabela 3.4 - Principais métodos utilizados no algoritmo da β -sequence.

Método	Descrição
iniciar()	Responsável por inicializar o código, definindo o valor das primeiras variáveis a serem utilizadas.
retornarEstadoInicial()	Retorna estado inicial da MEF.
criarListaFilhos(estadoInicial)	Retorna a lista de estados destinos (<i>filho</i>) do estado corrente.
percorrer(listaFilhos)	Método responsável por começar a percorrer a lista de estados destinos (<i>filhos</i>) para poder gerar os caminhos UES.
gerarCaminhos(listaFilhos)	Percorre a máquina capturando os estados origem (<i>pai</i>) dos estados correntes de forma recursiva até alcançar o estado inicial.
noVisitado(noBean)	Verifica se o estado corrente já foi visitado.

O fluxograma na Figura 3.6, apresenta uma visão geral de como ocorre o fluxo de execuções de todo o algoritmo implementado para o critério DS. Note, que podem existir MEF que não possuam uma SD.

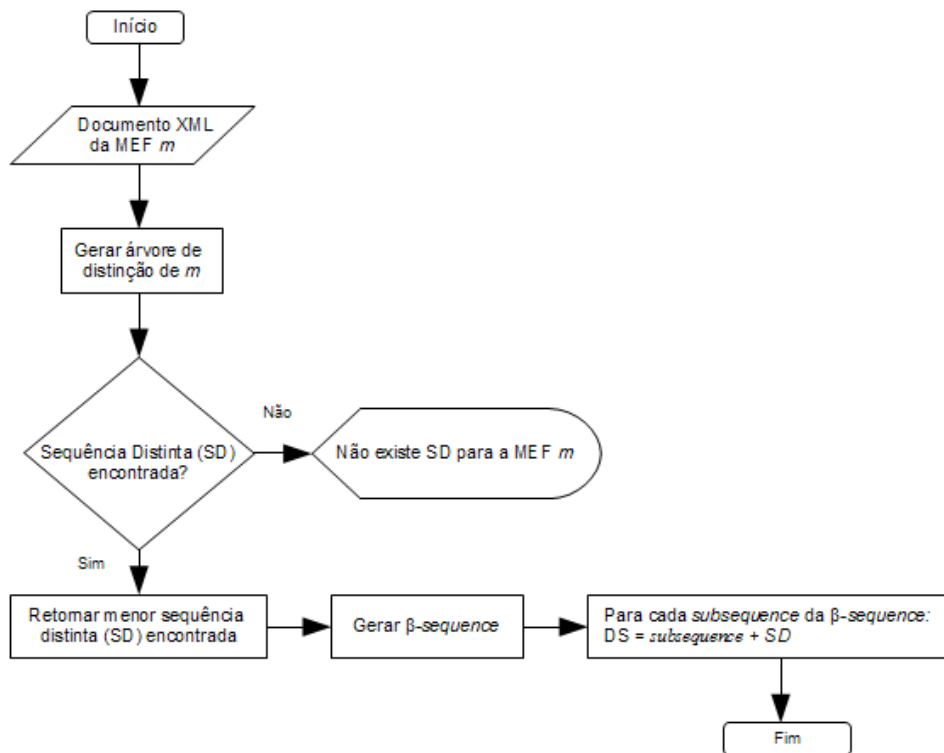


Figura 3.6 - Fluxograma do algoritmo do critério de teste DS.

3.3 Implementação do Critério UIO

O critério UIO utiliza as sequência UES para a geração de um conjunto de sequência de entrada para testar a MEF. Assim como o critério DS, o critério UIO também passa por duas etapas. Primeiro, verifica-se cada estado da MEF para encontrar sua respectiva sequência UES. Na segunda etapa deve-se encontrar a β -sequence. O

pseudo-código para encontrar a sequência UES é apresentada na Figura 3.7.

```

input :  $m$ 
output:  $listaUES$ 
1  $listaFilhos \leftarrow null$ ;
2  $entradas \leftarrow null$ ;
3 begin {
4 for  $i \leftarrow 0$  to  $m$  do
5    $isUio \leftarrow false$ ;
6    $estadoAnalisado \leftarrow i$ ;
7    $entradas \leftarrow gerarEntradas(m)$ ;
8    $percorrer(listaFilhos, listaCaminhos)$  ;
9 return  $listaUES$ ;
10 };
11  $percorrer(listaFilhos, estadoAnalisado, entradas)\{$ 
12 for  $j \leftarrow noBean$  to  $listaFilhos$  do
13    $entradas \leftarrow gerarEntradas(listaFilhos)$ ;
14    $isUio \leftarrow avaliarUIO(entradas, estadoAvaliado)$ ;
15 if  $isUio == false$  then
16    $listaFilhos \leftarrow criarListaFilhos(noBean)$ ;
17    $novasEntradas \leftarrow gerarEntradas(listaFilhos)$ ;
18    $percorrer(listaAuxiliar, estadoAnalisado, novasEntradas)$ ;
19 else  $isUio \leftarrow true$ ;
20  $listaUES \leftarrow estadoAnalisado$ ;
21 };

```

Figura 3.7 - Pseudo-código para encontrar a UES de cada estado da MEF

Conforme mostrado no Capítulo 2, a segunda etapa do critério UIO é similar ao critério DS, pois também é necessário encontrar a β -sequence da MEF. O algoritmo sofre uma pequena modificação no momento de fazer a junção das UES com a β -sequence, que leva em conta a qual estado a UES pertence. Para todas as subsequence, deve-se fazer: $UIO = subsequence + UIO$. As Tabelas 3.5 e 3.6 apresentam as principais variáveis e métodos do um pseudo-código para encontrar a UES de um determinado estado.

Tabela 3.5 - Variáveis utilizadas no código da UES.

Variável	Descrição
listaFilhos	Lista todos os estados considerados estados destinos (<i>filhos</i>) do estado em questão.
entradas	Refere-se aos eventos de entrada de uma transição.
isUIO	Variável booleana que representa se foi encontrado uma UES para o estado analisado.
estadoAnalisado	Refere-se ao estado que está sendo analisado no momento.
novasEntradas	Refere-se aos novos eventos de entrada de uma transição.

Tabela 3.6 - Principais métodos utilizados no código UES

Método	Descrição
gerarEntradas(m)	Retorna as entradas que serão analisadas para verificar se são UES para o estado corrente.
percorrer(listaFilhos, estadoAnalisado, entradas)	Método recursivo responsável por percorrer a MEF até encontrar a UES para o estado corrente.
avaliarUES(entradas, estadoAvaliado)	Método responsável por verificar se a <i>entrada</i> é uma UES para o estado corrente.
criarListaFilhos (no-Bean)	Retorna uma lista de estados destinos (<i>filhos</i>) do estado corrente.

O fluxograma na Figura 3.8, apresenta uma visão geral de como ocorre o fluxo de execuções de todo algoritmo implementado para o critério UIO.

3.4 Implementação do Critério *H-Switch Cover*

O critério *Switch Cover* já possui algumas versões apresentadas na literatura. No INPE, uma das versões já utilizadas refere-se à ferramenta Condado (MARTINS et al., 1999). No entanto, essa versão da ferramenta não consegue lidar com alguns tipos de MEF, principalmente aquelas que apresentam maior complexidade com um grande número de estados e transições. Para contornar isto, uma outra versão

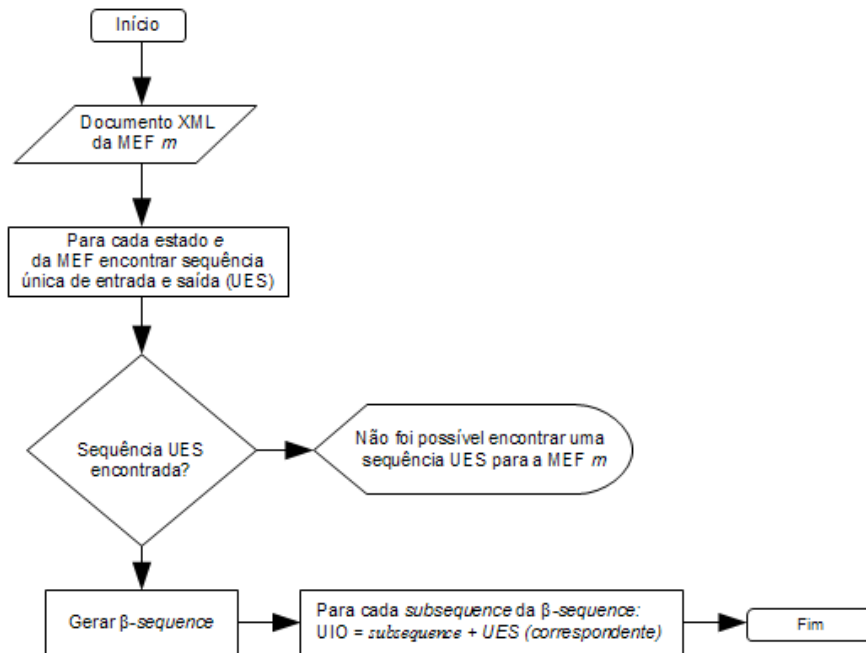


Figura 3.8 - Fluxograma do algoritmo do critério de teste **UIO**.

foi desenvolvida por [Arantes et al. \(2002\)](#), que foi incorporada ao ambiente *WEB-PerformCharts*. Entretanto, percebeu-se que o critério pode ainda ser melhorado no que diz respeito ao seu desempenho. Neste sentido, foi desenvolvida outra versão do *switch cover*, chamada de *H-Switch Cover*, que foi incorporada a um outro ambiente também desenvolvido no **INPE - GTSC**. A nova versão está sendo também incorporada no ambiente *WEB-PerformCharts*.

O critério *Switch Cover* é conhecido por gerar sequências de teste muito extensas, tornando-se, em muitos casos, inviável a sua aplicação. Dessa forma, o objetivo inicial deste trabalho de pesquisa constituía o desenvolvimento do critério de teste *Switch Cover*, que abrangesse **MEF** mais complexas. Durante o desenvolvimento do critério, foram criadas, ao longo da pesquisa, diversas versões da implementação do critério, incorporando melhorias no código. Tais melhorias estão relacionadas com a quantidade de sequências de teste e a quantidade de eventos dos casos de teste que o critério deve gerar, que deu origem ao novo critério: o *H-Switch Cover*.

Esta seção apresenta a atual versão do critério *H-Switch Cover* incorporada ao ambiente **GTSC**, no **INPE**. Na Seção 2.3.3, foram apresentadas, brevemente, algumas características e passos da implementação do critério *Switch Cover*. Conforme foi

apresentado, o critério possui diversas etapas a serem seguidas, sendo elas: *i*) Criação de um Grafo Dual a partir do original; *ii*) Criação das novas transições no Grafo Dual; *iii*) Balanceamento do Grafo Dual; e *iv*) Geração dos casos de teste.

A primeira e a segunda etapa foram implementadas em um único algoritmo. Ao mesmo tempo em que as transições eram transformadas em estados, as novas transições para o novo grafo a ser formado eram criadas também. Para estas duas etapas foi gerado o algoritmo apresentado na Figura 3.9.

```

input  : m
output: grafoDual
1 begin {
2 for i ← 0 to m do
3   | listaTransicoes ← i.getTransicoes();
4 for j ← 0 to listaTransicoes do
5   | listaEstados ← criaListaEstados(j);
6   | grafoDual ← listaEstados;
7 for y ← 0 to m do
8   | listaTranEstado ← y.getTransicoes();
9   | for k ← 0 to listaTranEstado do
10  |   | estadoDestino ← retornaEstadoDestino(tranEC);
11  |   | for tranED ← 0 to listaTranEDestino do
12  |   |   | adicionaNovasTransicoes(i);
13 return grafoDual;
14 };
```

Figura 3.9 - Pseudo-código para transformar as transições da MEF em estados e adicionar as novas transições.

As Tabelas 3.7 e 3.8, apresentam as principais variáveis e métodos do pseudo-código para converter as transições em estados.

Tabela 3.7 - Principais variáveis utilizados no código da conversão das transições em estados

Variáveis	Descrição
listaTransicoes	Lista de transições.
listaEstados	Lista de estados.
grafoDual	Representa a nova MEF (Grafo Dual).
listaTranEstado	Representa lista de transições que partem do estado.
estadoDestino	Representa estado destino.
tranED	Transições do estado destino.

Tabela 3.8 - Principais métodos utilizados no código da conversão das transições em estados

Métodos	Descrição
getTransicoes()	Retorna todas as transições da MEF.
criaListaEstados (lista-Transicoes)	Percorre as transições transformando-as em estados e retorna uma lista com os estados.
retornaEstadoDestino (tranEC)	Retorna estado destino.
adicionaNovasTransicoes (novoEstado)	Adiciona as novas transições no Grafo Dual.

A terceira etapa pode ser considerada a mais importante e o ponto chave para melhoria do tamanho dos casos de testes gerados pelo critério, que é o balanceamento do grafo Dual, onde novas regras foram criadas para decidir que posição adicionar às novas transições de balanceamento. Para balancear a MEF, deve-se criar novas transições em estados não balanceados, e isso deve ser realizado até que todos os estados da MEF estejam balanceados, ou seja, a quantidade de transições de entrada deve ser a mesma da quantidade de transições de saída no estado. Novas transições só podem ser adicionadas onde já existem transições que ligam os estados que receberão a nova transição, ou seja, se o estado s_i estiver desbalanceado e for criada uma nova transição para balanceá-lo do estado s_i a s_j , essa transição só poderá ser criada se já existir uma transição ligando s_i a s_j .

Problemas com o balanceamento das MEF costumam aparecer nesta etapa como, por exemplo, nunca conseguir balancear a MEF pelo fato de que toda vez que se balanceia um estado, acaba-se desbalanceando outro, ou por criar centenas de transições até conseguir balancear, fazendo com que os casos de teste gerados posteriormente sejam enormes. Por isso, neste trabalho, foram elaboradas algumas regras que definem a posição mais relevante para poder adicionar a nova transição. Se um estado for considerado desbalanceado, as seguintes regras são aplicadas:

- Encontrar o melhor estado de origem - se o número de transições de entrada for menor que o número de transições de saída, verifica-se quem são os estados de origem do estado para criar uma nova transição para eles. Se já estiverem balanceados, podem comprometer o balanceamento do grafo. Então, num primeiro momento, é considerado que nenhuma transição será adicionada em algum estado de origem. Se depois de percorrer toda a máquina, for constatado que deverá ser desbalanceado um estado de origem para balancear o estado corrente, então é analisado qual é o melhor estado de origem a ser desbalanceado;
- Encontrar o melhor estado destino - se o número de transições de entrada for maior que o número de transições de saída, verifica-se quem são os estados destino do estado para criar uma nova transição para eles. Se os estados destinos já estiverem balanceados, podem comprometer o balanceamento do grafo. Então, num primeiro momento, é considerado que nenhuma transição será adicionada em algum estado destino. Se depois de percorrer toda a máquina, for constatado que deverá ser desbalanceado um estado destino para balancear o estado corrente, então é analisado qual é o melhor estado destino a ser desbalanceado;
- Contador de balanceamento - se existir estados que já foram balanceados mais do que outros, será dado prioridade para o estado que foi menos balanceado para atribuir uma transição. Para esse controle, cada estado tem um contador que mostra quantas vezes ele foi balanceado e desbalanceado; e
- Contador de transições - cada estado também contém um contador com o número de transições que possui. Caso seja necessário balancear um estado, independente de ter que ser criada uma nova transição para um estado

destino ou para um estado origem, será dado prioridade em escolher um estado que esteja desbalanceado em relação ao estado origem e que tenha um menor número de transições.

O pseudo-código 3.10 mostra o mecanismo de balanceamento da MEF para o critério *H-Switch Cover*. Como pode ser observado no pseudo-código trabalha de forma recursiva, enquanto existir estados na máquina ainda não estejam balanceados, o tratamento para balancear a máquina é chamado novamente adicionando novas arestas conforme as regras implementadas. O algoritmo implementado só termina quando a máquina estiver balanceada, podendo passar para a próxima etapa do algoritmo.

Os pseudo-códigos 3.11 e 3.12, representam os métodos *retornarMelhorPai()* e *retornarMelhorFilho()* respectivamente, que fazem parte do algoritmo de balanceamento. Conforme mostrado anteriormente, tais métodos fazem parte das regras criadas para definir a posição mais relevante para poder adicionar a nova transição na máquina balanceada e assim melhorar o desempenho do critério *H-Switch Cover*.

