MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/12.21.23.15-TDI

MINIMIZAÇÃO DE CASOS/PASSOS DE TESTE PARA LINGUAGENS DE PROGRAMAÇÃO VISUAL

Paulo Nolberto dos Santos Alarcon

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 14 de dezembro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3Q8M792>

INPE
São José dos Campos
2017

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GBDIR)
Serviço de Informação e Documentação (SESID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
E-mail: pubtc@inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/12.21.23.15-TDI

MINIMIZAÇÃO DE CASOS/PASSOS DE TESTE PARA LINGUAGENS DE PROGRAMAÇÃO VISUAL

Paulo Nolberto dos Santos Alarcon

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 14 de dezembro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3Q8M792>

INPE
São José dos Campos
2017

Dados Internacionais de Catalogação na Publicação (CIP)

Alarcon, Paulo Nolberto dos Santos.

Al12m Minimização de casos/passos de teste para linguagens de programação visual / Paulo Nolberto dos Santos Alarcon. – São José dos Campos : INPE, 2017.

xxii + 63 p. ; (sid.inpe.br/mtc-m21b/2017/12.21.23.15-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2017.

Orientador : Dr. Valdivino Alexandre de Santiago Júnior.

1. Teste de software. 2. Minimizacão de casos de teste. 3. Verificação formal. 4. Linguagens de programação visual. 5. Satélites. I.Título.

CDU 004.415.26

Aluno (a): ***Paulo Nolberto dos Santos Alarcon***

Título: **"MINIMIZAÇÃO DE CASOS/PASSOS DE TESTE PARA LINGUAGENS DE PROGRAMAÇÃO VISUAL"**

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

Computação Aplicada

Dra. **Karine Reis Ferreira**

Presidente / INPE / São José dos Campos - SP

Dr. **Valdivino Alexandre de Santiago Júnior**

Orientador(a) / INPE / São José dos Campos - SP

Dr. **Evandro Marconi Rocco**

Membro da Banca / INPE / SJCampos - SP

Dr. **Álvaro Luiz Fazenda**

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

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

São José dos Campos, 14 de dezembro de 2017

"Saber muito não lhe torna inteligente. A inteligência se traduz na forma que você recolhe, julga, maneja e, sobretudo,onde e como aplica essa informação".

Carl Sagan

Dedicatoria

Dedico este trabalho aos meus pais por todo o esforço que fizeram para me prover educação de qualidade.

Dedico este trabalho também aos meus avós que não tiveram as mesmas oportunidades que eu tive, porém me ensinaram lições valiosíssimas.

Dedico este trabalho também à minha noiva Isabela Caroline Silva Rodrigues, por ser a pessoa que esteve sempre ao meu lado, me apoiando e cuidando de mim.

AGRADECIMENTOS

Agradeço acima de tudo a Deus.

Agradeço a minha família pelo apoio recebido.

Agradeço em especial a minha noiva Isabela Caroline Silva Rodrigues por me suportar e estar sempre ao meu lado ao longo de todo o curso. Por me ajudar e cuidar de mim, mesmo nos momentos em que negligenciei minha própria saúde.

Agradeço ao meu orientador, Professor Valdivino, pela paciência, ajuda no desenvolvimento deste trabalho e todo o conhecimento que tem me passado.

Agradeço à FITec e à Johnson & Johnson por me apoiarem neste curso e me liberarem para as aulas.

Agradeço aos Dr. Adenilson Roberto da Silva e Dr. Fabrício Galende Marques de Carvalho por permitirem o uso do estudo de caso.

E finalmente, agradeço a todos os professores pelos conhecimentos adquiridos ao longo de toda essa jornada.

RESUMO

Linguagens de Programação Visual têm sido amplamente utilizadas no contexto de Desenvolvimento/Projeto Baseado em Modelos, e as mesmas encontram um apelo particular para o desenvolvimento de subsistemas de satélites e outras aplicações espaciais, tais como o Subsistema de Controle de Órbita e Atitude (SCOA) e o Subsistema de Controle de Atitude (SCA). A modelagem de um SCA é complexa, e assegurar a qualidade desse tipo de modelagem é algo muito desafiador. Por sua vez, a comunidade de teste de software vem colocando esforços para garantir que produtos tenham alta qualidade, possuindo o menor número possível de defeitos. Dado que a geração e execução exaustivas de casos de teste para software é inviável na prática, uma das iniciativas é reduzir os conjuntos de casos de testes necessários para se testar um Software/Sistema Sob Teste, mas que ainda tenha uma efetividade (habilidade de encontrar defeitos no produto, cobertura de código) significativa. Essa dissertação de mestrado apresenta um novo método para gerar casos de teste para modelos de Linguagem de Programação Visual, objetivando minimizar o conjunto de casos/passos de teste requeridos e, ao mesmo tempo, aumentando a efetividade do mesmo. A abordagem, denominada specification Patterns, modified Condition/Decision coverage, and formal Verification to support Testing (PCDVT), combina o critério de Cobertura de Decisões/Condições Modificadas (MC/DC), Verificação Formal (Model Checking), padrões de especificação, e abordagem de minimização por identificação de testes insubstituíveis em um único método, tirando proveito dos diversos benefícios de todos esses esforços em uma abordagem unificada. Duas avaliações experimentais foram conduzidas, considerando como estudos de caso modelos de Linguagens de Programação Visual para SCA similar ao do satélite Lattes-1 do Instituto Nacional de Pesquisas Espaciais (INPE). Em uma primeira avaliação experimental, que considerou um modelo SCA de complexidade intermediária, resultados demonstraram que duas instâncias do PCDVT, uma baseada no padrão de ausência e outra baseada no padrão de resposta encadeada, tiveram custo (quantidade de passos de teste) similar, mas uma efetividade (cobertura do modelo) significativamente maior se comparadas a uma abordagem manual e ad hoc. Dessa forma, pôde ser verificado um melhor desempenho do método PCDVT comparado a uma abordagem manual. Em uma segunda avaliação experimental, onde o modelo SCA completo foi considerado, percebeu-se um custo (quantidade de passos de teste) consideravelmente melhor se for comparada a geração de casos de teste via PCDVT com e sem a aplicação do algoritmo de minimização proposto no método.

Palavras-chave: Teste de Software. Minimizacão de Casos de Teste. Verificação Formal. Linguagens de Programação Visual. Satélites. Subsistema de Controle de Atitude.

TEST CASE/STEP MINIMIZATION FOR VISUAL PROGRAMMING LANGUAGE MODELS

ABSTRACT

Visual Programming languages have been widely used in the context of Development/Model-Based Design, and they find particular appeal for the development of satellite subsystems, such as the Attitude and Orbit Control Subsystem (AOCS) and the Attitude Control Subsystem (ACS). Modeling a ACS is complex, and ensuring the quality of this type of system is very challenging. In turn, the software testing community has been making efforts to ensure that products present high quality, with as few defects as possible. Since the exhaustive software test case generation and execution is actually impractical, one of the initiatives is to reduce the sets of test cases required to test a Software/System Under Test but still keeping the effectiveness (ability to find product defects , code coverage). This dissertation presents a new method to generate test cases for Visual Programming Language models, aiming to minimize the required set of test cases/steps and, at the same time, increasing its effectiveness. The approach, called specification Patterns, modified Condition/Decision coverage, andformal Verification to support Testing (PCDVT), combines the Modified Condition/Decision Coverage (MC/DC), Model Checking, specification patterns, and minimization by identifying irreplaceable tests in a single method, taking advantage of the many benefits of all these efforts in a unified approach. Two experimental evaluations were conducted, considering as case studies ACS models similar to the Lattes-1 satellite of the National Institute for Space Research (INPE). In a first experimental evaluation, which ACS model of intermediate complexity was considered, results demonstrate that two instances of the PCDVT, a default basead on the absence absence pattern and another based on the chain response pattern, had similar cost (number of test steps), but one effectiveness (model coverage) significantly higher compared to a manual and ad-hoc approach. In this way, a better performance of the PCDVT method could be verified when compared with a manual approach. In a second experimental evaluation, where the complete ACS model was considered, a considerably better cost (number of test steps) was perceived compared to the generation of test cases via PCDVT with and without the application of the minimization algorithm proposed in the method.

Keywords: Software testing. Minimization test cases. Formal Verification. Visual Programming Languages. Satellites. Attitude Control Subsystem.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

ECD	–	Estados Com Decisão
eIET	–	*eIrreplace-able for Early Tests*
EPM	–	*Earth Pointing Mode*
ESD	–	Estados Sem Decisão
FDIR	–	*Failure Detection Isolation and Recovery*
GE	–	*Greedy Essential Algorithm*
GRE	–	*Greedy Redundant Essential Algorithm*
INPE	–	Instituto Nacional de Pesquisas Espaciais
JSON	–	*JavaScript Object Notation*
LPV	–	Linguagem de Progamação Visual
LTL	–	Lógica Temporal Linear
MC/DC	–	*Modified Condition/Decision Coverage*
mtype	–	*message type*
NuSMV	–	*New Symbolic Model Verifier*
PCDVT	–	*specification Patterns, modified Condition/Decision coverage, and formal Verification to support Testing*
PCDVT-A	–	Instância do método PCDVT criada usando o padrão de especificação de Ausência em escopo entre Q e R
PCDVT-RC	–	Instância do método PCDVT criada usando o padrão de especificação de Resposta encadeada em escopo global
PROMELA	–	*PROcess or PROtocol MEta LAnguage*
SCA	–	Subsistema de Controle de Atitude
SCOA	–	Subsistema de Controle de Órbita e Atitude
SPIN	–	*Simple Promela INterpreter*
SPM	–	*Sun Pointing Mode*
SST	–	Sistema Sob Teste
TBM	–	Teste Baseado em Modelos
XML	–	*eXtended Markup Language*
XPATH	–	*XML Path Language*

SUMÁRIO

Pág.

1 INTRODUÇÃO

A garantia da qualidade é imprescindível para avanços tecnológicos e aprimoramento das tecnologias, visto que, defeitos em diversos tipos de aplicações, podem implicar em grandes perdas financeiras e até mesmo de vidas humanas. Uma área de estudo que tem se destacado nesse sentido é a engenharia espacial, cujos os sistemas desenvolvidos apresentam alta complexidade(WERTZ; LARSON, 1999). Na engenharia de sistemas espaciais, a garantia da qualidade representa uma etapa importante no desenvolvimento, devido à natureza crítica, complexa e ao alto custo de sistemas espaciais (e.g. satélites, aplicações de balão, ...). Assim, o desenvolvimento de metodologias, métodos e técnicas que assegurem a qualidade de tais produtos antes do lançamento é altamente necessário (PISACANE, 2005).

Um dos subsistemas mais importantes que compõe aplicações espaciais, tais como satélites, é o Subsistema de Controle de Órbita e Atitude (SCOA) (WERTZ; LARSON, 1999). O SCOA deve prover informações de atitude e manter a adequada atitude e órbita do satélite durante todas as fases da missão, desde a separação do satélite do lançador até a sua vida operacional. Quando o subsistema controla apenas atitude, então este é denominado Subsistema de Controle de Atitude (SCA).

