sid.inpe.br/mtc-m19/2011/11.04.20.08-TDI

GERAÇÃO DE TESTES ESTRUTURAIS PARA APLICAÇÕES MULTITHREADS: ABORDAGEM POR STATECHARTS

Rogério Marinke

Dissertaçao de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Nandamudi Lankalapalli Vijaykumar, e Edson Luiz França Senne, aprovada em 18 de novembro de 2011.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3ANLBSE>

INPE
São José dos Campos
2011

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) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

sid.inpe.br/mtc-m19/2011/11.04.20.08-TDI

GERAÇÃO DE TESTES ESTRUTURAIS PARA APLICAÇÕES MULTITHREADS: ABORDAGEM POR STATECHARTS

Rogério Marinke

Dissertaçao de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Nandamudi Lankalapalli Vijaykumar, e Edson Luiz França Senne, aprovada em 18 de novembro de 2011.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3ANLBSE>

INPE
São José dos Campos
2011

Dados Internacionais de Catalogação na Publicação (CIP)

Marinke, Rogério.
M339g Geração de testes estruturais para aplicações multithreads: abordagem por statecharts / Rogério Marinke. – São José dos Campos : INPE, 2011.
xxii + 73 p. ; (sid.inpe.br/mtc-m19/2011/11.04.20.08-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011.
Orientadores : Drs. Nandamudi Lankalapalli Vijaykumar, e Edson Luiz França Senne.

1. teste estrutural. 2. programa concorrente. 3. diagrama de Estado. I.Título.

CDU 004.415.532.2

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

Computação Aplicada

Dr. Solon Venâncio de Carvalho

Presidente / INPE / SJCampos - SP

Dr. Edson Luiz França Senne

Orientador(a) / UNESP/GUARA / Guaratinguetá - SP

Dr. Nandamudi Lankalapalli Vijaykumar

Orientador(a) / INPE / SJCampos - SP

Dr. Alexandre Carlos Brandão Ramos

Convidado(a) / UNIFEI / Itajubá - MG

Dr. Celso Massaki Hirata

Convidado(a) / ITA / SJCampos - SP

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

Aluno (a): Rogério Marinke

São José dos Campos, 18 de novembro de 2011

"Para onde quer que fores, vai todo, leva junto teu coração".

CONFÚCIO

A meus pais Benedito(in memoriam) e Marcilia(in memoriam)

AGRADECIMENTOS

Agradeço primeiramente a Deus por permitir transformar o sonho em realidade.

Agradecimentos aos meus orientadores, Dr. Nandamudi Lankalapalli Vijaykumar e Dr. Edson Luiz França Senne pela oportunidade e por toda a confiança em mim depositada. Vocês foram mestres para uma vida inteira. Muito obrigado!

Aos professores membros da Banca Examinadora pelas relevantes contribuições.

Agradeço aos meus colegas do Laboratório Associado de Computação e Matemática Aplicada - LAC - pelos momentos agradabilíssimos que passamos juntos: Rudinei, Marlon, Luis, Sóstenes, Laurita, Teodora e Fábio. A cada um vocês fica aqui registrado o meu muito obrigado.

Agradeço ao amigo Alessandro Arantes pelo auxílio com a ferramenta WEB-PerformCharts.

Agradecimentos à Érica Ferreira pela amizade sempre presente e pelas diversas discussões referente ao meu trabalho.

Agradeço aos meus amigos Gabriel Negreira, Nelson Alves, Fabiano Sousa e Mário Shimanuki pela convivência agradável.

Agradecimentos aos colegas do projeto Harpia e em especial ao amigo Rogério Tsufa por todo o apoio na realização deste trabalho.

Agradeço também aos professores doutores Haroldo Fraga, Yano, Demísio(in memoriam), José Ernesto, Hélio Kuga, Ijar Milagre, Fabiano Sousa, Maurício, Reinaldo Rosa e Marcelo Lopes pelas inúmeras contribuições na minha vida acadêmica e pessoal.

A minha família, pelo apoio constante.

A minha amada esposa, pelo amor e compreensão incondicionais.

RESUMO

A modelagem em *Statecharts* do código fonte de softwares *multithreads* é proposta neste trabalho. O objetivo deste trabalho é explorar testes caixa branca utilizando *Statecharts*. O modelo proposto realiza a modelagem de softwares *multithreads* em *Statecharts*. Então, uma ferramenta desenvolvida no Instituto Nacional de Pesquisas Espaciais (INPE), chamada WEB-PerformCharts converterá a especificação obtida para uma Máquina de Estados Finita (MEF) plana. Isto torna possível a implementação de critérios para derivar casos de testes para testes tipo caixa branca para sistemas concorrentes. Um experimento de avaliação do modelo proposto é descrito. Os resultados obtidos comprovam que o modelo é capaz de realizar a especificação de softwares *multithreads* em *Statecharts* e derivar casos de testes que possibilitam exercitar as características de concorrência e paralelismo de sistemas concorrentes.

WHITE BOX TEST GENERATION FOR MULTITHREADS APPLICATIONS: APPROACH BY STATECHARTS

ABSTRACT

A computational model capable to perform the modeling in Statecharts of the source code of a multithread software is proposed in this work. The objective of this dissertation is to explore the white box testing using Statecharts. The proposed model performs modeling in Statecharts. Then, a developed tool at INPE, called WEB-PerformCharts converts the specification into flat FSM. With FSM, it is possible to implement methods to generate white box test cases for concurrent systems. An experiment to evaluate the proposed model is described. The results show that the model is capable of specifying multithread software in Statecharts and generate test cases that exercise the characteristics of concurrency and parallelism of concurrent systems.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

AOCS *Attitude and Orbit Control System*

BCEL *Byte Code Engineering Library*

BIRD *Bi-spectral InfraRed Detection*

DS *Distinguishing Sequence*

DTE Diagrama de Transição de Estado

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

GFC Grafo de Fluxo de Controle

GPS *Global Positioning System*

IPC *Interprocess Communication*

JSE *Java Standard Edition*

JME *Java Micro Edition*

JVM *Java Virtual Machine*

INPE Instituto Nacional de Pesquisas Espaciais

JABUTi *Java Bytecode Understanding and Testing*

MEF Máquina de Estados Finita

NASA *National Aeronautics and Space Administration*

PCFG *Parallel Control Flow Graph*

PCFGsm *Parallel Control Flow Graph for Shared Memory*

PcML *PerformCharts Markup Language*

POKE-TOOL *Potential Uses Criteria Tool for Program Testing*

RTSJ *Real Time Specification for Java*

TBM Teste Baseado em Modelo

TT *Transition Tour*

UML *Unified Modeling Language*

VVT Verificação, Validação & Teste

XML *eXtensible Markup Language*

VANT Veículo Aéreo Não Tripulado

UIO *Unique Input/Output*

SUMÁRIO

Pág.

1 INTRODUÇÃO

A garantia de qualidade de um software envolve várias atividades, como padronização da codificação, utilização de métricas, documentação, e, dentre outras está a atividade de teste. Estas atividades possuem como objetivo comum a minimização da quantidade de erros nos softwares e consequentemente diminuir os impactos financeiros e com refatoração de código, por exemplo. Segundo Pressman (2006), a atividade de teste consiste em executar um software com a intenção de encontrar erros. As exigências por softwares com maior qualidade motiva pesquisadores a desenvolverem e aperfeiçoarem metodologias e ferramentas de desenvolvimento de software a fim de atender aos requisitos de qualidade e, também, elaborar estratégias com grande capacidade de revelar erros.

O desenvolvimento da tecnologia para computadores pessoais tem propiciado que softwares estejam presentes na maior parte das atividades humanas e, em muitos casos, softwares são empregados em tarefas consideradas críticas, como aquelas que envolvem vidas humanas e altos valores financeiros. Como exemplos é possível citar:

- Os profissionais da área médica têm à disposição softwares que auxiliam na emissão de diagnósticos a pacientes;
- Engenheiros químicos utilizam softwares para conduzir experimentos científicos; e
- Na engenharia aeroespacial, softwares são empregados para simular em computador condições de vibração, comportamento em voo, impacto e controle.

A computação científica, por exemplo, é uma outra área que possui uma demanda crescente por softwares capazes de realizar o processamento de grande volumes de dados. Neste contexto, com o objetivo de minimizar o tempo de processamento dos dados os softwares têm sido projetados para fazer uso da multiprogramação, tornando mais eficiente o uso de recursos do computador. A multiprogramação permite que um mesmo software possa ser executado diversas vezes ao mesmo tempo por usuários distintos e também permite que existam no software *threads*, ou processos leves, que permitem que partes do software sejam executadas de forma concorrente(YANG; POLLOCK, 1997).

Naturalmente, o uso da multiprogramação possui características que tornam as tarefas de modelagem e desenvolvimento mais complexas, e, consequentemente aumentam as possibilidades de que erros sejam inseridos na aplicação à medida em que os desenvolvedores e analistas realizam as suas atividades. Durante a atividade de implementação o desenvolvedor deve se preocupar, por exemplo, com o aspecto da comunicação entre os processos do software e possivelmente realizar algum tipo de sincronização. Em caso de erros de sincronização, uma execução não-determinística do software pode provocar que os resultados de uma execução sejam diferentes dos resultados de outra execução utilizando os mesmos dados de entrada(YANG; POLLOCK, 1997). Segundo Almasi e Gottlieb (1994), essa interação é essencial para que o software seja considerado concorrente e apresenta-se como um desafio para os desenvolvedores que fazem uso da multiprogramação.

Assim como softwares sequenciais, softwares concorrentes também são vulneráveis a erros de programação. Adicionalmente, outros aspectos como velocidade de execução dos processos, quais processos serão concorrentes ou ainda o exato momento em que o sistema operacional realizou um chaveamento de contexto torna a atividade de teste para identificação e reprodução de erros mais complexa. Entretanto, conforme já citado, mesmo com todas estas dificuldades a multiprogramação é empregada em muitas áreas e principalmente naquelas que dispõem de ambientes com vários processadores onde é possível explorar os recursos da computação paralela. Em sistemas com apenas um processador o uso da multiprogramação pode ser justificado pela interatividade e pela sensação de velocidade fornecida ao usuário ao executar os processos do software(YANG; POLLOCK, 2003).

Consta na literatura que erros de softwares já provocaram prejuízos de toda magnitude e para muitos dos acidentes provocados, os prejuízos causados são conhecidos, como é o caso do acidente com o foguete Ariane 5, ocorrido devido a um erro de conversão de ponto flutuante de 64 *bits* para inteiro de 16 *bits* (DOWSON, 1997), que gerou um problema de imprecisão numérica. Neste contexto, a atividade de teste é considerada atualmente indispensável no processo de desenvolvimento de aplicações, visto que, ela atesta a qualidade de um software desenvolvido. Adicionalmente, à medida em que os erros revelados pelos testes são corrigidos, a confiabilidade é aprimorada e os eventuais prejuízos minimizados.

Os erros encontrados num software podem ter origem na implementação incorreta de uma funcionalidade anteriormente especificada, ou por exemplo, ter como causa uma

entrada não prevista durante a fase de modelagem e especificação, que o software não é capaz de processar adequadamente. Após a realização dos testes um software pode apresentar um comportamento adequado, entretanto, segundo Beizer (1995), não é possível garantir que ele esteja livre de erros, uma vez que o conjunto de casos de testes utilizados pode ter sido insuficiente para evidenciar erros. Adicionalmente, a quantidade de entradas possíveis num software, mesmo muito simples, é tão vasto que torna o teste exaustivo uma tarefa impraticável, por isso existe a necessidade de gerar casos de testes variados que permitam explorar o maior número possível de funcionalidades de um software, bem como os resultados fornecidos (MYERS, 1979). Um exemplo simples, é um software que realiza o cálculo da raiz quadrada de um valor informado pelo usuário, no qual, os valores que podem ser utilizados como entrada inviabilizam o teste exaustivo.

Para verificar se um software contém erros são utilizadas técnicas de teste que procuram através do efeito observado determinar a causa. No contexto desta dissertação o termo erro é utilizado para referenciar um defeito no software, ou seja, um retorno diferente do planejado, e o termo falha para indicar o comportamento incorreto do software, este provocado pelo defeito ou erro que fornece um resultado diferente do esperado de acordo com a especificação inicial (ELECTRICAL; (IEEE), 1990). Um estudo detalhado sobre causa e efeito no contexto de erros de software pode ser encontrado no trabalho realizado por Myers (1979).

Para que os possíveis erros de um software sejam descobertos antes da utilização em ambientes de produção, um conjunto de atividades denominadas de Verificação, Validação & Teste (VVT) devem ser exercitadas no software objeto de teste. O conjunto de atividades de verificação tem como objetivo assegurar que uma função específica está implementada no software. As atividades de validação visam garantir que o produto correto está sendo desenvolvido, ou seja, se foi obtido um produto conforme se esperava e que tenham sido minimizadas as chances de encontrar erros(PRESSMAN, 2006). Estas atividades são descritas em detalhes na Norma (ISO) (2003) e estão resumidas na Tabela 1.1.

Realizar a atividade de teste de software é, em muitos casos, uma tarefa complexa, pois um erro pode ser detectado em qualquer parte do software. Dessa forma, adotar uma abordagem única para realização dos testes pode ser uma tarefa difícil e inadequada. Para amenizar tal dificuldade a atividade de teste é organizada geralmente em quatro fases: 1) Planejamento de teste - apresenta o planejamento para a exe-

Tabela 1.1 - Atividades de Verificação, Validação e Testes.

Validação	Possui o objetivo de assegurar que o produto final corresponde aos requisitos do software.
Verificação	Tem como meta assegurar a consistência, completitude e corretude do produto em cada fase e entre fases consecutivas do ciclo de vida do software.
Teste	É a atividade que tem como objetivo examinar o comportamento do software através da execução.

cução do teste, identifica os itens e as funcionalidades a serem testados, as tarefas, além dos riscos associados com a atividade de teste; 2) Projeto de casos de teste - identifica os casos e os procedimentos de teste, critérios de aprovação, define dados de entrada e resultados esperados; 3) Execução dos resultados de teste - especifica os passos para a execução dos testes; 4) Avaliação dos resultados de teste - análise dos critérios admitidos e resultados observados para determinar conformidade com os requisitos (ELECTRICAL; (IEEE), 1998).

Qualquer software a ser testado poderá passar ao longo de seu processo de desenvolvimento por tais fases. No entanto, é indispensável determinar o que será testado. Com este objetivo Pressman (2006), propôs as seguintes etapas:

- Teste de Unidade- teste de cada unidade (método, classe) contida no código fonte do software;
- Teste de Integração- objetivo é explorar as interfaces entre as unidades do software, procurando identificar erros de comunicação, visando garantir o funcionamento de todos os possíveis módulos do software; e
- Teste de Sistema- teste do software com outros elementos do sistema, como hardwares, pessoas e softwares.

Para cada uma das etapas citadas é necessário a organização em fases conforme citado anteriormente.

1.1 Classificação dos Testes

A etapa de planejamento da atividade de teste fornece recursos e informações essenciais ao restante do ciclo da atividade. Uma das informações é a escolha do método

de seleção de casos de testes que será utilizado. Segundo Delamaro et al. (2007), os métodos de seleção de casos de teste podem ser classificados em:

- Especificação (Caixa preta);
- Falhas; e
- Implementação (Caixa branca).

Os testes baseados na especificação, ou testes funcionais, têm por objetivo determinar se os aspectos requeridos foram implementados. Nessa técnica os detalhes de implementação não são considerados e o software é avaliado segundo o ponto de vista do usuário, ou seja, o foco de todas as atividades está somente na funcionalidade e não no que acontece internamente no software. Desta forma, o objetivo é garantir que o mesmo está implementado de acordo com os requisitos especificados e que está em conformidade com as expectativas do usuário.

Para selecionar um conjunto de dados de entrada para posterior execução dos testes, a técnica de teste funcional utiliza alguns critérios, entre os quais estão:

- Particionamento em Classes de Equivalência - Testar um software inserindo todas as possíveis entradas pode ser algo inviável. Logo, se escolhe um representante da classe e assume-se que membros escolhidos possuem comportamento similar aos demais membros da classe. Portanto, se um membro apresentar erro ou ausência de erro supõe-se que os demais membros se comportem de maneira semelhante (MYERS, 1979; DELAMARO et al., 2007);
- Análise do Valor Limite - Critério complementar ao Particionamento em Classes de Equivalência. Esse tipo de critério escolhe valores próximos às extremidades das classes, ou seja, valores acima e abaixo dos limites possíveis (MYERS, 1979; DELAMARO et al., 2007);
- Grafo de Causa-Efeito - Um grafo é construído a partir de possíveis entradas (causas) e saídas (efeitos) do software, que possui ligações entre as ações e reações. O grafo é convertido numa tabela de decisão, da qual se derivam das colunas da tabela os casos de teste (MYERS, 1979).

Outros tipos de critérios utilizados para definir casos de testes, como *Syntax Testing* podem ser encontrados em (KIT; FINZI, 1995; VERGILIO et al., 2001).

Os testes baseados em falhas utilizam critérios que fornecem dados para compor uma medida da eficiência dos casos de testes. Segundo Zhu et al. (1997), há uma série de formas de se aplicar estes testes e estabelecer critérios, entre as quais estão:

- Introdução de erros - a informação selecionada é utilizada para inferir que certos erros não estão presentes num software. O objetivo do teste é diferenciar o comportamento do software original das novas versões criadas;
- Teste de mutação - tem como objetivo verificar a qualidade dos testes. O teste consiste em inserir falhas no código original para a derivação de mutantes. A cada mutante são aplicados os testes originais estabelecidos. Caso os testes detectem os erros artificiais assume-se que o teste detectará erros reais (SHAN; ZHU, 2006). Estudos adicionais sobre utilização de mutantes em testes pode ser encontrados em (SOUZA, 2009); e
- Teste de perturbação - é uma técnica para atestar a robustez de um software. Este tipo de teste se utiliza de três hipóteses de falhas: 1)Execução - a falha deve ser executada; 2) Infecção - a falha deve alterar os dados do estado; 3) Propagação - o estado incorreto deve ser propagado para uma variável de saída (MORELL et al., 1997).

Nos testes baseados em implementação, também chamados de testes caixa branca ou estrutural, a seleção dos testes é baseada na informação obtida a partir do código fonte do software e possui como objetivo garantir que as diversas partes do código exercitadas não violam a especificação (MYERS, 1979). A modelagem utilizada para se derivar os casos de teste pode ser feita numa representação conhecida como Grafo de Fluxo de Controle (GFC). Nesse tipo de representação o software é decomposto em um conjunto de blocos tal que, a execução de um comando do bloco realiza a execução dos comandos dos demais blocos. O GFC estabelece uma relação entre os *nós* e os blocos do grafo, onde cada *nó* e aresta do grafo representam respectivamente o comando e o caminho[1] possível do software. A partir do grafo construído é possível escolher quais componentes devem ser executados, o que de fato caracteriza

[1]Um caminho é definido por uma sequência finita de *nós*, tal que existe uma aresta de n_i para n_{i+1}.

Tabela 1.2 - Definição de *caminhos* em GFC.

Caminho simples	é definido por um caminho em que todos os *nós* são diferentes, exceto possivelmente o primeiro e o último *nó*.
Caminho livre de laço	todos os *nós* do caminho, sem exceção, são diferentes.
Caminho completo	é um caminho em que o primeiro *nó* é o *nó* de entrada e o último *nó* é o *nó* de saída do grafo.

Fonte: Delamaro et al. (2007)

o teste estrutural (DELAMARO et al., 2007). A partir do GFC podem ser definidos os caminhos que podem ser encontrados no software. A Tabela 1.2 resume os tipos de caminhos no contexto deste trabalho.

Para execução dos testes utilizando a técnica estrutural são aplicados alguns critérios de teste, a saber:

- Critério baseado em fluxo de controle - relaciona-se com aspectos ligados a execução do software, tais como, comandos ou desvios. A Tabela 1.3 lista algumas estratégias que podem ser utilizadas com este critério;
- Critério baseado em fluxo de dados - utiliza o fluxo de dados do software para obter os casos de teste; e
- Critério baseado na complexidade - esse critério baseia-se na complexidade do software para obter os casos de teste. Para determinar a complexidade é utilizada a métrica complexidade ciclomática. Usada para teste de caminho simples, o valor estabelecido pela métrica é o número de caminhos simples linearmente independentes, o que estabelece um número máximo de casos de uso que irá garantir a cobertura de que todos os nós do GFC, ou seja, que o bloco de comandos, seja alcançado ao menos uma vez durante a execução (RAPPS; WEYUKER, 1985).