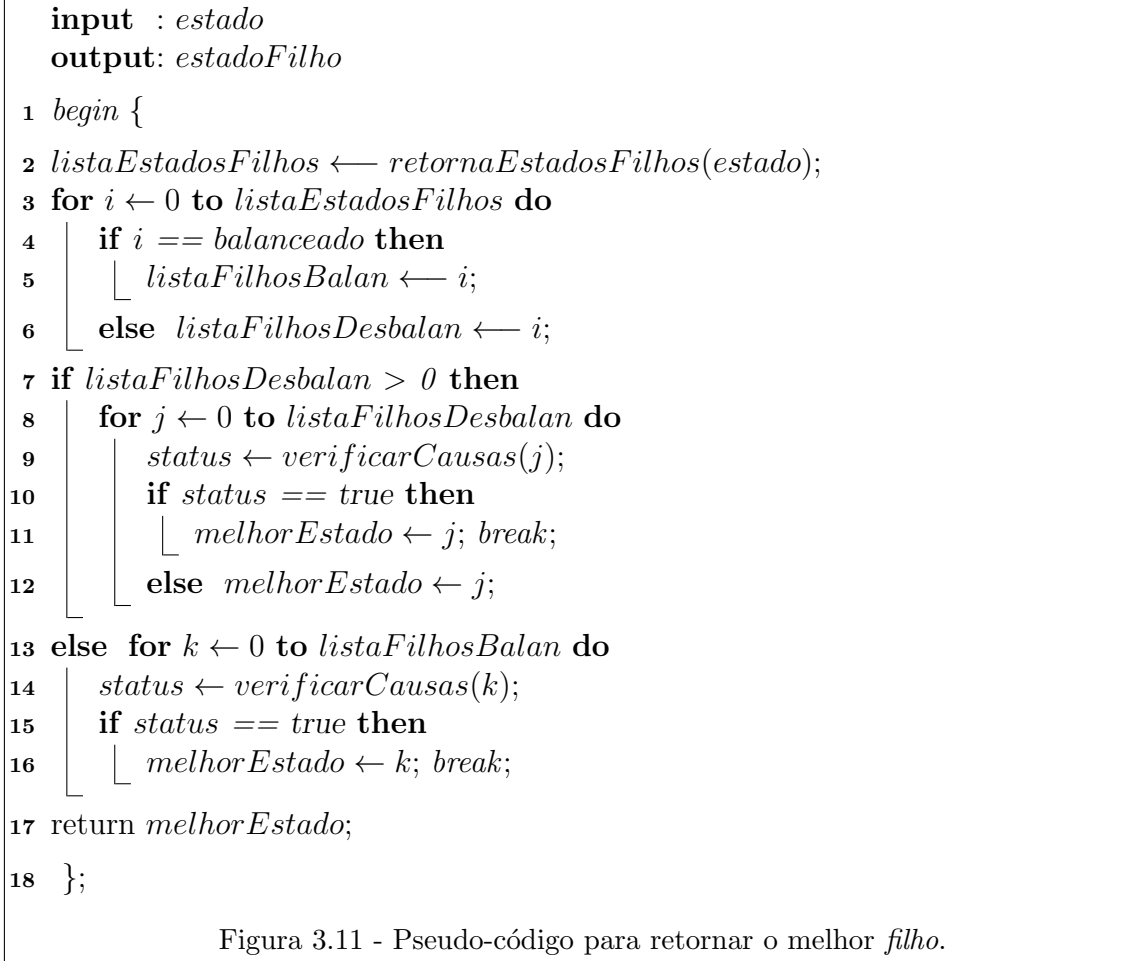
```

input : grafoDual
output: grafoBalanceado

1 begin {
2 while balanceamento(grafoDual) == false do
3   for  $i \leftarrow 0$  to grafoDual do
4     balanceiaEstado(estado){
5       balanceiaEstado(estado){
6         if numTransicaoEntrada < numTransicaoSaida then
7           estadoPai  $\leftarrow$  retornarMelhorPai();
8           adicionarAresta(estadoPai, estado);
9         else estadoFilho  $\leftarrow$  retornarMelhorFilho();
10        adicionarAresta(estadoFilho, estado);
11      return grafoBalanceado;
12    };
13  };
14 };

```

Figura 3.10 - Pseudo-código para balanceamento do Grafo Dual.



É apresentado na Tabelas 3.9 e 3.10 as principais variáveis e métodos do pseudo-código do balanceamento do Grafo Dual.

Tabela 3.9 - Principais variáveis utilizados no código de balanceamento

Variáveis	Descrição
<i>numTransicaoEntrada</i>	Representa o número de transições de entrada de um estado s_i .
<i>numTransicaoSaida</i>	Representa o número de transições de saída de um estado s_i .
<i>estado</i>	Representa o estado corrente.
<i>estadoPai</i>	Representa o estado origem.
<i>estadoFilho</i>	Representa o estado destino.

continua na próxima página

Tabela 3.9 - Principais variáveis utilizados no código de balanceamento

Variáveis	Descrição
<i>listaEstadosFilhos</i>	Lista com estados destino de um determinado estado s_i .
<i>status</i>	Variável booleana que mostra se a MEF está balanceada.
<i>melhorEstado</i>	Representa melhor estado para se adicionar uma transição.
<i>estadoDes</i>	Representa um estado desbalanceado.
<i>estadoBalan</i>	Representa um estado balanceado.

Tabela 3.10 - Principais métodos utilizados no código de balanceamento

Métodos	Descrição
<code>balanceiaEstado(<i>estado</i>)</code>	Método responsável por fazer o balanceamento de um estado s_i , se este estiver desbalanceado.
<code>adicionarAresta(<i>estadoPai</i>, <i>estado</i>)</code>	Adiciona uma nova aresta partindo do estado de origem (<i>pai</i>).
<code>adicionarAresta(<i>estadoFilho</i>, <i>estado</i>)</code>	Adiciona uma nova aresta indo para o estado de destino (<i>filho</i>).
<code>verificarCausas(<i>estadoDes</i>)</code>	Verifica os contadores de transição e de balanceamento para verificar as causas de um desbalanceamento de um estado para poder balancear outro.
<code>retornaEstadosPai(<i>estado</i>)</code>	Retorna estado origem de um determinado estado s_i .
<code>retornaEstadosFilhos(<i>estado</i>)</code>	Retorna estado destino de um determinado estado s_i .
<code>balanceamento(<i>grafoDual</i>)</code>	Método responsável por verificar se o grafo dual está desbalanceado.

O último passo refere-se à geração das sequências de teste a partir do grafo balanceado. Assim como na terceira etapa, a quarta etapa deste critério também sofreu

```

input : estado
output: estadoFilho

1 begin {
2 listaEstadosPai  $\leftarrow$  retornaEstadosPai(estado);
3 for i  $\leftarrow$  0 to listaEstadosPai do
4   if i == balanceado then
5      $\lfloor$  listaPaiBalan  $\leftarrow$  i;
6   else listaPaiDesbalan  $\leftarrow$  i;
7 if listaPaiDesbalan > 0 then
8   for j  $\leftarrow$  0 to listaPaiDesbalan do
9     status  $\leftarrow$  verificarCausas(j);
10    if status == true then
11       $\lfloor$  melhorEstado  $\leftarrow$  j; break;
12    else melhorEstado  $\leftarrow$  j;
13 else for k  $\leftarrow$  0 to listaPaiBalan do
14   status  $\leftarrow$  verificarCausas(k);
15   if status == true then
16      $\lfloor$  melhorEstado  $\leftarrow$  k; break;
17 return melhorEstado;
18 };
```

Figura 3.12 - Pseudo-código para retornar o melhor *pai*.

modificações com o objetivo de melhorar a qualidade dos casos de teste. Em outras versões do critério *Switch Cover* implementadas no GTSC, a última etapa obedecia a seguinte regra: o critério parte do estado inicial, percorre a MEF até retornar ao estado inicial. Enquanto houver transições de saída no estado inicial, novas sequências devem ser geradas, de forma que não passem pelas transições já visitadas. Porém, em algumas MEF foi percebido que o método voltava para o estado inicial sem passar por todos os estados, pois ele encontrava o estado inicial antes de passar por todos os estados. Outra situação notada foi que algumas das transições eram simplesmente puladas e os casos de teste gerados tinham uma sequência de execuções que não respeitavam o modelo, não sendo possível exercitá-las. O modelo da Figura 3.13 representa um dos problemas que ocorriam com outras versões do critério.

Foram gerados os casos de teste com uma das antigas versões do critério *Switch Cover* implementada no GTSC. Os casos de teste obtidos foram:

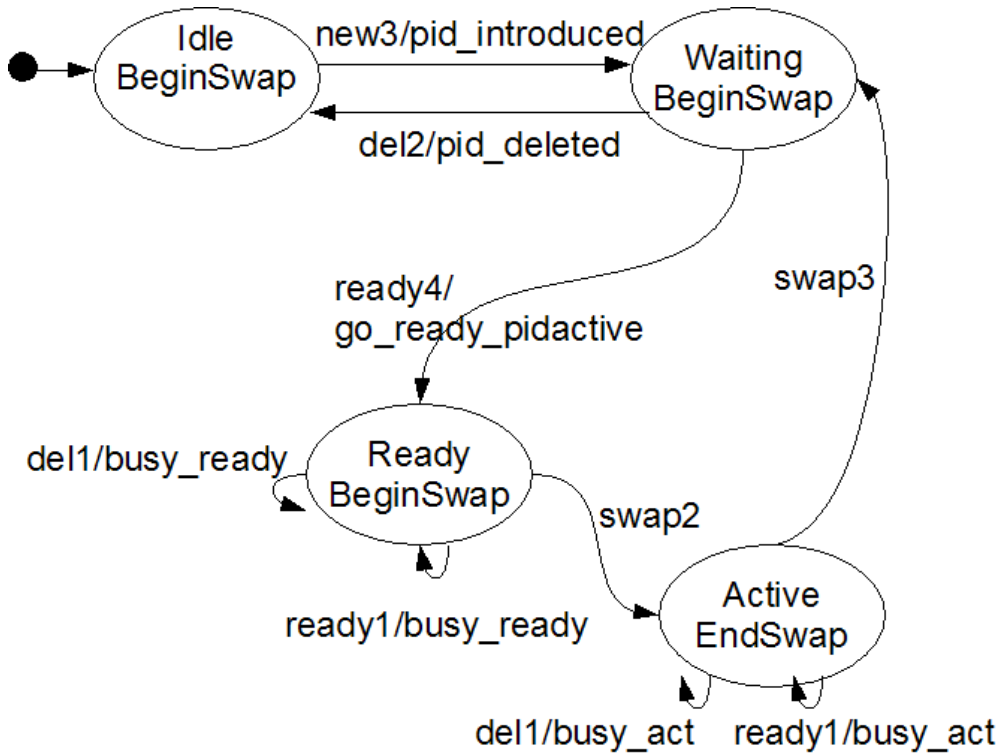


Figura 3.13 - Exemplo de uma MEF para o critério *Switch Cover*.

- 1- *new3/del2*
- 2- *new3/ready4/del1/del1/ready1/del1/swap3/del2*

Note que o evento *swap2* não aparece em nenhum momento e não seria possível exercitar o segundo caso de teste, pois para passar pelo evento *swap3* é necessário passar pelo evento *swap2* antes. Considerando a versão do *Switch Cover* implementada na *WEB-PerformCharts* (ARANTES et al., 2002), percebeu-se que o evento *del1/busy_ready* não foi visitado pelos casos de teste. Neste trabalho, para resolver o problema das implementações anteriores referente a esta etapa, foi utilizado o algoritmo de *Hierholzer* (LIPSCHUTZ; LIPSON, 1997), que deu origem ao nome do novo critério de teste elaborado: o *H-Switch Cover*.

O algoritmo de *Hierholzer* é um algoritmo para a construção de um ciclo euleriano sugerido a partir da prova do teorema de Euler (LIPSCHUTZ; LIPSON, 1997). Segundo Lipschutz e Lipson (1997), o teorema de Euler diz que um grafo conexo finito é euleriano se e somente se cada vértice tem um grau par. Por isso, da realização da terceira etapa desse critério, que é o balanceamento da MEF. Considerando que a

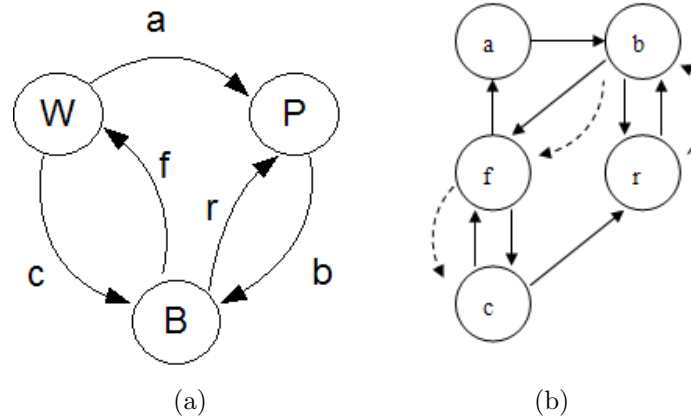


Figura 3.14 - MEF inicial e MEF balanceada.

MEF esteja balanceada, é possível aplicar o algoritmo de *Hierholzer* e gerar um caminho euleriano. O algoritmo é aplicado de acordo com a quantidade de estados iniciais que a MEF possui. Quando as transições da MEF original são transformadas em estados, as transições que partem do estado inicial, passam a ser estados iniciais no grafo Dual. Se duas transições no grafo original saem do estado inicial, o grafo dual passará a ter dois estados iniciais e, conseqüentemente, duas sequências de teste, ou dois casos de teste. Os principais passos para aplicação do algoritmo são:

- A partir do estado inicial, percorrer a MEF até voltar para o estado inicial;
- Comece em qualquer transição t do estado inicial e percorra aleatoriamente as arestas ainda não visitadas até fechar um ciclo;
- Se sobrirem transições não visitadas, deve-se recomençar a partir de uma transição do ciclo já formado; e
- Se não existirem mais transições não visitadas, deve ser construído o ciclo euleriano a partir dos ciclos formados, unindo-os a partir de uma transição comum.

Utilizando o mesmo exemplo mostrado na Seção 2.3.3, (Figuras 3.14(a) e 3.14(b)), é apresentado logo abaixo os casos de teste gerados com a utilização do algoritmo de *Hierholzer*.

Usando o estado a como sendo inicial têm-se:

- **Ciclo 1** - a b r b f a
- **Ciclo 2** - b f c f c r b
- **Ciclo Euleriano** - a b f c f c r b r b f a

O algoritmo desenvolvido para percorrer a MEF balanceada e gerar os casos de teste, é apresentado na Figura 3.15.

```

input : grafoBalanceado
output: casoTeste
1 begin {
2 listaEstadosIniciais  $\leftarrow$  retornaEstadosIniciais(grafoBalanceado);
3 for i  $\leftarrow$  0 to listaEstadosIniciais do
4   | percorrer(i);
5   | percorrer(i) {;
6   | listaTransicoes  $\leftarrow$  retornaListaTrans(i);
7   | for j  $\leftarrow$  0 to listaTransicoes do
8   |   | estadoDestino  $\leftarrow$  retornaEstadoDestino(j);
9   |   | if estadoDestino  $\neq$  estadoInicial then
10  |   |   | transicao  $\leftarrow$  retornaMelhorTransicao(estadoDestino);
11  |   |   | if transicao.getVisitada  $==$  false then
12  |   |   |   | caminhoPercorrido  $\leftarrow$  caminhoPercorrido + transicao;
13  |   |   |   | percorrer(estadoDestino);
14  |   |   | else listaCiclos  $\leftarrow$  caminhoPercorrido;
15  |   |   | if transicoesNaoVisitadas  $==$  true then
16  |   |   |   | novoEstadoIncial  $\leftarrow$ 
17  |   |   |   |   | retornaNovoEstadoIncial(transicoesNaoVisitadas);
18  |   |   |   |   | percorrer(novoEstadoIncial);
19 casoTeste  $\leftarrow$  montaCaminhoEuleriano(listaCiclos);
20 return casoTeste;
21 };
```

Figura 3.15 - Pseudo-código para geração dos casos de teste.

As Tabelas 3.11 e 3.12, mostram respectivamente as principais variáveis e métodos utilizados.

Tabela 3.11 - Principais variáveis utilizados no código de geração dos casos de teste.

Variáveis	Descrição
listaEstadosIniciais	Representa a lista de estados iniciais.
listaTransicoes	Lista de transições de um determinado estado.
transicao	Representa uma determinada transição de um estado.
caminhoPercorrido	Caminho percorrido pelo algoritmo para montar um ciclo.
listaCiclos	Todos os caminhos que foram gerados para poder percorrer a MEF inteira passando por todos os estados e transições.
estadoDestino	Estado destino de um determinado estado corrente.
novoEstadoIncial	Retorna um novo estado inicial para começar a percorrer a MEF e gerar um novo ciclo.
casoTeste	Caso de teste final gerado pelo algoritmo.

Tabela 3.12 - Principais métodos utilizados no código de geração dos casos de teste.

Métodos	Descrição
retornaEstadosIniciais (grafo-Balanceado)	Retorna os estados iniciais da grafo balanceado.
percorrer(i)	Método recursivo responsável por percorrer o grafo balanceado até encontrar o estado inicial, gerando um caminho chamado de ciclo.
retornaListaTrans(i)	Retorna lista de transições do estado s_i .
retornaMelhorTransicao (estadoDestino)	Este método retorna o melhor estado destino para montar o caminho. Dessa forma será montado ciclos maiores que cobrem maior parte do grafo nos primeiros ciclos montados.
retornaNovoEstadoIncial (transicoesNaoVisitadas)	Depois de fechar um ciclo se ainda existir estados sem visitar, é retornado um novo estado inicial para iniciar um novo ciclo.

continua na próxima página

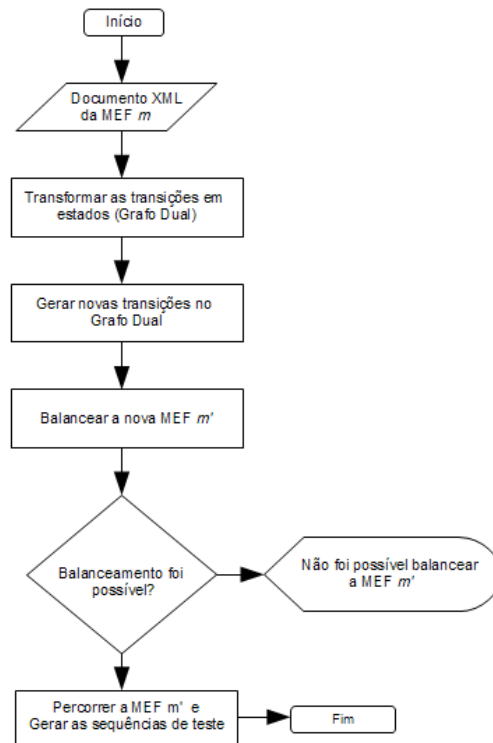


Figura 3.16 - Fluxograma do algoritmo do critério de teste *Switch Cover*.

Tabela 3.12 - Principais métodos utilizados no código de geração dos casos de teste.

Métodos	Descrição
retornaEstadoDestino(j)	Retorna estado destino do estado s_j .
montaCaminhoEuleriano (listaCiclos)	Método responsável por percorrer todos os ciclos formados e gerar apenas um caminho euleriano.

O fluxograma na Figura 3.16, apresenta em uma visão geral de como ocorre o fluxo de execuções do algoritmo implementado para o critério *H-Switch Cover*.

Utilizando o mesmo exemplo que apresenta um dos problemas da versão anterior do critério *Switch Cover* (Figura 3.13), foi realizada uma comparação de quantidade de eventos gerados pela versão antiga do critério *Switch Cover*, implementada no ambiente *GTSC*, pela versão implementada na *WEB-PerformCharts* (ARANTES et al., 2002) e a versão desenvolvida neste trabalho, o critério *H-Switch Cover*, também implementado no *GTSC*. A Tabela 3.13 apresenta a quantidade de casos de teste

e eventos gerados pelas três versões do critério. Pode-se perceber, que todas as transições são visitadas com o critério *H-Switch Cover*, enquanto que isso não ocorre com as duas outras versões do critério *Switch Cover*.