Devido a natureza crítica de uma aplicação como um satélite, onde defeitos podem causar grandes perdas financeiras, é de extrema importância assegurar a qualidade do sistema como um todo. Consequentemente, o SCA deve possuir alta qualidade e robustez. E uma das formas de se obter a qualidade de um sistema é por meio de processos bem definidos tal como o processo de **Teste de Software**. No contexto de software, Teste (AMMANN; OFFUTT, 2017; MATHUR, 2008; DELAMARO et al., 2007) é um dos vários processos relacionadas à Verificação & Validação (SANTIAGO JÚNIOR, 2011).

1.1 Motivação

O processo de Teste de Software é composto por várias atividades, sendo que uma das mais estudadas é a geração de casos de teste. No fundo, a atividade de geração de casos de teste para software tenta responder a seguinte questão: dado que a geração/execução de casos de teste exaustiva é inviável (SANTIAGO JÚNIOR et al., 2008), de que maneira pode-se selecionar os casos de teste de forma a se obter um conjunto de casos de teste que seja eficaz e efetivo?

Existe uma quantidade significativa de iniciativas na comunidade acadêmica e na

indústria com diferentes técnicas para gerar casos de teste, desde as mais clássicas como particionamento por classes de equivalência e análise do valor limite (MYERS, 2004), passando por muitas outras como teste aleatório (PACHECO et al., 2007), teste de interação combinatória (KUHN et al., 2013; PETKE et al., 2015; BALERA; SANTIAGO JÚNIOR, 2015), teste de mutação (DELAMARO et al., 2007), teste caixa branca (MYERS, 2004), otimização em Teste de Software (HARMAN et al., 2015), Teste Baseado em Modelos (TBM) (UTTING; LEGEARD, 2007; DIAS NETO, 2009), entre outras.

Em particular, TBM tem chamado bastante atenção da comunidade científica e da indústria. Em TBM, casos de teste são gerados parcial ou completamente a partir de um modelo que descreve algum aspecto (e.g. funcionalidade, desempenho, etc.) de um produto de software. Importante enfatizar que esse modelo pode ser realizado via uma linguagem/modelo formal tais como Z (HELKE et al., 1997), B (BERNARD et al., 2004), Statecharts (HAREL, 1987; SANTIAGO JÚNIOR, 2011) e Máquinas de Estados Finitos (HIERONS et al., 2009; ALARCÓN et al., 2014), ou pode ser elaborado em notações não formais como a Unified Modeling Language (LI et al., 2007), ou mesmo via Linguagens de Programação Visual (LPVs) tais como SciLab/Xcos (JANÍK; ŽÁKOVÁ, 2011) e Simulink (HANSELMAN; LITTLEFIELD, 2005).

Independentemente da estratégia escolhida para gerar casos de teste, um problema que existe é a explosão de casos de teste. Em outras palavras, se são considerados modelos bastante detalhados do sistema, a quantidade de casos de teste que devem ser executados é consideravelmente grande (WANG et al., 2013), inviabilizando que estes casos de teste possam ser executados em tempo hábil.

Portanto, muitos estudos objetivam minimizar a quantidade de casos de teste necessários para serem executados (CAMPOS; ABREU, 2013; FERRANTE et al., 2014a; FRASER; WOTAWA, 2007; LIN et al., 2014; YOO; HARMAN, 2012). Em outras palavras, o problema da minimização de *suites* de teste (minimização de conjuntos de casos de teste) é diminuir a quantidade necessária de casos de teste que precisam ser executados pela eliminação daqueles que são redundantes.

Entretanto, a minimização de casos de teste tem sido explorada na literatura mais no contexto de testes de regressão (YOO; HARMAN, 2012). Portanto, é interessante investigar e propor novas soluções que considerem que o produto está sendo testado pela primeira vez.

LPVs permitem o desenvolvimento de sistemas, modelos de forma totalmente visual

ao invés da codificação textual, via o agrupamento/conexão de blocos com variadas funções. Uma LPV permite programar com expressões visuais, arranjos espaciais de texto e símbolos gráficos, usados ou como elementos da sintaxe ou notação secundária. Por exemplo, muitas LPVs são baseadas na ideia de "blocos e setas", onde os blocos, ou outros objetos, são tratados como entidades e conectados por meio de setas, linhas ou arcos que representam relações.

Dado que sistemas/modelos complexos podem ser desenvolvidos via LPVs, é relevante propor novas abordagens para assegurar a qualidade de tais modelos. Desse modo, propor novos métodos para gerar conjuntos de casos de teste que sejam mínimos mas que possam, ao mesmo tempo, ter alta efetividade e baseados em modelos elaborados via LPVs, é algo relevante de ser investigado. Particularmente no contexto aeroespacial, o Instituto Nacional de Pesquisas Espaciais (INPE) vem desenvolvendo satélites que fazem uso de abordagens baseadas em modelos usando LPVs.

1.2 Objetivo e Metodologia de Pesquisa

Esse trabalho tem como objetivo dar uma contribuição ao problema da minimização de casos de teste para modelos elaborados em LPVs.

Para isso, é proposto um novo método (ALARCON; SANTIAGO JÚNIOR, 2017), denominado specification Patterns, modified Condition/Decision coverage, and formal Verification to support Testing (PCDVT - Padrões de Especificação, Cobertura de Decisões/Condições Modificada, e Verificação Formal para apoiar o Teste), para a geração de casos/passos de testes a partir de modelos LPV garantindo a Cobertura de Decisões/Condições Modificada (MC/DC) dos mesmos. Além disso, o conjunto de casos de teste deverá ser o menor possível, de forma que haja menos esforço para que seja executado totalmente.

O PCDVT combina o critério MC/DC (KAPOOR; BOWEN, 2003), Verificação Formal (*Model Checking* (BAIER; KATOEN, 2008)), padrões de especificação (DWYER et al., 1998), e abordagem de minimização por identificação de testes insubstituíveis (LIN et al., 2014) em um único método, tirando proveito dos diversos benefícios de todos esses esforços em uma abordagem unificada.

Além disso, o método PCDVT foi implementado, em uma ferramenta computacional de mesmo nome (PCDVT), de forma que a geração de casos de teste seja feita de maneira automatizada.

Duas avaliações experimentais foram conduzidas, considerando como estudos de

caso modelos LPV de um SCA para controle de atitude em 1 eixo, similar ao do satélite Lattes-1 do INPE. A avaliação experimental foi efetuada considerando-se a efetividade, custo e eficiência das suítes de teste geradas pelo PCDVT. A efetividade foi avaliada via a execução dos testes e medição da cobertura MC/DC obtida. O custo consiste no número total de passos de teste da suíte de teste. A eficiência foi medida por meio do percentual de redução do custo da suíte de testes, antes e depois do processo de minimização.

Em uma primeira avaliação experimental, que considerou um modelo SCA de complexidade intermediária, resultados demonstraram que duas instâncias do PCDVT, uma baseada no padrão de ausência e outra baseada no padrão de resposta encadeada, tiveram custo (quantidade de passos de teste) similar, mas uma efetividade (cobertura do modelo) significativamente maior se comparadas a uma abordagem manual e ad hoc. Dessa forma, pôde ser verificado um melhor desempenho do método PCDVT comparado a uma abordagem manual. Em uma segunda avaliação experimental, onde o modelo SCA completo foi considerado, percebeu-se um custo (quantidade de passos de teste) consideravelmente melhor se for comparada a geração de casos de teste via PCDVT, com e sem a aplicação do algoritmo de minimização proposto no método.

1.3 Contribuições e Limitações

As principais contribuições dessa pesquisa são as seguintes:

- Proposta de um novo método, PCDVT, pela combinação de diversas áreas de conhecimento, para gerar conjuntos de casos de teste para modelos criados em LPVs, os quais são viáveis de serem executados (custo viável) e com uma alta cobertura MC/DC (efetividade), se comparado com abordagens ad hoc;
- No contexto do método proposto, uso de *Model Checking* para geração de casos de teste, o que é interessante dado a verificação exaustiva no espaço de estados que *Model Checking* realiza;
- No contexto do método PCDVT, adaptação da técnica de minimização por identificação de testes insubstituíveis (LIN et al., 2014) com a criação de um novo algoritmo;
- No contexto do método proposto, uso de padrões de especificação para simplificar o processo de geração de casos testes por meio de *Model Checking*,

apresentando uma descrição clara e padronizada da propriedade verificada;

- No contexto do PCDVT, uso de MC/DC (critério de teste caixa branca para desenvolvimento de software aviônico de acordo com a norma DO-178C (HOLLOWAY, 2012));
- Realização de duas avaliações experimentais para avaliar o desempenho do PCDVT, considerando estudos de caso da área espacial;
- Implementação do método PCDVT em uma ferramenta, também denominada PCDVT, para apoiar a automatização da geração de casos de teste de software para modelos LPV.

Como limitações, pode-se mencionar o fato de que a ferramenta PCDVT, no estado atual, ainda é incapaz de processar decisões em algumas classes de blocos presentes em algumas LPVs, e também é incapaz de fazer a análise MC/DC de blocos contendo *scripts*. Essas limitações são discutidas em maiores detalhes no Capítulo 4 .

1.4 Organização do texto

A estruturação do texto relacionada a essa dissertação de mestrado é descrita a seguir:

a) Capítulo 2. Nesse capítulo, apresenta-se conceitos relacionados ao processo de Teste de Software com foco na atividade de geração de casos de teste baseado em modelos (particularmente via *Model Checking*), critério MC/DC, minimização de casos/passos de teste onde, nesse caso, são salientados os pontos fortes e fracos das estratégias para minimização de casos de teste. Padrões de especificação também são apresentados nesse capítulo;

b) Capítulo 3. Nesse capítulo apresenta-se, em detalhes, o método PCDVT;

c) Capítulo 4. Nesse capítulo é mostrado um detalhamento da implementação do método PCDVT por meio da linguagem GO. Além disso, são mostradas as duas avaliações experimentais para avaliar o desempenho do PCDVT, considerando modelos LPV de SCA similar ao do satélite Lattes-1 do INPE;

d) Capítulo 5. As conclusões dessa pesquisa e trabalhos futuros são apresentados nesse capítulo.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Teste de Software

Teste de Software é o processo de investigação do sistema a fim de encontrar defeitos no mesmo (PRESSMAN, 2011). Os testes de software podem ser divididos em duas grandes classes, conhecidas como testes de caixa branca e testes de caixa preta.

O teste de caixa branca é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste (PRESSMAN, 2011). Usando métodos de teste caixa-branca, o engenheiro de software pode criar casos de teste que garantam que todos os caminhos independentes de um módulo foram exercitados ao menos uma vez, exercitam todas as decisões lógicas nos seus estados verdadeiro e falso, executam todos os ciclos em seus limites e dentro de suas fronteiras operacionais, e exercitam estruturas de dados internas para assegurar a sua validade. Essas técnicas trabalham diretamente sobre o código fonte do componente de software para avaliar aspectos tais como teste de condição, teste de fluxo de dados, teste de ciclos, teste de caminhos lógicos, códigos nunca executados, entre outros.