Os critérios baseados no fluxo de controle, fluxo de dados e na complexidade, são utilizados pela técnica de teste caixa branca e um estudo detalhado pode ser encontrado em (PRESSMAN, 2006; WEYUKER, 1984; MALDONADO, 1991).

Tabela 1.3 - Critérios de teste estrutural.

Todos-Nós	Exige que quando executado todos os nós do GFC sejam alcançados ao menos uma vez.
Todas-Arestas	Exige que quando executado todas as arestas do GFC sejam percorridas ao menos uma vez.
Todos-Caminhos	Exige que quando executado todos os caminhos possíveis do software sejam executados.

Fonte: Delamaro et al. (2007)

No trabalho desenvolvido por Maldonado et al. (1989), os autores propuseram uma família de critérios estruturais denominada Potenciais Usos e uma ferramenta de teste de apoio aos critérios propostos. A ferramenta nomeada de *Potential Uses Criteria Tool for Program Testing* (POKE-TOOL) também permite trabalhar com outros critérios de teste estruturais baseados em fluxo de controle e fluxo de dados (CHAIM, 1991). A ferramenta foi inicialmente desenvolvida para testes de softwares escritos na linguagem C. Entretanto, alguns outros trabalhos foram desenvolvidos criando melhorias e extensões para a POKE-TOOL, como exemplos podem ser citados: no trabalho realizado por Junior (1992), foi proposta uma extensão para softwares desenvolvidos na linguagem de programação Cobol; Fonseca (1993) aprimorou os recursos da ferramenta para softwares escritos em Fortran.

A ferramenta *Java Bytecode Understanding and Testing* (JABUTi) proposta por Vicenzi et al. (2003), foi projetada com o objetivo de realizar testes em softwares escritos na linguagem de programação Java. Segundo os autores a principal vantagem da ferramenta é que ela não requer o código fonte para realizar suas atividades. Em resumo, a ferramenta possibilita que a partir de um arquivo *bytecode* (arquivo *class* binário que contém todos os dados da classe) o testador execute critérios de teste estruturais, derive o grafo de fluxo de controle.

1.2 Motivação

Dado o contexto exposto, são apresentadas a seguir as principais motivações para a realização deste trabalho:

- a importância da atividade de teste dentro do contexto de garantia de qualidade de software *multithreads*;

- o aumento de satélites artificiais que se utilizam de sistemas embarcados multiprogramados que fazem uso de protocolos de comunicação e que necessitam que o nível de confiabilidade do software seja aprimorado;

- softwares embarcados em sistemas de computação de missões espaciais utilizam-se com frequência da programação concorrente e paralela para agilizar o processamento de dados;

- Utilizar e verificar a eficácia de critérios para gerar testes para sistemas concorrentes, que tratem principalmente a sincronização e a comunicação entre processos;

- A análise de casos de testes para softwares *multithreads* pode auxiliar na escolha futura de qual critério utilizar para derivar casos de testes; e

- Possibilidade de contribuir com a melhoria das ferramentas de teste de software do INPE.

1.3 Objetivo do trabalho

O objetivo desta dissertação é realizar a automatização da modelagem em *Statecharts* de códigos fonte de softwares *multithreads* escritos na linguagem de programação Java. A partir do modelo *Statecharts* obtido é derivada a MEF plana utilizando a ferramenta WEB-PerformCharts. Então, é realizada a implementação de critérios de testes tipo caixa branca para sistemas concorrentes. Os critérios para sistemas concorrentes implementados nesta dissertação foram propostos por Souza et al. (2008). Adicionalmente, foram realizadas as análises preliminares dos casos de testes obtidos e as funcionalidades oriundas da contribuição deste trabalho serão incorporadas nas ferramentas: Geração Automática de Casos de Teste Baseada em *Statecharts* (GTSC) e WEB-PerformCharts.

1.4 Organização do texto

Neste Capítulo foi apresentado o contexto no qual este trabalho se insere e as principais motivações para a sua realização e também foram definidos os objetivos desta dissertação.

Os demais capítulos desta dissertação estão organizados da seguinte maneira:

- CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA: contempla um estudo sobre a área de programação concorrente e paralela fornecendo informações sobre os aspectos da implementação de softwares *multithreads*, memória compartilhada e comunicação entre processos. O capítulo também enumera os principais trabalhos referente a softwares *multithreads* em missões espaciais, Teste Baseado em Modelo (TBM), *Statecharts*, testes estruturais e testes para sistemas concorrentes.

- CAPÍTULO 3 - MODELAGEM DE CÓDIGO MULTITHREADS: apresenta o modelo proposto neste trabalho, discute as etapas da metodologia adotada, apresenta como foi utilizada a ferramenta WEB-PerformCharts para a geração da MEFs planas, detalha o critério proposto por (SARMANHO, 2010) para geração de sequências de casos de teste para sistemas concorrentes implementado neste trabalho. Neste capítulo também é apresentada uma análise preliminar dos casos de teste obtidos e as funcionalidades e limitações da implementação realizada.

- CAPÍTULO 4 - RESULTADOS: apresenta alguns resultados obtidos utilizando alguns softwares de exemplo. Num primeiro teste é utilizado um software sem código *multithreads*; este exemplo é útil para uma discussão mais sucinta do modelo proposto. Num outro exemplo é utilizado o código fonte do software *multithreads* para resolver o problema do produtor-consumidor com *buffer* limitado. Num terceiro e último estudo de caso é abordado um software *multithreads* proposto por Corbett et al. (2000). Os resultados e as estratégias de implementação são discutidos.

- CAPÍTULO 5 - CONSIDERAÇÕES FINAIS: apresenta as considerações e conclusões sobre o trabalho realizado e sugere alguns trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas paralelos e distribuídos

A utilização de redes de computadores pode tornar possível a melhora no tempo de resposta das requisições, uma maior confiabilidade nos softwares, uma quantidade maior de usuários que podem ser atendidos e um maior volume de processamento e etc. A popularização das redes de computadores tornou possível a programação distribuída, que consiste em executar em máquinas diferentes, softwares que permitem a troca de informações. Os softwares são executados em máquinas diferentes, interligadas por uma rede Intranet, Internet e outras redes públicas ou privadas.

Com o desenvolvimento de sistemas operacionais multi-tarefa, *multithreads* e paralelos, tornou-se possível a execução simultânea de várias partes de um mesmo software. Os softwares podem ser executados simulando paralelismo num mesmo processador, num hardware multiprocessado ou num grupo de máquinas interligadas que se comportam como uma única máquina. No entanto, para que ocorra o paralelismo real dos processos é necessária uma máquina multiprocessada ou um processador com núcleo múltiplo(TANENBAUM; STEEN, 2007).

Os sistemas paralelos são fortemente acoplados, compartilham hardware ou se comunicam através de um barramento de alta velocidade. Por outro lado, os sistemas distribuídos são fracamente acoplados. Outra característica importante é a previsibilidade. O comportamento dos sistemas paralelos é mais previsível quando comparado com os sistemas distribuídos, dado ao fato de que os sistemas distribuídos estão sujeitos às falhas da rede.

2.2 Programação Concorrente

O recurso de multiprogramação está disponível em muitas das linguagens de programação atuais, como é o caso de Java, C e *.Net Framework*. através da criação de *threads*. Uma *thread* é similar a um processo que corresponde a um conjunto de instruções que podem ser escalonadas para execução num dado processador. Entretanto, as *threads* definidas num dado software fazem uso do mesmo espaço de endereçamento que o processo principal que lhes deu origem(TANENBAUM; STEEN, 2007).

Segundo Tanenbaum e Steen (2007), utilizar *multithreading* em softwares é vantajoso

por alguns motivos, um deles é a capacidade do software executar diversas tarefas ao mesmo tempo, como, enquanto o usuário atualiza uma célula de uma planilha eletrônica, outras tarefas são disparadas e executadas sem a intervenção do usuário. Uma outra vantagem citada pelo autor é a possibilidade de explorar o paralelismo real quando o software é executado numa máquina multiprocessada.

A cooperação entre softwares é implementada por meio do mecanismo conhecido como comunicação entre processos, *Interprocess Communication* (IPC). Nos sistemas *Unix*, este mecanismo inclui fila de mensagens, memória compartilhada e *pipes*.

Quando um processo é removido do acesso ao processador, deve ser armazenada a informação referente ao seu estado operacional, a fim de que, quando for executada pelo processador novamente, o processo seja executado da posição em que se encontrava. Estes dados são conhecidos como contexto, e a operação de remover uma *thread* do processo de execução do processador e colocar outra em processamento é conhecida como troca de contexto. Tanenbaum e Steen (2007), apontam que um dos inconvenientes dos mecanismos do IPC é a requisição frequente para troca de contexto.

Para que ocorra a execução em paralelo é necessária a definição de primitivas que determinem quais trechos serão executados em paralelo e quais serão executados sequencialmente. Além disso, para garantir a ordem em que os trechos serão executados é necessário definir primitivas que permitam realizar a sincronização do software. Portanto, a comunicação citada anteriormente é responsável por garantir a troca de dados entre os processos, e a sincronização é responsável pela coordenação das operações realizadas pelos processos. Almasi e Gottlieb (1994), propõem dois tipos de sincronização: sincronização de *controle de sequência* e sincronização de *controle de acesso*.

2.3 Java em sistemas embarcados

A *Java Standard Edition* (JSE) é comumente conhecida como uma plataforma que executa software Java em servidores e computadores desktops. A JSE foi otimizada para o uso embarcado nos mais variados dispositivos, como televisores, celulares, caixas eletrônicos e sistemas embarcados em satélites.

A *Oracle* fornece basicamente duas versões para o Java Embarcado, a saber:

- JSE para dispositivos com no mínimo 32MB; e

- *Java Micro Edition* (JME) para dispositivos com no mínimo 8MB.

Muitos trabalhos têm sido propostos com o objetivo de explorar o uso da linguagem Java em sistemas embarcados. Como exemplo, é possível citar o trabalho desenvolvido por Blum et al. (2003), no qual os autores apresentam a versão Java do *framework Attitude and Orbit Control System* (AOCS) para sistemas de controle em tempo real de satélites. Segundo os autores, a versão Java do *framework* AOCS fornece um conjunto de padrões de projeto, uma arquitetura adaptável e um conjunto de componentes configuráveis que permitem a implementação de sistemas de controle para satélites. Ainda segundo os autores, uma vantagem de se utilizar Java embarcado em sistemas é que o uso da *Java Virtual Machine* (JVM) torna as diferenças entre o ambiente *desktop* e embarcado muito pequenas, o que possibilita uma vantagem no desenvolvimento do sistema.

As missões espaciais planetárias, como aquelas que exploram Marte e Saturno, empregam uma grande variedade de espaçonaves, sondas e veículos lançadores. Cada um destes sistemas depende de sistemas de controle em tempo real. Dvorak et al. (2004), propõem uma implementação comercial chamada *Real Time Specification for Java* (RTSJ); em seu trabalho, os autores também discutem os desafios de engenheiros e analistas de sistemas ao projetar, analisar e verificar sistemas de controle complexos implementados em Java.

No trabalho desenvolvido por Baker et al. (2006), os autores apresentam uma JVM para sistemas de tempo real baseada na RTSJ. Em seu trabalho, são discutidos os resultados da implementação por meio de um estudo de caso num Veículo Aéreo Não Tripulado (VANT) chamado *ScanEagle* desenvolvido em parceria com a *Boeing*. Os autores reportam no trabalho que o projeto com o VANT foi o primeiro RTSJ a ser aprovado nos testes de qualificação da *Boeing*.

O uso do Java embarcado em sistemas críticos e de tempo real também é discutido no trabalho realizado por Dautelle (2007). No trabalho em questão, o autor propõe uma biblioteca de programação chamada *Javolution* adaptada para o uso em sistemas críticos, a qual possui entre outras características a troca de dados em tempo real entre softwares escritos em Java e C/C++.

2.4 Teste baseado em modelo

Especificar comportamentos e características de um software num documento é uma tarefa que pode ser realizada através de diversas técnicas de modelagem existentes. No entanto, dependendo da técnica selecionada e do software a ser modelado, podem surgir algumas inconsistências no modelo causadas por características do software que não podem ser representadas ou por possíveis ambiguidades oriundas da dificuldade de representação, como, por exemplo, quando se utiliza linguagem natural para modelagem.

Modelo é uma forma de representar um software e a devida atenção deve ser dada para que esta representação seja feita de forma rigorosa e clara, visto que é necessário que todos que irão utilizar o modelo compreendam sem dificuldades qual é o objetivo do software. Porém, antes de se perguntar se o resultado do teste está correto, há de se saber se o modelo está correto, ou seja, tanto o software quanto o modelo precisam ser testados. Entretanto, obter o modelo para um software pode não ser uma atividade trivial.

Conforme Binder (2001), todo modelo possui quatro elementos:

- Assunto - é a ideia principal sob a qual o modelo foi criado. Nas atividades de testes, o modelo do sistema auxilia na seleção de casos de testes;
- Ponto de vista - deve estabelecer a informação necessária para produzir e avaliar casos de testes;
- Representação - uma técnica de modelagem deve ter formas para expressar o comportamento de um sistema; e
- Técnica - modelos podem ser desenvolvidos usando qualquer técnica formal, assim como *Unified Modeling Language* (UML), ou utilizar simplesmente um artefato de uso geral.

No trabalho desenvolvido por Usman et al. (2008), é realizado um estudo relativo às técnicas de verificação e consistência para modelos de aplicações orientadas a objeto utilizando a notação UML. Em seu trabalho, as técnicas de controle são analisadas e alguns parâmetros de análise e construção tem sua coerência avaliada utilizando a estratégia de monitoramento. Segundo os autores, os diagramas de classe, sequência

e *Statecharts* estão presentes na maioria das técnicas de verificação de consistência existentes e que estas são capazes de validar consistências em nível intra e inter modelos, ou seja, é possível atestar a exatidão de um modelo verificando a transição entre os estados do modelo, bem como a troca de mensagens com outros modelos.

Segundo Harel (1987), um estado captura uma situação ou um contexto de um objeto, ou seja, descreve a atividade realizada ou a espera de um evento. Transição é um relacionamento entre dois estados que é disparado por algum evento que acarreta na execução de uma ação que leva a um estado específico (veja Tabela 2.1). Segundo Binder (2001) uma MEF é uma abstração que possui estados, eventos, ações e transições. Uma MEF pode ser representada por um grafo dirigido *G=(V,E)*, onde *V*, é um conjunto de vértices, que representa o conjunto de estados, *S*; e um conjunto de arcos E=((v_i,v_n; x/y) v_i, v_n $\in$ V que representam as transições da MEF; por exemplo, cada arco e_{in} = (v_i,v_n; x/y), representa uma transição do estado v_i para v_n com entrada x e saída y. Um exemplo de uma MEF é mostrado na Figura 2.1, na qual:

- *S0,S1,S2* são os estados, sendo *S0* o estado inicial;
- *a*, *b* são os eventos/entradas; e
- *0,1* são as saídas esperadas.

Nesta MEF, como exemplo, caso seja aplicado a entrada *a* para o estado *S0*, a saída esperada será *1* e o estado final será *S1*. De forma análoga caso a entrada seja *a* para o estado *S1*, a saída esperada será *0* e o estado final será *S2*.

Os critérios de geração de casos de teste baseados em MEF, normalmente tratam o

Tabela 2.1 - Componentes de uma MEF.

Estados	Descrevem comportamentos de um software.
Transições	Indicam a mudança de estado.
Eventos	Considerados interferências no software que implicam em alteração nos comportamentos.
Ações	Resultado de um evento.

Fonte: Vijaykumar (1999)

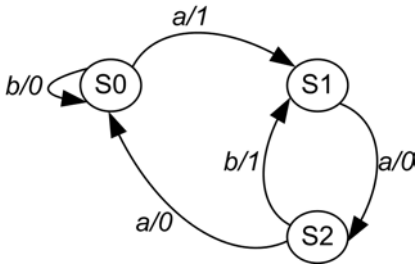


Figura 2.1 - Exemplo de uma MEF.
Fonte: Adaptada de Fuhrer (1999).

fluxo de controle de protocolos de comunicação e são baseados em técnicas de testes de transições de estados que possuem como objetivo detectar falhas de transição e de transferência(BOCHMANN; PETRENKO, 1994). Estes, geralmente se utilizam de algum algoritmo, por exemplo busca em largura, para realizar percursos em grafos. O funcionamento e propósito destes critérios, numa visão alto nível, podem ser explicados como: a partir de uma MEF é criada uma árvore de transição de estados. Nesta árvore, é aplicado um critério que irá utilizar as entradas dos estados para a derivação dos casos de testes. Neste caso, os casos de testes são sequências de fluxo extraídas do modelo e podem ser utilizados, por exemplo, para validar se a implementação corresponde aos fluxos previstos no modelo. Dentre os critérios mais comuns para a geração de sequências de testes encontrados na literatura estão: *Transition Tour* (TT), *Unique Input/Output* (UIO), *Distinguishing Sequence* (DS), *Switch-Cover*, critério *W* e *Round-Trip Path* (CHOW, 1978; MYERS, 1979; SIDHU; LEUNG, 1989; LEE; YANNAKAKIS, 1996; BINDER, 2001). A Tabela 2.2 mostra um resumo do estudo destes critérios realizado por Sidhu e Leung (1989). No estudo, os autores evidenciam que para todos os critérios analisados há necessidade de uma MEF totalmente especificada, fortemente conectada e mínima. A característica fortemente conectada garante que uma MEF possa alcançar outro estado a partir de qualquer outro estado. A característica totalmente especificada exige que todos os estados de uma MEF contenham eventos e ações. A característica mínima se refere ao fato de que não devem existir estados equivalentes ou estados desnecessários na MEF.

Segundo Binder (2001), estes critérios permitem que sejam detectadas transições, respostas e estados incorretos ou não implementados. Estes procedimentos podem ser visualizados na Figura 2.2. Um estudo referente aos testes de software utilizando

Tabela 2.2 - Análise de critérios de testes.

W	O critério requer uma MEF totalmente especificada e fortemente conectada para a geração dos casos de testes.
TT	Mostrou baixa capacidade de detecção de falhas e requer uma MEF fortemente conectada para a geração dos casos de testes.
UIO	Perfeito na detecção de falhas simples, entretanto não foi capaz de detectar falhas que requerem uma combinação com muitos estados e transições.
DS	Não foi possível obter resultados utilizando os protocolos abordados no estudo, visto que o critério requer uma máquina totalmente especificada.

Fonte: Sidhu e Leung (1989)

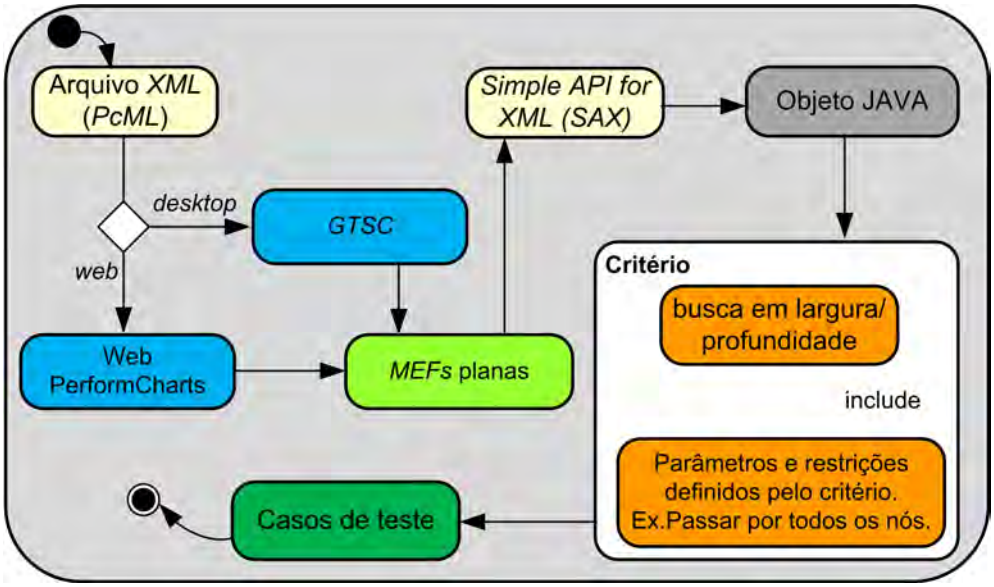


Figura 2.2 - Procedimentos para derivar casos de teste.
Fonte: Adaptada de Souza (2009).