Tabela 3.13 - Comparação das versões do critério *Switch Cover*

Versão	Número de casos de teste	Número de eventos	Transições não visitadas
<i>Switch Cover</i> (Versão anterior - GTSC)	2	10	1
<i>Switch Cover</i> (WEB-PerformCharts)	1	28	1
<i>H-Switch Cover</i> (GTSC)	1	13	0

Os três critérios de teste, *UIO*, *DS* e *H-Switch Cover*, foram incorporados ao ambiente GTSC, como contribuição inicial, e, também, estão sendo incorporados ao ambiente *WEB-PerformCharts*.

4 AVALIAÇÃO DOS CRITÉRIOS DE TESTE

Este Capítulo descreve as avaliações dos critérios de teste [DS](#), [UIO](#) e *H-Switch Cover*, realizadas, em termos de custo e eficiência, em dois estudos de caso da área espacial: [APEX](#) e [SWPDC](#). Para realizar essa avaliação foi utilizado apenas o ambiente [GTSC](#). Inicialmente, este Capítulo apresenta uma breve descrição de cada estudo de caso utilizado, bem como da estratégia de avaliação adotada. Em seguida, são apresentados os experimentos realizados e os principais resultados alcançados.

Foram avaliados, no total, 21 (vinte e um) modelos elaborados em *Statecharts* dos estudos de caso propostos para avaliação dos 03 (três) critérios de teste desenvolvidos. O primeiro modelo refere-se ao primeiro estudo de caso, e os outros vinte modelos referem-se ao segundo estudo de caso. O segundo estudo de caso é considerado um *software* maior e mais complexo. Foi necessária a criação de diversos casos de uso que contemplassem os cenários nele existentes. Para validação dos critérios de teste, a estratégia utilizada neste trabalho de pesquisa consistiu na mensuração dos casos de teste gerados por esses critérios, através da utilização de métricas como medidas de custo e eficiência, já descritos na Seção [2.3](#).

Os casos de teste gerados pelos 3 (três) critérios de teste, implementados no [GTSC](#), foram executados pela ferramenta Qualidade do *Software* Embarcado em Aplicações Espaciais - Teste Automatizado de *Software* ([QSEE-TAS](#)) ([SILVA, 2008](#)). A [QSEE-TAS](#) foi projetada e implementada no escopo do projeto Qualidade do *Software* Embarcado em Aplicações Espaciais ([QSEE](#)) ([SANTIAGO et al., 2007](#)), objetivando a automatização tanto da execução de casos de teste como da geração da documentação do processo de teste.

4.1 Planejamento e Operação

Para a realização dos estudos de casos propostos, primeiramente os modelos em *Statecharts* foram elaborados baseando-se em documentos como Especificações de Requisitos de *Software* e Protocolos de Comunicação. Após isto, o conjunto de casos de teste foi gerado, de acordo com cada modelo e de acordo com cada critério, pelo ambiente [GTSC](#).

Após serem gerados automaticamente, os casos de teste foram cadastrados na ferramenta [QSEE-TAS](#) que se comunica com outro computador através de um protocolo de comunicação para o envio e o recebimento de mensagens.

Neste trabalho de pesquisa, as medidas utilizadas para avaliar os critérios de teste, em termos de custo, baseiam-se no tamanho dos conjuntos de casos de teste gerados e também no tempo ¹ que levam para encontrar um defeito no código. Em relação à eficiência, como já mencionado, foi utilizado o critério de análise de mutantes para medir a eficiência de um determinado critério de teste em encontrar defeitos no código.

A Figura 4.1, apresenta um exemplo de um conjunto de casos de teste gerado pelo critério UIO. Cada linha representa um caso de teste, podendo conter diversos eventos associados a ele como, por exemplo, o caso de teste de número 07 (sete) contém 04 (quatro) eventos: *0xEB*, *0x92*, *Type07* e *Size01*.

```

01 - NotEB
02 - 0xEB Not0x92
03 - 0xEB Not0x92 NotEB
04 - 0xEB 0x92 Type_NotOk
05 - 0xEB 0x92 TimerTimeout NotEB
06 - 0xEB 0x92 Type_NotOk EnteredCancel
07 - 0xEB 0x92 Type07 Size01
08 - 0xEB 0x92 Type1A Size04
09 - 0xEB 0x92 Type1B Size38
10 - 0xEB 0x92 Type1F Size01
11 - 0xEB 0x92 Type01 TimerTimeout Cks_Ok
12 - 0xEB 0x92 Type_NotOk EnteredCancel NotEB
13 - 0xEB 0x92 Type07 TimerTimeout NotEB
14 - 0xEB 0x92 Type07 Size01 Data00 Data01
15 - 0xEB 0x92 Type07 TypeOkSizeNotOK EnteredCancel
16 - 0xEB 0x92 Type1A TimerTimeout NotEB
17 - 0xEB 0x92 Type1A Size04 DataE100E500
18 - 0xEB 0x92 Type1A TypeOkSizeNotOK EnteredCancel
19 - 0xEB 0x92 Type1B TimerTimeout NotEB
20 - 0xEB 0x92 Type1B Size38 Data0x100
21 - 0xEB 0x92 Type1B TypeOkSizeNotOK EnteredCancel
...

```

Figura 4.1 - Exemplo de uma sequência de casos de teste gerada pelo critério UIO.

4.2 Estudo de Caso I: *Alpha, Proton and Electron monitoring eXperiment in the magnetosphere* (APEX)

O APEX é um experimento de astrofísica, previsto para ser embarcado em satélites científicos em desenvolvimento no INPE, responsável por monitorar o fluxo de partículas subatômicas na altitude de magnetosfera interna. Um protocolo proprietário foi especificado para comunicação entre o APEX e o computador OBDH (MATTIELLO-FRANCISCO; SANTIAGO, 1998). Para a realização deste trabalho usou-

¹No contexto deste trabalho o termo tempo se refere ao instante em que foi encontrado um defeito no código, ou seja, o número da sequência de teste que conseguiu identificar o mutante.

se uma versão simulada do experimento, desenvolvido na linguagem de programação Java/C/C++.

A comunicação é em modo mestre-escravo, onde o APEX é totalmente controlado pelo OBDH (MATTIELLO-FRANCISCO; SANTIAGO, 1998). APEX e OBDH foram executados em diferentes computadores. A Figura 4.2 ilustra a estrutura, envolvendo o OBDH e um emulador do APEX (*EXPEmulator*), que implementa o experimento.

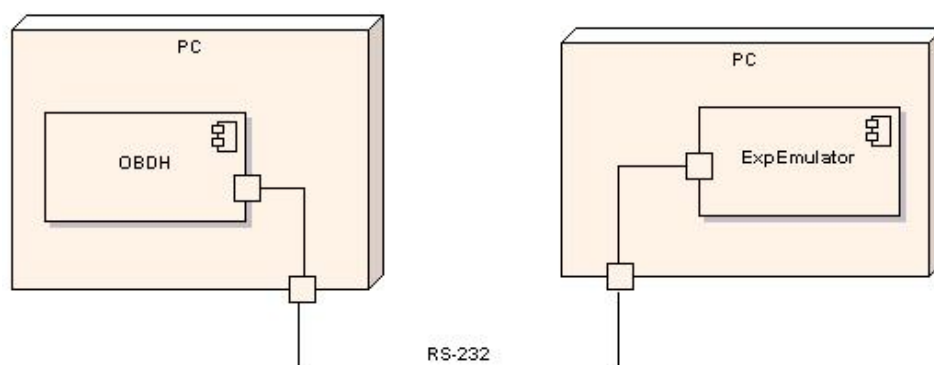


Figura 4.2 - Canal de comunicação entre o OBDH e o *EXPEmulator*.

Fonte: Mattiello-Francisco e Santiago (1998)

O comportamento do *software* APEX pode ser descrito da seguinte forma: o computador OBDH e o *software* do experimento, comunicam-se via canal de comunicação com padrão de interface RS-232 ou *Universal Serial Bus* (USB). Ao reconhecer um comando, o APEX processa e responde para o OBDH. Os comandos que podem ser reconhecidos pelo *EXPEmulator* são: reconfiguração, transmissão de dados, carga de memória (*load*), descarga de memória (*dump*), reinicialização (*reset*), obtenção de tempo (*clock*), inicialização e interrupção da aquisição de dados científicos e carga de parâmetros. O formato da mensagem de comando, definido no protocolo, é composto de 06 (seis) campos: *SYNC* (valor de sincronismo EB9), *EID* (identificação do experimento), *TYPE* (especifica os comandos aceitos), *SIZE* (quantidade de *Bytes* no campo *DATA*), *DATA* e *CKSUM* (*checksum* de 8 *bits*). Os campos *SIZE* e *DATA* são opcionais, dependendo do tipo do comando (SANTIAGO et al., 2008). A Figura 4.3 representa o formato da mensagem de comando recebida pelo APEX.

A descrição de cada um dos comandos contidos é:

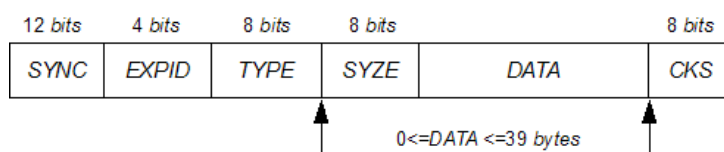


Figura 4.3 - Formato da mensagem de comando enviada pelo OBDH para o APEX
 Fonte: Mattiello-Francisco e Santiago (1998)

- *SYNC* - palavra de 12 (doze) *bits*, enviada ao APEX, que indica o sincronismo. Seu valor no campo é EB9;
- *EXP ID* - palavra de 04 (quatro) *bits* que identifica o experimento. O valor binário usado é 0010;
- *TYPE* - comando com 08 (oito) *bits* que identifica o tipo. Os comandos que representam o tipo são apresentados na Tabela 4.1;
- *DATA* - campo opcional que depende do tipo de comando. Se o comando necessitar transmitir dados, esse campo poderá ser usado várias vezes (até 38 (trinta e oito) *bytes* por comando); e
- *CKS* - palavra de 08 (oito) *bits* que verifica a integridade da mensagem.

Tabela 4.1 - Descrição dos comandos enviados pelo OBDH.

Nome do comando	Tipo	Descrição	Resposta do <i>EXPE-mulador</i>
Reinicialização	01H	Reinicia o microcontrolador	80H - Mensagem Recebida
Obtenção do tempo	02H	Solicita dados do relógio	82H - Dados do <i>Clock</i>
Inicia aquisição de dados	03H	Ativa a aquisição de dados	80H - Mensagem Recebida
Pára a aquisição de dados	04H	Desativa a aquisição de dados	80H - Mensagem Recebida
Transmissão de dados	05H	Solicita a transmissão de dados	85H - Transmite o pacote de dados ou 89H

continua na próxima página

Tabela 4.1 - Descrição dos comandos enviados pelo **OBDH**.

Nome do co- mando	Tipo	Descrição	Resposta do <i>EXPE- mulator</i>
Reconfiguração	07H	Modifica o valor do modo de operação. Especifica esse valor no campo Data	80H - Mensagem Recebida
Descarga de memória	1AH	Solicita a carga de dados na memória. Esse comando é transmitido ao OBDH como resposta de um comando de transmissão de dados	80H - Mensagem Recebida
Carga de memória	1BH	Carrega na memória do microcontrolador até 36 <i>bytes</i> de dados	80H - Mensagem Recebida
Carga de parâmetros	1FH	O <i>EXPEmulator</i> identifica os tipos de parâmetros recebidos e executa-os	80H - Mensagem Recebida

A Figura 4.4 apresenta o modelo do *software* **APEX**. Esse modelo refere-se ao principal componente do *software*: reconhecimento de comando. Esse modelo é de fundamental importância, pois ele é o responsável por analisar a conformidade dos comandos recebidos do **OBDH** em relação ao protocolo de comunicação estabelecido. Como pode ser observado no modelo, a modelagem foi desenvolvida em *Statecharts*. Conforme foi abordado no Capítulo 3, para geração dos casos de teste é necessário que as estruturas de hierarquia e concorrência do modelo sejam transformadas em uma **MEF**, para que seja possível os critérios percorrerem sua estrutura. Sendo assim, para gerar os casos de teste, foi usado o ambiente **GTSC** (**SANTIAGO et al., 2008**), que realiza essa conversão dos modelos.

4.2.1 Avaliação de eficiência

Para realizar a avaliação, em termos de eficiência dos critérios de teste, foram utilizados 19 (dezenove) operadores de mutação para a linguagem de programação Java, tanto em nível de método como em nível de classe. A partir destes 19 (dezenove) operadores de mutação foram gerados 202 (duzentos e dois) mutantes para o pri-

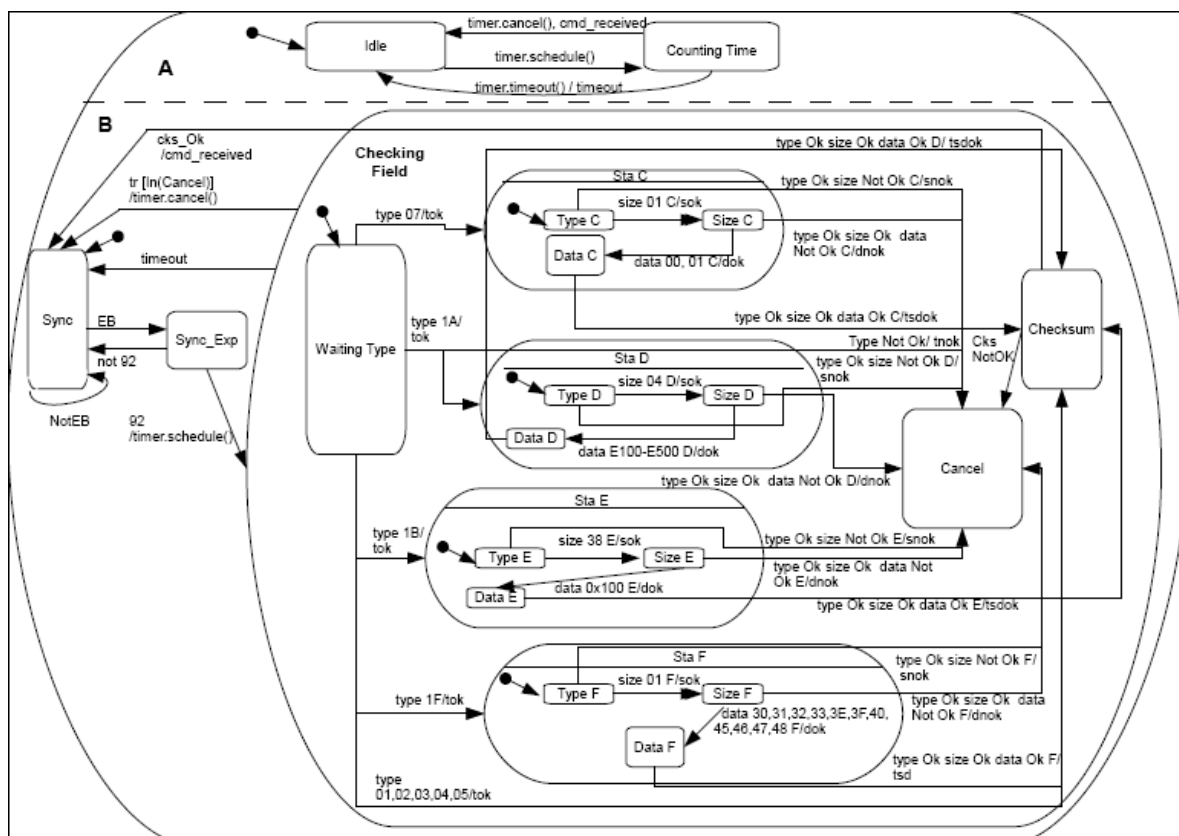


Figura 4.4 - Modelo *Statecharts* do principal componente do *software* APEX.
 Fonte: Santiago et al. (2010)

meiro estudo de caso. A Tabela 4.2, apresenta os operadores que foram utilizados e a quantidade de mutantes gerados para cada operador.

Tabela 4.2 - Operadores de Mutação para linguagem Java usados no *software* APEX.

Operador	Descrição	Quantidade
ROR	<i>Relational Operator Replacement</i>	38
AOIS	<i>Arithmetic Operator Insertion</i>	20
STRI	<i>Condition Trap of the command if</i>	11
JTI	<i>This keyword insertion</i>	8
ASRS	<i>Short-Cut Assignment Operator Replacement</i>	3
COR	<i>Conditional Operator Replacement</i>	11
JID	<i>Member variable initialization deletion</i>	6
AMC	<i>Access Modifier Change</i>	5

continua na próxima página

Tabela 4.2 - Operadores de Mutação para linguagem Java usados no *software* APEX.

Operador	Descrição	Quantidade
JSI	<i>Static modifier insertion</i>	2
JSD	<i>Static modifier deletion</i>	6
EOC	<i>Reference comparison and content comparison re- placement</i>	4
AOIU	<i>Arithmetic Operator Insertion</i>	25
COI	<i>Conditional Operator Insertion</i>	8
LOI	<i>Logical Operator Insertion</i>	21
JTD	<i>This keyword deletion</i>	4
PRV	<i>Reference assignment with other comparable varia- ble</i>	12
IOP	<i>Overriding method calling position change</i>	3
SSDL	<i>Overriding method deletion</i>	13
JDC	<i>Java-supported default constructor create</i>	2
TOTAL		202

A Tabela 4.3 apresenta os resultados relacionados a eficiência dos 03 (três) critérios de teste de acordo com o estudo de caso. Conforme mostra a Tabela 4.3, pelos escores de mutação obtidos, todos os critérios apresentaram boa eficiência. O critério DS e o critério *H-Switch Cover* apresentaram ser levemente superiores ao critério UIO. É importante ressaltar que basta que apenas um caso de teste, do conjunto de casos de teste total, identifique um mutante (ou seja, que ele mate o mutante) para que o mutante seja considerado morto. Observou-se também que nem todos os mutantes foram mortos, ou seja, nenhum caso de teste foi capaz de identificá-los. Porém, dentre os mutantes que continuaram vivos, alguns podem ser considerados equivalentes, ou seja, embora os mutantes sejam sintacticamente diferentes do original, podem vir a apresentar o mesmo comportamento, dizendo-se equivalentes.