Testes caixa preta, também conhecidos como testes funcionais, avaliam o comportamento externo do componente de software, sem se considerar o comportamento interno do mesmo. As técnicas de teste caixa-preta permitem derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa (PRESSMAN, 2011). Devido ao teste caixa-preta propositadamente desconsiderar a estrutura de controle, a atenção é focalizada no domínio das informações.

O teste caixa preta é classificado em diversos níveis de dentre os quais se destacam:

- Teste Unitário: Busca exercitar as entradas e saídas das unidades do sistema, isto é, a menor parte testável do sistema;

- Teste de Integração: Exercita as interfaces e o funcionamento conjunto de mais de um módulo;

- Teste de Sistema: Exercita o comportamento do sistema de acordo com seus requisitos num ambiente de produção. Está no escopo da técnica de teste de caixa-preta, e dessa forma não requer conhecimento da estrutura interna do sistema;

- Teste de Regressão: Busca verificar se o comportamento implementado anteriormente permanece com a adição de código novo.

2.1.1 Critério MC/DC

MC/DC (*Modified Condition/Decision Coverage*) é um critério de cobertura de código desenvolvido no contexto dos testes estruturais com o objetivo de garantir que todas as condições e decisões sejam executadas ao menos uma vez durante o processo de verificação, maximizando o código coberto durante os testes. Este critério é oriundo do teste caixa-branca sendo consequentemente dependente da implementação do sistema (AWEDIKIAN et al., 2009). O critério MC/DC foi criado com a ideia de que uma maior cobertura de código aumenta as chance de se encontrar defeitos, porém a cobertura total do código para todas os possíveis cenários é muitas vezes impossível. Portanto busca-se reduzir o número de cenários selecionando-se somente os mais relevantes levando-se em conta as decisões presentes no código.

Tabela 2.1 - Exemplo de geração de casos de teste usando o critério MC/DC.

A	B	C	$(A \vee B) \wedge C$	
T	T	T	T	
F	T	T	T	Teste 1
T	F	T	T	Teste 2
F	F	T	F	Teste 3
T	T	F	F	
F	T	F	F	
T	F	F	F	Teste 4
F	F	F	F	

Fonte: Adaptado de Kapoor e Bowen (2003)

Cada ponto de entrada e saída no programa deve ser invocado ao menos uma vez, toda condição em uma decisão no programa deve assumir todos valores possíveis ao menos uma vez, e cada condição que independentemente afete a saída de uma decisão deve assumir cada valor possível ao menos uma vez (KAPOOR; BOWEN, 2003). A Tabela 2.1 mostra um exemplo de geração de casos de teste usando o critério MC/DC, onde para a decisão $(A \vee B) \wedge C$, C pode ser classificada como a condição que afeta independentemente a decisão e então A e B são variados para explorar as possíveis saídas. Assim, são gerados 4 casos de teste, o que é metade das possíveis combinações.

2.1.2 Teste Baseado em Modelos

TBM é uma técnica onde se utiliza de alguma forma de especificação formal de um sistema, tipicamente descrita em linguagem matemática ou grafos, para se obter casos de teste. O modelo descrevendo o comportamento do SST é usualmente uma representação abstrata ou parcial do mesmo e os casos de teste obtidos são testes funcionais no mesmo nível de abstração do modelo (UTTING; LEGEARD, 2007).

Uma das formas de derivação de casos de teste baseados em modelos é por meio de *Model Checking*(BAIER; KATOEN, 2008; FRASER et al., 2009), originalmente desenvolvida como uma técnica para verificar se uma propriedade formalizada é satisfeita por (por um determinado estado em) um modelo (mais detalhes sobre *Model Checking* na Seção 2.2). Em um processo de *Model Checking*, um modelo (também conhecido como Sistema de Transição) do SST, $\mathcal{M}$, e uma propriedade formalizada, Φ, são providas ao *Model Checker*, ou seja uma ferramenta de software que possui uma realização da teoria de *Model Checking*. No processo de verificação, se o modelo não satisfaz a propriedade ($\mathcal{M} \not\models \Phi$), então o Model Checker gera um contraexemplo. O contraexemplo é um traço do modelo que mostra que uma certa propriedade formalizada não é satisfeita pelo modelo. A ideia geral no contexto de Teste de Software, portanto, é considerar os contraexemplos como casos de teste. Diversos trabalhos foram desenvolvidos utilizando essa perspectiva tais como Fraser et al. (2009), Ferrante et al. (2014a), Ferrante et al. (2014b), Zeng et al. (2007), Enoiu et al. (2014), Yeolekar et al. (2013), Gent e Hsiao (2013), Mohalik et al. (2014), Santiago Júnior e Silva (2017).

2.1.3 Minimização de Casos de Teste

Minimização de casos de teste é o processo de remoção de testes com o intuito de reduzir o tamanho da suíte de testes de forma que apresente pouco impacto na capacidade de encontrar defeitos. O processo de minimização de testes é baseado em dois pontos chave:

- Estratégia de Avaliação: Consiste na forma como os testes são avaliados e suas pontuações são construídas. Elas podem usar diversos critérios, como cobertura de código (FRASER; WOTAWA, 2007), cobertura de requisitos (CAMPOS; ABREU, 2013; LIN et al., 2014), defeitos previamente encontrados (DANDAN et al., 2013) e tempo de execução (LIN et al., 2014) e são associadas a uma função de avaliação;

Tabela 2.2 - Exemplos de estratégias de avaliação

Simples	Multiobjetivo
Utiliza um único objetivo para executar a avaliação sendo normalmente cobertura de código (FRASER; WOTAWA, 2007) ou cobertura de requisitos (CAMPOS; ABREU, 2013).	Utiliza uma relação que envolvem diversos objetivos simultaneamente, como por exemplo o trabalho de Lin (LIN et al., 2014) é um exemplo desse tipo de avaliação, que também é chamada de "cost aware"assim como Wang (WANG et al., 2013) onde é a cobertura de features, falhas detectadas, frequência de execução e tempo total de execução da suíte de testes.

- Estratégia de Seleção: Consiste no método utilizado para selecionar ou remover casos de teste na suíte de teste usando como base a pontuação obtida durante a avaliação, podendo utilizar abordagens heurísticas como o como o Greedy Algorithm, GE *Greedy Essential*, GRE *Greedy Redundant Essential* (LIN et al., 2014) , e estratégias meta-heurísticas como algoritmos genéticos (PRADHAN et al., 2017);

Diversos estratégias de avaliação foram propostas para guiar o processo de minimização (ver tabela 2.2), utilizando diversos critérios para avaliação, dentre os quais se destacam cobertura de código (FRASER; WOTAWA, 2007), cobertura de requisitos (CAMPOS; ABREU, 2013; LIN et al., 2014), defeitos previamente encontrados (DANDAN et al., 2013) e tempo de execução (LIN et al., 2014). A maioria das abordagens modernas são do tipo Relação (LIN et al., 2014) ou Multiobjetivo (WANG et al., 2013) compostas por dois ou mais critérios sendo que normalmente um destes critérios funciona como restrição contra a perda de qualidade da suíte de testes enquanto o outro busca minimizar o custo da mesma (SINGH; SANTOSH, 2013).

Dentre as estratégias de seleção, as mais usadas são as heurísticas, destacando-se entre elas o *Greedy Algorithm*. O *Greedy Algorithm* é descrito no Algoritmo 1, onde:

- R_s : Suíte de requisitos contendo todos os requisitos satisfeitos por algum teste;
- t_i : Teste pertencente à suíte de testes original (T_i);
- t_m : Teste melhor avaliado e que será incluído na suíte de testes (T_m);
- $p(t)$: Função de avaliação de testes, retornando a pontuação do teste t.

O *Greedy Algorithm* gerou diversas variações que normalmente incluem processos de filtragem na suíte de testes com objetivos que podem variar entre encontrar testes que são únicos para um requisito (GE - *Greedy Essential*) e remoção de testes redundantes (GRE - *Greedy Redundant Essential*) (LIN et al., 2014). A maioria dos trabalhos que utilizam estratégias de seleção heurísticas utilizam as estratégias de seleção lineares ou por relação.

Algoritmo 1: Pseudocódigo para o *Greedy Algorithm* adaptado de Lin et al.(2014)

Entrada: Suíte de testes original T_i, Conjunto de todos os requisitos não cobertos por testes R_i

Saída: Suíte de testes minimizada T_m

```
1  início
2      R_s = ∅
3      T_m = ∅
4      repita
5          t_m = 0
6          para cada t_i ∈ T_i faça
7              se (p(t_i) > p(t_m)) ∧ (t_i ∋ T_m) então
8                  t_m = t_i
9              fim
10         fim
11         R_s ← R_s ∪ r(t_m)
12         T_m ← T_m ∪ t_m
13     até R_i = R_s;
14 fim
15 retorna T_m
```

A maioria das técnicas foram desenvolvidas para serem aplicadas no contexto de testes de regressão, utilizando-se de informações obtidas após um histórico de execuções (DANDAN et al., 2013; SINGH; SANTOSH, 2013; LIN et al., 2014; QIU et al., 2015). Entretanto, tais informações não estão disponíveis durante o processo de geração de testes, uma vez que os mesmos nunca foram executados.

Fora deste contexto, destaca-se o trabalho de Fraser e Wotawa (2007) em que é feita a minimização durante a geração de suítes de teste. Neste caso utiliza-se uma análise de redundância aonde são removidos testes que estejam duplicados ou contidos em outros. Entretanto, tal análise não leva em consideração o tamanho de cada caso de teste.

No trabalho de Ayav (2017), o processo de minimização é feito de forma que a suíte

de testes obtida cubra o critério de cobertura MC/DC, esse trabalho porém tem um objetivo bastante distinto do apresentado neste trabalho, que é o de minimizar uma suíte de testes de regressão de forma que a resultante cubra o critério MC/DC enquanto que o presente trabalho propõe uma técnica para gerar casos de teste que cubram o critério MC/DC e então minimizar a suíte de testes resultante de forma a viabilizar a execução dos mesmos.

2.2 Verificação Formal e *Model Checking*

Verificação formal é uma estratégia de garantia de qualidade em que um modelo do software é avaliado buscando-se encontrar defeitos nas fases iniciais do projeto (BAIER; KATOEN, 2008). Os principais métodos de verificação formal são a verificação por provadores de teorema e o *Model Checking*.

Segundo Ouimet e Lunqvist (2007), a verificação formal de um programa envolve a formulação de uma propriedade a ser verificada usando uma lógica adequada, tal como lógica temporal. Propriedades tipicamente tomam a forma de um predicado sobre valores de uma variável.

Ao avaliar um programa, são utilizadas duas abordagens distintas usando propriedades - condição pré/pós e asserção invariante. As abordagens de condição pré/pós formulam o problema de correção como a relação entre uma fórmula que se supõe manter no início da execução do programa (condição pré) e uma fórmula que deve ser realizada no final da execução do programa(condição pós). Avaliar o programa envolve determinar se a semântica do programa estabelece que a condição pós ocorre dado que a condição pré ocorreu. Abordagens baseadas em asserção invariante definem o comportamento de um programa como uma fórmula invariante que deve se manter durante toda a execução do programa. Os invariantes podem ser especificados pelo usuário ou podem ser inferidos automaticamente do código do programa.