MEFs pode ser encontrado em Lee e Yannakakis (1996).

Utilizando a abordagem TBM, o teste pode ser sistemático, focado e automatizado. Neste tipo de teste, a combinação entre entradas e estados pode ser enumerada. Visando entender o sistema e consequentemente aplicar os TBM, a equipe de teste pode criar seus próprios modelos, ou ainda, criar um novo modelo para substituir

um modelo existente. Como por exemplo, alguns softwares, como os utilizados em sistemas críticos de tempo real, protocolos de comunicação e sistemas embarcados em missões espaciais, veja Seção 2.6, exigem formas eficazes de verificação da qualidade. Estes tipos de aplicações podem ser classificadas como sistemas reativos, os quais caracterizam-se por interagirem fortemente com o ambiente em que estão inseridos, de forma que uma mesma entrada pode provocar reações distintas dependendo do contexto ou estado no qual ocorre. Para representar este tipo de sistema pode ser realizada uma modelagem através do Diagrama de Transição de Estado (DTE), o qual representa uma MEF. Essas técnicas são capazes de descrever o comportamento de um sistema, mostrar os estados possíveis em que um objeto pode estar e como o estado do objeto muda ao receber eventos. No entanto, não permitem hierarquia, modularidade, profundidade e concorrência, ou seja, são planas. Desta forma, à medida em que a complexidade do sistema cresce linearmente, o número de estados e transições cresce exponencialmente, gerando diagramas grandes e confusos.

Segundo Harel (1987), não é possível realizar a modelagem da concorrência de softwares em MEF, ou seja, somente um estado pode estar ativo. Adicionalmente, segundo Binder (2001), a escalabilidade de modelos criados utilizando MEF também é limitada. De acordo com o autor, o entendimento e a representação se tornam complicados em diagramas que possuem mais de 20 estados.

Devido a estas dificuldades de representação, foram propostas algumas técnicas para aprimoramento, como *Stelle* (BUDKOWSKI; DEMBINSKI, 1987), *Model Checking* (CLARKE et al., 1999), *Assume-Guarantee Testing* (BLUNDELL et al., 2005) e *Statecharts* (HAREL, 1987). Os *Statecharts* serão o enfoque deste trabalho de dissertação.

2.5 *Statecharts* e trabalhos correlacionados

A técnica gráfica de *Statecharts* é uma extensão à MEF, permitindo a representação da composição hierárquica de estados(profundidade), atividades paralelas(ortogonalidade), sincronismo e interdependência através de comunicação entre componentes *broadcast* (HAREL; POLITI, 1998). *Statecharts* são constituídos basicamente de estados, transições e eventos.

As transições podem possuir condições; neste caso é considerado que a transição é protegida por uma condição, em outras palavras, uma transição é realizada somente

se a condição for verdadeira e se o evento associado estiver habilitado, ou seja, mesmo que um evento esteja habilitado a transição irá ocorrer somente se a condição associada for satisfeita. O teste na condição da transição é feito toda vez que o software estiver no estado de origem da transição. Em algumas situações, é possível existirem ações nas transições associadas a eventos, que algumas vezes modificam os valores de condições e itens de dados, iniciam ou terminam atividades e disparam uma transição num outro componente paralelo (AMARAL, 2005).

Em *Statecharts*, os eventos estão associados à ocorrência de uma alteração no software, que podem ser de caráter de geração de sinais, valor de uma variável ou uma transição de um estado para outro(AMARAL, 2005; HAREL; POLITI, 1998). A classificação dos eventos ocorre em função da sua origem, que podem ser: externos e internos. Eventos externos são provocados por um evento físico, enquanto que os eventos internos são disparados pela própria aplicação (DELAMARO et al., 2007). A Figura 2.3 exibe um diagrama *Statecharts* com estados, transições e eventos.

A representação hierárquica permitida em *Statecharts* é realizada através de *superestados* ou *subestados*. Essa representação é útil quando, por exemplo, existem transições oriundas de estados diferentes, porém com o mesmo destino, pode-se encapsular esses estados num *superestado* (AMARAL, 2005). É possível visualizar este encapsulamento na Figura 2.4.

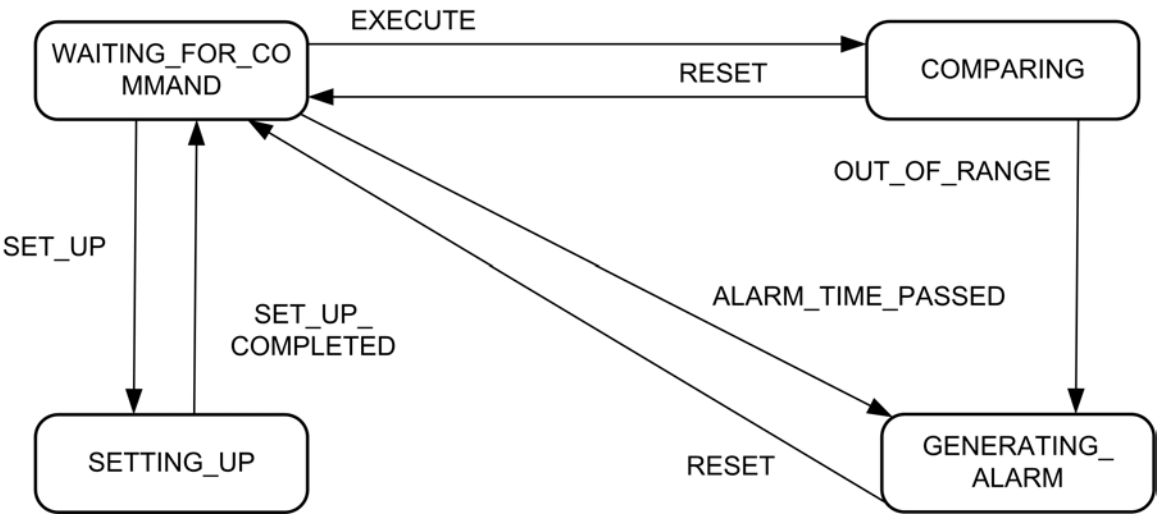


Figura 2.3 - Estados, transições e eventos.
Fonte: Adaptada de Harel e Politi (1998).

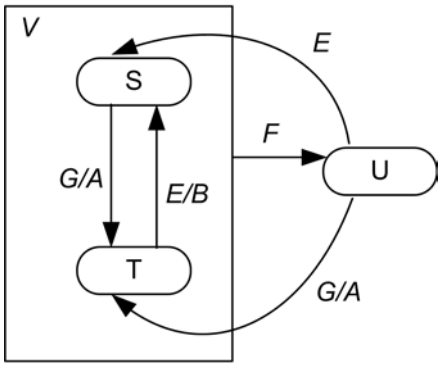


Figura 2.4 - Encapsulamento dos estados S e T num superestado V.
Fonte: Adaptada de Harel e Politi (1998).

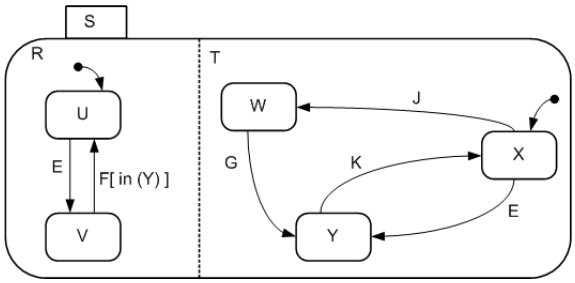


Figura 2.5 - *Statecharts* - Comunicação entre estados.
Fonte: Adaptada de Harel e Politi (1998).

Algumas funcionalidades contidas em softwares complexos recorrem muitas vezes a características de execução de funcionalidades em paralelo e que fazem uso de comunicação entre os seus processos. A representação destas características em *Statecharts* pode ser visualizada na Figura 2.5; a figura mostra um estado S, superestado, composto por dois outros estados R e T paralelos. O estado R contém os estados U e V e estes possuem as transições E e F. O estado T é composto pelos estados W, X e Y os quais são vinculados através das transições E, G, J e K. Os estados iniciais definidos em R e T são U e X respectivamente.

Caso ocorra a entrada E em (U,X), o sistema simultaneamente transfere para (V,Y), ou seja, um simples evento dispara duas ações simultâneas, caracterizando uma concorrência sincronizada.

A utilização de *Statecharts* ao invés de MEF, pode facilitar a leitura e entendimento

das ações, entradas e saídas contidas num software. Binder (2001), com o intuito de evidenciar as diferenças entre essas técnicas, propôs as modelagens em MEF e *Statecharts* para um sistema de semáforo. Os diagramas são exibidos nas Figuras 2.6 e 2.7, respectivamente.

O comportamento descrito para o sistema em ambos os modelos é o mesmo, no entanto, é possível notar que o modelo em *Statecharts* simplifica a notação sem perder informação e formalismo. Algumas destas simplificações são listadas a seguir:

- As transições (*FlashRedOn, Fault, LiteOff*, são comuns para os estados *Red, Yellow e Green*, por este motivo estes estados foram agrupados no diagrama *Statecharts* num superestado chamado *Cycling*;
- O superestado *On* representa que o sistema pode estar tanto em *Cycling* como em *FlashingRed*;
- Dependendo do estado que se encontra ativo, *FlashRedOn* dispara a transição para um dos estados *Red, Yellow e Green*;
- *Reset* dispara a transição *Off-On*, somente se a condição *NoFaults* for ver-

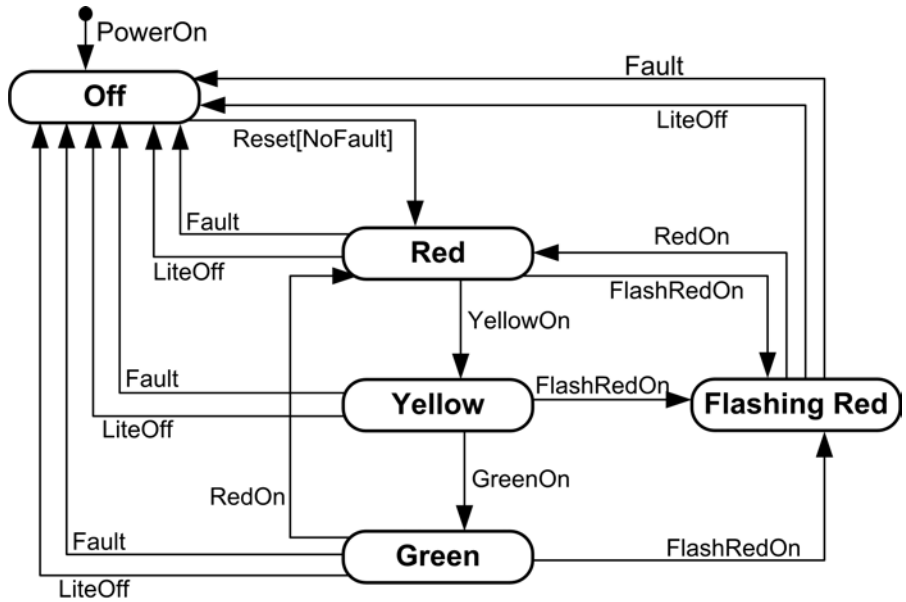


Figura 2.6 - Diagrama de Estado para sistema *Traffic Light*.
Fonte: Adaptada de Binder (2001).

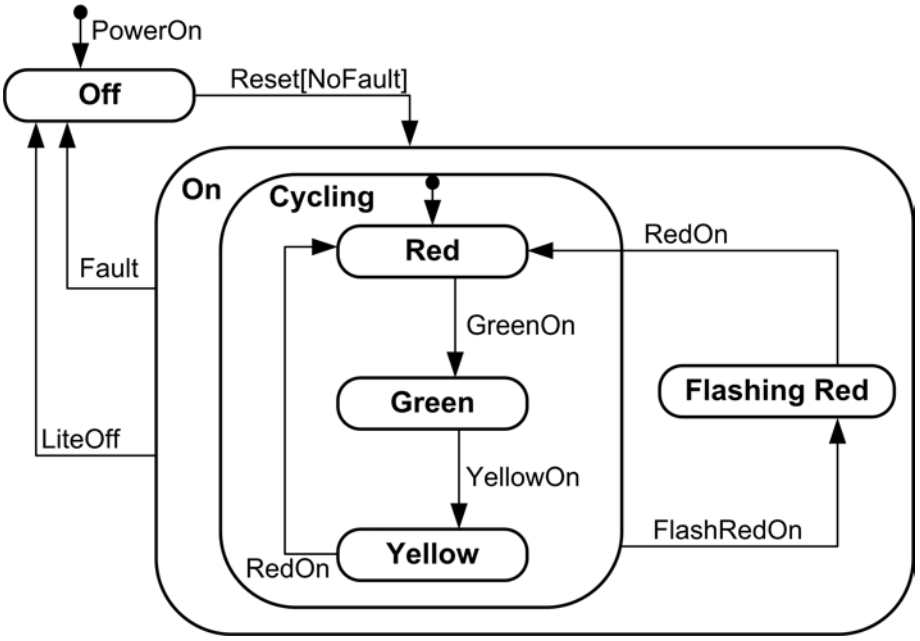


Figura 2.7 - *Statecharts* para sistema *Traffic Light*.
Fonte: Adaptada de Binder (2001).

dadeira;

- A transição sem rótulo dentro de *Cycling* indica que *Red* é o estado *default* de *Cycling*; e

- *Red* está marcado como estado *default* nos superestados *On* e *Cycling*, então *Reset* dispara a transição *Off-RedOn*.

No trabalho desenvolvido por Vijaykumar (1999), é proposta a ferramenta PerformCharts, a qual permite a especificação de um sistema reativo em *Statecharts*. Nesta primeira versão, a especificação *Statecharts* é convertida para uma Cadeia de Markov para avaliação de desempenho. Com a Cadeia de Markov obtém-se as probabilidades limite, as quais representam a porcentagem de tempo ocupado por cada estado; bases para cálculo de outras medidas de desempenho.

O trabalho proposto por Amaral (2005), utiliza uma metodologia que permite especificar modelos *Statecharts* em *eXtensible Markup Language* (XML). A linguagem denominada *PerformCharts Markup Language* (PcML) desenvolvida em seu trabalho é capaz de representar o *Statecharts* de um sistema reativo. PcML é uma

representação textual em XML de um diagrama *Statecharts*. A metodologia consiste em converter a especificação *Statecharts* para uma MEF estendida. A partir daí, faz-se uso da ferramenta *Condado* para a geração automática de casos de teste. Esta abordagem permite tratar os aspectos controle (MEF) e dados (MEF estendida) de maneira unificada. Um estudo detalhado sobre a ferramenta *Condado* pode ser encontrado em Martins et al. (2000).

No trabalho realizado por Santiago et al. (2008), é proposto o ambiente denominado GTSC, o qual permite ao projetista de teste modelar o comportamento de um software utilizando MEF ou *Statecharts* com o objetivo de gerar automaticamente sequências de casos de teste utilizando alguns critérios para MEF, como *Swith cover*, DS e UIO. O núcleo principal do ambiente GTSC é a ferramenta PerformCharts. A Figura 2.8 exibe a arquitetura da versão atual do GTSC.

Arantes et al. (2008), apresentam em seu trabalho a ferramenta WEB-PerformCharts, a qual permite a execução através da Web. Em seu trabalho, o critério para geração de casos de testes chamado *Transition Tour* foi implementado e integrado à ferramenta.

No trabalho desenvolvido por Souza (2009), foi realizada a implementação dos critérios *Switch-Cover*, DS e UIO. As implementações foram integradas às ferramentas GTSC e WEB-PerformCharts, desta forma o uso da ferramenta *Condado* não é mais necessário. No trabalho também foi realizada uma investigação preliminar do custo

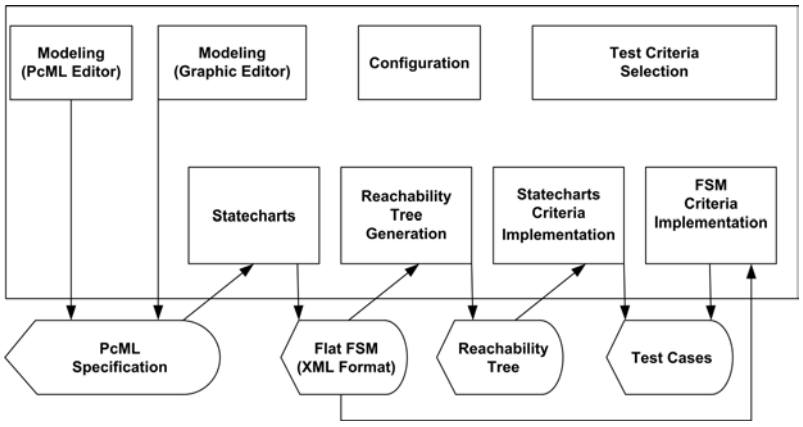


Figura 2.8 - Arquitetura do ambiente GTSC.
Fonte: Adaptada de Santiago et al. (2008).

e da eficiência desses critérios com base em avaliações empíricas mediante o uso de alguns estudos de caso.

As ferramentas citadas anteriormente, GTSC e WEB-PerformCharts são utilizadas pelo grupo de pesquisa de testes de software do INPE e são atualizadas à medida em que as pesquisas evoluem.

Em (SOUZA et al., 2000), é proposto um conjunto de critérios de testes de cobertura para especificações baseadas em *Statecharts*. Os critérios propostos utilizam uma árvore de alcançabilidade e têm como objetivo complementar as abordagens de simulação e análise de propriedades utilizadas para validação e teste de especificações *Statecharts*. Alguns destes critérios foram implementados e integrados à ferramenta GTSC.

2.6 Sistemas concorrentes em missões espaciais

Os softwares utilizados nas missões espaciais da *National Aeronautics and Space Administration* (NASA), por exemplo, estão cada vez maiores em número de linhas e mais complexos conforme as novas exigências vão surgindo. A Tabela 2.3 exibe algumas missões espaciais e os respectivos números de linhas dos softwares utilizados (THOMPSON et al., 2010).

Além da grande quantidade eminente de linhas, na maioria dos casos, esses softwares possuem trechos de código *multithreads*. Muitas vezes, estes softwares são tão complexos que um único processador seria incapaz de realizar os cálculos necessários para finalizar a execução dentro de um determinado intervalo de tempo. Neste contexto, muitos trabalhos têm sido realizados na tentativa de investigar e propor novas abordagens que explorem as características de sistemas concorrentes (SUGDEN et al., 2001; EZEKIEL et al., 2007; JAMES, 2007; CURTIS, 2009; THOMPSON et al., 2010).

Tabela 2.3 - Softwares utilizados em missões espaciais

Missão *Voyager*	3 mil linhas.
Missão *Cassini*	30 mil linhas.
Missão *Mars Path Finder*	160 mil linhas.
Space Shuttle	500 mil linhas.
Estação Espacial Internacional	3 milhões de linhas.
Constellation Project	50 milhões de linhas.

Gill et al. (2000) desenvolveram um software *multithreads* embarcado para o satélite alemão *Bi-spectral InfraRed Detection* (BIRD). O software fornece informações em tempo-real sobre a determinação da órbita baseado em *Global Positioning System* (GPS). Como parte adicional dos objetivos tecnológicos da missão BIRD, Gill e Montenbruck (2001) implementaram um software *multithreads* para classificação de dados utilizando uma rede neural artificial. O software classifica fumaça e nuvens(BRIEB et al., 2001).

No trabalho realizado por Surka et al. (2001), é proposta uma arquitetura de software embarcado baseada em *agente* para sistemas distribuídos autônomos. *Agentes* são utilizados para implementar todas as funcionalidades do software e se comunicarem através de mensagens simplificadas em linguagem natural. Numa demonstração preliminar simulada em laboratório, o software foi utilizado para realizar uma reconfiguração de *cluster* com três satélites.