Tabela 4.3 - Resultados sobre eficiência dos critérios de teste no *software* APEX.

Critério	Mortos	Vivos	Equivalentes	Escore de Mutação
DS	141	61	58	0,98
UIO	136	66	62	0,97
<i>H-Switch Cover</i>	152	50	45	0,98

Avaliando cada mutante que permaneceu vivo, identificaram-se quais são os mutantes considerados equivalentes. Análises realizadas no código do *software* mostraram que, durante o uso de alguns operadores de mutação, as mutações aplicadas no código não eram percebidas pelos conjuntos de casos de teste, mas poderiam ser considerados equivalentes.

O mutante AMC refere-se à mudança do modificador de acesso na linguagem Java. Tal operador foi usado em variáveis de classe. Dos mutantes criados com o operador AMC, nenhum foi morto pelos casos de teste gerados, pois para o estudo de caso proposto e os casos de teste gerados, a alteração do modificador de acesso das variáveis não causou nenhum tipo de impacto ao sistema. Por exemplo, trocar o modificador *private* para *public*. O mesmo ocorreu com os operadores JSI e JSD que, respectivamente, inserem e removem o modificador de acesso *static*. Outra situação que se refere a mutantes considerados equivalentes é o que acontece com o operador PRV, onde variáveis criadas recebem outra variável do mesmo tipo. Outro exemplo a ser levantado, é o que ocorre com o operador JDI, que apaga a inicialização de variáveis. Na maior parte dos casos, a remoção da inicialização de variáveis ao programa não causou nenhum impacto, pois na linguagem Java, dependendo do tipo de variável e do valor atribuído a ela, pode ser considerado equivalente. Por exemplo, a variável “*Object shutdownObj = null*”, continuará com o mesmo valor se tirarmos a atribuição *null* dela, pois na linguagem Java, uma variável de classe do tipo *Object*, por padrão é *null*.

Os mutantes que ficaram vivos se referem aos operadores de mutação AOIU e ao JTI. Nenhum caso de teste exercitado no código conseguiu matar estes mutantes. Dessa forma, são necessários que nas especificações modeladas, sejam incorporados novos eventos que geram casos de teste para matar tais mutantes.

4.2.2 Avaliação de Custo

Esta seção descreve a avaliação de custo realizada para os 03 (três) critérios de teste, considerando o estudo de caso [APEX](#).

Avaliando o fator tamanho, para o cenário principal do modelo proposto para o primeiro estudo de caso, a Tabela 4.4, mostra os valores referente ao tamanho e à quantidade total de eventos (transições) dos casos de teste. Considerando a quantidade de casos de teste gerados para avaliar o custo, o critério *H-Switch Cover*

apresenta ser mais relevante em relação aos outros critérios. Porém, se for considerado a quantidade de eventos associados a todos os casos de teste, tanto o critério **UIO**, como o critério **DS**, apresentam ser mais relevantes que o critério *H-Switch Cover*, ficando a cargo do projetista de teste escolher qual situação se enquadra melhor ao seu projeto de teste.

Tabela 4.4 - Avaliação de Custo - Tamanho dos casos de teste.

Critério	Quantidade de casos de teste	Quantidade de eventos
DS	35	183
UIO	35	203
<i>H-Switch Cover</i>	2	722

Outro fator utilizado como medida de custo é o tempo necessário para que um critério de teste encontre um mutante, ou seja, encontre um defeito inserido no código. Como existe uma quantidade relativamente grande de mutantes, foram criados diversos gráficos, mostrando, por operador de mutação, o tempo (ou instante) que o conjunto de caso de teste gerado pelos critérios de teste, levaram para encontrar um mutante. Nas Figuras 4.5(a) e 4.5(b), são apresentados os gráficos de custo em relação ao tempo com os operadores de mutação ROR (*Relational Operator Replacement*) e AOIU (*Arithmetic Operator Insertion*) e o instante em que uma sequência de teste encontra o mutante no código. Os demais gráficos encontram-se no Apêndice B.

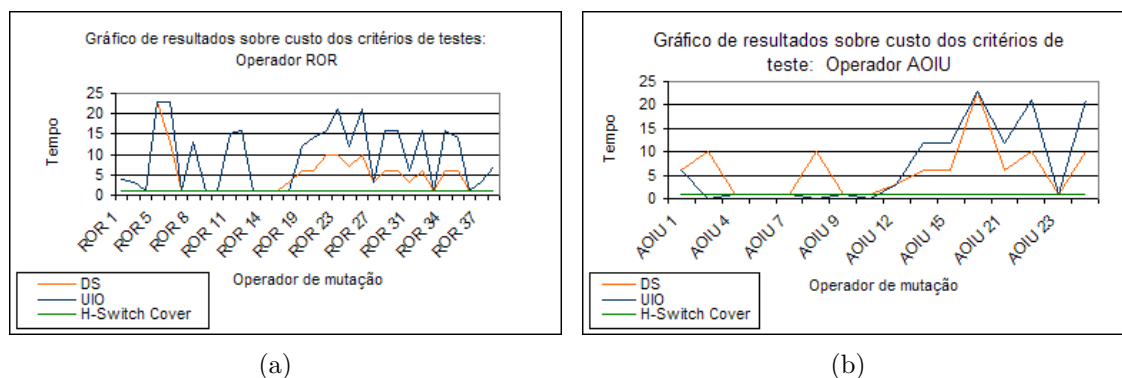


Figura 4.5 - Gráficos de resultados sobre custo dos critérios com os operadores de mutação ROR e AOIU

Como pode ser observado, o critério *H-Switch Cover* sempre apresenta uma maior rapidez para encontrar um defeito no código, mas isso se deve ao fato de o *H-Switch Cover* possuir apenas dois casos de teste para este estudo de caso, e suas sequências são maiores, fazendo com que aumentem as chances de um defeito inserido no código ser encontrado. Em relação aos critérios *DS* e *UIO*, avaliando todos os gráficos de custo gerados, o critério *DS* demonstra ser mais rápido que o critério *UIO*, ou seja, seu custo é levemente menor que o do critério *UIO*.

4.3 Estudo de Caso II: *Software of the Payload Data Handling Computer* (SWPDC)

O segundo estudo de caso é um software piloto, denominado *SWPDC*, que deverá ser embarcado em balões estratosféricos e/ou satélites científicos em desenvolvimento na coordenadoria de *CEA*. O *SWPDC* é um *software* que gerencia a aquisição de dados científicos, de diagnóstico, a partir de câmeras de raio X (*Event Pre-Processor (EPP) H1* e *EPP H2*) e um Sistema de Computação denominado *Central Electronics Unit (CEU)*. Além disso, o *SWPDC* possui funcionalidades tais como gerenciamento de memória, carregamento de novos programas (*firmware update*), e coletas periódicas de temperatura em dois sensores. Resumidamente, a principal função do *SWPDC* é adquirir e formatar dados, principalmente científicos, pela recepção e execução de comandos do computador *OBDH* e pelo envio de respostas e transmissão de dados para o *OBDH* (SILVA, 2008; SANTIAGO et al., 2007). O *Software SWPDC* foi desenvolvido na linguagem de programação C. A Figura 4.6 mostra uma visão geral do escopo do *software SWPDC*.

As principais mensagens de entrada do *SWPDC* são:

- Provenientes do *OBDH* - comandos para atuar no hardware do *Payload Data handling Computer (PDC)* e solicitar a transmissão de dados científicos;
- Provenientes do *EPP H1* - simulação de dados a serem gerados pelo *EPP H1*, de acordo com as informações obtidas pela câmera *Hard X-Ray Imager (HXI) 1*; e
- Provenientes do *EPP H2* - simulação de dados a serem gerados pelo *EPP H2*, de acordo com as informações obtidas pela câmera *HXI 2*.

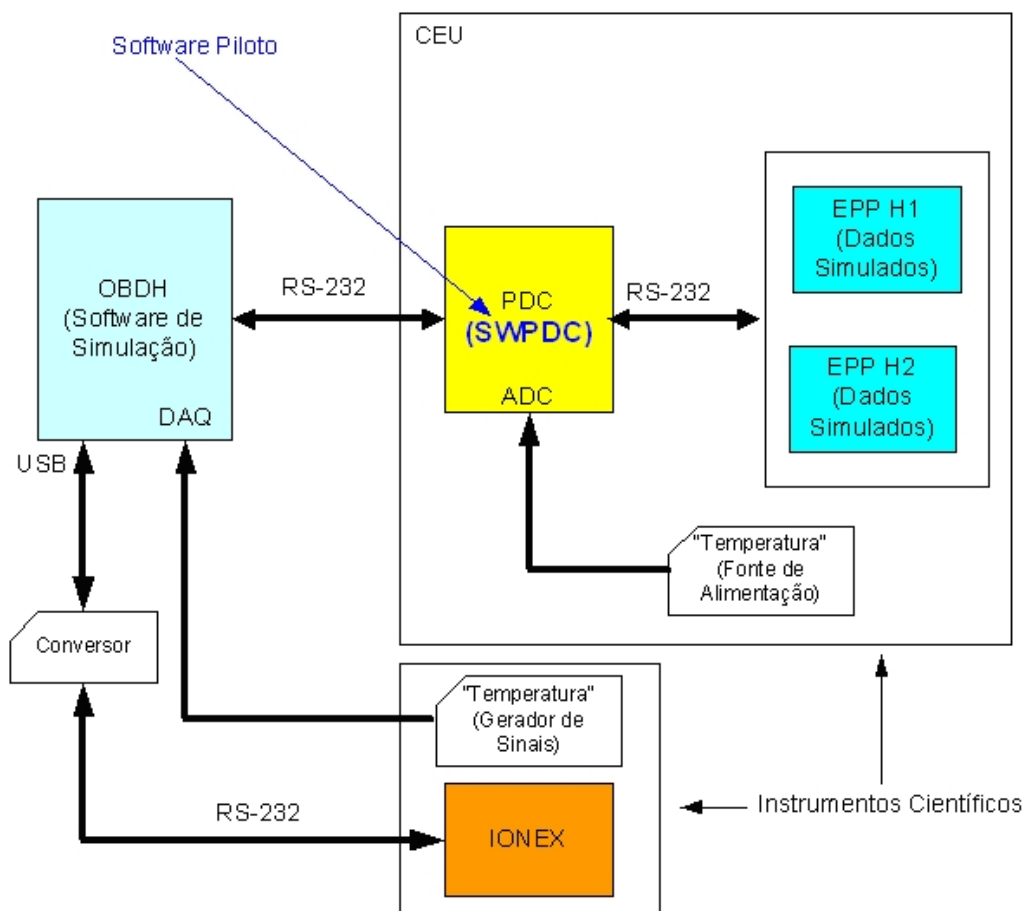


Figura 4.6 - Visão geral do escopo do *software* SWPDC.
Fonte: Santiago et al. (2007)

As principais mensagens de saída do SWPDC são:

- Para o OBDH - dados científicos;
- Para o EPP H1 - comandos para o EPP H1 para solicitar a transmissão de dados, para desligar o EPP H1 e a câmera HXI 1; e
- para o EPP H2 - comandos para o EPP H2 para solicitar a transmissão de dados, para desligar o EPP H2 e a câmera HXI 2.

Conforme mencionado anteriormente, o *software* SWPDC é composto de 20 (vinte) modelos para teste, ou seja, foram utilizados 20 (vinte) cenários para a geração dos casos de teste. Para fins de ilustração, apenas o cenário 03 é apresentado na

Figura 4.7. Este cenário, refere-se a troca de parâmetros e transmissão de dados de *housekeeping*. Os dados de *housekeeping* são quaisquer dados sobre os subsistemas do satélite (plataforma ou cargas úteis), tais como temperatura, pressão, contadores de erro de processador ou memória, relatos de eventos (anomalias detectadas), dentre outros. A Figura 4.8 apresenta uma expansão do estado *SafeM_ChangingSwPar* e a Figura 4.9 apresenta uma expansão do estado *SafeM_LoopingHK*.

Alguns cenários possuem outros subcenários referentes à expansão de alguns estados do cenário principal. Devido ao tamanho dos modelos, não serão apresentados todos neste texto, no entanto, alguns cenários poderão ser encontrados no Apêndice C.

4.3.1 Avaliação de eficiência

Para a avaliação de eficiência dos casos de teste, gerados pelos critérios no segundo estudo de caso, foram utilizados 09 (nove) operadores de mutação para linguagem de programação C. A partir destes 09 operadores de mutação, foram gerados 57 (cinquenta e sete) mutantes para cada um dos 20 (vinte) cenários modelados para o *software* SWPDC. A Tabela 4.5 apresenta os operadores utilizados e a quantidade de mutantes gerados para cada um dos operadores.

Tabela 4.5 - Operadores de Mutação para linguagem C usados no *software* SWPDC.

Operador	Descrição	Quantidade
ORRN	<i>Change Relational operator</i>	8
SBRC	<i>Break replacement by continue</i>	6
Vsrr	<i>Scalar Variable Reference Replacement</i>	9
SSDL	<i>Statement deletion</i>	7
STRI	<i>Trap on if condition</i>	7
OSSN	<i>Shift assignment mutation</i>	5
SSOM	<i>Sequence Operator Mutation</i>	4
OAAN	<i>Change Arithmetic operator</i>	5
OCNG	<i>Logical context negation</i>	6

A Tabela 4.6 apresenta os resultados relacionados à eficiência dos 03 (três) critérios de teste, de acordo com cada cenário. É possível perceber, pelos escores de mutação obtidos, que todos os critérios apresentaram boa eficiência. Em alguns cenários, por exemplo, nos cenários 15 (quinze), 18 (dezoito) e 19 (dezenove), os critérios DS e *H-Switch Cover* apresentaram ser levemente superiores ao critério UIO em

termos de eficiência. E nos cenários 04 (quatro), 06 (seis) e 08 (oito), o critério *H-Switch Cover* apresentou uma melhor eficiência em relação aos critérios **UIO** e **DS**. Os demais cenários avaliados apresentaram o mesmo desempenho em função dos valores obtidos nos escores de mutação. De forma geral, o critério *H-Switch Cover* destacou-se mais em termos de eficiência.

Tabela 4.6 - Resultados sobre a eficiência dos 20 (vinte) cenários de teste no *software* SWPDC.

Cenários	Critérios	Mortos	Vivos	Equivalentes	Escore
01	DS	9	48	48	1
	UIO	9	48	48	1
	<i>H-Switch Cover</i>	9	48	48	1
02	DS	15	42	42	1
	UIO	15	42	42	1
	<i>H-Switch Cover</i>	15	42	42	1
03	DS	35	22	20	0,95
	UIO	35	22	20	0,95
	<i>H-Switch Cover</i>	35	22	20	0,95
04	DS	28	29	28	0,97
	UIO	28	29	28	0,97
	<i>H-Switch Cover</i>	29	28	28	1
05	DS	29	28	26	0,94
	UIO	29	28	26	0,94
	<i>H-Switch Cover</i>	29	28	26	0,94
06	DS	29	61	25	0,91
	UIO	29	66	25	0,91
	<i>H-Switch Cover</i>	30	29	27	1
07	DS	35	22	19	0,92

continua na próxima página

Tabela 4.6 - Resultados sobre a eficiência dos 20 (vinte) cenários de teste no *software* SWPDC.

Cenários	Critérios	Mortos	Vivos	Equivalentes	Escore
08	UIO	35	22	19	0,92
	<i>H-Switch Co-ver</i>	35	22	19	0,92
	DS	35	22	20	0,95
09	UIO	35	22	20	0,95
	<i>H-Switch Co-ver</i>	36	21	20	0,97
	DS	40	17	14	0,95
10	UIO	40	17	14	0,95
	<i>H-Switch Co-ver</i>	41	16	13	0,95
	DS	36	21	17	0,90
11	UIO	36	21	17	0,90
	<i>H-Switch Co-ver</i>	36	21	17	0,90
	DS	35	22	19	0,92
12	UIO	35	22	19	0,92
	<i>H-Switch Co-ver</i>	35	22	19	0,92
	DS	38	19	16	0,93
13	UIO	38	19	16	0,93
	<i>H-Switch Co-ver</i>	38	19	16	0,93
	DS	39	18	16	0,95
14	UIO	39	18	16	0,95
	<i>H-Switch Co-ver</i>	39	18	16	0,95
	DS	40	17	16	0,98
	UIO	40	17	16	0,98
	<i>H-Switch Co-ver</i>	40	16	45	0,98

continua na próxima página

Tabela 4.6 - Resultados sobre a eficiência dos 20 (vinte) cenários de teste no *software* SWPDC.

Cenários	Critérios	Mortos	Vivos	Equivalentes	Escore
15	DS	37	20	16	0,98
	UIO	37	20	16	0,97
	<i>H-Switch Co-ver</i>	37	16	45	0,98
16	DS	36	21	19	0,95
	UIO	36	21	19	0,95
	<i>H-Switch Co-ver</i>	36	21	19	0,95
17	DS	37	20	18	0,95
	UIO	37	20	18	0,95
	<i>H-Switch Co-ver</i>	37	20	18	0,95
18	DS	38	19	17	0,95
	UIO	37	18	16	0,90
	<i>H-Switch Co-ver</i>	38	19	17	0,95
19	DS	39	18	16	0,95
	UIO	38	19	14	0,88
	<i>H-Switch Co-ver</i>	39	18	16	0,95
20	DS	37	20	17	0,93
	UIO	37	20	17	0,93
	<i>H-Switch Co-ver</i>	37	20	17	0,93

Dentre os mutantes que continuaram vivos em cada cenário, foi realizada uma análise no código fonte do *software* do estudo de caso e também nos casos de teste exercitados, para identificar quais mutantes permaneceram vivos e, conseqüentemente, considerados equivalentes. Na maioria dos casos, onde os mutantes continuaram vivos, os casos de teste gerados não alcançaram o local onde estava o defeito no código. Isso se deve ao fato de que alguns cenários foram modelados para casos de uso específicos. Por exemplo, a maioria dos cenários criados requerem funcionalidades

referente às câmeras [EPP](#) . O cenário 04 (quatro) não requer essa funcionalidade, no entanto, alguns operadores foram utilizados para criar defeitos no código exatamente nessa funcionalidade. Dessa forma, todos os defeitos que não estavam relacionados às câmeras de [EPP](#), permaneceram vivos e são considerados equivalentes.