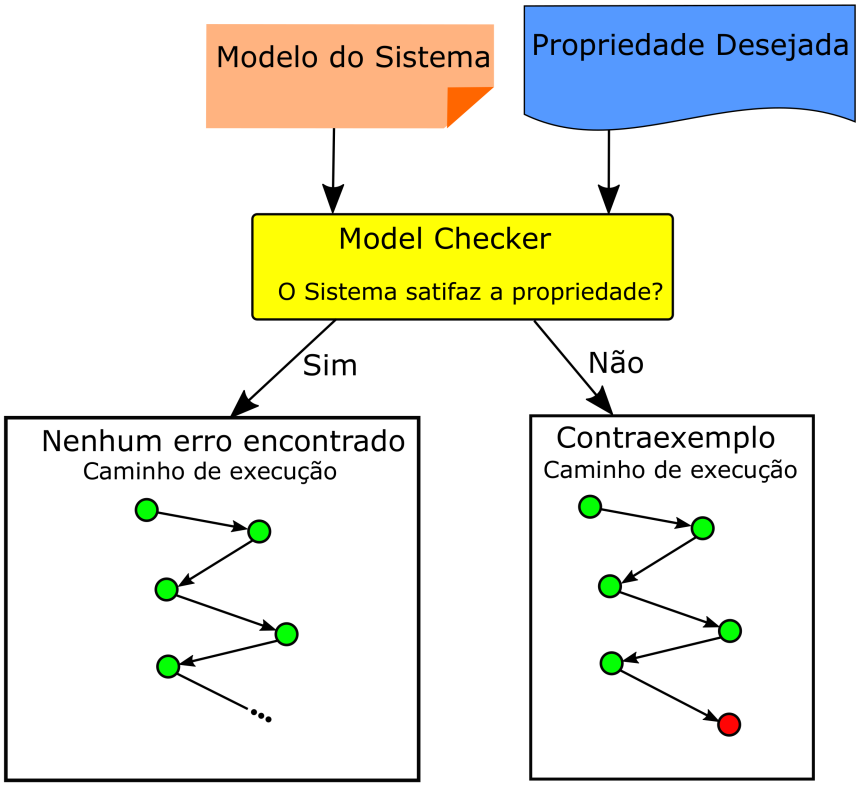
Provador de teorema (OUIMET; LUNDQVIST, 2007) é uma técnica usada para verificação aonde um modelo do sistema é verificado matematicamente através da derivação de um teorema o qual a ferramenta tenta provar automaticamente. Se não for possível chegar a uma resolução, então o sistema é considerado incorreto.

Model Checking é uma técnica usada para verificar se modelos de sistemas correspondem às suas especificações (BAIER; KATOEN, 2008). Usando uma especificação formal, tipicamente descrita em lógica temporal, o *model checker* explora o modelo buscando por inconsistências e retornando um contra exemplo demonstrando o erro

(ver figura 2.1. O contra exemplo consiste em um caminho de execução do modelo no qual a propriedade especificada é violada.

A técnica apresenta como vantagens da técnica o fato de ela ser uma abordagem utilizável em diversas aplicações, sua capacidade de prover informações para diagnóstico do erro (BAIER; KATOEN, 2008) e ser totalmente automatizada em relação aos provadores de teorema(OUIMET; LUNDQVIST, 2007). Como desvantagens, a técnica é pouco adaptável a aplicações com grande fluxo de dados, apresenta alto risco de explosão combinatória e é incapaz de garantir a completude do modelo (BAIER; KATOEN, 2008).

Figura 2.1 - Exemplo:*Model Checking*



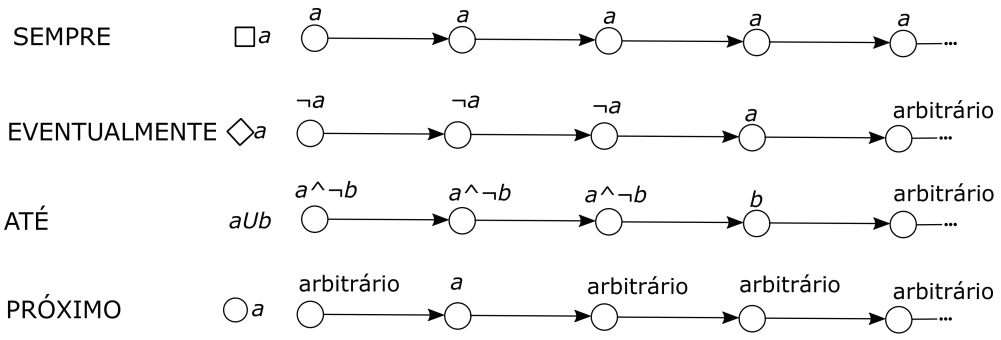
2.2.1 LTL e Padrões de Especificação

Lógica Temporal Linear (*Linear Temporal Logic* - LTL) é uma expansão da Lógica Proposicional proposta por Amir Pnueli (PNUELI, 1977) em 1977 para verificação formal de software, descrevendo propriedades temporais. A LTL adiciona os seguin-

tes modalidades temporais, exemplificadas na figura 2.2:

- $\Box$: SEMPRE indica que uma fórmula é sempre verdadeira;
- $\Diamond$: EVENTUALMENTE indica que uma fórmula será verdadeira em algum momento da execução do programa;
- $\cup$: ATÉ é um operador binário que indica que uma condição a é somente verdadeira até que outra fórmula b também seja;
- $\circ$: PRÓXIMO indica que uma fórmula se tornará verdadeira em um próximo passo.

Figura 2.2 - Exemplo: modalidades temporais LTL.



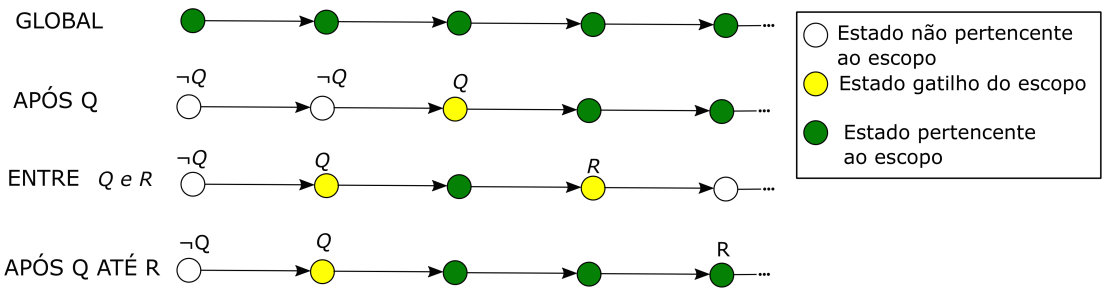
Fonte: Adaptado de Baier e Katoen (2008)

Dwyer *et al.* (1998) definiu padrões de especificação que descrevem diversas situações encontradas no processo de verificação formal de sistemas. Um padrão de especificação de propriedade pode ser definido como uma descrição generalizada de um requisito frequente nas sequências de estados ou eventos permitidas em modelos de estados finitos de um sistema (DWYER et al., 1998). Um padrão de especificação de propriedades descreve a estrutura essencial de algum aspecto do comportamento do sistema e provê expressões deste comportamento em um grupo de formalismos comuns. Cada padrão de especificação é associado a um um escopo (figura 2.3), que é a extensão da execução do programa sobre a qual o padrão é válido. Diversos escopos foram definidos sendo alguns deles (DWYER et al., 1998), exemplificados na figura 2.3:

- Global: A execução completa do programa ;

- Após: execução após um dado estado ou evento;
- Entre: qualquer parte da execução de um dado estado ou evento para outro estado ou evento;
- Após-até: como entre mas a parte designada da execução continua mesmo se o segundo estado ou evento não ocorre.

Figura 2.3 - Escopos de Padrões de Especificação



Dwyer (DWYER et al., 1998) define os seguintes padrões de especificação:

- Ausência: Um dado estado ou evento não deve ocorrer dentro de um escopo;
- Existência: Um dado estado ou evento deve ocorrer dentro de um escopo;
- Existência Limitada: Um dado estado ou evento deve ocorrer n vezes dentro de um escopo;
- Universalidade: Um dado estado ou evento deve ocorrer ao longo de um escopo;
- Precedência: Um estado ou evento P deve sempre ser precedido por um estado ou evento Q dentro de um escopo;
- Resposta: Um estado ou evento P deve sempre ser seguido por um estado ou evento Q;
- Precedência encadeada: Uma sequência de estados ou eventos P_1 ,...,P_n deve sempre ser precedida por uma sequência de estados ou eventos Q_1,...,Q_m;

Tabela 2.3 - Exemplos de Padrões de Especificação e Escopos

Padrão	Escopo	Modelo de Fórmula	Significado
Ausência	Global	$[](!P)$	P não deve ser verdadeiro durante a execução do programa
	Antes R	$<> R \rightarrow (!P \cup R)$	P não deve ser verdadeira até que R seja verdadeiro
	Após Q	$[](Q \rightarrow [](!P))$	Fórmula não deve ser verdadeira após Q ser verdadeiro
	Entre Q e R	$[]((Q\&!R\& <> R) \rightarrow (!P \cup R))$	P não deve ser verdadeira após Q ser verdadeiro e enquanto R não for verdadeiro
Existência	Global	$<> (P)$	Eventualmente P será verdadeiro
	Após Q até R	$[](Q\&!R \rightarrow (!RU(P\&!R)))$	Fórmula sempre P deve ser verdadeira após S e T serem verdadeiros
Resposta Encadeada	Global	$[](S\& \circ <> T \rightarrow \circ(<> (T\& <> P)))$	Fórmula sempre P deve ser verdadeira após S e T serem verdadeiros
	Antes de R	$<> R \rightarrow (S\& \circ (!R \cup T) \circ (!R \cup (T\& <> P))) \cup R$	Fórmula sempre P deve ser verdadeira após S e T serem verdadeiros e antes que R seja verdadeiro

- Resposta encadeada: Uma sequência de estados ou eventos P_1,...,P_n deve sempre ser seguida por uma sequência de estados ou eventos Q_1,...,Q_m.

Os padrões de especificação em conjunto com os escopos são exemplificados na tabela 2.3.