As taxas de telemetria de veículos espaciais têm aumentado na última década e se apresentado como um problema para o processamento em tempo real por instalações terrestres. O trabalho proposto por Kizhner et al. (2002) propõe uma solução para o problema dentro do contexto de um aplicativo de processamento de imagem. Os resultados apresentados incluem uma biblioteca de funções e aplicações espaciais de processamento de imagem usando a tecnologia de computação embarcada, incluindo os resultados de desempenho e comparações com os sistemas existentes.

Uma validação sobre recebimento e processamento de sinais de rádio transmitidos por satélite é realizado em Vejrazka et al. (2003). Em seu trabalho, são analisados o desempenho do receptor e o software *multithreads* utilizados numa das fases da missão *GALILEO*. Segundo os autores, os resultados obtidos validam uma correta concepção do receptor e do software.

Uma ferramenta computacional para analisar algoritmos e detectar possíveis condições de *deadlock* em software *multithreads* é apresentado em Bensalem et al. (2005). No trabalho do citado autor, foram realizados experimentos com uma aplicação utilizada num Veículo Explorador desenvolvido pela NASA chamado *K9* e bons resultados foram alcançados segundo a análise dos autores.

Um software para computadores de alto desempenho utilizado pela NASA para simular e analisar sistemas ópticos controlados foi desenvolvido no trabalho de Lou

et al. (2006). Os testes na versão paralelizada demonstraram que o desempenho incorporado, ao se utilizar *multithreads*, foi proporcional ao número de processadores utilizados.

2.7 Teste para Sistemas Concorrentes

Howden (1976) introduz dois tipos básicos de erros em softwares, quais sejam: erros de computação e erros de domínio. Segundo o autor, erros de computação ocorrem quando o resultado de uma computação para uma entrada do domínio do software é diferente do esperado. O segundo tipo de erro ocorre quando um caminho diferente do esperado é exercitado/percorrido.

Os erros que podem ser encontrados em softwares que possuem as características de paralelismo e concorrência são definidos como: erros de sincronização e erros de computação. Um estudo detalhado sobre estes tipos de erros pode ser encontrado em Tai (1989).

No trabalho realizado por Taylor et al. (1992), os autores mostram o conceito de critérios de teste estrutural para softwares concorrentes e propõem uma hierarquia de apoio à técnica de teste estrutural. Entre as propostas estão: critério de cobertura para estados concorrentes, cobertura de transição de estados e cobertura de sincronização.

No trabalho realizado por Bogdan et al. (1994), é realizado um estudo sobre os erros em softwares concorrentes e softwares sequenciais. No trabalho, os autores evidenciam que nos softwares sequenciais há basicamente duas classificações de erros. Uma está relacionada à garantia de qualidade pertinente ao desenvolvimento do ciclo de vida do software e a outra classificação está vinculada à lógica do software. No trabalho, os autores ilustram suas ideias por meio de exemplos, quando estas classificações se aplicam a softwares concorrentes.

Testes de software orientado a objeto utilizando MEF são discutidos no trabalho realizado por Hong et al. (1995). No trabalho em questão, é proposta a realização da modelagem dos membros internos de uma classe em MEFs, e a partir deste ponto, as máquinas obtidas são convertidas em GFC e gerados os testes utilizando as técnicas de fluxo de dados.

Testes baseados em autômatos são realizados no trabalho desenvolvido por Seo et al.

(2001) para executar teste em softwares *multithreads* escritos na linguagem Java. Na pesquisa citada, são desenvolvidos algoritmos que derivam os autômatos conforme os casos de testes selecionados e algoritmos que conduzem a execução dos testes.

Bader et al. (1998), descrevem uma abordagem para realizar testes em sistemas concorrentes dentro de um contexto de software orientado a objeto. Os autores utilizam *Statecharts* para especificar o software. Uma extensão do critério W é utilizado para derivar os casos de testes.

Na pesquisa realizada por Chung et al. (1999), os autores apresentam uma abordagem na qual é possível realizar TBM em sistemas concorrentes utilizando *Message Sequence Charts*.

Stoller (2000), explora padrões de sincronização da linguagem Java para propor uma redução na quantidade de dados e consequentemente estados em softwares *multithreads*.

No trabalho realizado por Vergilio et al. (2005), os autores apresentam uma família de critérios de teste para softwares concorrentes que utilizam o paradigma de passagem de mensagem. O modelo de teste proposto considera aspectos de sincronização e comunicação que são independentes da linguagem de programação.

Luo et al. (2006), apresentam um método para geração automática de casos de teste para sistemas concorrentes e protocolos de comunicação. Os autores propõem uma modelagem utilizando MEF não determinística e definem uma relação de conformidade, chamada de traço de equivalência, que serve como um guia para realização da geração automática dos casos de teste.

O trabalho desenvolvido por Seo et al. (2006), apresenta uma abordagem para TBM para gerar casos de testes para sistemas concorrentes especificados em *Statecharts*. Os autores selecionam inicialmente alguns casos de testes que representam as características de concorrência e paralelismo do software e, em seguida, derivam autômatos capazes de gerar a mesma sequência dos casos de teste selecionados anteriormente e estes conduzem o restante do experimento. Os resultados e demais discussões podem ser encontradas em Seo et al. (2006).

A extração de MEF a partir do código fonte de uma aplicação é proposto em (CORBETT et al., 2000). Segundo os autores, a ferramenta desenvolvida, **Bandera**, recebe

o código de um software e deriva o modelo em MEF. No trabalho, o autor aborda como os resultados podem ser utilizados para a verificação e correção do modelo de um software Java.

Sarmanho (2010), propõe um modelo de teste para softwares concorrentes que utilizam memória compartilhada. O trabalho trata a sincronização e a comunicação de *threads* de forma independente e entre outras características apresenta um método baseado em *timestamps* (ou sequência de caracteres para rotular data e/ou hora) para determinar as comunicações exercitadas numa dada execução do software.

2.8 Considerações finais

A demanda por softwares capazes de realizar o processamento de uma grande quantidade de dados em intervalos de tempo cada vez menores tem potencializado o uso da programação concorrente em muitos sistemas espaciais, por exemplo. Muitas abordagens para tratar os recursos da multiprogramação têm sido propostos, o que tem possibilitado um aprimoramento nas técnicas, ferramentas, hardwares e metodologias utilizadas no desenvolvimento de software concorrente.

No contexto de TBM, percebe-se uma necessidade de ferramentas de teste de software que auxiliem e permitam que a execução de testes para softwares *multithreads* sejam realizadas de forma automatizada. Embora ferramentas como *ValidPThread*, por exemplo, disponham de critérios estruturais adaptados para softwares *multithreads*, ela não realiza a geração automática dos casos de testes, sendo estes criados manualmente, e que por sua vez limita ainda mais as possibilidades de variação dos casos de testes. Neste mesmo caminho, as ferramentas JABUTi e POKE-TOOL, por exemplo, embora potencializem o uso de critérios para testes estruturais, não dispõem até o momento de critérios de teste para softwares concorrentes.

Quanto maior for a quantidade de casos de testes, maiores serão as chances de se evidenciar um erro num software. Softwares concorrentes podem possuir um número não-determinístico de caminhos devido a quantidade de estados, entradas e saídas. Disponibilizar uma ferramenta que possibilite a geração automática de casos de testes utilizando critérios estruturais adaptados para sistemas concorrentes é uma necessidade que requer atenção. A Tabela 2.4 exibe as ferramentas citadas anteriormente e algumas das suas características.

Adicionalmente, muitos pesquisadores têm aplicado TBM para derivar casos de teste

Tabela 2.4 - Ferramentas de apoio ao teste estrutural.

Ferrramenta	Critérios estruturais	Critérios *multithreads*	Geração automática de casos de testes
ValidPThread		χ	
JABUTi	χ		χ
POKE-TOOL	χ		χ
GTSC	χ		χ
WEB-PerformCharts	χ		χ

e realizar diversas análises decorrentes da atividade de teste. A comunidade científica geralmente considera o TBM como um tipo de teste caixa preta. Entretanto, segundo a classificação proposta por Utting et al. (2006), o TBM pode ser considerado um teste estrutural (caixa branca) quando critérios de cobertura são adaptados para trabalhar com modelos.

Este trabalho apresenta um modelo que permite realizar a geração de casos de testes para softwares *multithreads* utilizando diagramas especificados em *Statecharts*. No próximo capítulo são discutidos os detalhes da abordagem proposta para a realização desta dissertação de mestrado.

3 MODELAGEM DE CÓDIGO MULTITHREADS

Este capítulo apresenta a abordagem desta dissertação de mestrado. Inicialmente, são realizadas algumas considerações para delinear o escopo do modelo proposto. Posteriormente, são apresentados alguns critérios de teste para sistemas concorrentes e finalmente os detalhes da metodologia e da implementação são discutidos.

Neste trabalho de mestrado, para representar os grafos de fluxo de controle dos softwares concorrentes é utilizado o modelo *Parallel Control Flow Graph* (PCFG) proposto no trabalho realizado por Vergilio et al. (2005). Em seu trabalho, os autores propõem o modelo PCFG para aplicar critérios de teste em softwares concorrentes com passagem de mensagem. Algumas das considerações realizadas pelos autores a respeito do PCFG são apresentadas a seguir e são utilizadas para a realização desta dissertação.

- o modelo considera que o software concorrente $SC = \{p^0, p^1, ...p^{n-1}\}$ é um conjunto de n processos que executam concorrentemente;
- o número de processos n é fixo e conhecido em tempo de compilação e são criados apenas uma vez durante a inicialização do software concorrente;
- a comunicação/sincronização dos processos é realizada por meio das arestas do GFC do PCFG que é construído para representar o software concorrente;
- São utilizadas primitivas *send* e *receive* para realizar a comunicação/sincronização. A comunicação pode ser realizada das seguintes formas: um processo envia uma mensagem para outro específico (ponto-a-ponto); um processo envia uma mensagem para um grupo de processos ou todos utilizando diversos *send/receive* básicos; e
- no PCFG o i-ésimo nó do p-ésimo processo é representado por N_i^p. O conjunto N denota o conjunto de nós do PCFG e é dividido em dois subconjuntos: N_s conjunto dos nós *send* e N_r conjunto dos nós *receive*. O conjunto E denota o conjunto de arestas. E_i^p denota o conjunto de arestas intra-processo pertencentes ao processo p, enquanto que E_s denota o conjunto de arestas de sincronização.

Utilizando o modelo PCFG proposto é possível utilizar critérios de teste para softwares sequenciais, no entanto, dado que o contexto desta dissertação é realizar testes

para softwares concorrentes é necessário testar a comunicação/sincronização existente no PCFG derivado. Esta dissertação de mestrado utiliza uma família de critérios baseados em fluxo de controle e comunicação adaptados para gerar sequências de testes para softwares concorrentes propostos por Souza et al. (2008), mostrada pela Tabela 3.1. No trabalho citado, também foi proposta uma família de critérios baseados em fluxo de dados e passagem de mensagem. Maiores detalhes sobre as famílias de critérios podem ser encontrados em Souza et al. (2008).

No trabalho realizado por Sarmanho (2010), o modelo PCFG é estendido para o contexto de softwares concorrentes com memória compartilhada utilizando *time-stamps* (rótulos de tempo) para determinar a comunicação entre *threads*. O modelo proposto pelo autor, *Parallel Control Flow Graph for Shared Memory* (PCFGsm), é utilizado nesta dissertação de mestrado como ponto inicial para propor uma nova abordagem de teste para software concorrente. Entretanto, é importante destacar que o trabalho desenvolvido por Sarmanho (2010), não deriva os casos de testes para sistemas concorrentes, os quais são gerados manualmente e posteriormente são realizadas as análises. Em comparação com o modelo PCFGsm, as principais contribuições deste trabalho de mestrado são:

Tabela 3.1 - Família de critérios baseados em fluxo de controle e comunicação.

Critério All-nodes-r	requer que todos os nós $n_i^p \in N_r$, sejam exercitados ao menos uma vez. Em outras palavras, todos os nós que executem o recebimento (*receive*) de uma mensagem seja exercitado.
Critério All-nodes-s	requer que todos os nós $n_i^p \in N_s$, sejam exercitados ao menos uma vez. Ou seja, todos os nós que executem o envio (*send*) de uma mensagem seja exercitado.
Critério All-nodes	requer que todos os nós $n_i^p \in N$ sejam exercitados. Ou seja, todos os nós do PCFG sejam exercitados ao menos uma vez.
Critério All-edges-s	requer que todas as arestas $(n_i^f, n_j^g) \in E_s$ sejam exercitadas. Ou seja, toda aresta referente à comunicação/sincronização entre processos deve ser exercitada.
Critério All-edges	requer que todas as arestas $(n_i^f, n_j^g) \in E$ sejam exercitadas. Ou seja, requer que todas as arestas, sem exceção, do grafo PCFG sejam exercitadas.

Fonte: Souza et al. (2008)

- o uso de *Statecharts* neste trabalho contribui para que as características de softwares reativos e concorrentes sejam especificadas. Adicionalmente, viabiliza em termos computacionais a integração do modelo proposto com a ferramenta WEB-PerformCharts;

- torna possível a geração de casos de teste utilizando critérios estruturais para softwares concorrentes, nos quais a modelagem realizada utiliza uma abordagem TBM;

- automatização do processo de interpretação de código fonte, geração do PCFG e geração automática das sequências de teste utilizando critérios estruturais adaptados para softwares concorrentes; e

- possibilidade de uma maior variedade e quantidade das sequências de teste geradas devido ao processo automatizado.

No decorrer deste capítulo, tais contribuições e a metodologia empregada serão detalhadas e ilustradas por meio de alguns estudos de casos utilizados para validar o modelo proposto.

3.1 Abordagem proposta

A proposta para realizar os TBM utilizando uma abordagem caixa branca foi dividida neste trabalho em algumas etapas conforme citado na Figura 3.1. Estas etapas são detalhadas nas seções a seguir.

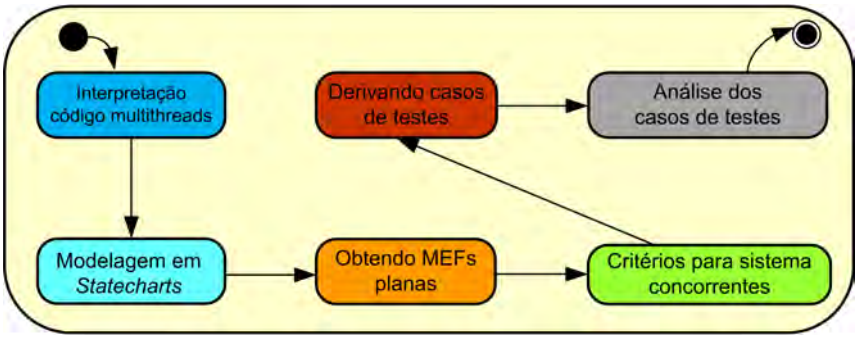


Figura 3.1 - Abordagem proposta.

3.1.1 Interpretação do código *multithread*

Nesta etapa o modelo proposto nesta dissertação realiza a interpretação do código fonte do software *multithreads* num PCFG. Num primeiro momento, essa interpretação deriva um GFC conforme a instrução contida no código fonte, e na sequência os *nós* que compõem este grafo são adicionados a um PCFG.

O modelo de derivação do PCFG proposto neste trabalho considera o conhecimento a priori da quantidade de *threads* do software *multithreads* submetido ao modelo proposto para interpretação do código fonte. Esta simplificação é realizada para reduzir a complexidade do modelo ao tratar as questões de sincronização e concorrência oriundas do código fonte *multithreads*. Por exemplo, considerando um software *multithreads* composto por 3 (três) *threads* que realizam a comunicação entre processos, o modelo proposto irá criar em tempo de execução uma instância de PCFG para cada método *run* (no caso de threads) e *syncronized* (no caso de métodos sincronizados) identificados. Estas instâncias serão criadas quantas vezes se fizer necessário conforme o conteúdo do código fonte da *thread*.

Foram definidas estruturas para representação do PCFG criado em tempo de execução. Estas estruturas são capazes de armazenar os estados contidos no respectivo PCFG, os estados alcançáveis a partir de um determinado estado, a lista de transições, os rótulos associados aos eventos e ainda mantém informações referentes aos estados e transições percorridas. O diagrama de classe completo é exibido na Figura 3.2, o qual é composto pelas seguintes classes:

- ICriterioGenerico - interface que possui métodos genéricos que são utilizados por todos os critérios de teste implementados nesta dissertação;
- GrafoFluxoControleParalelo - classe que contém as informações referentes ao PCFG;
- Estado - classe que contém os dados referentes aos estados do PCFG;
- Transicao - classe responsável por armazenar os dados das transições, como: estados origem e destino;
- AllEdges - classe que contém a implementação do critério de teste *All-edges* para derivação dos casos de teste;

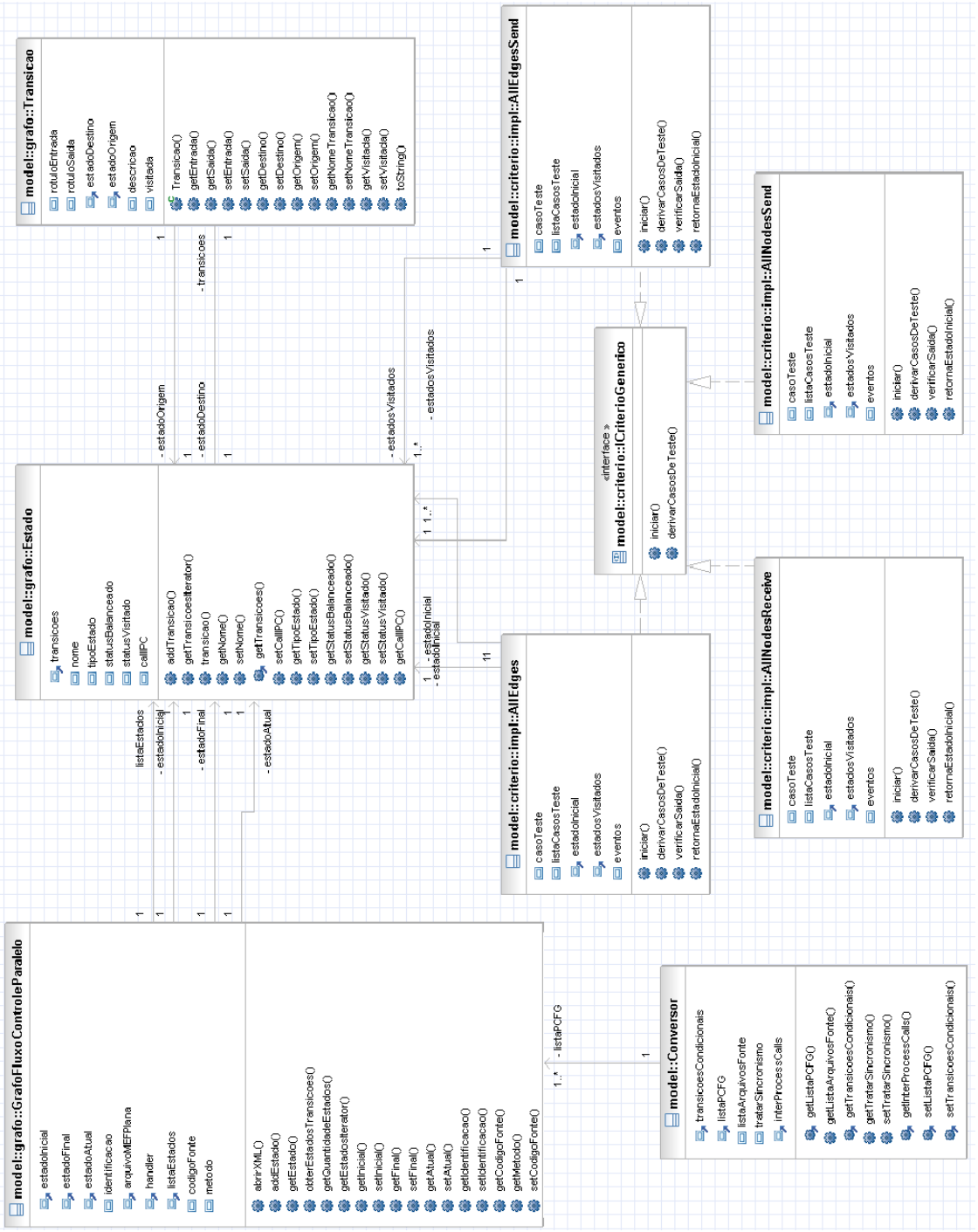


Figura 3.2 - Diagrama de classe do modelo proposto.