Outra situação que fizesse com que diversos defeitos fossem considerados equivalentes foi referente ao operador de mutação SSOM. Este operador muda a ordem de algumas expressões. Por exemplo, as expressões apresentadas a seguir podem gerar diversos defeitos que são equivalentes ao programa principal, pois no geral algumas expressões não dependem uma das outras.

Expressões: $s[i]=s[j]$, $s[j]=c$, $c=s[i]$.

Mutante gerado: $s[j]=c$, $c=s[i]$, $s[i]=s[j]$.

Depois de decidido quais mutantes são equivalentes ao programa original, foram avaliados os mutantes que permaneceram vivos. Os mutantes que ficaram vivos se referem ao operador de mutação OSSN. Nenhum caso de teste exercitado no código conseguiu matar estes mutantes. Dessa forma, seria necessário que novos modelos fossem criados para que gerar casos de teste que matem tais mutantes.

4.3.2 Avaliação de Custo

Esta Seção descreve as avaliações e discussões sobre os principais resultados obtidos, em termos de custo dos casos de teste gerados pelos critérios de teste [UIO](#), [DS](#) e *H-Switch Cover*, nos 20 (vinte) cenários do estudo de caso [SWPDC](#).

Para avaliar o custo em relação à quantidade de casos de teste gerados pelos critérios de teste do segundo estudo de caso, foram levantadas informações referentes a cada cenário proposto. Tais informações podem ser observadas na Tabela [4.7](#). Note que, o critério *H-Switch Cover* sempre gerou apenas uma sequência de teste, logo sempre apresentou um custo menor em todos os cenários, considerando o fator tamanho do conjunto de casos de teste gerados. Por outro lado, o critério [UIO](#) apresentou menor custo comparado com o critério [DS](#). A Figura [4.10](#) mostra, em forma de gráfico, os valores de custo para cada critério de teste para o segundo estudo de caso.

Tabela 4.7 - Resultados de custo dos critérios de teste no software *SWPDC*.

Cenário	DS	UIO	<i>H-Switch</i> <i>Cover</i>
1	5	5	1
2	11	9	1
3	24	16	1
4	21	10	1
5	43	12	1
6	40	18	1
7	31	19	1
8	31	24	1
9	17	14	1
10	27	21	1
11	28	14	1
12	27	20	1
13	27	20	1
14	27	20	1
15	27	20	1
16	28	14	1
17	27	20	1
18	25	20	1
19	27	20	1
20	27	20	1

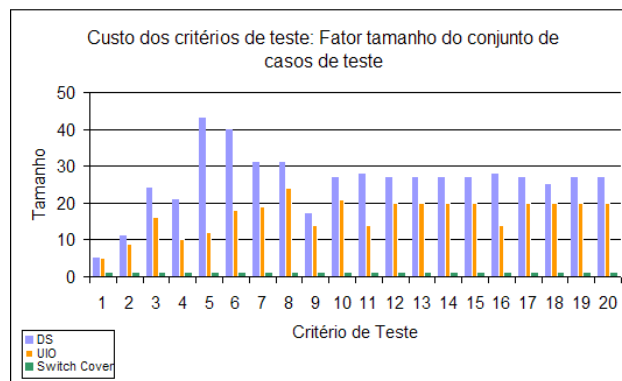


Figura 4.10 - Gráfico de custo por cenário dos critérios de teste: Quantidade de casos de teste.

Em relação à quantidade de eventos (transições) pertencentes a todos os casos de teste gerados, é mostrado na Tabela 4.8 a quantidade de eventos dos casos de teste referente a cada cenário. Observe que a quantidade de eventos para os critérios **DS**

e **UIO** são muito similares, são maiores do que do critério *H-Switch Cover* em todos os cenários. Isso se deve ao fato de que os critérios **DS** e **UIO** apresentarem uma quantidade maior de casos de teste, somando-se todos os eventos relacionados a cada caso de teste, aumentando consideravelmente o custo em relação à quantidade de eventos de teste a serem exercitados no *software*.

Tabela 4.8 - Quantidade de eventos dos casos de teste dos critérios **DS**, **UIO** e *H-Switch Cover*.

Cenários	Quantidade DS	Quantidade UIO	Quantidade <i>H-Switch Cover</i>
1	22	20	4
2	77	77	10
3	324	324	23
4	252	252	20
5	989	989	36
6	860	860	39
7	527	495	30
8	434	464	31
9	170	170	16
10	350	377	27
11	434	434	27
12	350	350	27
13	350	350	27
14	350	350	27
15	350	350	27
16	15	15	27
17	434	434	27
18	350	350	27
19	350	350	27
20	350	350	27

Considerando o fator tempo para medir o custo do critério de teste, foi avaliado em cada cenário quanto tempo (ou instante) é necessário para conjunto de caso de teste, de um determinado critério, encontrar um defeito no código. Em cada cenário é verificado qual caso de teste identifica um defeito no código por primeiro. As Figuras 4.11(a) e 4.11(b) apresentam, para o cenário 01 (um) e cenário 02 (dois), os gráficos relacionados com o tempo. Os gráficos gerados para os outros 18 (dezoito) cenários encontram-se no Apêndice B.

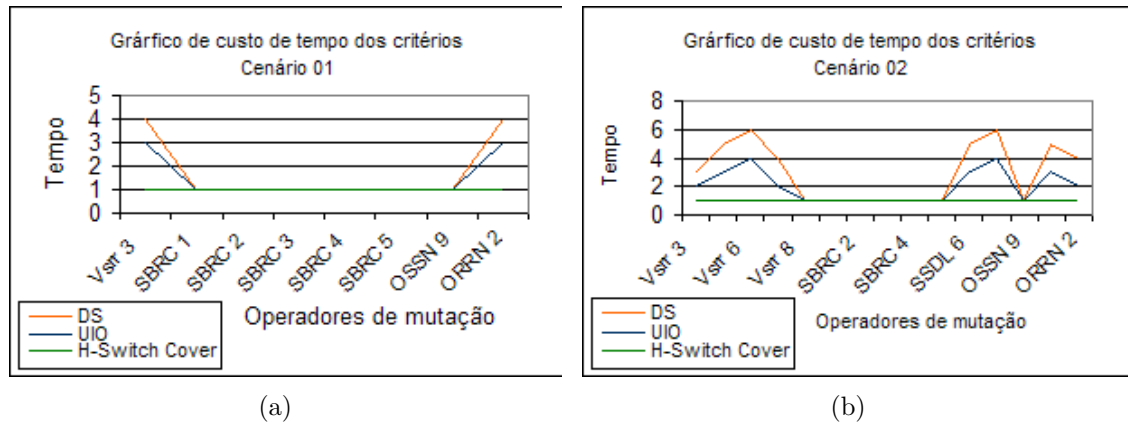


Figura 4.11 - Gráfico de custo para os cenários 01 (um) e 02 (um): Fator tempo.

Avaliando todos os gráficos gerados foi possível perceber que o critério *H-Switch Cover* sempre demonstrou ter um melhor custo, devido a sua quantidade de casos de teste. Como já mencionado, para todos os cenários, o critério *H-Switch Cover* gerou 01 (um) caso de teste. Mas se avaliado de forma separada o critério *UIO* e o critério *DS*, o critério *UIO*, na maioria dos cenários mostrou-se mais rápido para encontrar defeitos no código, ou seja, dos casos de teste gerados pelo critério *UIO*, os primeiros casos gerados conseguiram identificar um defeito no código testado, enquanto que, em relação ao critério *DS* era necessário que mais casos de teste fossem executados.

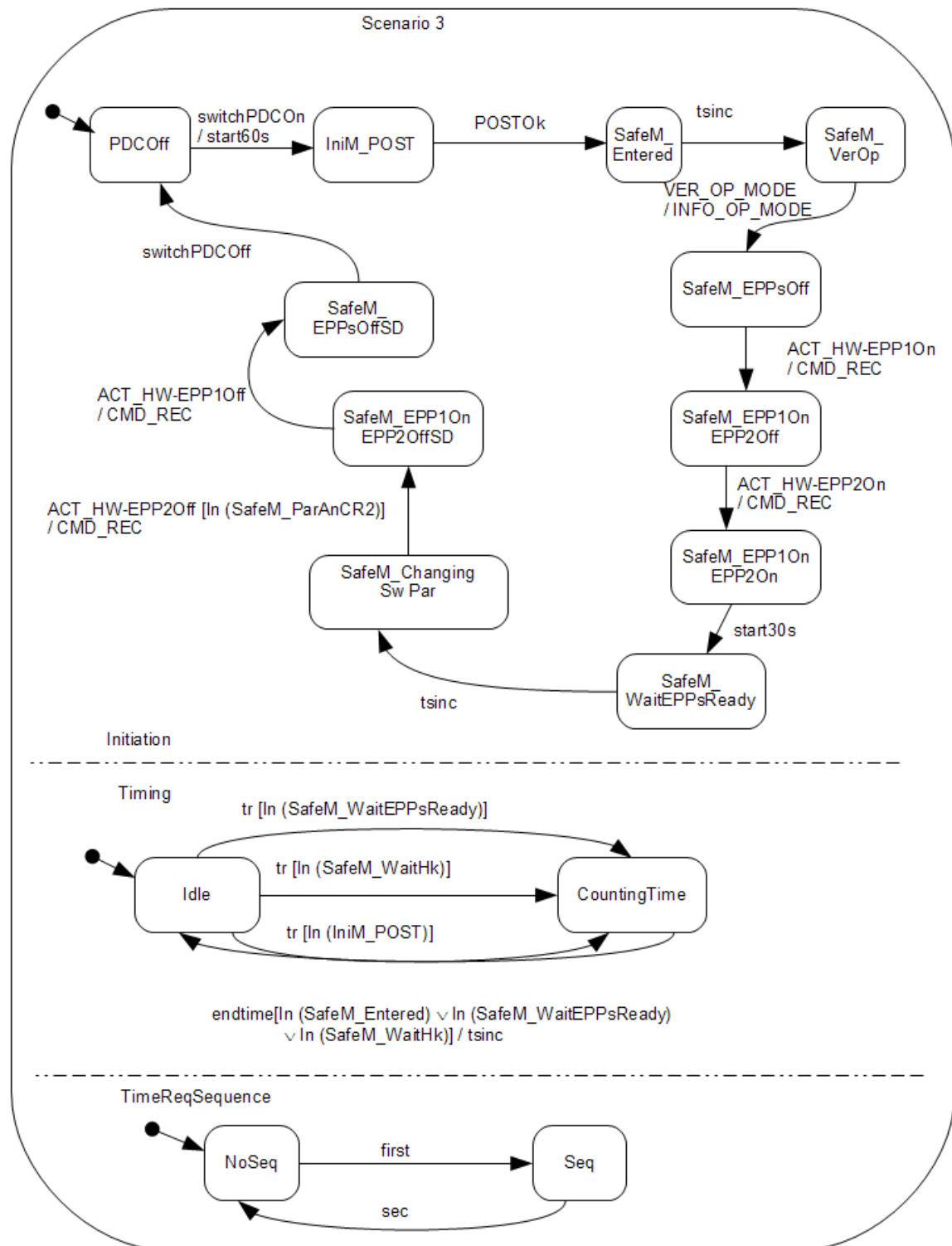


Figura 4.7 - Modelo *Statecharts* do cenário 03 (três) do *software* SWPDC.
Fonte: Santiago et al. (2010)

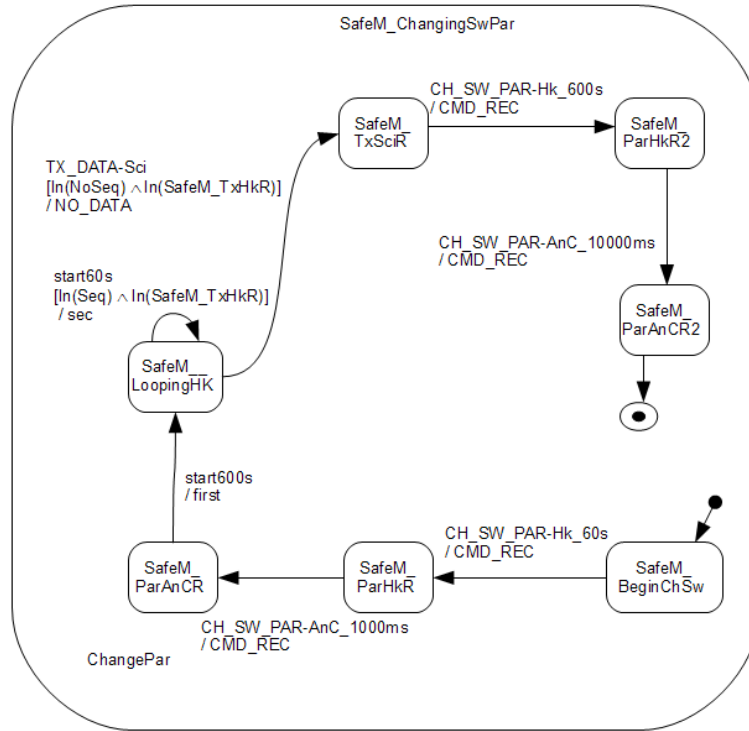


Figura 4.8 - Modelo referente a expansão do estado *SafeM_ChangingSwPar*.
Fonte: Santiago et al. (2010)

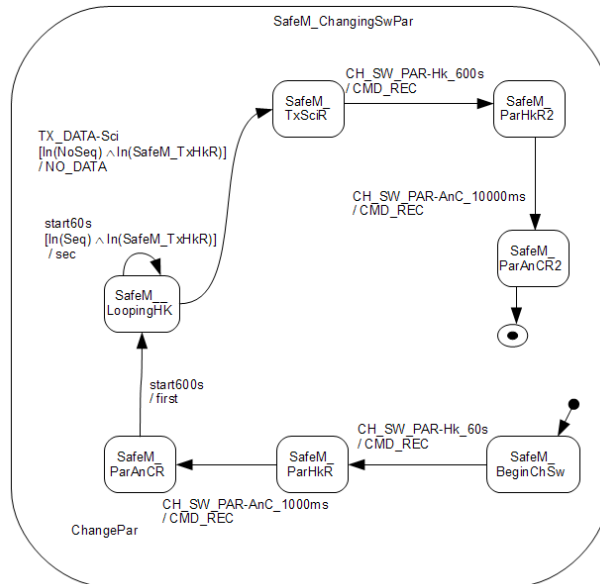


Figura 4.9 - Modelo referente a expansão do estado *SafeM_LoopingHK*.
Fonte: Santiago et al. (2010)

5 CONCLUSÃO

Este Capítulo apresenta as principais conclusões, considerações, contribuições, bem como as limitações e dificuldades encontradas durante a pesquisa. Ao final, também são apresentadas algumas sugestões para realização de trabalhos futuros.

Este trabalho teve como objetivo principal o desenvolvimento dos critérios de teste **UIO**, **DS** e *Switch Cover*, visando sua integração a dois ambientes de teste do **INPE**, bem como uma investigação preliminar do custo e da eficiência destes critérios concebidos com base em uma avaliação empírica.

5.1 Considerações Gerais

Este trabalho apresentou a implementação e a avaliação empírica da relação custo e eficiência de alguns critérios de teste para **MEF**, sendo eles **DS UIO** e *Switch Cover*. Melhorias no critério *Switch Cover* foram realizadas, dando origem a uma nova versão do critério, que foi denominada de *H-Switch Cover*. Tais critérios foram integrados a dois ambientes de teste do **INPE**: ambiente **GTSC** e ambiente *WEB-PerformCharts*. É importante ressaltar que os resultados aqui obtidos levaram em conta as características do *software* utilizado e do ambiente de teste proposto, e isso determinou: *i*) o esforço de teste atribuído; *ii*) a quantidade e quais operadores de mutação serão utilizados; *iii*) como ocorrerá a execução dos casos de teste; e *iv*) quais técnicas serão utilizadas para avaliar os resultados gerados. Para a sua avaliação, os critérios foram aplicados na execução de dois estudos de caso inseridos no contexto da área espacial. O primeiro, *software APEX*, é responsável por monitorar o fluxo de partículas subatômicas na altitude de magnetosfera interna. O segundo, *software SWPDC*, é responsável por adquirir e formatar dados, principalmente científicos, pela recepção e execução de comandos do **OBDH**.

Foram desenvolvidos 03 (três) critérios de teste para **MEF**, utilizando a linguagem Java. Tais critérios recebem como entrada uma **MEF** no formato **XML** e de acordo com as regras de cada critério, esta **MEF** é percorrida, gerando os respectivos casos de teste. Os casos de testes gerados foram cadastrados na ferramenta **QSEE-TAS**, cujo objetivo é automatizar a execução de testes funcionais para *software* embarcado em satélites. Todos os casos de teste foram exercitados nos dois estudos de caso propostos. Foram utilizadas métricas de teste para medir custo e eficiência dos casos de teste gerados pelos critérios.

Nas avaliações realizadas sobre a relação custo e eficiência dos critérios de teste, constatou-se que, em termos de eficiência, os três critérios apresentaram ser eficientes em relação aos escores de mutação obtidos nos dois estudos de caso. Em relação a eficiência, os critérios **DS** e *H-Switch Cover* foram ligeiramente superiores do que o critério **UIO**. Em termos de custo, foi considerado tanto o tamanho do conjunto de casos de teste gerados, como também o momento em que o conjunto de casos de teste encontra o defeito no código. Considerando o tamanho do conjunto de casos de teste gerado pelos critérios, o *H-Switch Cover* apresentou-se melhor, devido ao fato dele gerar poucos casos de teste. Avaliando separadamente os critérios **DS** e **UIO**, o critério **UIO** gera menores conjuntos de casos de teste. Por outro lado, considerando a quantidade de eventos gerados pelos casos de teste, no primeiro estudo de caso, **APEX**, os critérios **UIO** e **DS** apresentaram uma quantidade menor de eventos gerados do que o critério *H-Switch Cover*. Já no segundo estudo de caso, **SWPDC**, os critérios **UIO** e **DS** apresentam uma quantidade bem maior de eventos do que a apresentada pelo critério *H-Switch Cover*. Isso se deve ao fato dos critérios **UIO** e **DS** apresentarem uma quantidade maior de casos de teste, e quando somados todos os eventos, o custo se torna maior.