2.3 Linguagens de Programação Visual

LPVs permitem o desenvolvimento de sistemas, modelos de forma totalmente visual ao invés da codificação textual, através do agrupamento/conexão de blocos com variadas funções. Uma LPV permite programar com expressões visuais, arranjos espaciais de texto e símbolos gráficos, usados ou como elementos da sintaxe ou notação secundária. Por exemplo, muitas LPVs são baseadas na ideia de "blocos e setas", onde os blocos,ou outros objetos, são tratados como entidades e conectados através de setas, linhas ou arcos que representam relações. As LPVs podem ser classificadas por:

- Aplicação:
  - Educacionais: Linguagens especificamente desenvolvidas para o ensino de lógica, tais como Scratch (MALONEY et al., 2010), Kodu (MACLAURIN, 2011);
  - Multimídia: Linguagens desenvolvidas para aplicações multimídia, tais como Blender (KULKARNI et al., 2015);
  - Sistemas/Simulação: Linguagens utilizadas para especificação, prototipação, simulação e validação de sistemas, tais como Simulink (HANSELMAN; LITTLEFIELD, 2005) e Scilab/xcos(JANÍK; ŽÁKOVÁ, 2011). Essas LPV são muito utilizadas na metodologia MBD e o foco do presente trabalho está nessas linguagens;
  - Data Warehousing / Business Intelligence: Liguagens com foco na visualização e extração de informações de grandes massas de dados, tais como Talend Studio e Microsoft SSIS (KATRAGADDA et al., 2015).
- Estrutura Sintática (ERWIG et al., 2017):
  - Grafos: Usa explicitamente nós que são conectados por arestas, sendo as arestas direcionadas ou não-direcionadas e os nós, assim como as arestas podem ser rotulados ou não-rotulados. O presente trabalho se aplica a um subconjunto dessas LPVs que possua como características arestas direcionadas e que seja completamente rotulada;
  - Partição: Divide o espaço em uma não-sobreposição de regiões, onde uma partição fechada executa todas as sua ações separada das demais partições e do plano de fundo enquanto uma partição aberta integra as demais e/ou o plano de fundo.

As LPVs foram desenvolvidas com o intuito de reduzir o esforço necessário para desenvolver um sistema. Em geral, apresentam uma curva de aprendizado acelerado, permitindo que pessoas com pouco conhecimento de programação possam desenvolver sistemas. Devido a isso tem ampla aplicação no ensino de programação, em áreas que envolvam pessoas sem conhecimento em desenvolvimento de software e em aplicações de alta complexidade.

2.4 Trabalhos relacionados

A despeito dos trabalhos que já existem e que são relacionados a geração de casos de teste via *Model Checking* (vide Seção 2.1.2), apenas um trabalho foi encontrado na

literatura que estaria mais diretamente relacionado a essa dissertação de mestrado. Esse trabalho é descrito a seguir.

O estudo de Mohalik et al. (MOHALIK et al., 2014) propõe a geração de casos de teste a partir de um tipo específico de LPV, o Simulink, utilizando-se de *Model Cheking* para tal tarefa, e permitindo a criação de casos de teste que satisfazem o critério MC/DC. O trabalho utiliza uma série de abordagens diferentes visto que o esforço é feito dentro do Matlab, usando ferramentas disponibilizadas pelo software em questão, sem necessidade de traduzir o modelo para outra linguagem. Já existem frameworks para tradução de modelos Simulink para o model checker NuSMV (CIMATTI et al., 2002), tanto embutidas no Matlab (MEENAKSHI et al., 2006) como desenvolvidas por terceiros (MILLER, 2009).

Porém, Mohalik et al. não abordam um dos pontos chave dessa dissertação de mestrado: minimização. O trabalho em questão não efetua um processo de minimização, estando altamente suscetível à explosão combinatorial. Além disso, Mohalik et al. não fazem uso de padrões de especificação como é proposto no PCDVT.

Devido à similaridade entre ambos os trabalhos, a comparação entre ambos será discutida em maiores detalhes ao longo do Capítulo 3.

2.5 Considerações finais sobre esse Capítulo

Esse capítulo apresentou a fundamentação teórica relacionada a essa dissertação de mestrado. Foi apresentada uma visão geral sobre o processo de Teste de Software, mas ênfase maior foi dada aos conceitos/temas relacionados a teste, e que foram usados para o desenvolvimento dessa dissertação: critério para geração de casos de teste caixa branca MC/DC, TBM (particularmente a geração de casos de teste via *Model Checking*), e minimização de casos de teste. Outros temas abordados por essa pesquisa foram Verificação Formal (*Model Checking*, LTL), padrões de especificação e LPVs e, dessa forma, também foram salientados nesse capítulo.

O próximo capítulo apresenta o método PCDVT.

3 O MÉTODO PCDVT

3.1 PCDVT

O método PCDVT (ALARCON; SANTIAGO JÚNIOR, 2017) foi desenvolvido com o intuito de viabilizar a geração de casos de teste diretamente a partir de modelos LPV, reduzindo assim o esforço para se efetuar a verificação da qualidade do modelo e, consequentemente, do sistema final desenvolvido. O método utiliza o critério MC/DC, sugerido na norma DO-178C (HOLLOWAY, 2012), para agilizar o processo de qualificação do sistema visto que esse critério é importante para aplicações aeroespaciais. Além do MC/DC, o PCDVT faz uso de *Model Checking*, padrões de especificação, e abordagem de minimização por identificação de testes insubstituíveis tirando proveito dos principais pontos positivos dessas áreas em uma abordagem unificada. O método é descrito em alto nível na Figura 3.1, sendo composto pelas seguintes etapas:

a) LPV;

b) MC/DC;

c) *Model Checking*;

d) Minimização.

Ao longo das próximas subseções o método será explicado em maiores detalhes. Para exemplificar será usado o modelo LPV apresentado na Figura 3.2, onde, ao longo das próximas seções, o método proposto será aplicado. Este modelo, é composto por 3 blocos, recebendo duas entradas e levando a uma saída.

3.1.1 LPV

Nesta etapa, o modelo é definido para o sistema, o PCDVT realiza a tradução do modelo LPV para a linguagem *Process Meta Language* (PROMELA) (EDELKAMP, 2003) do *Model Checker* SPIN (HOLZMANN, 1997), no entanto, existem algumas características que o modelo deve satisfazer para ocorrer esse processo de tradução.

O PCDVT lida com a tradução de modelos do tipo diagrama de blocos (DORF; BISHOP, 2011) e máquinas de estado (ALARCÓN et al., 2014) desconsiderando operações de blocos que não executem decisões. O processo de análise do modelo LPV consiste em identificar cada um dos estados existentes, classificando-os em estados

Figura 3.1 - Diagrama de Atividades do método PCDVT

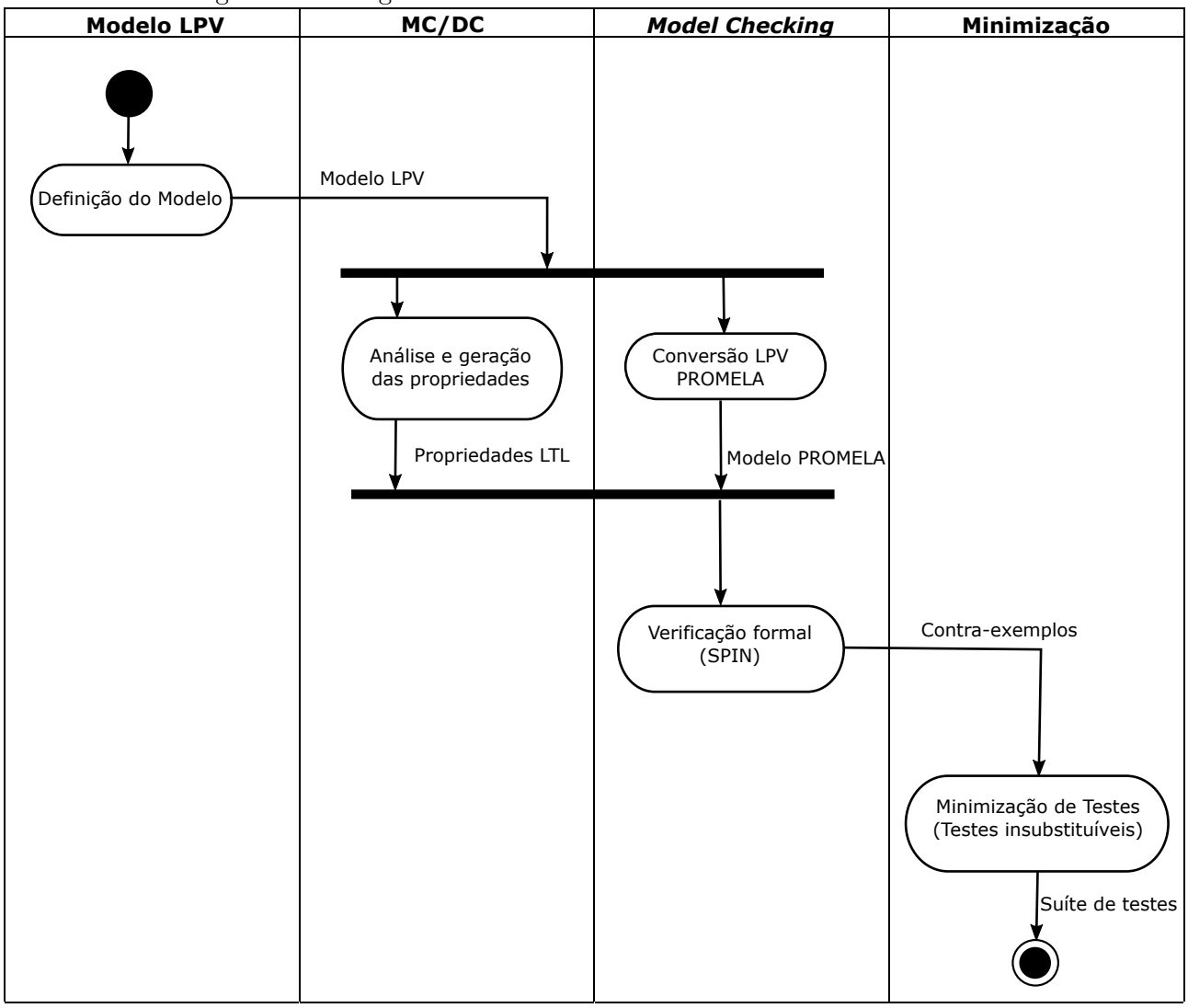
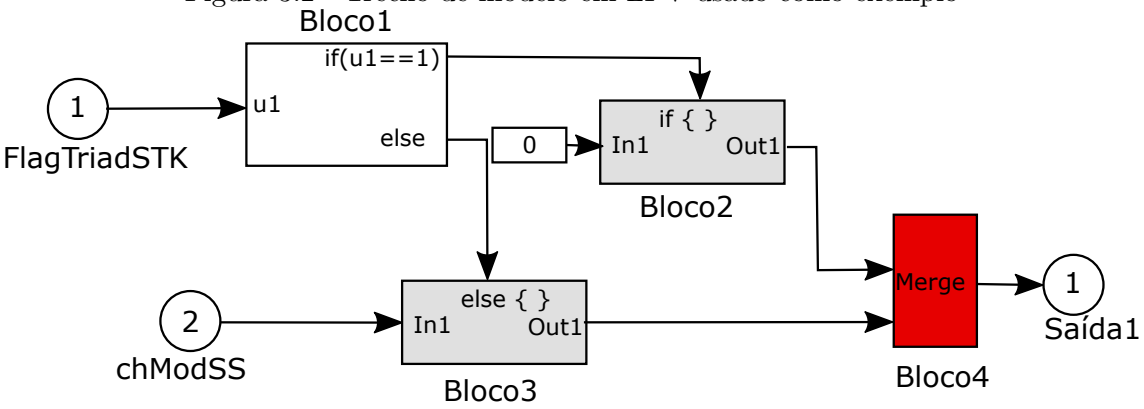


Figura 3.2 - Trecho de modelo em LPV usado como exemplo

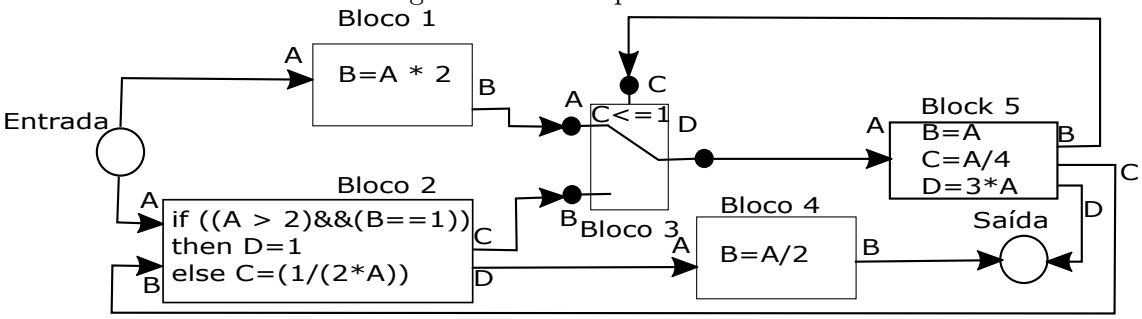


com decisão (ECD) e estados sem decisão (ESD). Em ECDs, é atribuído um identificador a cada uma das saídas possíveis. A Figura 3.3 mostra um exemplo de LPV suportada pelo PCDVT onde o Bloco 2 e o Bloco 3 são considerados ECDs e os demais blocos são ESD.