- AllEdgesSend - classe que implementa o critério de teste *All-edges-s* para derivação dos casos de teste;
- AllNodesSend - classe que contém a implementação do critério de teste *All-nodes-s* para derivação dos casos de teste;
- AllNodesReceive - implementa o critério de teste *All-nodes-r*; e
- Conversor - classe responsável por realizar a interpretação das *threads* e armazenar os PCFG derivados.

O modelo proposto inicia com a interpretação em sequência de todas as *threads* submetidas ao modelo. Para cada *thread* é adotada a mesma estratégia para geração dos vértices a serem adicionados ao PCFG referente ao método e *thread* em interpretação. A abordagem inicial consiste em criar uma instância de PCFG e adicioná-la a uma lista. Na sequência, o modelo verifica para cada linha do código fonte o tipo de instrução (tomada de decisão, declaração e inicialização de variáveis e laços de repetição) contido. Caso o tipo de instrução detectado seja **declaração e inicialização de variáveis** os códigos lidos farão parte de um único vértice até que seja encontrado uma linha de código que possua um outro tipo de instrução. Esta abordagem é utilizada para todos os outros tipos de instruções. Entretanto, os vértices e arcos criados são conectados de acordo com o tipo de instrução detectado. A Figura 3.3 mostra a representação dos vértices e arcos gerados quando os tipos detectados forem: sequenciais, tomada de decisão *if*, laço *while/for*, laço *until* e tomada de decisão *switch case*, respectivamente.

Durante a interpretação do código fonte o modelo identifica eventuais chamadas a

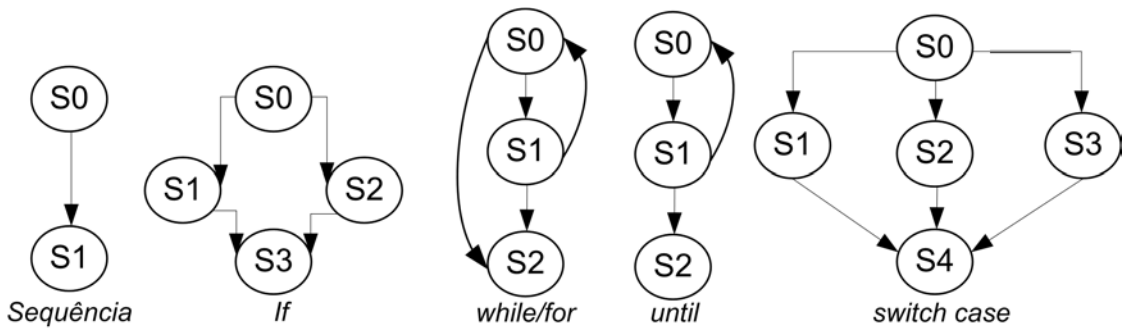


Figura 3.3 - Grafos de fluxo de controle.

métodos Java sincronizados[1]. Caso isso ocorra, é verificado no atributo *listaPCFG* da classe Conversor se o método destino chamado possui um PCFG derivado. Isto é realizado por meio dos atributos *codigoFonte* e *metodo* da classe Conversor. Caso o método possua uma instância de PCFG o modelo obtém o nome do estado inicial deste PCFG e cria uma transição para este estado, na qual este estado sendo o estado final. O estado inicial desta nova transição é a linha atual do código fonte lido do método que efetuou a chamada entre processos.

Caso o método chamado não possua uma instância de PCFG, então o modelo cria uma nova instância de PCFG, adiciona esta instância numa lista e realiza a interpretação do código fonte deste método identificando o grafo de fluxo de controle conforme citado anteriormente. Ao fim da interpretação do método, uma nova verificação da existência da instância de um PCFG para o método chamado é realizada para finalmente criar a nova transição. Em resumo, o modelo adiciona ao PCFG atual, a partir da linha de código lida, as informações referentes à chamada entre processos sincronizados, quais sejam:

- *Thread* origem - é a *thread* atual que está sendo lida; e
- *Thread* destino - é a *thread* que contém o método sincronizado chamado, a qual pode ser a mesma identificada como origem.

As informações citadas anteriormente a respeito das chamadas entre processos são armazenadas numa lista. Desta forma, o modelo será capaz de especificar em *Statecharts* os aspectos de concorrência e paralelismo contidos no software. Os detalhes desta especificação são tratados na próxima seção.

Na sequência e considerando o contexto atual de execução do modelo, no qual o último estado derivado fez uma chamada entre processos, a seguinte ação é realizada: o modelo modifica o valor do atributo booleano *callIPC* da instância do estado que efetuou a chamada entre processo para o valor *true*. Ao continuar a interpretação do código fonte e ao identificar um novo estado a ser adicionado ao PCFG o modelo cria uma nova transição caso o atributo *callIPC* tenha o valor *true*. Esta nova transição é armazenada numa lista específica de transições condicionais. Esta transição possui como origem o estado que fez a chamada entre processos e como estado destino o próximo estado derivado.

[1]Métodos que contém em sua declaração a palavra *syncronized*.

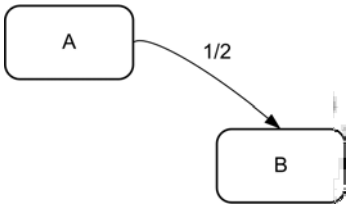


Figura 3.4 - MEF - Exemplo transições.

É importante destacar que, para a posterior geração automática dos casos de teste é necessário que exista um rótulo para as transições existentes entre os estados. Como exemplo, a Figura 3.4 exibe uma MEF que possui os estados A e B e uma única transição existente, a qual possui o rótulo 1/2 que pode ser lido da seguinte forma: o estado A ao receber a entrada (ou evento) 1 torna o estado B como estado ativo. Depois, é impressa a saída com rótulo 2. Estes rótulos são necessários para realizar os percursos entre os estados do PCFG.

Devido ao exposto no parágrafo anterior, para realizar adequadamente a geração dos rótulos das transições detectadas foi implementado para o modelo proposto um algoritmo que fornece rótulos de entrada e saída. Estes rótulos são gerados e nomeados de acordo com o tipo das transições detectadas pelo modelo, a saber:

- em caso de transições que não possuem condições o rótulo recebe o prefixo *e* mais um sufixo numérico;
- em caso de transições que possuem condições os rótulos seguem a seguinte regra de nomeação: prefixo *CONDEVENT* mais um sufixo numérico; e
- as transições que levam de um estado terminal para o estado inicial são nomeadas com o prefixo *eto* mais um sufixo numérico.

Para todos os rótulos criados o sufixo numérico inicia-se em *1* e recebe incremento a medida que outros rótulos são detectados. Por exemplo, o rótulo para a primeira transição sem condição detectada num diagrama *Statecharts* é nomeado como *e1*, o segundo como *e2* e assim por diante.

É importante destacar que os métodos implementados nesta dissertação requerem uma MEF fortemente conectada, ou seja, é necessário que a partir de qualquer estado seja possível alcançar qualquer outro estado da MEF. Desta forma, para atender esta

especificidade o modelo proposto cria no PCFG derivado um arco do tipo *time-out* em todos os estados terminais para o estado inicial do PCFG. Estados terminais são aqueles dos quais não partem arcos.

Após realizar a interpretação completa do código fonte da *thread* os objetos Java, estados e transições, que representam o PCFG são mantidos na memória e inicia-se a próxima etapa. A Figura 3.5 exibe o fluxograma referente aos passos descritos nesta seção.

3.1.2 Modelagem em *Statecharts*

Nesta etapa é realizada a leitura do PCFG passando por todos os vértices e arcos do grafo. A leitura tem como objetivo especificar o grafo em *Statecharts*. Para representar o *Statecharts* é utilizada a PcML, a qual permite representar com exatidão todos os componentes contidos num diagrama *Statecharts*, como estados, transições, condições, eventos de entrada e saída e super estados. O código PcML correspondente aos elementos lidos do PCFG são exibidos na Figura 3.6, na qual é possível verificar a estrutura geral de um arquivo PcML. A estratégia de tradução do grafo é realizada da seguinte forma:

- é criado um arquivo que irá armazenar o código PcML;
- adicionar o código de cabeçalho ao arquivo PcML;
- percorrer cada PCFG contido na lista de grafos detectados conforme explicado na seção anterior;
- cada PCFG é adicionado como um super estado no arquivo PcML dentro do grupo de *tag States*;
- utilizando o algoritmo busca em profundidade são percorridos todos os estados do grafo;
- para cada estado é gerada uma linha de código no arquivo PcML, a qual é adicionada no super estado correspondente;
- percorre-se a lista de transição de cada estado;
- para cada transição é gerado o código PcML correspondente dentro do grupo de *tags Transitions*. Cada transição contém os seguintes elementos: estado origem (*Source*), estado destino (*Destination*)e o rótulo (*Event*);

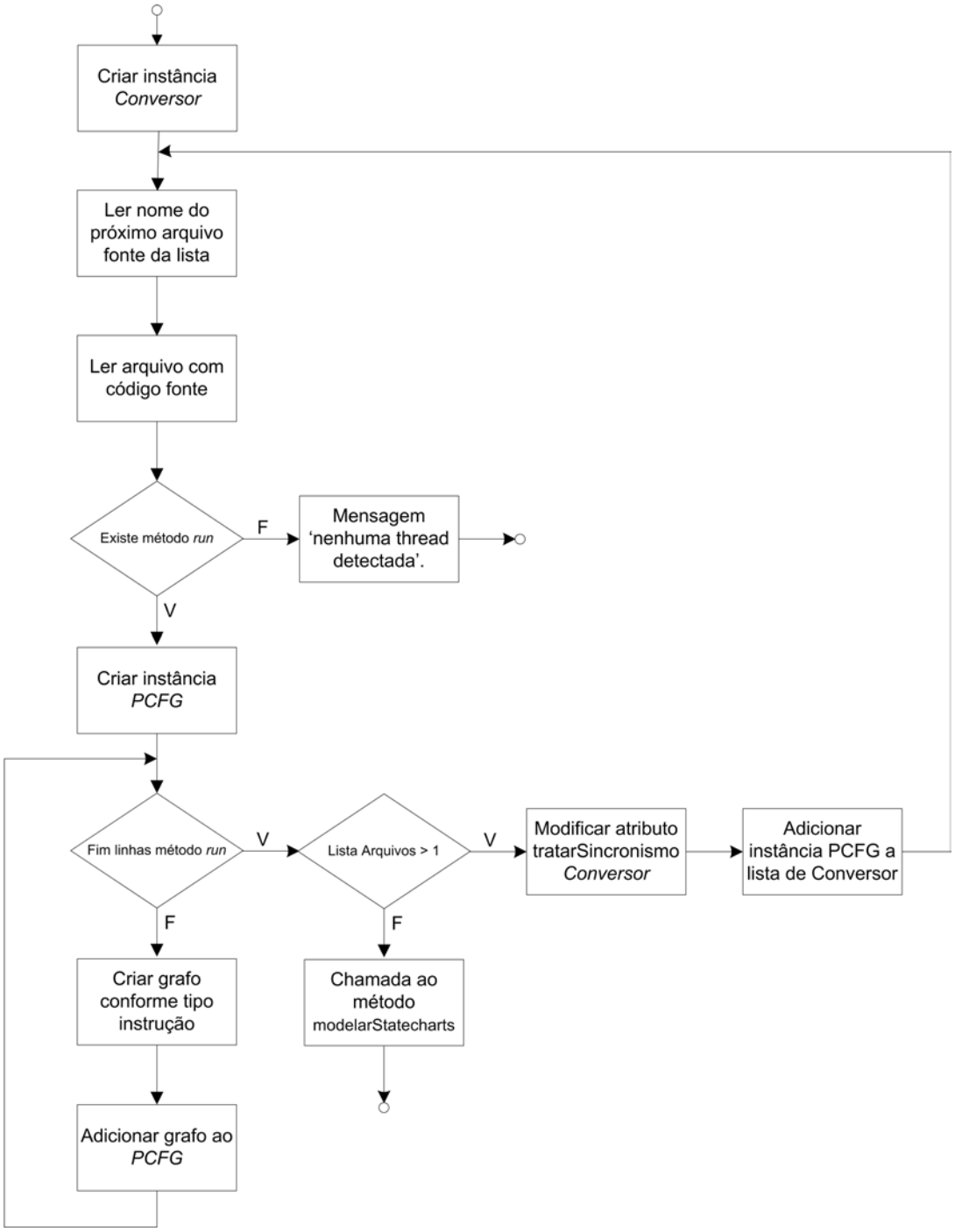


Figura 3.5 - Fluxograma da interpretação do código fonte.

- para cada transição contida na lista *interProcessCalls* do objeto Conversor é obtido o estado destino e deste a transição de saída; para esta transição é adicionado o código PcML dentro do grupo de tag *Actions*;

- para cada rótulo das transições são gerados os respectivos códigos PcML dentro do grupo de *tag events*;

- para cada transição condicional contida na lista de transições condicionais são adicionados ao arquivo os códigos PcML dentro da *tag Conditions*; e

- são adicionadas os códigos de fechamento ao arquivo PcML.

Em outras palavras, cada PCFG derivado na seção anterior, no qual cada PCFG corresponde a uma *thread*, corresponderá a um super estado no arquivo PcML, o qual é contido pelos estados que representam o fluxo de controle do código fonte lido. O grafo é percorrido e a partir dele também são convertidos em código PcML transições, condições e os rótulos de entrada e saída. Por fim, é percorrida o conteúdo da lista *transicoesCondicionais* do objeto Conversor, e para cada item da lista são geradas as linhas referente as condições das transições.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Pcml Title="estudoCaso" Date="2011-07-25" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="schema.xsd">
    <Info>
        <Author>
            <Name>Rogerio Marinke</Name>
            <Email>rogerio.marinke@lac.inpe.br</Email>
        </Author>
        <Description>
            Estrutura geral do arquivo PcML.
        </Description>
    </Info>
    <States>
        <Root Name="estudoCaso" Type="AND">
            <State Name="superEstado1" Type="XOR" Default="sensor1">
                <State Name="sensor1" Type="BASIC"/>
                <State Name="sensor2" Type="BASIC"/>
            </State>
        </Root>
    </States>
    <Conditions>
        <InState Name="" State=""/>
    </Conditions>
    <Actions>
        <EventTriggerAction Name="" Event=""/>
    </Actions>
    <Events>
        <Conditioned Name="" Condition="" Value=""/>
    </Events>
    <Transitions>
        <Transition Source="" Event="" Action="" Destination=""/>
    </Transitions>
</Pcml>
```

Figura 3.6 - Estrutura do arquivo PcML.