Quanto à análise de custo dos critérios em relação ao tempo que eles levam para detectar um defeito no código, o critério *H-Switch Cover* foi mais rápido que os outros critérios analisados, pelo fato de possuir menos casos de teste. Avaliando separadamente os critérios **DS** e **UIO**, no primeiro estudo de caso o critério **DS** apresentou-se levemente mais rápido que o critério **UIO**, já no segundo estudo de caso, o **UIO** mostrou-se ser mais rápido, porém as diferenças de custo em relação a tempo entre os critérios **UIO** e **DS** são poucas.

De forma geral, para os estudos de caso propostos, em termos de eficiência o critério *H-Switch Cover* apresentou-se melhor. Em termos de custo, considerando a quantidade de casos de teste, o critério *H-Switch Cover* também foi melhor, e considerando a quantidade de eventos, os critérios **UIO** e **DS** foram melhores no primeiro estudo de caso, porém no segundo estudo de caso, nos 20 (vinte) modelos avaliados, o critério *H-Switch Cover* apresentou uma quantidade melhor de eventos, tendo assim um menor custo.

5.2 Principais contribuições

Este trabalho de pesquisa em nível de mestrado possibilitou realizar as seguintes contribuições:

- O desenvolvimento de três critérios de teste para [MEF](#), possibilitando a geração automática dos casos de teste. Os critérios desenvolvidos foram: DS, UIO e *H-Switch Cover*. Tais critérios foram incorporados ao ambiente [GTSC](#) e estão sendo incorporados ao ambiente *WEB-PerformCharts*.
- Durante o desenvolvimento dos critérios foram realizadas melhorias no critério *Switch Cover*, no que diz respeito ao seu desempenho, dando origem a uma nova versão denominada *H-Switch Cover*. O critério atualmente gera um menor número de casos de teste, e a quantidade de eventos por caso de teste também diminuiu, mantendo a mesma eficiência; e
- Avaliações de custo e eficiência dos critérios de teste desenvolvidos para a mensuração da adequabilidade dos casos de teste gerados a partir dos critérios.

5.3 Limitações e dificuldades

Durante o desenvolvimento e a condução dos estudos empíricos para avaliar a eficiência dos critérios de teste, alguns fatores limitantes resultaram na necessidade de um esforço adicional para a realização deste trabalho. Alguns dos fatores são:

- Desenvolvimento dos critérios de teste de forma que eles conseguissem lidar com [MEF](#) complexas.
- O critério análise de mutantes da técnica de teste [TBD](#) possibilitou avaliar a eficiência dos casos de teste gerados pelos critérios de teste. Um esforço adicional foi necessário por causa da inexistência, até o término deste trabalho, de uma ferramenta de apoio a geração e execução de mutantes para o ambiente no qual se encontravam os estudos de caso propostos. A forma como é tratada a geração e a execução dos mutantes em um sistema relativo, mais precisamente, em um sistema embarcado, é diferente de um sistema considerado como básico ou de informação. Devido a isso, os mutantes gerados para os dois estudos de caso, tiveram que ser gerados de

forma manual e a execução dos casos de teste foi realizada em um mutante por vez, pois para cada mutante criado no código, fazia-se necessário embarcar o código novamente;

- Tempo necessário para cadastrar cada caso de teste gerado pelos critérios;
- Embora os casos de teste tenham sido cadastrados na ferramenta [QSEETAS](#) para se comunicar com o *software* embarcado e executá-los, não foi possível executá-los de uma única vez, pois isso inviabilizava a análise dos mutantes. Considerando o caso de teste como uma sequência de eventos, em muitos casos, devido a complexidade do *software* e do próprio caso de teste, não foi possível executar um caso de teste por completo e sim, um evento de cada vez, demandando muito tempo; e
- A identificação dos mutantes equivalentes normalmente é realizada manualmente, o que acarretou num aumento do custo da aplicação do critério de análise de mutantes.

5.4 Sugestões para trabalhos futuros

Para a continuidade deste trabalho de pesquisa, destacam-se algumas sugestões para trabalhos futuros, dentre elas:

- Realizar novos experimentos em sistemas da área espacial, pois, embora os *softwares* utilizados neste trabalho como estudos de caso tenham sido submetidos a 21 (vinte e um) modelos de teste diferentes, não é possível generalizar os resultados alcançados. No entanto, a análise apresentada é um bom ponto de partida para uma melhor conclusão sobre a importância de se saber qual o melhor critério de teste, no contexto de *software* embarcado em satélites, a ser considerado para geração de casos de teste;
- Avaliar outros critérios de teste para [MEF](#) e incorporar aos ambientes de teste do [INPE](#); e
- Utilizar outras métricas para mensurar custo e eficiência dos critérios de teste.

A autora deste trabalho acredita que o futuro desta importante área de pesquisa requer o desenvolvimento de novos critérios de teste, ou na melhoria dos já existentes,

como também de técnicas, ferramentas, modelos e estratégias de teste, capazes de auxiliar cada vez mais o desenvolvimento seguro de *softwares* críticos, em especial, na área espacial.

REFERÊNCIAS BIBLIOGRÁFICAS

- AMARAL, A. S. M. S. **Geração de casos de testes para sistemas especificados em Statecharts**. 162 p. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais, (INPE-14215-TDI/1116). Dissertação (Mestrado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2005. Disponível em: <<http://urlib.net/rep/sid.inpe.br/MTC-m13080/2006/02.14.19.24?languagebutton=pt-BR>>. Acesso em: 22 feb. 2010. 23, 31
- ANTONIOL, G.; BRIAND, L. C.; PENTA, M. D.; LABICHE, Y. A. A case study using the round-trip path strategy for state-based class testing. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING, 13., 2002, Annapolis. **Proceedings...** Annapolis, Maryland, USA: IEEE Computer Society, 2002. 35
- ARANTES, A. O.; VIJAYKUMAR, N. L.; SANTIAGO, V.; GUIMARAES, D. Test case generation for critical systems through a collaborative web-based tool. In: INTERNATIONAL CONFERENCE ON INOVATION IN SOFTWARE ENGINEERING (ISE 2008), 2008, Viena, Áustria. **Proceedings...** Viena: IEEE Computer Society, 2002. p. 163–168. 3, 4, 33, 34, 49, 57, 61
- BASTOS, A.; RIOS, E.; CRISTALLI, R.; MOREIRA, T. **Base de conhecimento em testes de software**. 2. ed. São Paulo: Ed. Martins, 2007. 6
- BINDER, R. V. **Testing object-oriented systems models, patterns, and tools**. 6. ed. San Diego: Addison Wesley, 2005. 1, 9, 11, 13, 35
- BRIAND, L.; LABICHE, Y.; WANG, Y. Using simulation to empirically investigate test coverage criteria based on statechart. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE'04), 2004, Edinburgh, Scotland. **Proceedings...** Edinburgh: IEEE Computer Society, 2004. p. 86–95. 36
- BUDKOWSKI, S.; DEMBINSKI, P. An introduction to estelle: A specification language for distributed systems. In: COMPUTER NETWORK AND ISDN SYSTEMS, 14., 1987, Amsterdam, The Netherlands. **Proceedings...** Amsterdam: Elsevier Science Publishers B. V., 1987. p. 3–23. 2, 9

CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, SE-4(3), p. 178–187, 1978. [10](#), [14](#), [35](#), [36](#)

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to algorithms**. 2. ed. Massachusettes, USA: McGraw-Hill book company, 2001. [35](#)

DEITEL, H. M.; DEITEL, P. J. **Java - como programar**. 6. ed. New York: Pearson Education, 2005. [41](#)

DELAMARO, E.; MANDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Rio de Janeiro: Ed. Elsevier, 2007. [7](#), [8](#), [9](#), [14](#), [15](#), [17](#), [27](#), [28](#), [30](#), [35](#)

ELLSBERGER, I.; HOGREFE, D.; SARMA, A. **SDL: Formal object-oriented language for communicating systems**. 2. ed. Prentice Hall Europe: Prentice Hall, 1997. 312 p. [2](#), [9](#)

FERREIRA, E.; SANTIAGO, V.; GUIMARAES, D.; VIJAYKUMAR, N. L. Evaluation of test criteria for space application software modeling in statecharts. In: INTERNATIONAL CONFERENCE ON INNOVATION IN SOFTWARE ENGINEERING, 2008, Viena, Áustria. **Proceedings...** Viena: Elsevier Science Publishers B. V., 2008. p. 157–162. [3](#), [29](#), [36](#), [37](#)

FUJIWARA, S.; BOCHMAN, G. V.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. In: TRANSACTIONS ON SOFTWARE ENGINEERING, 17., 1991, New York, USA. **Proceedings...** New York: IEEE Computer Society, 1991. p. 591–603. [27](#)

GILL, A. **Introduction to the theory of finite-state machine**. Nova York, NY, EUA: McGraw-Hill, 1962. [9](#), [14](#)

GONENC, G. A method for the design of fault detection experiments. **IEEE Transactions on Computers**, v. 19, p. 551–558, 1970. [14](#)

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, North-Holland, v. 8, p. 231–274, 1987. [2](#), [9](#), [12](#), [13](#)

HARROLD, M. J. Testing: a roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, 17., 2000, New York, USA. **Proceedings...** New York: ACM Press, 2000. p. 61–72. [31](#)

HECHT, H. **Flow analysis of Computer Programs**. New York, EUA: Elsevier Science Inc, 1977. 7

HERCULANO, P. F. R.; DELAMARO, M. E. Análise de cobertura de critérios de teste estruturais a partir de conjuntos derivados de especificações formais: um estudo comparativo. In: I BRAZILIAN WORKSHOP ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING (SAST), 2007, João Pessoa, Brasil. **Proceedings...** João Pessoa: IEEE Computer Society, 2007. 36

IEEE. Institute of electric and electronic engineers. **Standard glossary of software engineering terminology**, Standard 610.12, 1990. 5

KOHAVI, Z. **Switching and finite automata theory**. 2. ed. New York: McGraw-Hill, 1978. 15

LEVENSON, N. The role of software in recent aerospace accidents. In: INTERNATIONAL SYSTEM SAFETY CONFERENCE (ISSC), 2001, Huntsville, USA. **Proceedings...** Huntsville: IEEE Computer Society, 2001. p. 10–14. 1

LIPSCHUTZ, S.; LIPSON, M. **Matemática discreta**. 2. ed. Porto Alegre: Bookman - Coleção Schaum, 1997. 24, 57

MA, Y.; KWON, Y. R.; KIM, S. Statical investigation on class mutation operators. **ETRI Journal**, v. 31, p. 140–150, 2009. 28

MALDONADO, J. C. **Crítérios potenciais usos: Uma contribuição ao teste estrutural de software**. 262 p. Tese (Doutorado) — Universidade Estadual de Campinas (UNICAMP), Tese (doutorado) - Programa de Pós-Graduação em Engenharia Elétrica, Campinas, 1991. Disponível em: <<http://libdigi.unicamp.br/document/?code=vtls000031945>>. Acesso em: 05 dez. 2009. 7

MARTINS, E.; SABIAO, S. B.; AMBROSIO, A. M. Condata: a tool for automating specification-based test case generation for communication systems. In: INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 1999, Hawaii, USA. **Proceedings...** Hawaii: IEEE Computer Society, 1999. p. 303–319. 33, 48

MATIELLO, M. F.; SANTIAGO, V. A.; AMBROSIO, A. M.; COSTA, R.; JOGAIB; LEISE. Verificação e validação na terceirização de software embarcado em aplicações espaciais. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE

SOFTWARE (SBQS2006), 2006, Espirito Santo. **Anais...** Vila Velha: Instituto Nacional de Pesquisas Espaciais (INPE), 2006. Disponível em:
<<http://mtc-m16.sid.inpe.br/rep/sid.inpe.br/mtc-m16@80/2006/08.21.20.22?mirror=sid.inpe.br/banon/2003/08.15.17.40.18&metadataarepository=sid.inpe.br/mtc-m16@80/2006/08.21.20.22.48>>.
Acesso em: 15 nov. 2009. 1

MATTIELLO-FRANCISCO, M. F.; SANTIAGO, V. A. (**EXP OBDH**) **communication protocolo definition**: A case study for (plavis). São José dos Campos, 1998. 12 p. 64, 65, 66

MYERS, G. J. **The art of software testing**. 2. ed. Canada: John Wiley and Sons, 2004. 1, 3, 6, 13

NAITO, S.; TSUNOYAMA, M. Fault detection for sequential machines by transition tours. In: FAULT TOLERANT COMPUTING CONFERENCE (FTCS 1981), 1981. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1981. p. 238–243. 34, 35

NTAFOS, S. C. An evaluation of required element testing strategies. Florida, USA. 35

OFFUTT, A. J.; CRAFT, W. M. Using compiler optimization techniques to detect equivalent mutants. **Software Testing, Verification and Reliability**, v. 4, p. 131–154, 1994. 29, 30

PETERSON, J. L. **Petri net theory and the modeling of systems**. Englewood Cliffs, NJ: Prentice Hall, 1981. 2, 9

PETRENKO, A.; YEVTUSHENKO, N. Testing from partial deterministic fsm specifications. In: IEEE TRANSACTIONS ON COMPUTERS, 54., 2005, USA. **Proceedings...** Washington: IEEE Computer Society, 2005. p. 1154–1165. 2, 10, 36

PIMONT, S.; RAULT, J. A software reliability assessment based on a structural and behavioral analysis of programs. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 1976, NY, USA. **Proceedings...** San Francisco: IEEE Computer Society, 1976. p. 486–491. 2, 21, 22

PRESSMAN, R. S. **Software engineering - a practitioner's approach**. 6. ed. New York: McGraw-Hill series in computer science, 2006. 2, 6, 7, 8

ROBINSON, H. Graph theory techniques in model-based testing. In: INTERNATIONAL CONFERENCE ON TESTING COMPUTER SOFTWARE, 1999, CA, USA. **Proceedings...** Los Angeles, 1999. p. 1–13. 22

SABNANI, K. K. A protocol test generation procedure. **Computer networks and ISDN systems**, v. 15, p. 285–297, 1988. 18

SANTIAGO, V.; AMARAL, A.; VIJAYKUMAR, N.; MATTIELLO-FRANCISCO, M.; MARTINS, E.; LOPES, O. C. A practical approach for automated test case generation using statecharts. In: SECOND INTERNATIONAL WORKSHOP ON TESTING AND QUALITY ASSURANCE FOR COMPONENT-BASED SYSTEMS IN THE IEEE INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2006, Chicago, EUA. **Proceedings...** Chicago: IEEE Computer Society, 2006. p. 183–188. 4, 32

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARÃES, D.; AMARAL, A. S.; FERREIRA, E. An environment for automated test case generation from statechart based and finite state machine-based behavioral models. In: FISRT IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION AND VALIDATION, 2003, Lillehammer, Norway. **Proceedings...** Lillehammer: IEEE Computer Society, 2008. p. 63–72. 4, 31, 32, 65, 67

SANTIAGO, V. A.; CRISTIÁ, M.; VIJAYKUMAR, N. L. **Model-based test case generation using Statecharts and Z: A comparison and a combined approach**. São José dos Campos, 2010. 72 p. (INPE-16677-RPQ/850). Disponível em: <<http://urlib.net/sid.inpe.br/mtc-m19@80/2010/02.26.14.05>>. Acesso em: 14 abr. 2010. 32, 68, 82, 83, 97, 109, 110, 111

SANTIAGO, V. A.; MATTIELLO, F.; COSTA, R.; SILVA, W. P.; AMBRÓSIO, A. M. Qsee project: an experience in outsourcing software development for space. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING (SEKE'07), 14., 2006, Boston, USA. **Proceedings...** Boston: Springer-Verlag Berlin Heidelberg, 2007. p. 183–188. 4, 63, 72, 73

SIDHU, D. P.; LEUNG, T. Formal methods for protocol testing: A detailed study. **Transactions on Software Engineering**, Dept. of Comput. Sci., Maryland Univ., Baltimore, MD, v. 15, 1989. 2, 9, 14, 18, 19, 36

SILVA, W. P. **QSEE-TAS/SPAC: Execução automatizada de casos de teste para software embarcado e processamento de dados para um satélite de astrofísica do INPE**. 102 p. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais, (INPE-15662-TDI/1338). Dissertação (Mestrado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2008. Disponível em:

<<http://mtc-m18.sid.inpe.br/rep/sid.inpe.br/mtc-m18@80/2008/10.30.23.47?mirror=sid.inpe.br/mtc-m18@80/2008/03.17.15.17.24&metadataarepository=sid.inpe.br/mtc-m18@80/2008/10.30.23.47.18>>.

Acesso em: 10 nov. 2009. 63, 72

SILVEIRA, F. F. **Ferramenta de apoio ao teste de aplicações java baseada em reflexão computacional**. 132 p. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação., (38063377). Dissertação (Mestrado em Ciências Exatas e da Terra) - UFRGS, Rio Grande do Sul, 2001. Disponível em:

<<http://www.lume.ufrgs.br/bitstream/handle/10183/2283/000317273.pdf?sequence=1>>. Acesso em: 10 nov. 2009. 3, 31

SOUZA, S. R. S. **Validação de Especificações de Sistemas Reativos: Definição e Análise de Critérios de Teste**. 287 p. Tese (Doutorado) — Universidade de São Paulo, Instituto de Física de São Carlos Departamento de Física e Informática (IFSC), São Carlos, São Paulo, 2000. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/76/76132/tde-27112008-085629/>>. Acesso em: 10 nov. 2009. 25, 26, 33, 37

VIJAYKUMAR, N. L. **Statecharts: Their use in specifying and dealing with Performance Models**. 153 p. Tese (Doutorado) — Instituto Tecnológico de Aeronáutica (ITA), 1999. Disponível em: <http://www.bd.bibl.ita.br/tesesdigitais/lista_resumo.php?num_tese=000432296>. Acesso em: 12 nov. 2009. 3, 9, 11, 31, 33

W3C. **World Wide Web Consortium**. 1996. Extensible Markup Language (XML), Cambridge. Disponível em: <<http://www.w3.org/XML/>>. Acesso em: 08 dez. 2009. 39

APÊNDICE A - EXEMPLO DE PcML E MEF EM XML

Grande parte das especificações produzidos e manipulados pelas ferramentas GTSC e *Web-PerformCharts* possuem uma descrição no formato XML. A partir do modelo em *Statecharts* apresentado na Figura A.1 é apresentado nas próximas seções as respectivas representações das especificações em *PerformCharts Markup Language* (PcML) e da Máquina de Estados Finito (MEF), ambos em XML. O modelo abaixo se refere ao primeiro cenário utilizado como estudo de caso do *software* SWPDC.