O mapeamento de todas as transições, com origem e destino, também é feito. No caso de ECDs, é mapeado também qual das possíveis saídas dispara tal transição, associando-se o identificador da saída à decisão.

No exemplo mostrado na Figura 3.2, o Bloco 1 é um ECD enquanto os demais são ESDs que respondem a este bloco. Bloco 4 é um bloco do tipo merge que une as saídas dos Blocos 2 e 3.

Figura 3.3 - Exemplo de LPV



Diversas LPVs podem ser, portanto, consideradas no contexto do método PCDVT, desde que respeitando as características mencionadas acima. Algumas delas são SciLab/Xcos (JANÍK; ŽÁKOVÁ, 2011), Yed (WIESE et al., 2004), e Simulink (DABNEY; HARMAN, 2004). Nesse trabalho, considerou-se a LPV Simulink.

3.1.2 Análise MC/DC

A próxima etapa é a identificação das condições de acordo com o critério MC/DC. O Algoritmo 2 foi definido para esse propósito. Em outras palavras, este algoritmo utiliza como entrada o modelo e retorna todas as as representações de transições compostas por:

- s_i : Estado destino;
- s_f : Condições que afetam independentemente, a saída. Quando se tratar de um bloco do tipo ESD, a condição é definida como *True* quando se

trata de um ESD de forma que tal estado também passe pelas demais etapas do método. Tais condições são posteriormente usadas para gerar as propriedades LTL que apoiam o processo de *Model Checking*;

- M : Modelo;
- t : Transição de estados no modelo;
- c_i : Condição em uma dada transição;
- I : Conjunto de todas as condições que independentemente afetem a decisão;
- C: Conjunto de todas as condições ou *True* obtidas;
- s_i : Estado origem;
- s_f : Estado destino;
- Θ: Conjunto de representações de transições contendo s_i,s_f,c_i.

Algoritmo 2: Análise MC/DC

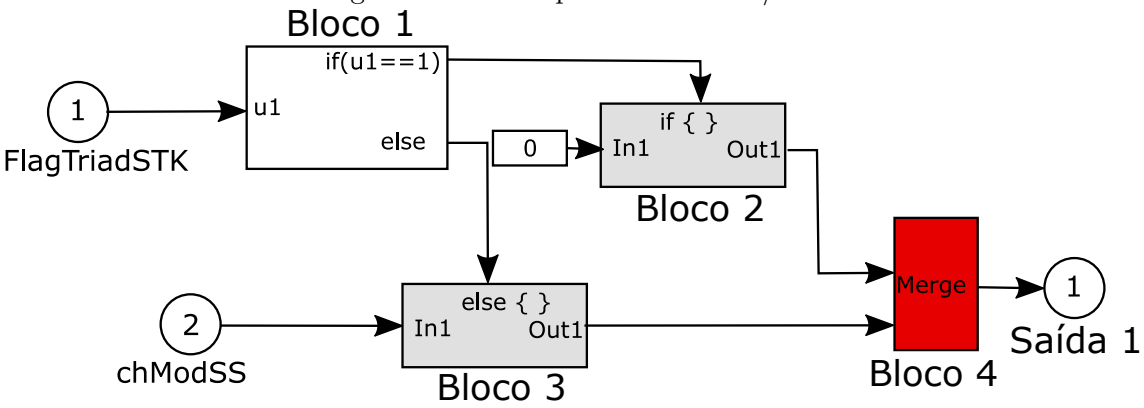
Entrada: Modelo, M
Saída: Conjunto de representações de transições, Θ

```
1  Θ ← ∅
2  C ← ∅
3  para cada t ∈ M faça
4  |   para cada c_i ∈ t faça
5  |   |   se c_i ∈ I então
6  |   |   |   C ← C ∪ c_i
7  |   |   fim
8  |   |   senão
9  |   |   |   C ← {C ∪ True}
10 |   |   fim
11 |   fim
12 fim
13 Θ ← Θ ∪ {(s_i, s_f, c_i)}
14 retorna Θ
```

A Figura 3.4 exemplifica como o Algoritmo 2 trabalha. Para o Bloco 1 (ECD), o algoritmo gerará 2 condições para serem testadas, enquanto para os demais blocos (ESD), vai gerar a condição *True*. Assim, o conjunto de condições (C) para todos os

blocos, considerando o algoritmo mostrado no Algoritmo 2, se torna $\{u_1 = 1, u_1! = 1, True, True, True\}$.

Figura 3.4 - Exemplo: Análise MC/DC



Bloco	Decisões	Condições	Próximo Bloco
1	1	u1=1 u1!=1	Bloco 2 Bloco 3
2	0	True	Bloco 4
3	0	True	Bloco 4
4	0	True	Saída 1

Para cada condição obtida através do Algoritmo 2, uma propriedade LTL é gerada. As propriedades LTL são derivadas por meio de 2 padrões de especificação para LTL. Considerando o padrão de Ausência com escopo entre Q e R (DWYER et al., 1998), o Algoritmo 3 descreve o processo utilizado para obtenção das propriedade LTLs, onde:

- s_i : Estado origem da transição do modelo, definido por Dwyer (DWYER et al., 1998) no padrão de especificação em questão como Q;
- s_f : Estado destino, definido por Dwyer (DWYER et al., 1998) no padrão de especificação em questão como R;
- c : Condição. definido como a fórmula P por Dwyer (DWYER et al., 1998) no padrão de especificação em questão;
- c : Fórmula P do presente padrão;

- l_a : Propriedade obtida para um dado padrão de especificação. De fato, essa propriedade é um conjunto cujos elementos informará como as fórmulas $(P, Q, R, \cdots)$ dos padrões de especificação serão substituídas. No fundo, as fórmulas serão substituídas por estados de origem (s_i), destino (s_f), ou condições (c_i);
- L_a : Conjunto de todas as propriedades LTL;
- Θ : Conjunto de todas as representações de transições obtidas previamente.

O padrão de especificação é completamente negado $(\neg)$ para forçar a geração do contraexemplo . Neste algoritmo, as seguintes modalidades temporais são utilizadas:

- $\Box$: Sempre;
- $\Diamond$: Eventualmente;
- $\cup$: Até;
- $\circ$: Próximo.

A Figura 3.5 apresenta um exemplo de derivação de propriedades LTL através do Algoritmo 3 para o Padrão de Ausência com escopo entre Q e R. Para cada bloco, pelo menos uma propriedade LTL é gerada. Para blocos ECD, cada condição identificada via análise MC/DC gera uma propriedade l_a para a transição em questão. Quando não há decisão, a condição é sempre definida como *True* e a propriedade l_a é gerada. Assim, no exemplo apresentado, para o Bloco 1 (ECD) são geradas duas propriedades sendo uma para cada possível saída, por se tratar de uma decisão simples. As condições c obtidas ($u1 == 1$ e $u1! = 1$) assumem a função da fórmula P definida por (DWYER et al., 1998) no Padrão de Ausência em Escopo entre Q e R. Para ambas as propriedades obtidas para o Bloco 1, o identificar do bloco vai assumir a função de fórmula Q enquanto o bloco destino, Bloco 2 para $u1 == 1$ e Bloco 3 para $u1! = 1$, assumem a função da fórmula R. Para os demais blocos é gerado apenas uma propriedade onde *True* é tratado como a fórmula P.

Outro padrão de especificação considerado é o Padrão de Resposta encadeada com escopo Global, é apresentado no Algoritmo 4. É considerada uma cadeia composta por 2 estímulos para uma resposta, por exemplo, P responde para S, T onde:

- s_i : Estado origem S da transição do modelo;

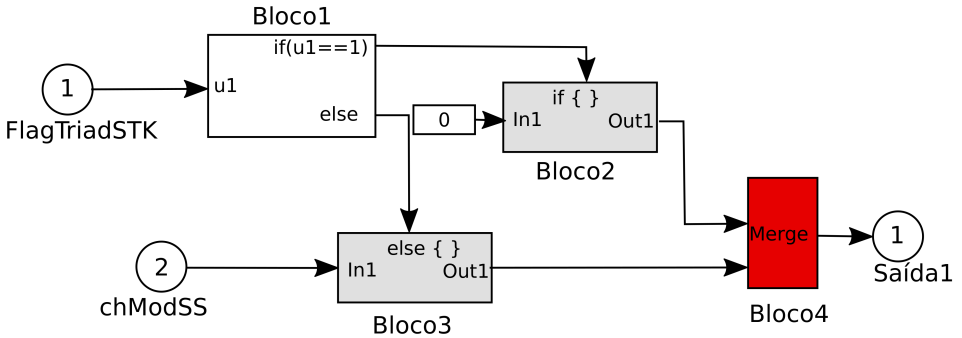
Algoritmo 3: Derivação das propriedades LTL para o Padrão de Ausência com escopo entre Q e R

Entrada: Conjunto de representação de transições, Θ

Saída: Propriedades LTL para o Padrão de Ausência em escopo entre Q (s_i) e R (s_f)

início
  $L_a \leftarrow \emptyset$
  para *cada* $\theta \subseteq \Theta$ **faça**
    $l_a \leftarrow \left\{\neg(\Box((s_i \wedge \neg s_f \wedge \Diamond_f) \rightarrow (\neg c_i \cup s_f))\right\}$
  fim
  $L_a \leftarrow L_a \cup l_a$
fim
retorna L_a

Figura 3.5 - Exemplo: propriedades LTL para o Padrão de Ausência



Bloco	Condições	Próximo Bloco	Padrão de Ausência em Escopo Entre Q e R
1	u1=1	Bloco 2	¬(□((Bloco1∧!Bloco2∧◇Bloco2)→(¬(u1==1)∪Bloco2)
	u1!=1	Bloco 3	¬(□((Bloco1∧!Bloco3∧◇Bloco3)→(¬(u1!=1)∪Bloco3)
2	True	Bloco 4	¬(□((Bloco2∧¬Bloco4∧◇Bloco4)→(¬(True)∪Bloco4)
3	True	Bloco 4	¬(□((Bloco3∧¬Bloco4∧◇Bloco4)→(¬(True)∪Bloco4)
4	True	Saída1	¬(□((Bloco4∧¬Saída1∧◇Saída1)→(¬(True)∪Saída1)

- c : Condição T que dispara a transição;
- s_f : Estado destino P.