```
<Conditions>
    <InState Name="COND1" State="S0T3"/>
</Conditions>
<Events>
    <Conditioned Name="CONDEVENT1" Condition="COND1" Value="e5"/>
</Events>
<Actions>
    <EventTriggerAction Name="ETA1" Event="etaNe1" />
</Actions>
<Transitions>
    <Transition Source="S4T1" Event="CONDEVENT1" Action="ETA1" Destination="S5T1" />
</Transitions>
```

Figura 3.7 - Exemplo de condição para transições especificadas em PcML.

Um exemplo de transição com condição especificada em PcML é exibido na Figura 3.7, a qual mostra que a transição do estado *S4T1* para o estado *S5T1* é condicionada a um evento que obedece à condição chamada *COND1* do tipo *inState*. Em outras palavras, esta transição somente ocorrerá caso o estado chamado *S0T3* esteja ativo. Uma informação adicional, é que após ocorrer a transição é executada a ação nomeada *ETA1*, a qual dispara o evento *etaNe1*. Ações do tipo *eventTriggerAction* correspondem a um evento a ser disparado num componente paralelo na ocorrência de uma transição (AMARAL, 2005).

Após a execução dos passos descritos acima é obtido o modelo em *Statecharts* do software especificado em PcML. Este arquivo representa o PCFG derivado na primeira etapa. A Figura 3.8 exibe o fluxograma referente aos passos descritos nesta seção.

3.1.3 Obtendo MEFs planas

O objetivo desta etapa é obter as MEFs planas a partir do diagrama *Statecharts* especificado em PcML na etapa anterior.

Esta conversão é realizada nesta dissertação para possibilitar a posterior implementação dos critérios de teste estruturais para geração dos casos de testes. Adicionalmente, realizando a implementação desta forma os esforços de integração destes critérios com a ferramenta WEB-PerformCharts desenvolvida no INPE serão minimizados.

A realização desta etapa consiste em utilizar o arquivo PcML obtido na etapa anterior como entrada para a ferramenta WEB-PerformCharts, a qual fornecerá como saída um arquivo no formato XML que representa a MEF plana. As Figuras Fi-

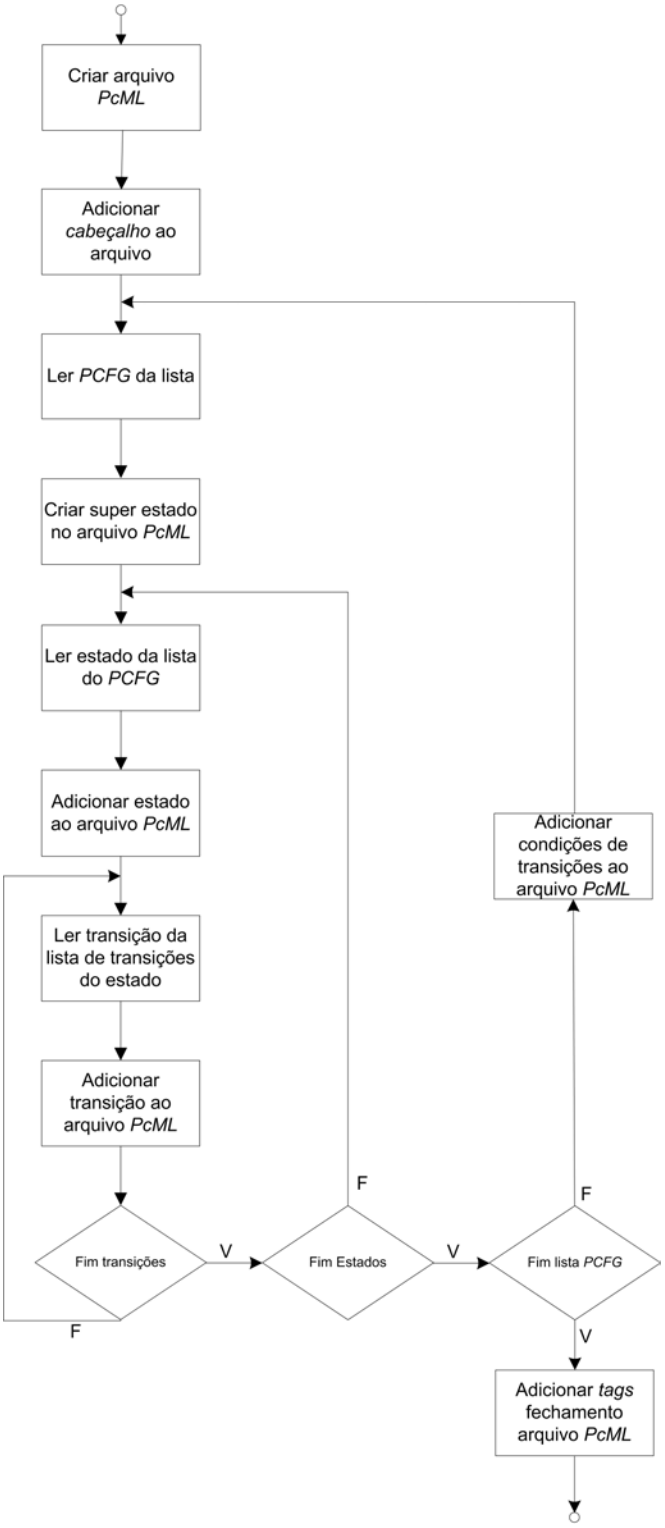


Figura 3.8 - Fluxograma modelagem do código em Statecharts.

Figura 3.9 - WEB-PerformCharts - Importação arquivo PcML.
Fonte: Adaptada de Arantes et al. (2008).

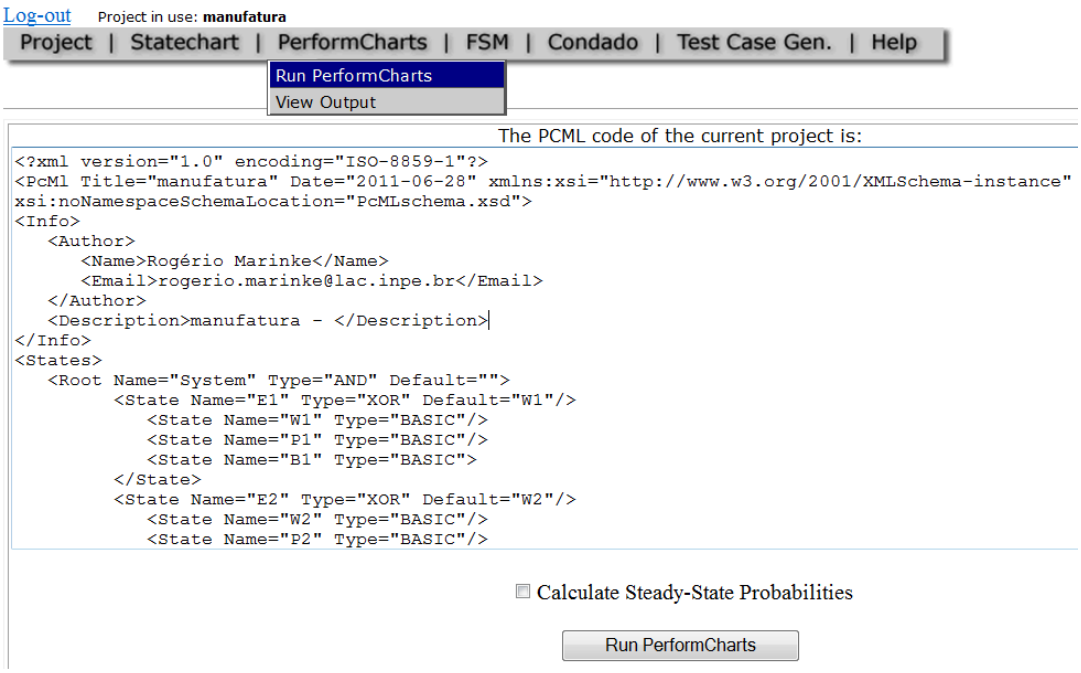


Figura 3.10 - WEB-PerformCharts - Gerando MEF plana.
Fonte: Adaptada de Arantes et al. (2008).

gura 3.9, Figura 3.10, Figura 3.11 exibem um exemplo de execução da ferramenta.

3.1.4 Derivando os casos de testes

Inicialmente neste trabalho, foi implementado o critério de teste estrutural chamado *All-edges*. Embora, este critério estrutural auxilie na revelação de erros de computação ele é restrito ao considerar apenas um PCFG, ou seja, os casos de testes derivados correspondem a uma execução isolada do PCFG. No entanto, como o pro-

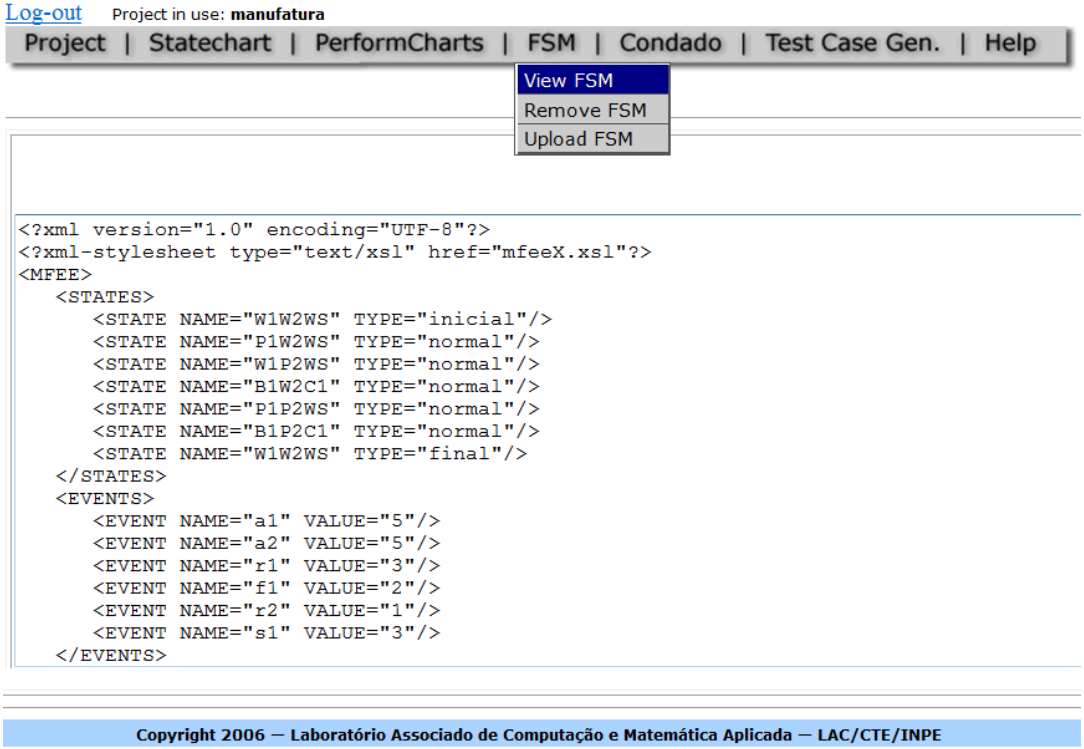


Figura 3.11 - WEB-PerformCharts - Visualizando MEF plana.
Fonte: Adaptada de Arantes et al. (2008).

pósito principal deste trabalho é gerar casos de testes para softwares concorrentes utilizando critérios estruturais, ou seja, deseja-se obter casos de testes que exercitem a concorrência e o paralelismo eventualmente contidos no modelo, os casos de testes gerados utilizando o critério *All-edges* podem não ser suficientes para algumas análises em software concorrente, como por exemplo o teste de comunicação entre processos. Entretanto, no contexto de softwares paralelos, um erro de computação pode estar relacionado a um erro de comunicação. Desta forma, foram implementados nesta dissertação os critérios estruturais para softwares concorrentes propostos por Souza et al. (2008), quais sejam: *All-nodes-r*, *All-nodes-s* e *All-edges-s*. Estes critérios foram explicados na Tabela 3.1.

Em resumo, para gerar os casos de testes são realizados os seguintes procedimentos: (a) utilizando o algoritmo busca em largura adaptado para atender as restrições do critério a MEF plana derivada é percorrida; (b) com base nos estados e transições do software e considerando as restrições do critério são derivados os casos de teste - para os critérios de teste para softwares concorrentes é utilizado o conteúdo da lista

de chamadas entre processos para derivar os casos de testes.

Os casos de teste obtidos nesta etapa serão analisados neste trabalho de uma forma preliminar. A abordagem utilizada para realização desta análise é detalhada na próxima seção.

3.1.5 Análise dos casos de teste obtidos

Esta seção descreve a abordagem utilizada para realizar uma análise do ponto de vista de cobertura dos casos de teste derivados. Uma análise dos casos de testes gerados pode, por exemplo, possibilitar que seja verificada a eficácia dos critérios dentro de determinados contextos do software submetido aos testes. Essa análise de cobertura é realizada neste trabalho nas seguintes perspectivas:

- são comparados a quantidade de estados percorridos ao se utilizar os critérios: *All-nodes-r*, *All-nodes-s*;
- são comparados a quantidade de transições exercitadas ao se utilizar os critérios: *All-edges-s*, *All-edges*; e
- as quantidades de estados e transições percorridas são comparadas ao total de estados e transições existentes na MEF plana derivada.

No contexto desta dissertação de mestrado as quantidades de estados e transições obtidas em cada um dos casos descritos acima são utilizadas para verificar se estas quantidades realmente existem no software *multithreads* submetido ao modelo proposto neste trabalho. Esta verificação é realizada de forma automática pelo modelo proposto. Portanto, esta análise preliminar auxilia na validação do modelo proposto e contribui para validar a qualidade dos casos de teste gerados, uma vez que os critérios revelam uma medida de cobertura dos testes realizados.

Este capítulo apresentou as características, restrições e detalhes da implementação do modelo proposto nesta dissertação de mestrado. O próximo capítulo discute os resultados obtidos a partir da aplicação deste modelo por meio de alguns estudos de casos de software *multithreads*.

4 RESULTADOS

Este capítulo apresenta e discute os resultados obtidos nesta dissertação de mestrado. Em resumo, os resultados são derivados a partir dos códigos fonte de softwares submetidos ao modelo proposto no capítulo anterior. Conforme anteriormente citado, o modelo lê o código fonte do software (pode ser composto por diversos arquivos), deriva os GFC e adiciona-os ao PCFG. Ao final da leitura do código fonte o PCFG obtido é percorrido, e, à medida que cada objeto do PCFG é lido é realizada a modelagem em *Statecharts*. Então, a especificação *Statecharts* é submetida a ferramenta WEB-Performcharts, a qual obtém a MEF plana correspondente, da qual são derivados os casos de testes após execução dos critérios implementados nesta dissertação.

4.1 Estudo de caso - Software *multithreads* Produtor/Consumidor

Segundo Blum et al. (2003), uma vantagem de se utilizar Java embarcado em sistemas é que o uso da JVM torna as diferenças entre o ambiente desktop e embarcado muito pequenas, o que possibilita uma vantagem no desenvolvimento do sistema, uma vez que as diferenças entre o ambiente simulado e o ambiente real são minimizadas pela JVM, possibilitando que o código Java permaneça praticamente o mesmo em ambos os ambientes.

4.1.1 Código fonte

Neste primeiro estudo de caso é submetido ao modelo proposto um software *multithreads*. O software *multithreads* selecionado realiza a leitura e escrita de valores numéricos num *buffer*, o qual possui o acesso as suas regiões críticas por meio de monitores. Na linguagem Java pode ser utilizada a palavra reservada *synchronized* na declaração de um método, por exemplo, para garantir que apenas uma *thread* terá acesso ao método/bloco num determinado instante.

O software *multithreads* exibido nas Figuras 4.1, 4.2, 4.3 implementa o exemplo clássico de concorrência do tipo produtor/consumidor e foi adaptado a partir de trechos de código do trabalho realizado por Baker et al. (2006). A Figura 4.4 exibe o diagrama de classes do software, o qual é composto pelas seguintes classes:

- Buffer - esta classe desempenha o papel de supervisor do processo de consumo e produção. Esta classe contém atributos para determinar a quan-

```java
package threads.synchronizedBuffer;

public class Buffer {
    private int memory = -1;
    private boolean occupied = true;
    private int sizeBuffer;
    private int timeSleep;

    public int getSizeBuffer() {
        return sizeBuffer;
    }

    public void setSizeBuffer(int sizeBuffer) {
        this.sizeBuffer = sizeBuffer;
    }

    public void setTimeSleep(int timeSleep) {
        this.timeSleep = timeSleep;
    }

    public int getTimeSleep() {
        return timeSleep;
    }

   public synchronized void writeBuffer(int pValue){
      while(!occupied) {
          try {
               wait();
          }
          catch (InterruptedException e) {
               e.printStackTrace();
          }
      }
      System.err.println( Thread.currentThread().getName() + " producing: " + pValue );
      this.memory = pValue;
      occupied = false;
      notify();
   }

   public synchronized int readBuffer(){
      while(occupied) {
          try {
               wait();
          }
          catch ( InterruptedException e) {
               e.printStackTrace( );
          }
      }
      System.err.println( Thread.currentThread().getName() + " consuming: " + this.memory );
      occupied = true;
      notify();
      return this.memory;
   }
}
```

Figura 4.1 - Código fonte da classe Buffer.

tidade máxima de valores numéricos que poderão ser armazenados pelo *buffer* (atributo *sizeBuffer*), o intervalo de tempo de *sleep* das *threads* (atributo *timeSleep*), o valor numérico produzido/consumido (atributo *memory*) e um atributo para informar se alguma *thread* está na região crítica do software (atributo *occupied*). A classe ainda possui os métodos sincronizados *writeBuffer()* e *readBuffer()*;

- Producer - *thread* responsável por produzir (escrever) valores numéricos de 1 até o valor máximo definido na classe Buffer; e
- Consumer - esta *thread* realiza o consumo (leitura) dos valores numéricos

```java
package threads.synchronizedBuffer;

public class Producer extends Thread{
        private Buffer buffer;

        public Producer(Buffer pBuffer) {
            super("Producer");
            buffer = pBuffer;
        }

        public void run() {
            for (int i = 1; i <= buffer.getSizeBuffer(); i++) {
                try {
                    Thread.sleep((int) (Math.random() * buffer.getTimeSleep()));
                } catch (InterruptedException exception) {
                    System.err.println(exception.toString());
                }
                buffer.writeBuffer(i);
            }
            System.err.println(getName() + " finished tasks.");
        }
}
```

Figura 4.2 - Código fonte da classe Producer.

```java
package threads.synchronizedBuffer;

public class Consumer extends Thread{
        private Buffer buffer;

        public Consumer(Buffer pBuffer) {
            super("Consumer");
            buffer = pBuffer;
        }

        public void run() {
            int value, total = 0;
            do {
                try {
                    Thread.sleep((int) (Math.random() * buffer.getTimeSleep()));
                } catch (InterruptedException exception) {
                    System.err.println(exception.toString());
                }
                value = buffer.readBuffer();
                total += value;
            } while (value != buffer.getSizeBuffer());
            System.err.println(getName() + " finished tasks. Total: " + total);
        }
}
```

Figura 4.3 - Código fonte da classe Consumer.

definidos na classe Buffer.

A partir deste ponto os passos da metodologia descrita no capítulo anterior serão exercitados. O primeiro passo é executar o modelo fornecendo como entradas os códigos fontes desse estudo de caso.

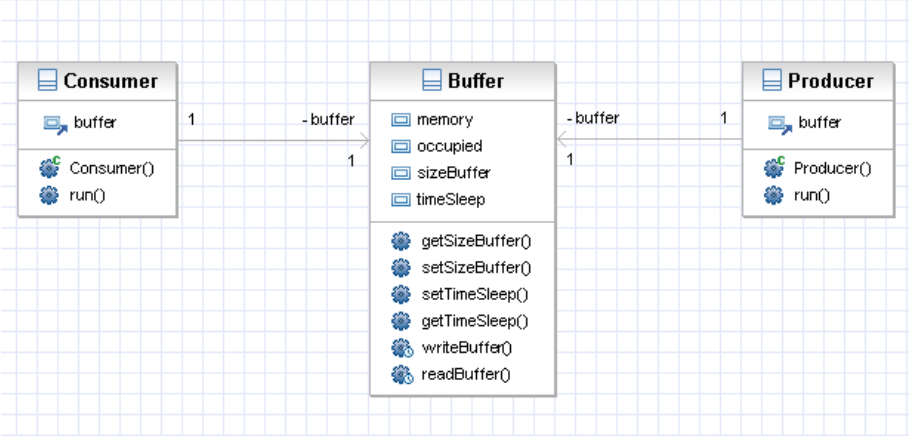


Figura 4.4 - Diagrama de classes estudo de caso Produtor/Consumidor.

4.1.2 *Statecharts*

Após realizar a leitura do código fonte e derivar os PCFGs, o modelo proposto percorre todos os PCFGs e produz como resultado um arquivo PcML, o qual representa o *Statecharts* do software lido. O conteúdo deste arquivo é exibido na Figura 4.5, no qual é importante destacar que os seguintes super estados foram derivados para esse estudo de caso:

- *CONSUMER* - possui como estado inicial o *S0T1* e contém os estados de *S0T1* até *S6T1*;
- *PRODUCER* - contém os estados de *S0T2* até *S6T2* e o estado inicial é o *S0T2*;
- *BUFFER1* - contém os estados de *S0T3* até *S3T3* e o estado inicial é o *S0T3*; e
- *BUFFER2* - contém os estados de *S4T3* até *S7T3* e o estado inicial é o *S4T3*;

Nota-se na Figura 4.5 que foram detectadas as transições, nas quais cada transição contém: estados origem e destino e o evento associado à transição. Cada evento gerado possui um nome (*tag Name*) e um valor (*tag Value*). Na Figura 4.5 nota-se também valores (no caso 0.1) associados aos eventos estocásticos. Estes valores são ignorados para o caso de geração de testes, pois a ferramenta Web-PerformCharts foi

Figura 4.5 - Especificação *Statecharts* em PcML do Produtor/Consumidor.

inicialmente criada para gerar medidas de desempenho associando *Statecharts* com Cadeias de Markov a Tempo Contínuo onde é necessário que estes valores sejam taxas de transição entre estados(VIJAYKUMAR, 1999).

Para auxiliar na leitura dos resultados foi gerado, a partir do arquivo PcML derivado, o diagrama *Statecharts* do problema proposto, o qual é exibido na Figura 4.6.

4.1.3 MEF plana

Após a execução dos passos anteriores, o arquivo *Statecharts* em PcML é submetido à execução na ferramenta WEB-Performcharts, a qual derivou a MEF plana correspondente. Devido a grande quantidade de linhas do arquivo gerado, somente alguns trechos são exibidos na Figura 4.7.

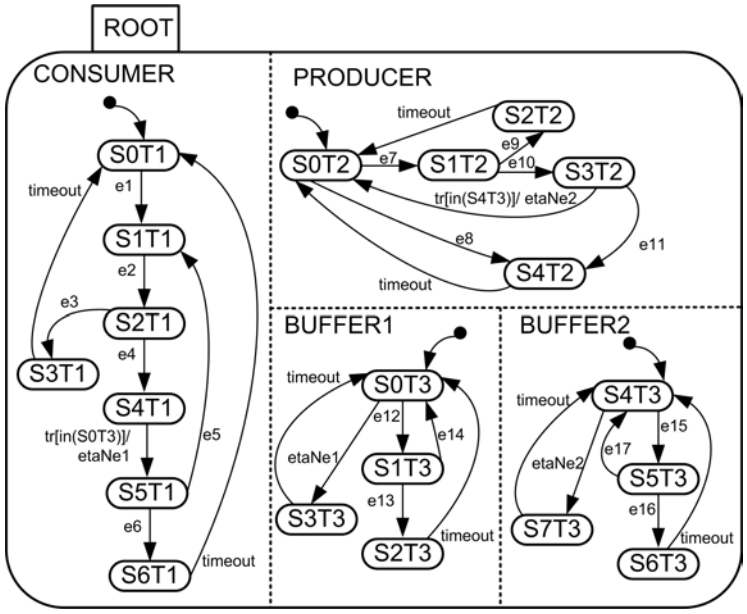


Figura 4.6 - Diagrama *Statecharts* do estudo de caso Produtor/Consumidor.

4.1.4 Casos de testes

O próximo passo da abordagem proposta consiste em submeter a MEF plana à execução dos critérios estruturais adaptados para sistemas concorrentes para obter os casos de testes. Foram gerados um total de 176 casos de teste. Alguns dos casos de teste obtidos para esse estudo de caso são exibidos na Tabela 4.1.

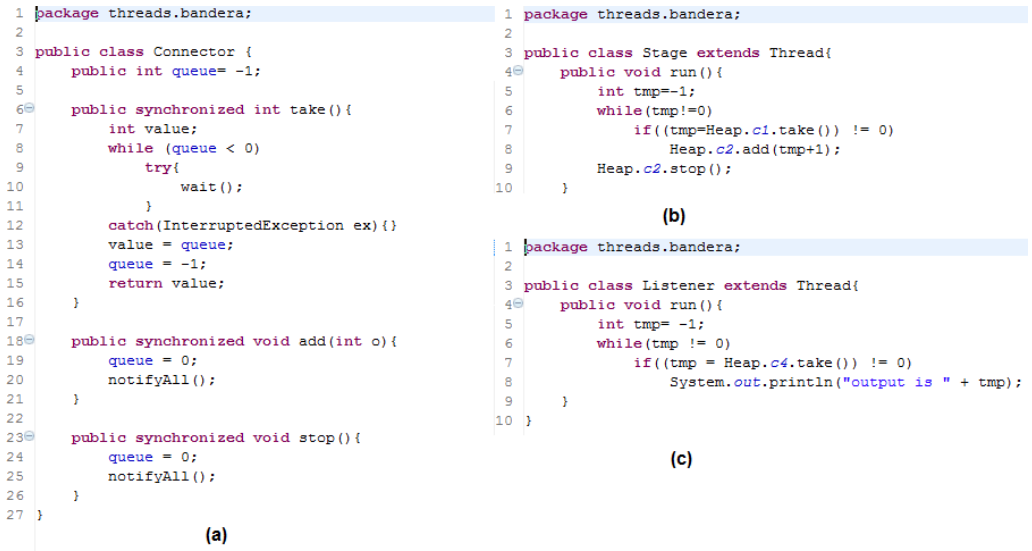
Tabela 4.1 - Alguns casos de teste - Estudo de caso Produtor/Consumidor.

Critério	Casos de teste gerados	Quantidade
All-nodes-s	[S4T1,S0T2,S1T3,S4T3], [S4T1,S1T2,S1T3,S4T3], [S4T1,S4T2,S1T3,S4T3], [S4T1,S1T2,S2T3,S4T3], [S4T1,S1T2,S1T3,S5T3], [S0T3,S1T2,S1T3,S4T3], [S2T1,S1T2,S2T3,S5T3], [S4T1,S0T2,S2T3,S4T3], ... , [S4T1,S0T2,S1T3,S5T3]	75
All-nodes-r	[S0T3,S0T2,S1T3,S4T3], [S0T3,S1T2,S1T3,S4T3], [S0T3,S4T2,S1T3,S4T3], [S0T3,S0T2,S2T3,S4T3], [S2T1,S4T3,S2T3,S5T3], [S0T3,S4T3,S1T3,S4T3], [S4T1,S4T3,S1T3,S5T3], [S0T3,S0T2,S1T3,S5T3], ... , [S4T1,S4T3,S2T3,S4T3]	75
All-edges	[e1, e2, ..., e17], [eto1, ..., eto5], [eipc1, eipc2]	24
All-edges-s	eipc1, eipc2	2

Figura 4.7 - MEF plana do estudo de caso Produtor/Consumidor.

Para o presente estudo de caso, após convertido para a MEF plana, contabilizou-se um total de 584 estados. Deste total, 150 estados foram exercitados pelos casos de testes derivados, ou seja, uma taxa de cobertura de 26 por cento. Em relação as transições, 100 por cento foram exercitadas. Em ambas as análises foram consideradas apenas os critérios de teste estruturais para sistemas concorrentes, ou seja: All-nodes-s, All-nodes-r e All-edges-s. Juntos, estes critérios geraram 152 casos de teste. A Tabela 4.4 exibe estes dados de forma resumida.

Vale salientar que os 150 estados exercitados pelos casos de teste estão contidos em chamadas entre processos, ou seja, as transições destes estados não seriam exercitadas caso fosse utilizado um critério estrutural para software sequencial.