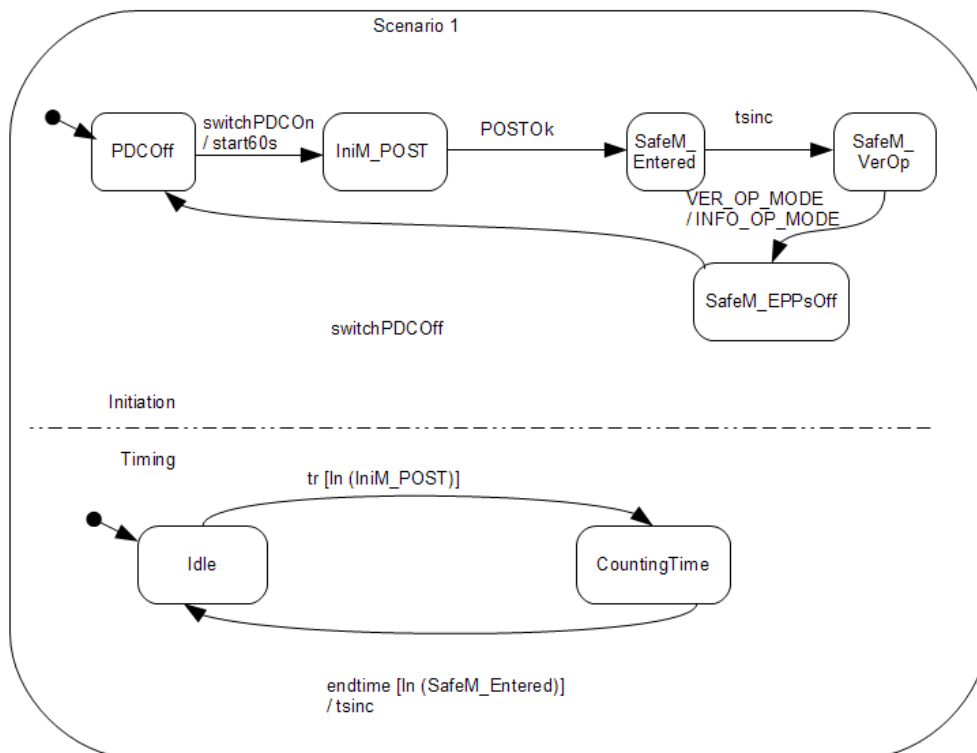


Figura A.1 - Cenário 01 usado no estudo de caso do SWPDC.
Fonte: Santiago et al. (2010)

A.1 Estrutura em XML do PcML

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<Pcml Title="Command Recognition part of the Communication Protocol"
Date="2009-08-07" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schema.xsd">

```

```

<States>
  <Root Name="Scenario1" Type="AND">
    <State Name="Initiation" Type="XOR" Default="PDCOff">
      <State Name="PDCOff" Type="BASIC"/>
      <State Name="IniM_Post" Type="BASIC"/>
      <State Name="SafeM_Entered" Type="BASIC"/>
      <State Name="SafeM_VerOp" Type="BASIC"/>
      <State Name="SafeM_EPPsOff" Type="BASIC"/>
    </State>
    <State Name="Timing" Type="XOR" Default="Idle">
      <State Name="Idle" Type="BASIC"/>
      <State Name="CoutingTime" Type="BASIC"/>
    </State>
  </Root>
</States>
<Conditions>
  <InState Name="in1" State="SafeM_Entered"/>
  <InState Name="in3" State="IniM_Post"/>
</Conditions>
<Outputs>
  <Output Name="start60s" Value="saida1"/>
  <Output Name="INFO_OP_MODE" Value="saida2"/>
</Outputs>
<Actions>
  <OutputTriggerAction Name="ota1" Output="start60s"/>
  <OutputTriggerAction Name="ota2" Output="INFO_OP_MODE"/>
  <EventTriggerAction Name="eta1" Event="tsinc"/>
</Actions>
<Events>
  <Stochastic Name="switchPDCOn" Value="5.0"/>
  <Stochastic Name="POSTOk" Value="5.0"/>
  <Stochastic Name="VER_OP_MODE" Value="5.0"/>
  <Stochastic Name="switchPDCOff" Value="5.0"/>
  <Stochastic Name="endtime" Value="5.0"/>
  <Conditioned Name="cev1" Condition="in1" Value="endtime"/>
  <TrueCondition Name="tevE1" Condition="in3"/>

```

```

    </Events>
<Transitions>
<Transition Source="PDCOff" Event="switchPDCOn" Action="ota1"
Destination="IniM_Post"/>
<Transition Source="IniM_Post" Event="POSTOk"
Destination="SafeM_Entered"/>
<Transition Source="SafeM_Entered" Event="tsinc"
Destination="SafeM_VerOp"/>
<Transition Source="SafeM_VerOp" Event="VER_OP_MODE" Action="ota2"
Destination="SafeM_EPPsOff"/>
<Transition Source="SafeM_EPPsOff" Event="switchPDCOff"
Destination="PDCOff"/>
<Transition Source="Idle" Event="tevE1" Destination="CoutingTime"/>
<Transition Source="CoutingTime" Event="cev1" Action="eta1"
Destination="Idle"/>
    </Transitions> </Pcml>

```

A.2 Estrutura em XML da MEF

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="mfeeTesteX.xsl"?>
<MFEE>
<STATES>
<STATE NAME="pDCOffIdle" TYPE="inicial"/>
<STATE NAME="iniMPostCoutingTime" TYPE="normal"/>
<STATE NAME="safeMEnteredCoutingTime" TYPE="normal"/>
<STATE NAME="safeMVerOpIdle" TYPE="normal"/>
<STATE NAME="safeMEPPsOffIdle" TYPE="normal"/>
<STATE NAME="pDCOffIdle" TYPE="final"/>
</STATES>
<EVENTS>
<EVENT VALUE="5" NAME="switchPDCOn"/>
<OUTPUT VALUE="saida1" EVENT="start60s"/>
<EVENT VALUE="5" NAME="POSTOk"/>
<EVENT VALUE="5" NAME="endtime"/>

```

```

<EVENT VALUE="5" NAME="VER_OP_MODE"/>
<OUTPUT VALUE="saida2" EVENT="INFO_OP_MODE"/>
<EVENT VALUE="5" NAME="switchPDCOff"/>
</EVENTS>
<INPUTS>
<INPUT EVENT="switchPDCOn"/>
<INPUT EVENT="POSTOk"/>
<INPUT EVENT="endtime"/>
<INPUT EVENT="VER_OP_MODE"/>
<INPUT EVENT="switchPDCOff"/>
</INPUTS>
<OUTPUTS>
<OUTPUT EVENT="start60s" VALUE="saida1"/>
<OUTPUT EVENT="INFO_OP_MODE" VALUE="saida2"/>
</OUTPUTS>
<TRANSITIONS>
<TRANSITION SOURCE="pDCOffIdle" DESTINATION="iniMPostCoutingTime">
<INPUT INTERFACE="L">switchPDCOn</INPUT>
<OUTPUT INTERFACE="L">start60s</OUTPUT>
</TRANSITION>
<TRANSITION SOURCE="iniMPostCoutingTime" DESTINATION="safeMEnteredCoutingTime">
<INPUT INTERFACE="L">POSTOk</INPUT>
<OUTPUT INTERFACE="L">null</OUTPUT>
</TRANSITION>
<TRANSITION SOURCE="safeMEnteredCoutingTime" DESTINATION="safeMVerOpIdle">
<INPUT INTERFACE="L">endtime</INPUT>
<OUTPUT INTERFACE="L">null</OUTPUT>
</TRANSITION>
<TRANSITION SOURCE="safeMVerOpIdle" DESTINATION="safeMEPPsOffIdle">
<INPUT INTERFACE="L">VER_OP_MODE</INPUT>
<OUTPUT INTERFACE="L">INFO_OP_MODE</OUTPUT>
</TRANSITION>
<TRANSITION SOURCE="safeMEPPsOffIdle" DESTINATION="pDCOffIdle">
<INPUT INTERFACE="L">switchPDCOff</INPUT>
<OUTPUT INTERFACE="L">null</OUTPUT>
</TRANSITION>

```

</TRANSITIONS></MFEE>

APÊNDICE B - GRÁFICOS DAS ANÁLISES

B.1 Gráficos de custo dos critérios de teste para o *software* APEX

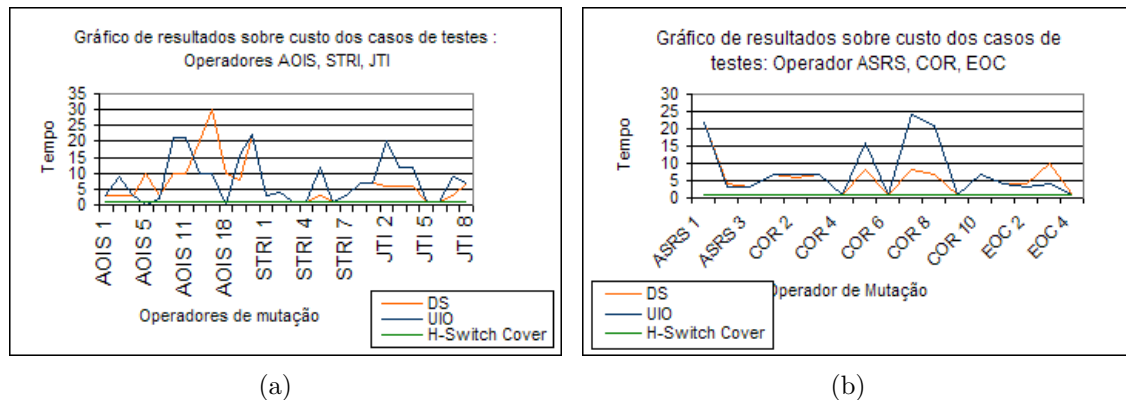


Figura B.1 - Gráficos de custo para os operadores AOIS,ATRI, JTI, ASRS, COR e EOC.

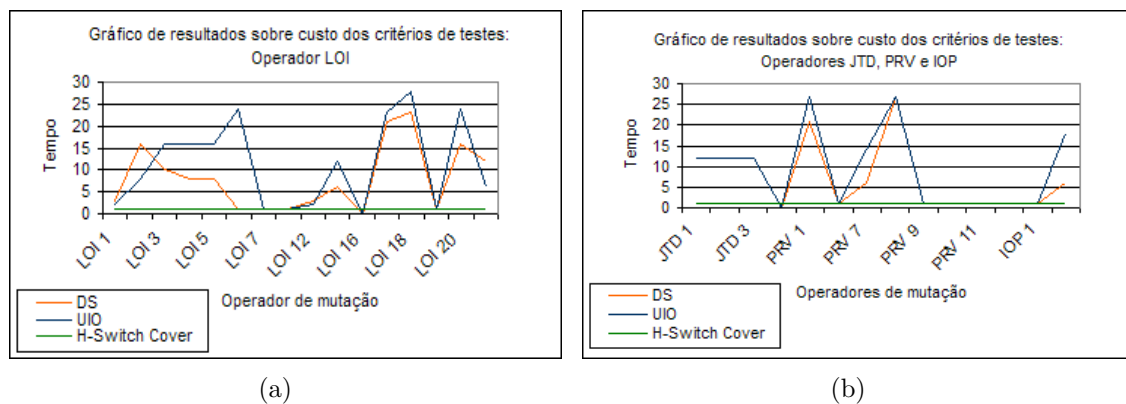


Figura B.2 - Gráficos de custo para os operadores LOI, JTD, PRV e IOP

B.2 Gráficos de custo dos critérios de teste para o *software* SWPDC

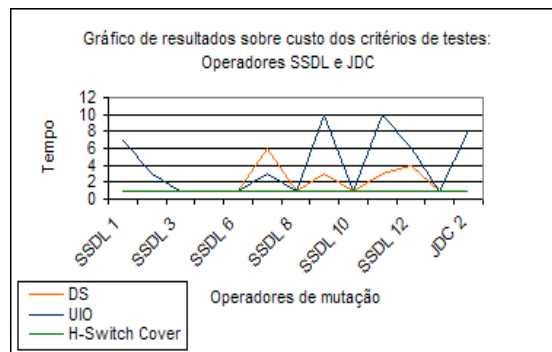
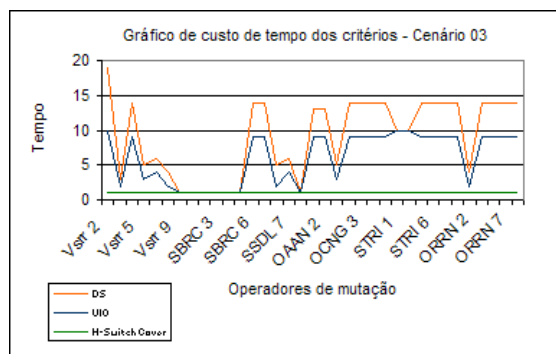
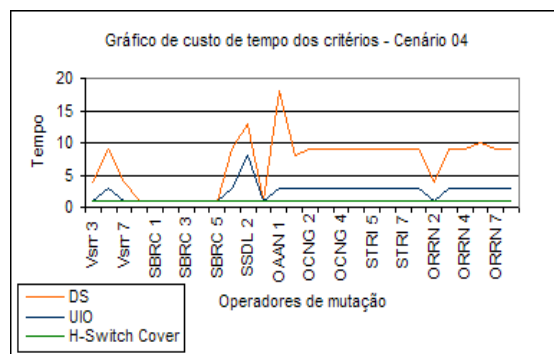


Figura B.3 - Gráfico de custo para os operadores SSDL e JDC.

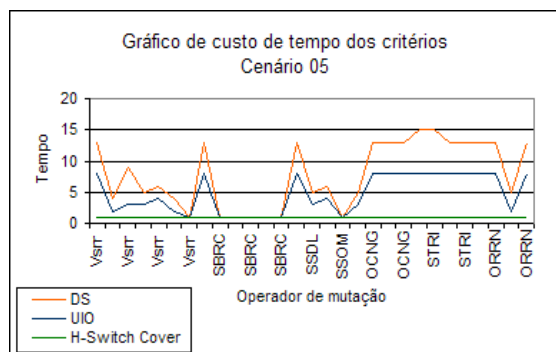


(a)

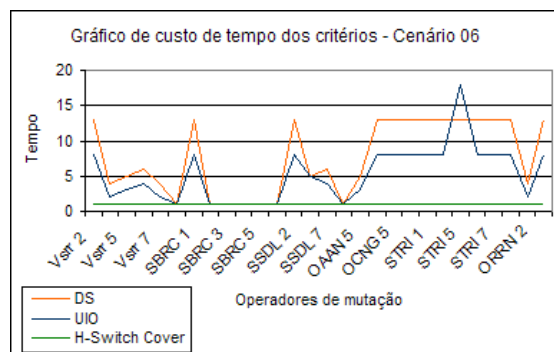


(b)

Figura B.4 - Gráfico de custo para o cenários 03 e 04.

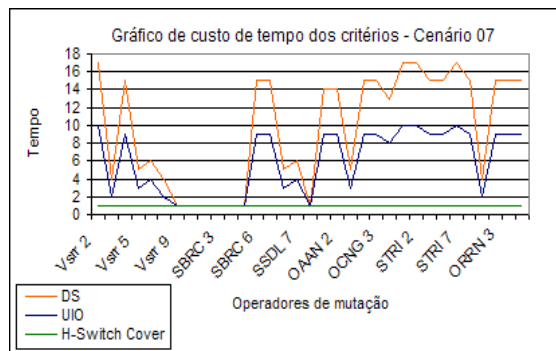


(a)

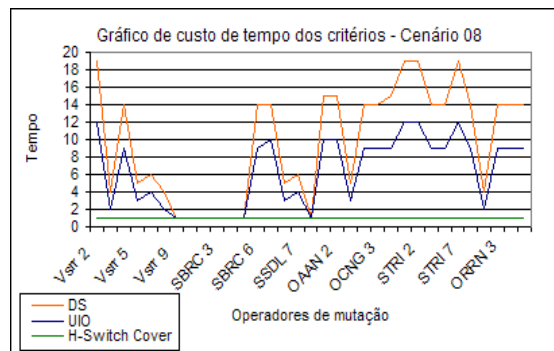


(b)

Figura B.5 - Gráfico de custo para os cenários 05 e 06.

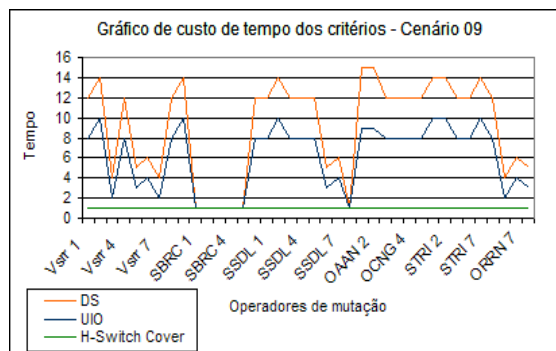


(a)

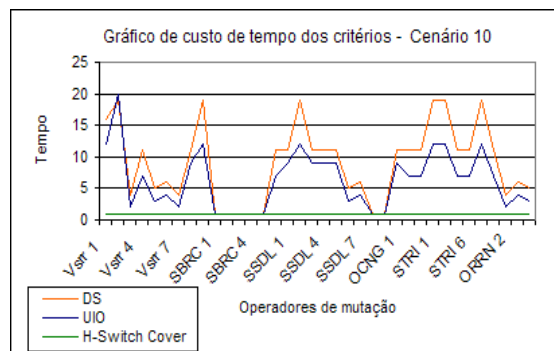


(b)

Figura B.6 - Gráfico de custo para os cenários 07 e 08.