Além disso, L_r é o conjunto de de todas as propriedades (l_r; vide comentário anterior sobre l_a, o qual também se aplica aqui) e Θ é o conjunto de todas as representações de transições obtidas previamente.

Novamente, o padrão inteiro é completamente negado ($\neg$) para forçar a geração de um contraexemplo. A Figura 3.6 exemplifica o processo de obtenção das propriedades LTL de acordo com o Padrão de Resposta encadeada com escopo Global definido

no Algoritmo 4. Para o Bloco 1, que é um ECD, são geradas duas propriedades l_a, cada uma delas para as condições c identificadas ($u1 == 1$ e $u1! = 1$), onde estas assumem a função de T no Padrão de Resposta Encadeada em escopo Global definido por (DWYER et al., 1998). O bloco origem da transição (s_i), neste caso o Bloco 1, assume a função de S enquanto o bloco destino (s_f), Bloco 2 para $u1 == 1$ e Bloco 3 para $u1! = 1$, assume a função de P. Para os demais blocos do exemplo, o comportamento se mantém sendo a condição c definida como *True*.

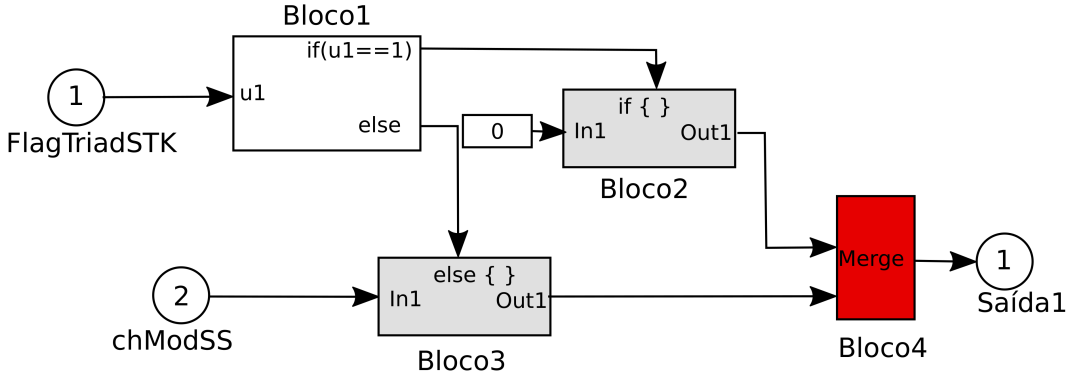
Algoritmo 4: Derivação das propriedades LTL para o Padrão de Resposta Encadeada com escopo Global

Entrada: Conjunto de representação de transições, Θ

Saída: Propriedades LTL para o Padrão de Resposta Encadeada com escopo Global

1 $L_r \leftarrow \emptyset$
2 **para** *cada* $\theta \subseteq \Theta$ **faça**
3 $\quad l_r \leftarrow \left\{\neg(\Box(s_i \wedge \circ\Diamond c \rightarrow \circ(\Diamond(c \wedge \Diamond s_f))))\right\}$
4 **fim**
5 $L_r \leftarrow L_r \cup l_r$
6 **retorna** L_r

Figura 3.6 - Exemplo: Propriedades LTL para o Padrão de Resposta encadeada com escopo global



Bloco	Condições	Próximo Bloco	Resposta Encadadea em Escopo Global
1	u1=1	Bloco 2	¬(□(Bloco1∧o◇(u1==1)→o(◇((u1==1)∧◇Bloco2))))
	u1!=1	Bloco 3	¬(□(Bloco1∧o◇(u1!=1)→o(◇((u1!=1)∧◇Bloco3))))
2	True	Bloco 4	¬(□(Bloco2∧o◇(True)→o(◇((True)∧◇Bloco4))))
3	True	Bloco 4	¬(□(Bloco3∧o◇(True)→o(◇((True)∧◇Bloco4))))
4	True	Saída1	¬(□(Bloco4∧o◇(True)→o(◇((True)∧◇Saída1))))

3.1.3 Model Checking

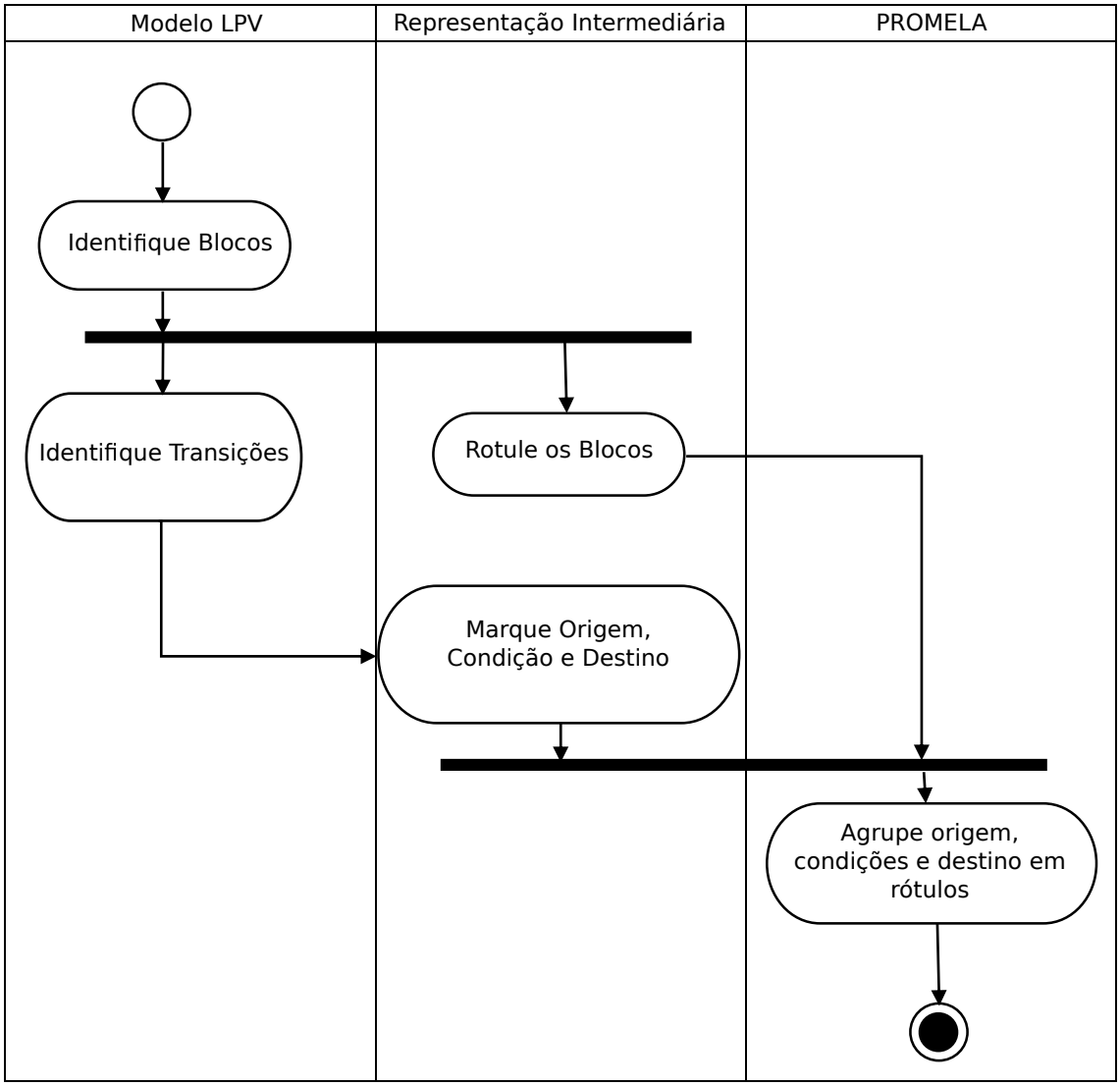
No PCDVT, utiliza-se de *Model Checking* considerando o *Model Checker* SPIN (HOLZMANN, 1997). Os contraexemplos gerados via SPIN são convertidos em casos de teste. Segundo Bayer (BAIER; KATOEN, 2008), *Model Checking* é comumente utilizado para verificar se o modelo comportamental de um sistema corresponde à especificação. O PCDVT utiliza o processo de *Model Checking* de maneira diferente de sua proposta original, usando os contraexemplos gerados por tal processo como caso de testes. As propriedades LTL definidas na Seção 3.1.2 descrevem uma condição existente, uma vez que é derivada do próprio sistema, utilizando a análise MC/DC. Devido a isso, em muitas ocasiões, um contraexemplo não é gerado ao exercitar o processo de *Model Checking*. Porém, ao negar o padrão de especificação, o sistema tenderá a deixa de satisfazê-la, levando à geração de um contraexemplo. Esse contra exemplo é um caminho de execução do modelo LPV que leva até o bloco testado.

Em adição à geração de propriedades LTL, é necessário construir um modelo formal (Sistema de Transição) que representa o modelo comportamental do sistema sob estudo. O SPIN foi o Model Checker selecionado e, uma das particularidades do SPIN é o fato de usar uma linguagem própria para a descrição do sistema de transição, chamada PROMELA, que permite descrever diversas propriedades, principalmente de concorrência (HOLZMANN, 1997). Essa característica se torna interessante para o PCDVT e para estudos de caso mais complexos, pois permite descrever sistemas concorrentes e simular a interação com ambientes externos, como por exemplo envio de telecomandos e falha de componentes. O PROMELA permite a definição de variáveis do tipo *mtype* (message type) que recebem valores texto a critério do usuário, sendo muito úteis para a codificação das condições encontradas nas decisões presentes no modelo sem necessitar processos mais complexos de interpretação do modelo LPV e que, consequentemente, demandariam mais tempo de desenvolvimento.

Para a utilização do *Model Checker* SPIN, foi necessário fazer uma conversão do modelo LPV para PROMELA, sendo utilizado como base o trabalho de Yamada e Miller (YAMADA; MILLER, 2016). No entanto, apresenta limitações em relação à extração de informações necessárias ao PCDVT além de uma descrição pouco detalhada dos passos intermediários. Desse modo, foram realizadas algumas adaptações de forma que o PCDVT pudesse realizar a conversão modelo LPV para PROMELA.