```
package threads.bandera;

public class Connector {
    public int queue= -1;

    public synchronized int take(){
        int value;
        while (queue < 0)
            try{
                wait();
            }
        catch(InterruptedException ex){}
        value = queue;
        queue = -1;
        return value;
    }

    public synchronized void add(int o){
        queue = 0;
        notifyAll();
    }

    public synchronized void stop(){
        queue = 0;
        notifyAll();
    }
}
```

(a)

```
package threads.bandera;

public class Stage extends Thread{
    public void run(){
        int tmp=-1;
        while(tmp!=0)
            if((tmp=Heap.c1.take()) != 0)
                Heap.c2.add(tmp+1);
        Heap.c2.stop();
    }
```

(b)

```
package threads.bandera;

public class Listener extends Thread{
    public void run(){
        int tmp= -1;
        while(tmp != 0)
            if((tmp = Heap.c4.take()) != 0)
                System.out.println("output is " + tmp);
    }
}
```

(c)

Figura 4.8 - Código fonte do estudo de caso *Bandera*.

4.2 Estudo de caso *Bandera*

O código fonte utilizado nesta seção como estudo de caso foi proposto no trabalho realizado por Corbett et al. (2000). No trabalho citado os autores utilizaram o código fonte *multithreads* para extrair as respectivas MEF.

4.2.1 Código fonte

O código fonte desse estudo de caso é exibido na Figura 4.8 e o respectivo diagrama de classes é exibido na Figura 4.9.

4.2.2 *Statecharts*

O diagrama *Statecharts* desse estudo de caso é exibido na Figura 4.10. O modelo proposto nesta dissertação de mestrado realizou a leitura do PCFG derivado a partir do código fonte e gerou a modelagem do software em *Statecharts* especificado em PcML, o qual é exibido na Figura 4.11.

4.2.3 MEF plana

A partir do arquivo PcML exibido na seção anterior, a ferramenta WEB-Performcharts gerou a respectiva MEF plana, a qual é exibida na Figura 4.12.

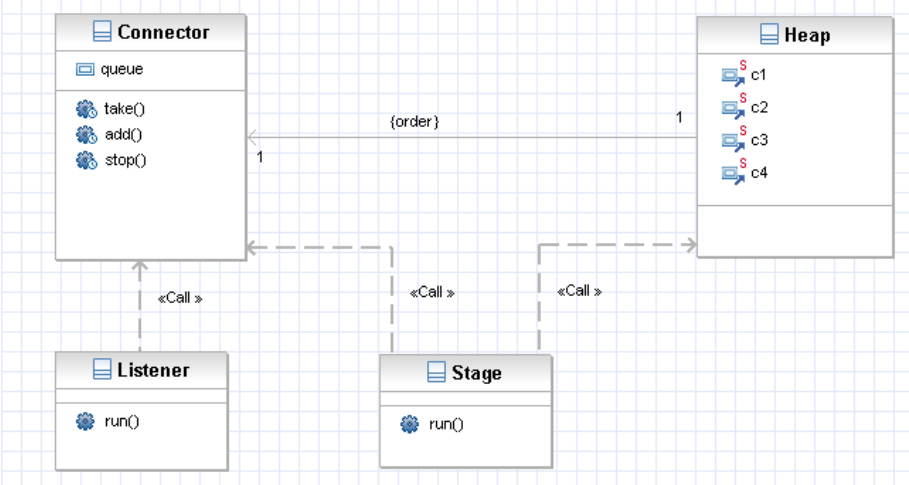


Figura 4.9 - Diagrama de classes do estudo de caso *Bandera*.

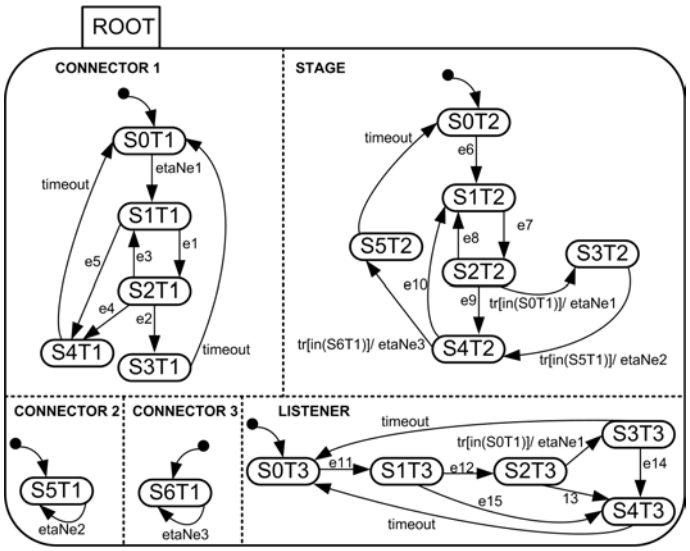


Figura 4.10 - Diagrama *Statecharts* do estudo de caso *Bandera*.

4.2.4 Casos de testes

Alguns casos de testes obtidos nesse estudo de caso são exibidos na Tabela 4.2. No total foram gerados 138 casos de teste. Neste estudo de caso, após convertido para a MEF plana, foram derivados um total de 345 estados. Deste total, 136 estados foram exercitados pelos casos de testes derivados, ou seja, uma taxa de cobertura de 39 por cento. Em relação as transições, 100 por cento foram exercitadas. A Tabela 4.4 exibe estes dados de forma resumida.

Figura 4.11 - Especificação *Statecharts* em PcML do estudo de caso *Bandera*.

Tabela 4.2 - Alguns casos de teste - Estudo de caso *Bandera*.

Critério	Casos de teste gerados	Quantidade
All-nodes-s	[S1T1,S5T1,S6T1,S5T2,S2T3], [S1T1,S5T1,S6T1,S2T2,S3T3], [S2T1,S5T1,S6T1,S5T2,S2T3], [S1T1,S5T1,S6T1,S2T2,S0T3], [S4T1,S5T1,S6T1,S5T2,S2T3], ..., [S1T1,S5T1,S6T1,S0T2,S2T3]	68
All-nodes-r	[S1T1,S5T1,S6T1,S5T2,S0T1], [S1T1,S5T1,S6T1,S0T1,S3T3], [S2T1,S5T1,S6T1,S5T2,S0T1], [S4T1,S5T1,S6T1,S5T2,S0T1], ..., [S1T1,S5T1,S6T1,S0T2,S0T1]	68
All-edges	[e1, e2, e3, e4, e5, e6, ..., e15], [eto1, eto2, ..., eto5], [eipc1, eipc4]	22
All-edges-s	eipc1, eipc4	2

Figura 4.12 - MEF plana do estudo de caso *Bandera*.

4.3 Estudo de caso - Software sequencial

Num primeiro momento durante a realização deste trabalho, com o objetivo de demonstrar e verificar se a implementação do modelo proposto realiza corretamente a modelagem do código em *Statecharts*, foi utilizado um código fonte de um software somente com estruturas e instruções consideradas fundamentais, como: declaração e inicialização de variáveis, laços de repetição e blocos para tomada de decisões.

4.3.1 Código fonte

O código fonte deste estudo de caso submetido ao modelo é mostrado na Figura 4.13.

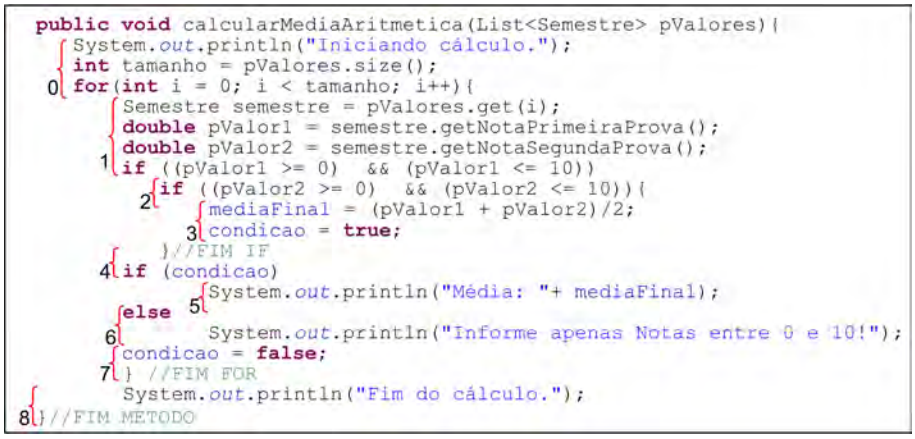


Figura 4.13 - Código fonte software sequencial.

4.3.2 *Statecharts*

Como este estudo de caso refere-se a um software sequencial, resolveu-se gerar o GFC para auxiliar na análise dos resultados, o qual é exibido na Figura 4.14. No GFC gerado é possível observar todos os blocos de execução do software. O primeiro bloco de comandos (nó) é caracterizado da linha 2 à linha 4. O segundo bloco é formado pelas linhas de 5 a 8 e assim por diante. Ao todo o GFC é composto por 9 nós.

Após derivar o GFC o modelo realiza a modelagem dos objetos do grafo em *Statecharts*. O conteúdo do arquivo gerado é exibido na Figura 4.15, no qual é importante destacar que o modelo proposto derivou um super estado chamado *THREAD1*, o qual possui como estado *default* ou inicial o estado *S0*. O super estado *THREAD1* é composto por todos os estados derivados (*S0* até *S8*)do GFC.

É possível ainda verificar na Figura 4.15 que o modelo gerou os eventos que serão associados às transições. Cada evento gerado possui um nome (*tag Name*) e um valor (*tag Value*). Também foram gerados os códigos PcML das transições existentes no GFC.

4.3.3 MEF plana

O arquivo PcML que representa a MEF plana derivada pela ferramenta WEB-Performcharts é exibido na Figura 4.16, o qual exibe os estados, eventos e as entradas derivadas a partir do *Statecharts*. Um dos detalhes a ser destacado na figura citada

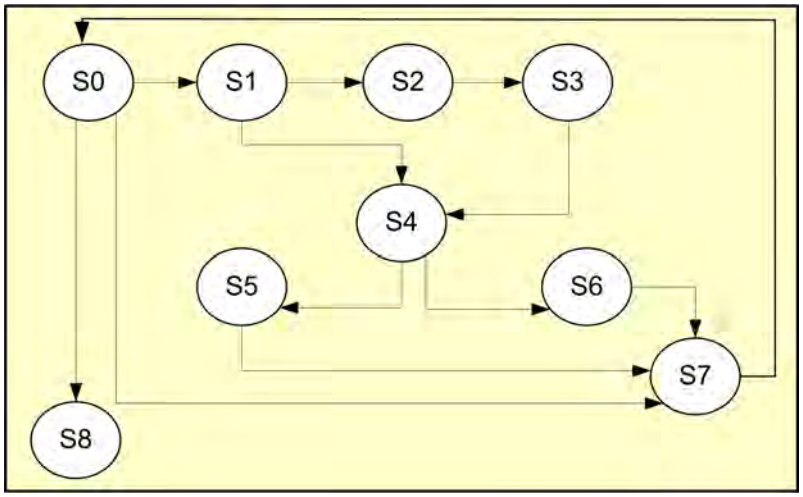


Figura 4.14 - Grafo de fluxo de controle.

anteriormente é o estado *S0*, o qual aparece duas vezes na *tag states* da MEF plana, uma como estado inicial e outra como estado final. Esta informação é utilizada pelos algoritmos de busca em largura e/ou profundidade implementados no modelo ao percorrer o grafo. Estes algoritmos são utilizados pelos critérios de teste caixa branca para derivar os casos de testes. Outra informação relevante, é a listagem de eventos e entradas, sendo que, estes dados serão utilizados em conjunto com as transições para aplicar critérios caixa branca. A Figura 4.17 exibe as transições e a *tag* de fechamento do arquivo.

4.3.4 Casos de testes

Considerando que o código fonte utilizado neste estudo de caso corresponde a um software sequencial foi selecionado o critério de teste *All-Edges* para derivar os casos de teste. Alguns dos casos de testes gerados são exibidos na Tabela 4.3.

Foram gerados um total de 9 casos de teste, os quais realizam uma cobertura de 100 por cento do total de estados existentes no estudo de caso. Em relação as transições 100 por cento delas foram cobertas pelos casos de testes.

Tabela 4.3 - Alguns casos de teste - Estudo de caso software sequencial.

Critério	Casos de teste	Quantidade
All-edges	[e1, e2, e3, e4, e5, e6, e7, e8, e9]	9

```
<States>
        <Root Name="RAIZ" Type="AND" Default="THREAD1">
                <State Name="THREAD1" Type="XOR" Default="S0">
                        <State Name="S0" Type="BASIC"/>
                        <State Name="S1" Type="BASIC"/>
                        <State Name="S2" Type="BASIC"/>
                        <State Name="S3" Type="BASIC"/>
                        <State Name="S4" Type="BASIC"/>
                        <State Name="S5" Type="BASIC"/>
                        <State Name="S6" Type="BASIC"/>
                        <State Name="S7" Type="BASIC"/>
                        <State Name="S8" Type="BASIC"/>
                </State>
        </Root>
</States>
<Events>
        <Stochastic Name="e1" Value="0.1"/>
        <Stochastic Name="e2" Value="0.1"/>
        <Stochastic Name="e3" Value="0.1"/>
        <Stochastic Name="e4" Value="0.1"/>
        <Stochastic Name="e5" Value="0.1"/>
        <Stochastic Name="e6" Value="0.1"/>
        <Stochastic Name="e7" Value="0.1"/>
        <Stochastic Name="e8" Value="0.1"/>
        <Stochastic Name="e9" Value="0.1"/>
        <Stochastic Name="e10" Value="0.1"/>
        <Stochastic Name="e11" Value="0.1"/>
        <Stochastic Name="e12" Value="0.1"/>
</Events>
<Transitions>
        <Transition Source="S0" Event="e1" Destination="S7"/>
        <Transition Source="S0" Event="e2" Destination="S8"/>
        <Transition Source="S7" Event="e3" Destination="S0"/>
        <Transition Source="S0" Event="e4" Destination="S1"/>
        <Transition Source="S1" Event="e5" Destination="S2"/>
        <Transition Source="S1" Event="e6" Destination="S4"/>
        <Transition Source="S2" Event="e7" Destination="S3"/>
        <Transition Source="S3" Event="e8" Destination="S4"/>
        <Transition Source="S4" Event="e9" Destination="S5"/>
        <Transition Source="S4" Event="e10" Destination="S6"/>
        <Transition Source="S5" Event="e11" Destination="S7"/>
        <Transition Source="S6" Event="e12" Destination="S7"/>
</Transitions>
```

Figura 4.15 - Especificação *Statecharts* do software sequencial.

Tabela 4.4 - Resumo da análise dos casos de teste gerados.

Estudo de caso	Estados na MEF plana	Estados exercitados	Casos de teste	Taxa cobertura
Produtor/Consumidor	584	150	152	26 %
Bandera	345	136	138	39 %

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="mfeeX.xsl"?>
        <MFEE>
                <STATES>
                        <State Name="S0" Type="inicial"/>
                        <State Name="S1" Type="normal"/>
                        <State Name="S2" Type="normal"/>
                        <State Name="S3" Type="normal"/>
                        <State Name="S4" Type="normal"/>
                        <State Name="S5" Type="normal"/>
                        <State Name="S6" Type="normal"/>
                        <State Name="S7" Type="normal"/>
                        <State Name="S8" Type="normal"/>
                        <State Name="S0" Type="final"/>
                </STATES>
                <EVENTS>
                        <EVENT NAME="e1" VALUE="1"/>
                        <EVENT NAME="e2" VALUE="2"/>
                        <EVENT NAME="e4" VALUE="4"/>
                        <EVENT NAME="e3" VALUE="3"/>
                        <EVENT NAME="e5" VALUE="5"/>
                        <EVENT NAME="e6" VALUE="0.1"/>
                        <EVENT NAME="e7" VALUE="0.2"/>
                        <EVENT NAME="e9" VALUE="0.4"/>
                        <EVENT NAME="e10" VALUE="0.5"/>
                        <EVENT NAME="e8" VALUE="0.3"/>
                        <EVENT NAME="e11" VALUE="0.6"/>
                        <EVENT NAME="e12" VALUE="0.7"/>
                </EVENTS>
                <INPUTS>
                        <INPUT EVENT="e1"/>
                        <INPUT EVENT="e2"/>
                        <INPUT EVENT="e4"/>
                        <INPUT EVENT="e3"/>
                        <INPUT EVENT="e5"/>
                        <INPUT EVENT="e6"/>
                        <INPUT EVENT="e7"/>
                        <INPUT EVENT="e9"/>
                        <INPUT EVENT="e10"/>
                        <INPUT EVENT="e8"/>
                        <INPUT EVENT="e11"/>
                        <INPUT EVENT="e12"/>
                </INPUTS>
```

Figura 4.16 - MEF plana do software sequencial - Estados, eventos e entradas.

```
<TRANSITIONS>
      <TRANSITION SOURCE="S0" DESTINATION="S7">
         <INPUT INTERFACE="L">e1</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S0" DESTINATION="S8">
         <INPUT INTERFACE="L">e2</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S0" DESTINATION="S1">
         <INPUT INTERFACE="L">e4</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S7" DESTINATION="S0">
         <INPUT INTERFACE="L">e3</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S1" DESTINATION="S2">
         <INPUT INTERFACE="L">e5</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S1" DESTINATION="S4">
         <INPUT INTERFACE="L">e6</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S2" DESTINATION="S3">
         <INPUT INTERFACE="L">e7</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S4" DESTINATION="S5">
         <INPUT INTERFACE="L">e9</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S4" DESTINATION="S6">
         <INPUT INTERFACE="L">e10</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S3" DESTINATION="S4">
         <INPUT INTERFACE="L">e8</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S5" DESTINATION="S7">
         <INPUT INTERFACE="L">e11</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
      <TRANSITION SOURCE="S6" DESTINATION="S7">
         <INPUT INTERFACE="L">e12</INPUT>
         <OUTPUT></OUTPUT>
      </TRANSITION>
</TRANSITIONS>
</MFEE>
```

Figura 4.17 - MEF plana do software sequencial - Transições.