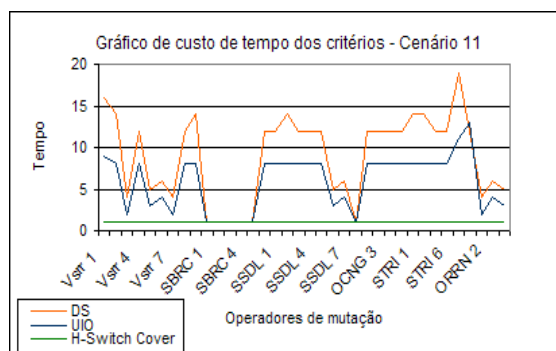


(a)

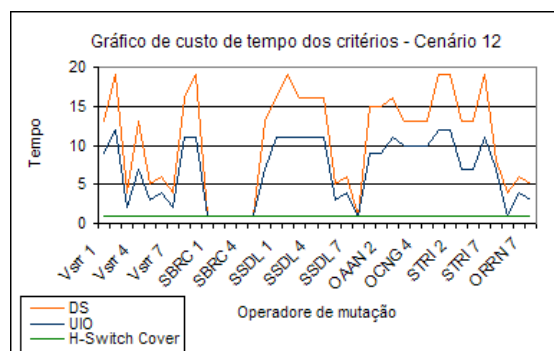


(b)

Figura B.7 - Gráfico de custo para os cenários 09 e 10

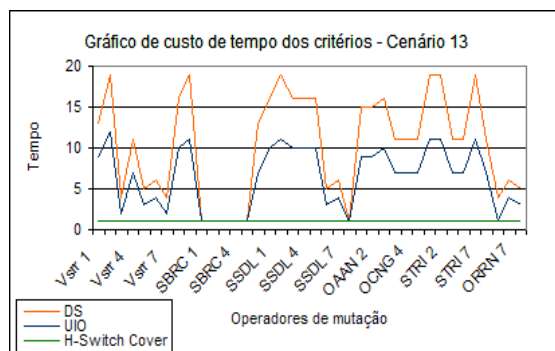


(a)

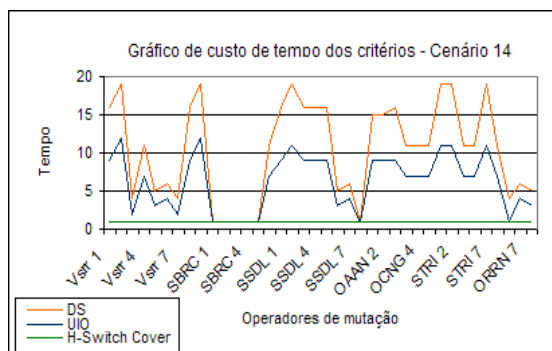


(b)

Figura B.8 - Gráfico de custo para os cenários 11 e 12

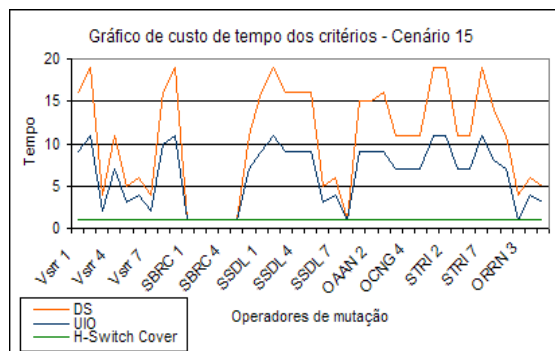


(a)

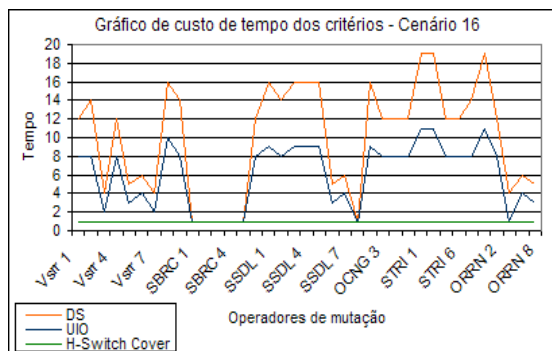


(b)

Figura B.9 - Gráfico de custo para os cenários 13 e 14

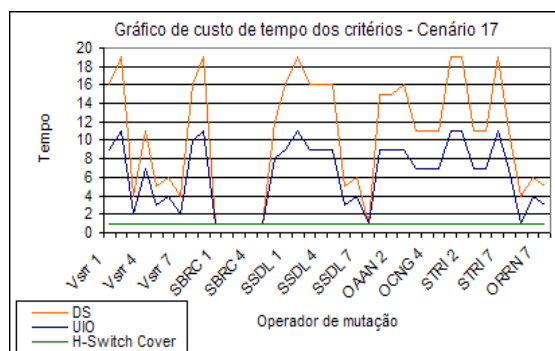


(a)

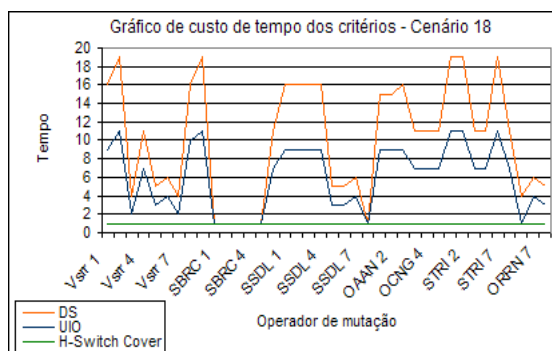


(b)

Figura B.10 - Gráfico de custo para os cenários 15 e 16

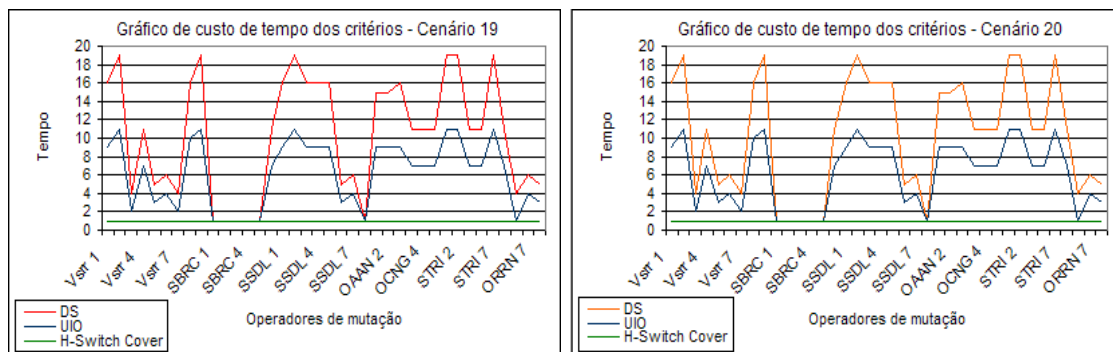


(a)



(b)

Figura B.11 - Gráfico de custo para os cenários 17 e 18



(a)

(b)

Figura B.12 - Gráfico de custo para os cenários 19 e 20

APÊNDICE C - CENÁRIOS DO *SOFTWARE* SWPDC

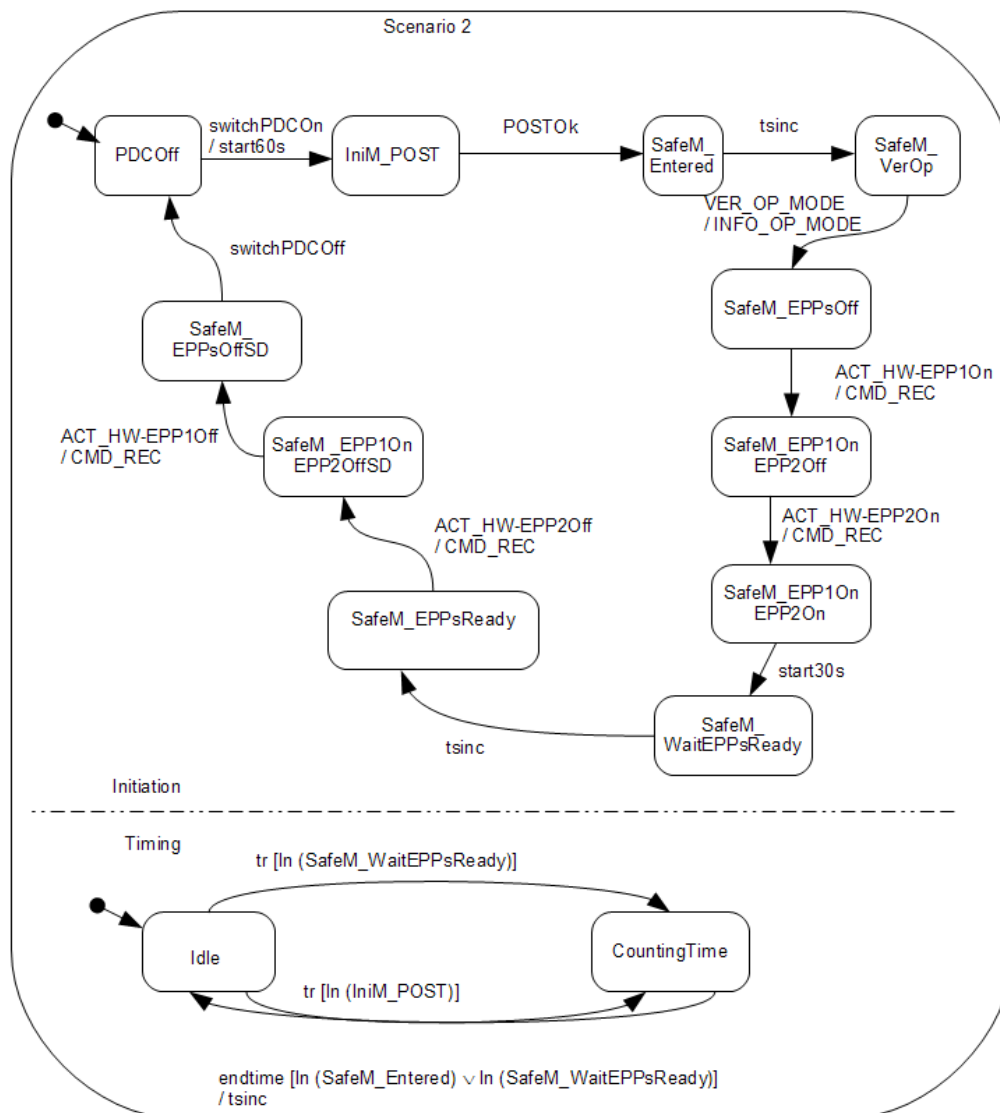


Figura C.1 - Modelo *Statecharts* do cenário 02 (dois) do *software* SWPDC.
Fonte: Santiago et al. (2010)

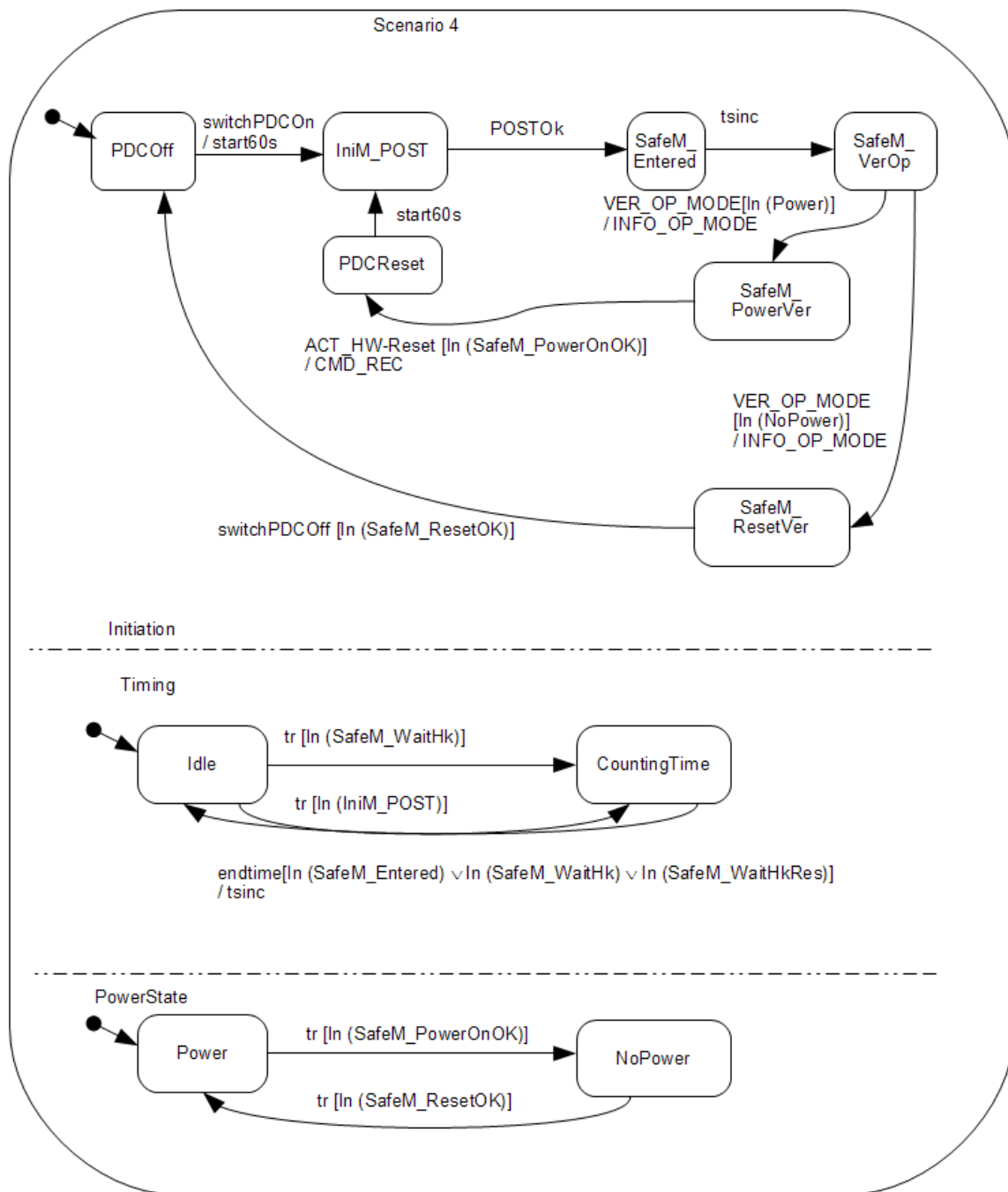


Figura C.2 - Modelo *Statecharts* do cenário 04 (quatro) do *software* SWPDC.
 Fonte: Santiago et al. (2010)

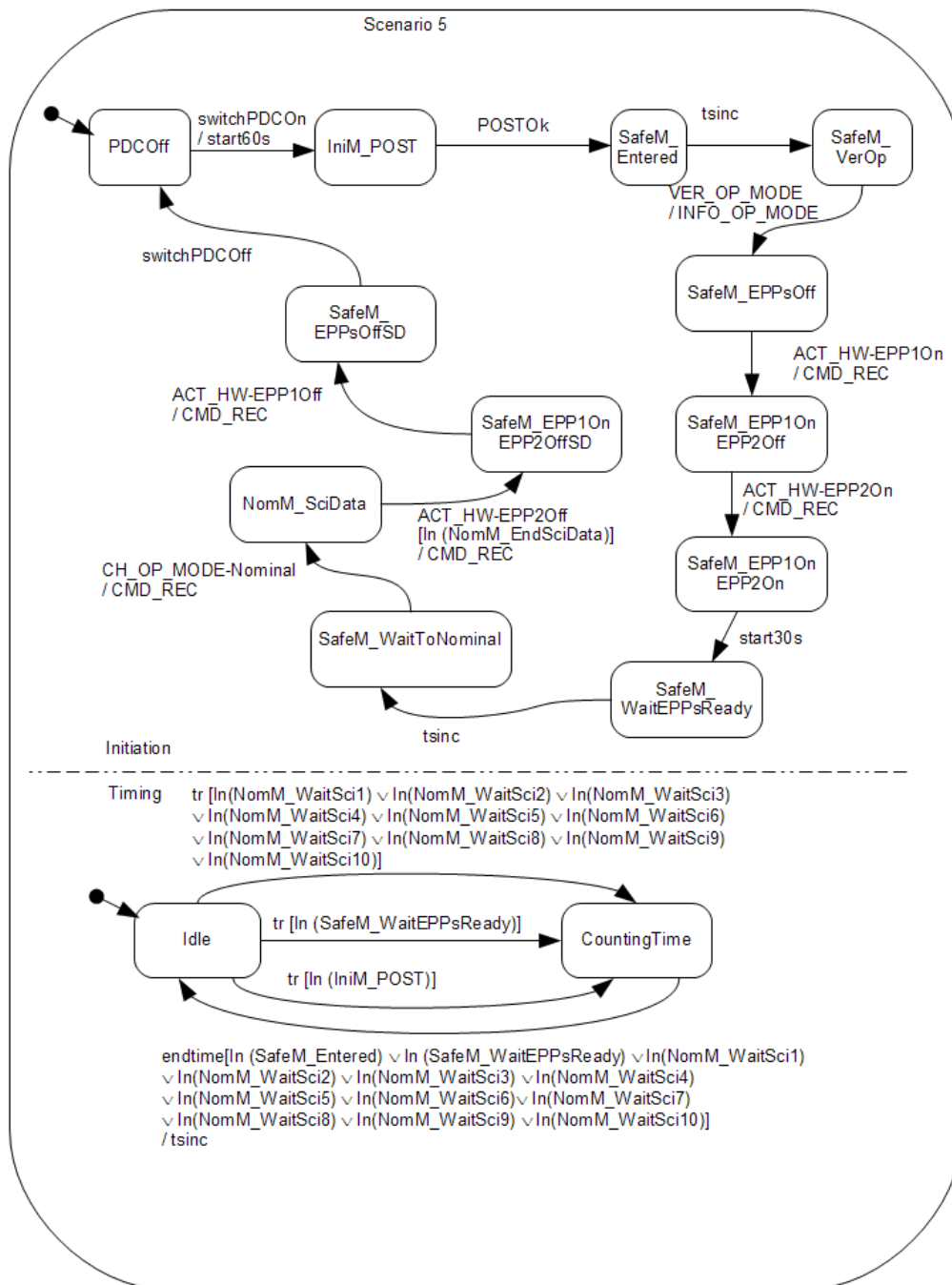


Figura C.3 - Modelo *Statecharts* do cenário 05 (cinco) do *software* SWPDC.
Fonte: Santiago et al. (2010)

PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programas de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. Aceitam-se tanto programas fonte quanto os executáveis.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.