5 CONCLUSÕES

Os softwares embarcados em sistemas de missões espaciais devem ser submetidos a rigorosos procedimentos de testes. Esta dissertação de mestrado explorou testes estruturais utilizando *Statecharts*. O modelo proposto lê o código fonte de softwares *multithreads* e realiza a modelagem correspondente especificada em PcML. No modelo proposto foram realizadas as implementações de critérios de teste, os quais derivam casos de testes adaptados para softwares concorrentes.

5.1 Contribuições do trabalho

Utilizar modelos para a realização de testes de software *multithreads* pode tornar possível a obtenção de um maior rigor das expectativas que o sistema deve satisfazer, sendo que a ambiguidade, considerando a hipótese de existirem estados com comportamento idêntico, por exemplo, é reduzida na medida em que é utilizada uma técnica formal, como *Statecharts*, para representar corretamente os comportamentos do software.

Este trabalho possui duas principais contribuições. A primeira refere-se a modelagem automática em *Statecharts* de códigos fonte de softwares *multithreads*. Esta modelagem é realizada de forma manual no trabalho realizado por Sarmanho (2010).

A segunda contribuição deste trabalho está vinculada a implementação dos critérios de teste para softwares concorrentes. Os resultados obtidos a partir da abordagem exposta nesse trabalho de realizar a modelagem em *Statecharts* do código fonte de software *multithreads*, possibilita a correta especificação de softwares que contenham as características de concorrência e paralelismo. O uso da ferramenta WEB-PerformCharts e a implementação dos critérios de testes para sistemas concorrentes contribui para que a equipe de testes do INPE, por exemplo, possa ter uma alternativa ao realizar os TBM, visto que, atualmente estes testes são realizados utilizando apenas a metodologia caixa preta.

O modelo proposto nesta dissertação está sendo incorporado a ferramenta WEB-PerformCharts, que após a fase de integração poderá derivar casos de testes utilizando uma metodologia caixa branca.

5.2 Sugestões para trabalhos futuros

Como trabalhos futuros sugere-se a integração do modelo proposto nesta dissertação de mestrado à ferramenta WEB-PerformCharts, a implementação de outros critérios e o aprofundamento das análises dos resultados.

Uma alternativa para trabalhos futuros é realizar as análises dos casos de teste gerados nesta dissertação conforme a abordagem proposta por Sarmanho (2010).

Outra possibilidade para trabalho futuro é evoluir o modelo proposto para que o mesmo seja capaz de realizar a modelagem a partir de códigos fonte escritos em outras linguagens de programação, como *C* e *Fortran*.

Futuramente, poderá ser estudada uma abordagem para que o modelo proposto seja capaz de realizar a modelagem a partir de arquivos *bytecode* Java utilizando a *Byte Code Engineering Library* (BCEL).

Como trabalho futuro também pode ser realizada uma análise detalhada dos casos de testes gerados considerando eficiência, variedade e quantidade de passos em cada caso de teste. A análise poderá, por exemplo, realizar uma comparação das características mencionadas comparando critérios caixa branca sequenciais e critérios caixa branca para softwares concorrentes.

REFERÊNCIAS BIBLIOGRÁFICAS

ALMASI, G. S.; GOTTLIEB, A. **Highly parallel computing**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 0-8053-0177-1. 2, 12

AMARAL, A. S. M. S. d. **Geração de casos de teste para sistemas especificados em Statecharts**. 162 p. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2005-11-18 2005. Disponível em: <http://urlib.net/sid.inpe.br/MTC-m13@80/2006/02.14.19.24>. Acesso em: 31 ago. 2010. 19, 22, 42

ARANTES, A. O.; VIJAYKUMAR, N. L.; JUNIOR, V. A. de S.; GUIMARãES, D. Web-performcharts: a collaborative web-based tool for test case generation from statecharts. In: INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS & SERVICES, 10., 2008, Linz, Austria. **Proceedings...** Linz: ACM, 2008. p. 374–381. ISBN 978-1-60558-349-5. 23, 44, 45

BADER, A.; SAJEEV, A. S. M.; RAMAKRISHNAN, S. Testing concurrency and communication in distributed objects. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING, 5., 1998, Washington, USA. **Proceedings...** Washington, 1998. p. 422–428. 27

BAKER, J.; CUNEI, A.; FLACK, C.; PIZLO, F.; PROCHAZKA, M.; VITEK, J.; ARMBRUSTER, A.; PLA, E.; HOLMES, D. A real-time java virtual machine for avionics - an experience report. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMP, 12., 2006, Washington, USA. **Proceedings...** Washington: IEEE Computer Society, 2006. p. 384–396. 13, 47

BEIZER, B. **Black-box testing: techniques for functional testing of software and systems**. [S.l.]: John Wiley and Sons, 1995. 3

BENSALEM, S.; HAVELUND, K.; VERIMAG, U. J. F. Scalable dynamic deadlock analysis of multithreaded programs. In: ACM PARALLEL AND DISTRIBUTED SYSTEMS: TESTING AND DEBUGGING, 3., 2005, New York, USA. **Proceedings...** New York: ACM, 2005. 25

BINDER, R. **Testing object-oriented systems. models, patterns, and tools**. [S.l.]: Addison-Wesley, 2001. 14, 15, 16, 18, 21, 22

BLUM, A.; CECHTICKY, V.; PASETTI, A.; SCHAUFELBERGER, W. A java-based framework for real-time control systems. In: IEEE CONF. EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, 2., 2003, Lisbon, Portugal. **Proceedings...** lisbon, 2003. v. 2, p. 447–453. 13, 47

BLUNDELL, C.; GIANNAKOPOULOU, D.; PASAREANU, C. S. Assume-guarantee testing. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 31, September 2005. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/1108768.1123060>. 18

BOCHMANN, G.; PETRENKO, A. Protocol testing: Review of methods and relevance for software testing. **Proc. Int l Symposium on Software Testing and Analysis**, p. 109–124, 1994. 16

BOGDAN, H. K.; WISZNIEWSKI, B.; MORK, P. **Classification of software defects in parallel programs**. [S.l.: s.n.], 1994. 26

BRIEB, K.; BARWALD, W.; LURA, F.; MONTENEGRO, S.; OERTEL, D.; STUDEMUND, H.; SCHLOTZHAUER, G. The bird mission is completed for launch with the pslv-c3 in 2001. In: IAA SYMPOSIUM ON SMALL SATELLITES FOR EARTH OBSERVATION, 3., 2001, Berlin, Germany. **Proceedings...** Berlin, 2001. 25

BUDKOWSKI, S.; DEMBINSKI, P. An introduction to estelle: a specification language for distributed systems. **Comput. Netw. ISDN Syst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 14, n. 1, p. 3–23, 1987. ISSN 0169-7552. 18

CHAIM, M. L. **POKE-TOOL - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados**. Dissertação (Mestrado) — DCA/FEE/UNICAMP, Campinas,SP,Brasil, abr. 1991. 8

CHOW, T. Testing software desing modeled by finite-state machines. **IEEE Transaction on Software Engineering**, v. 4, n. 3, p. 178–187, may 1978. 16

CHUNG, I. S.; KIM, H. S.; BAE, H. S.; KWON, Y. R.; LEE, B. S. Testing of concurrent programs based on message sequence charts. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE ENGINEERING FOR PARALLEL AND DISTRIBUTED SYSTEMS, 99., 1999, Washington, USA. **Proceedings...** Washington: IEEE Computer Society, 1999. (PDSE '99), p. 72–82. ISBN 0-7695-0191-5. Disponível em: <http://portal.acm.org/citation.cfm?id=554221.826291>. 27

CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. **Model checking**. [S.l.]: MIT Press, 1999. 18

CORBETT, J. C.; DWYER, M. B.; HATCLIFF, J.; LAUBACH, S.; PASAREANU, C. S.; ZHENG, H. Bandera: Extracting finite-state models from java source code. In: ACM INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22., 2000, San Francisco Bay, USA. **Proceedings...** San Francisco Bay: ACM Press, 2000. p. 439–448. 10, 27, 54

CURTIS, S. A. **Evolvable Neural Software System**. [S.l.: s.n.], 2009. 24

DAUTELLE, J.-M. Fully time deterministic java. In: AIAA SPACE CONFERENCE AND EXPOSITION, 2007, Long Beach, California. **Proceedings...** Long Beach, 2007. 13

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. [S.l.]: Elsevier, 2007. 5, 7, 8, 19

DOWSON, M. The ariane 5 software failure. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 22, n. 2, p. 84, 1997. ISSN 0163-5948. 2

DVORAK, D.; BOLLELLA, G.; CANHAM, T.; CARSON, V.; CHAMPLIN, V.; GIOVANNONI, B.; INDICTOR, M.; MEYER, K.; MURRAY, A.; REINHOLTZ, K. Project golden gate: towards real-time java in space missions. In: IEEE INT OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING SYMP, 17., 2004, Vienna, Austria. **Proceedings...** Vienna, 2004. p. 15–22. 13

ELECTRICAL, I. O.; (IEEE), E. E. **IEEE Standard Glossary of software engineering terminology Standard (610-12)**. [S.l.: s.n.], 1990. 3

_____. **Standard for Software Test Documentation (IEEE-829)**. [S.l.: s.n.], 1998. 4

EZEKIEL, J.; LüTTGEN, G.; SIMINICEANU, R. I. **A Parallel Saturation Algorithm on Shared Memory Architectures**. [S.l.: s.n.], 2007. 24

FONSECA, R. P. da. **Suporte ao teste estrutural de programas Fortran no ambiente POKE-TOOL**. Dissertação (Mestrado) — DCA/FEEC/UNICAMP, Campinas, SP, Brasil, jan. 1993. 8

FUHRER, R. **Sequential optimization of asynchronous and synchronous finite-state machines: algorithms and tools**. Tese (Doutorado) — Columbia University, Graduate School of Arts and Sciences, 1999. 16

GILL, E.; MONTENBRUCK, O. Spaceborne autonomous navigation for the bird satellite mission. In: **ESA Workshop on On-Board Autonomy**. [S.l.: s.n.], 2001. 25

GILL, E.; MONTENBRUCK, O.; BRIEB, K. Gps-based autonomous navigation for the bird satellite. In: **15th International Symposium on Spaceflight Dynamics**. [S.l.: s.n.], 2000. 25

HAREL, D. Statecharts: A visual formalism for complex systems. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 8, n. 3, p. 231–274, 1987. ISSN 0167-6423. 15, 18

HAREL, D.; POLITI, M. **Modeling reactive systems with Statecharts: the statemate approach**. New York, NY, USA: McGraw-Hill, Inc., 1998. ISBN 0070262055. 18, 19, 20

HONG, H. S.; KWON, Y. R.; CHA, S. D. Testing of object-oriented programs based on finite state machines. **Asia-Pacific Software Engineering Conference**, p. 234 – 241, 1995. 26

HOWDEN, W. E. Reliability of the path analysis testing strategy. n. 3, p. 208–215, 1976. 26

(ISO), I. O. F. S. **Information Technology Software Process Assessment (ISO/IEC-15504)**. [S.l.: s.n.], 2003. 3

JAMES, M. **Integrated Hardware and Software for No-Loss Computing**. [S.l.: s.n.], 2007. 24

JUNIOR, P. de S. L. **Suporte ao teste estrutural de programas Cobol no ambiente POKE-TOOL**. Dissertação (Mestrado) — DCA/FEEC/UNICAMP, Campinas, SP, Brasil, ago. 1992. 8

KIT, E.; FINZI, S. **Software testing in the real world: improving the process**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. ISBN 0-201-87756-2. 6

KIZHNER, S.; FLATLEY, T. P.; HESTNES, P.; JENTOFT-NILSEN, M.; PETRICK, D. J. Pre-hardware optimization of spacecraft image processing software algorithms and hardware implementation. **Aerospace Conference Proceedings, 2002. IEEE**, v. 4, p. 1975–1992, 2002. 25

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines-a survey. v. 84, n. 8, p. 1090–1123, 1996. 16, 17

LOU, J.; BEDDING, D.; BASINGER, S. **Optics Program Modified for Multithreaded Parallel Computing**. [S.l.: s.n.], 2006. 26

LUO, G.; BOCHMANN, G. V.; PETRENKO, R. Test selection based on communicating nondeterministic finite state machines using a generalized wp-method. **IEEE Trans.**, Dept. d'Inf. et de Recherche Oper., Montreal Univ., Que., v. 20, n. 2, p. 149–162, 2006. 27

MALDONADO, J. C. **Critérios potenciais usos: uma contribuição ao teste estrutural de software**. Tese (Doutorado) — DCA/FEEC/UNICAMP, 1991. 7

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Arquitetura de uma ferramenta de teste de apoio aos critérios potencias usos. **XXII Congresso Nacional de Informática**, São Paulo, Set. 1989. 8

MARTINS, E.; aO, S. B. S.; AMBROSIO, A. M. Condata: a tool for automating specification-based test case generation for communication systems. In: IEEE HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 33., 2000, Washington, USA. **Proceedings...** Washington: IEEE Computer Society, 2000. (HICSS '00, v. 8), p. 8012–8028. ISBN 0-7695-0493-0. 23

MORELL, L.; MURRILL, B.; RAND, R. Perturbation analysis of computer programs. p. 77–87, 1997. 6

MYERS, G. J. **The art of software testing**. New York: John Wiley and Sons, 1979. 3, 5, 6, 16

PRESSMAN, R. S. **Engenharia de software**. Rio de Janeiro: McGraw-Hill, Inc., 2006. 1, 3, 4, 7

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, v.11 n.4, p. 367–375, 1985. 7

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARAES, D.; AMARAL, A. S.; FERREIRA, E. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION AND VALIDATION WORKSHOP, 2008, Washington, USA. **Proceedings...** Washington: IEEE Computer Society, 2008. p. 63–72. ISBN 978-0-7695-3388-9. Disponível em: <http://portal.acm.org/citation.cfm?id=1439281.1440221>. 23

SARMANHO, F. S. **Teste de programas concorrentes com memória compartilhada**. Dissertação (Mestrado) — ICMC/USP, 2010. 10, 28, 32, 63, 64

SEO, H.-S.; CHUNG, I. S.; KIM, B. M.; KWON, Y. R. The design and implementation of automata-based testing environment for java multi-thread programs. **Asia-Pacific Software Engineering Conference**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 221, 2001. 27

SEO, H.-S.; CHUNG, I. S.; KWON, Y. R. Generating test sequences from statecharts for concurrent program testing. **IEICE - Trans. Inf. Syst.**, Oxford University Press, Oxford, UK, E89-D, p. 1459–1469, April 2006. ISSN 0916-8532. Disponível em: <http://portal.acm.org/citation.cfm?id=1184857.1184986>. 27

SHAN, L.; ZHU, H. Testing software modelling tools using data mutation. In: ACM INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, 2006, Shanghai, China. **Proceedings...** Shanghai: ACM, 2006. (AST '06), p. 43–49. ISBN 1-59593-408-1. Disponível em: <http://doi.acm.org/10.1145/1138929.1138938>. 6

SIDHU, D. P.; LEUNG, T.-k. Formal methods for protocol testing: A detailed study. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 15, p. 413–426, April 1989. ISSN 0098-5589. Disponível em: <http://portal.acm.org/citation.cfm?id=63379.63384>. 16, 17

SOUZA, E. F. **Geração de casos de teste para sistemas da área espacial usando critérios de teste para máquinas de estados finitos**. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais, 2009. 6, 17, 23

SOUZA, S. R. S.; FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C. Statecharts specifications: A family of coverage testing criteria. **XXVI Conferência Latinoamericana de Informática (CLEI 2000)**, Monterrey, a, p. 18–22, 2000. 24

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMãO, A. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 20, p. 1893–1916, November 2008. ISSN 1532-0626. Disponível em: <http://portal.acm.org/citation.cfm?id=1455689.1455692>. 9, 32, 45

STOLLER, S. D. Model-checking multi-threaded distributed java programs. In: INTERNATIONAL SPIN WORKSHOP ON SPIN MODEL CHECKING AND SOFTWARE VERIFICATION, 7., 2000, London, UK. **Proceedings...** London: Springer-Verlag, 2000. p. 224–244. ISBN 3-540-41030-9. Disponível em: <http://dl.acm.org/citation.cfm?id=645880.672084>. 27

SUGDEN, P. P.; RAU, M. A.; KENNEY, P. S. Platform-independence and scheduling in a multi-threaded real-time simulation. In: **AIAA Modeling and Simulation Technologies Conference and Exhibit**. [S.l.: s.n.], 2001. 24

SURKA, D. M.; BRITO, M. C.; HARVEY, C. G. The real-time objectagent software architecture for distributed satellite systems. In: IEEE PROCEEDINGS AEROSPACE CONFERENCE, 7., 2001, Big Sky, Montana. **Proceedings...** Big Sky, 2001. 25

TAI, K. Testing of concurrent software. **Proceedings of the 13th Annual International. Computer Software and Applications Conference.**, p. 62 – 64, 1989. 26

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems, principles and paradigms**. [S.l.]: Printice Hall, 2007. 11, 12

TAYLOR, R. N.; LEVINE, D. L.; KELLY, C. D. Structural testing of concurrent programs. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 18, p. 206–215, March 1992. ISSN 0098-5589. Disponível em: <http://portal.acm.org/citation.cfm?id=129809.129812>. 26

THOMPSON, S.; BRAT, G.; VENET, A. Software model checking of arinc-653 flight code with mcp. In: MUÑOZ, C. (Ed.). **Proceedings...** Langley Research Center, 2010. p. 171–181. 24

USMAN, M.; NADEEM, A.; KIM, T.-h.; CHO, E.-s. A survey of consistency checking techniques for uml models. In: IEEE ADVANCED SOFTWARE ENGINEERING AND ITS APPLICATIONS, 2008, Hainan Island, China. **Proceedings...** Hainan Island: IEEE Computer Society, 2008. p. 57–62. ISBN 978-0-7695-3432-9. 14

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing. **Working Paper Series**, The University of Waikato, Hamilton, New Zealand, 2006. ISSN 1170-487X. 29

VEJRAZKA, F.; KOVAR, P.; SEIDL, P. K. L. Experimental software receiver of signals of satellite navigation systems. In: IAIN WORLD CONGRESS ON SMART NAVIGATION SYSTEMS AND SERVICES, 11., 2003, Berlin, Germany. **Proceedings...** Berlin, 2003. 25

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Constraint based criteria: An approach for test case selection in the structural testing. **J. Electron. Test.**, Kluwer Academic Publishers, Norwell, MA, USA, v. 17, n. 2, p. 175–183, 2001. ISSN 0923-8174. 6

VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. de. Coverage testing criteria for message-passing parallel programs. **6th IEEE Latin-American Test Workshop (LATW 2005)**, IEEE, v. 1, n. 2, p. 161–166, 2005. 27, 31

VICENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C. **JaBUTi Java Bytecode Understanding and Testing**. São Carlos, SP, Brasil: [s.n.], 2003. 8

VIJAYKUMAR, N. L. **Statecharts: their use in specifying and dealing with performance models**. Tese (Doutorado) — Instituto Tecnológico de Aeronáutica, São José dos Campos, 1999. 15, 22, 51

WEYUKER, E. J. The complexity of data flow criteria for test data selection. **Inf. Process. Lett.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 19, n. 2, p. 103–109, 1984. ISSN 0020-0190. 7

YANG, C.-S.; POLLOCK, L. L. The challenges in automated testing of multithreaded programs. In: IN PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON TESTING COMPUTER SOFTWARE, 14., 1997, Melbourne, Victoria. **Proceedings...** Melbourne, 1997. p. 157–166. 1, 2

YANG, C.-S. D.; POLLOCK, L. L. All-uses testing of shared memory parallel programs. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd., v. 13, n. 1, p. 3–24, 2003. ISSN 1099-1689. Disponível em: <http://dx.doi.org/10.1002/stvr.262>. 2

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 29, n. 4, p. 366–427, 1997. ISSN 0360-0300. 6

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.