A Figura 3.7 mostra o processo de conversão do modelo LPV para PROMELA resumidamente. O processo de conversão consiste em identificar cada um dos blocos que compõem o modelo e extrair dos mesmos as transições. Em seguida, os blocos rece-

Figura 3.7 - Processo de conversão da LPV para PROMELA



bem identificadores únicos e as transições são explicitamente associadas ao bloco de origem em um arquivo intermediário. Os dados contidos nesse arquivo intermediário, são escritos de acordo com a sintaxe da linguagem PROMELA.

O Algoritmo 5 é uma adaptação do primeiro passo do do algoritmo de Yamada e Miller (YAMADA; MILLER, 2016) e também do processo definido na Figura 3.7, onde foi acrescentado a segregação de blocos ECD e ESD:

- LPV : Modelo LPV;
- b : Bloco;
- δ : Decisões presentes no bloco b;

- Δ : Conjunto de todos as decisões contidas no bloco b.

Algoritmo 5: Identificação de estados no modelo LPV

Entrada: Modelo LPV
Saída: Conjunto de estados classificados B

```
1  B ← ∅
2  para cada b ∈ LPV faça
3  |  b.tipo == tipo(b)
4  |  b.subsistema == subsistema(b)
5  |  se b.tipo ==ECD então
6  |  |  para cada δ ∈ b faça
7  |  |  |  b.Δ ← b.Δ ∪ δ
8  |  |  fim
9  |  fim
10 |  b.Δ ← b.Δ ∪ True
11 |  B ← B ∪ b
12 fim
13 retorna B
```

O processo de tradução do modelo consiste em um primeiro momento (ver Figura 3.7), buscar todos os blocos do modelo através do Algoritmo 5, classificando-os como ECDs e ESDs. Neste algoritmo também é identificado se o bloco se trata de um subsistema e, quando aplicável, a qual subsistema pertence, permitindo que o processo possa ser feitos para partes de um modelo.

Uma vez que os blocos tenham sido mapeados em estados, analisam-se as transições através do Algoritmo 7, buscando o estado origem da transição, à qual das decisões pertence (para ECDs) e então associando ao respectivo destino. Em caso de ESDs, as saídas são associadas à condição o valor *True*. Simultaneamente, no processo de tratamento das decisões é feito a identificação das variáveis envolvidas na decisão que serão posteriormente transformadas em canais de comunicação.

Com os estados mapeados, essa estrutura de dados é convertida, através do Algoritmo 7, em um arquivo intermediário cuja gramática é definida na Tabela 3.1. Nesta etapa, os estados são escritos no modelo PROMELA na forma de *labels*, onde são agrupadas as transições. A transição passa então a ser um comando *go to* para o *label* do respectivo estado destino.

No cabeçalho do arquivo PROMELA é definido um tipo *mtype* contendo os valores

Algoritmo 6: Identificação de transições no modelo LPV

Entrada: Modelo LPV, Conjunto de estados classificados B
Saída: Conjunto de estados classificados B

para *cada* $t \in LPV$ **faça**
  para *cada* $b \in B$ **faça**
    se $(t.origem \in b.nome) \wedge (t.destino \in B)$ **então**
      para *cada* $\delta \in b$ **faça**
        se $\delta.identificador = t.identificador$ **então**
          $\delta.condição \leftarrow t.condição$
          $\delta.destino \leftarrow t.destino$
        fim
      fim
    fim
  fim
fim
retorna B

Algoritmo 7: Geração do arquivo intermediário

Entrada: Conjunto de estados classificados B
Saída: Modelo intermediário M_{in}

$M_{in} \leftarrow \emptyset$
para *cada* $b \in B$ **faça**
  estado $\leftarrow \emptyset$
  se $b.tipo == ECD$ **então**
    lista-de-transições-ecd $\leftarrow \emptyset$
    para *cada* $b.\delta \in b.\Delta$ **faça**
      lista-de-transições-ecd $\leftarrow$ lista-de-transições-ecd $\cup\ \delta$
    fim
    estado $\leftarrow$b.identificador,b.nome,"ECD", $lista-de-transições-ecd$
    senão
      lista-de-transições-esd $\leftarrow \emptyset$
      para *cada* $b.\delta \in b.\Delta$ **faça**
        lista-de-transições-esd $\leftarrow$ lista-de-transições-esd $\cup\ \delta$
      fim
      estado $\leftarrow$b.identificador,b.nome,"ESD", $lista-de-transições-esd$
    fim
  fim
  M_{in}.subsistema(b.subsistema).lista-de-estados $\leftarrow$
   M_{in}.subsistema(b.subsistema).lista-de-estados $\cup$ estado
fim
retorna M_{in}

Tabela 3.1 - Regras gramáticas do arquivo intermediário definidas na Forma Backus-Naur

<modelo>	::=	<lista-de-subsistemas>
<lista-de-subsistemas>	::=	<subsistema><lista-de-subsistemas> | <subsistema>
<subsistema>	::=	<lista-de-estados>
<lista_de_estados>	::=	<lista-vazia> | <estado><lista-de-estados> | <estado>
<estado>	::=	<identificador>,<nome>,"ECD",<lista-de-transições-ecd> | <identificador>,<nome>,"ESD",<lista-de-transições-esd>
<identificador>	::=	<dígito>
<nome>	::=	<caractere>
<lista-de-transições-esd>	::=	<TRUE><destino>
<lista-de-transições-ecd>	::=	<lista-de-transições-ecd><decisão>,<destino> | <lista-de-transições-ecd><TRUE>,<destino> | <decisão>,<destino> | <TRUE>,<destino>
<destino>	::=	<nome><identificador>
<decisão>	::=	<condição> | <condição> <operador> <decisão>
<condição>	::=	<variável><comparação><valor>
<variável>	::=	<palavra>
<valor>	::=	<palavra>
<comparação>	::=	!= | == | < | <= | > | >=
<caractere>	::=	"A"| "B"| "C"| "D"| "E"| "F"| "G"| "H"| "I"| "J"| "K"| "L"| "M"| "N"| "O"| "P"| "Q"| "R"| "S"| "T"| "U"| "V"| "W"| "X"| "Y"| "Z"| "a"| "b"| "c"| "d"| "e"| "f"| "g"| "h"| "i"| "j"| "k"| "l"| "m"| "n"| "o"| "p"| "q"| "r"| "s"| "t"| "u"| "v"| "w"| "x"| "y"| "z"| "_"| "0"| "1"| "2"| "3"| "4"| "5"| "6"| "7"| "8"| "9"
<palavra>	::=	<caractere> | <caractere><palavra>
<operador>	::=	∧|∨
<TRUE>	::=	"True"
<lista-vazia>	::=	"[]"

possíveis das variáveis que fazem parte das decisões existentes no modelo. Em seguida são criados canais de comunicação do tipo *mtype*, sendo um para cada variável presente nas decisões do modelo. Em seguida, um processo *active process* é declarado e nele são embutidos todos os estados, definindo cada estado como um *label* e cada transição como um *go to*. As decisões são feitas em cima dos dados recebidos nos canais de comunicação com os nomes das variáveis presentes.

Um segundo processo *active process* é criado contendo as condições que disparam as transições definidas no modelo. Esse processo roda paralelamente ao anterior e envia, através dos canais de comunicação definidos para cada variável, os valores definidos nas decisões encontradas no sistema de forma que sejam gerados considerando as saídas verdadeiro e falso.

Figura 3.8 exemplifica o processo de tradução de um modelo LPV para PROMELA através do método proposto, onde as informações são organizadas em um arquivo intermediário. Posteriormente, as informações contidas no arquivo intermediário são escritas no modelo PROMELA dentro do processo *system*. Para o Bloco 1, a variável $u1$ se tornou o canal *u1* o qual é verificado se o valor é 1 ou diferente de 1, levando respectivamente para o Bloco 2 e Bloco 3. O processo *Send_u1* efetua a troca dos valores da variável $u1$ disparando as transições do Bloco 1 enquanto o processo *SendGeneral* envia o valor $True$ que dispara as demais transições do modelo.

Após a tradução do modelo para PROMELA, o mesmo é analisado pelo *model checker* SPIN para cada uma das propriedades obtidas anteriormente e os contra-exemplos passam então a ser casos de teste, usando o caminho de execução e a sequência de valores de variáveis obtidos.

3.1.4 Minimização

O último passo do PCDVT é a minimização da *suite* de testes (conjunto de casos de teste) obtida previamente através de *Model Checking*. Isso se faz necessário pois o processo de geração de casos de teste leva à geração de, pelo ao menos um caso de teste para cada bloco do modelo, considerando que todos os blocos passam pelo processo de geração de testes através de *Model Checking*. Como muitos dos blocos não possuem decisões, uma grande quantidade de casos de teste apresentará redundância, isto é os caminhos percorridos nos testes serão comuns em muitos deles, agregando pouco para a qualidade do sistema.

O processo de minimização de casos de teste começa com uma análise dos caminhos

de execução obtidos no passo anterior, para identificação de redundâncias entre casos de teste. O algoritmo, baseado no trabalho de Fraser (FRASER; WOTAWA, 2007), busca redundâncias considerandos casos de teste idênticos, casos de teste contidos em outros casos de testes, e é mostrado no Algoritmo 8, onde:

- CE: Conjunto de todos os contra-exemplos obtidos na etapa de Model Checking;
- ce: Contra exemplo pertencente ao conjunto CE;
- $L(ce)$: Conjunto de todas as propriedades LTL, equivalentes a requisitos, verificadas pelo contraexemplo ce.

Algoritmo 8: ANÁLISE DE REDUNDÂNCIA

Entrada: CE

Saída: Suíte de Contra-exemplos melhorada

início
  para *cada* $ce_i \in CE$ **faça**
    para *cada* $ce_j = ce_{i+1} \in CE$ **faça**
      se $ce_i == ce_j$ **então**
        $L(ce_i) \leftarrow \{L(ce_i), L(ce_j)\}$
        $L(ce_j) \leftarrow \{L(ce_j), L(ce_i)\}$
      fim
      senão
        se $ce_i > ce_j ce_j \in ce_i$ **então**
          $L(ce_i) \leftarrow \{L(ce_i), L(ce_j)\}$
        fim
        senão
          se $ce_i < ce_j ce_i \in ce_j$ **então**
            $L(ce_j) \leftarrow \{L(ce_j), L(ce_i)\}$
          fim
        fim
      fim
    fim
  fim
fim
retorna CE

A Figura 3.9 exemplifica o Algoritmo 8, mostrando a comparação entre dois casos de teste, onde os passos do Caso de Teste 1 estão contidos no Caso de teste 2. Com

a aplicação do Algoritmo 8, a redundância é identificada e o requisito *Req1* é então associado ao Caso de Teste 2. Na situação em que ambos os casos de teste são iguais, todos os requisitos são associados a ambos.

Por se tratar de um processo de minimização que ocorre durante o processo de geração de casos de teste, o uso do Algoritmo 8 é importante importante pois o algoritmo de avaliação usado nesse trabalho é baseado em cobertura de requisitos. Ao analisar a similaridade entre casos de teste e então associar os códigos das propriedades LTL (tratando como requisitos) a outros testes com caminhos de execução similares, o algoritmo de minimização pode ser aplicado, removendo da *suite* de testes aqueles que tenham baixa cobertura de propriedades em prol daqueles que apresentem maiores coberturas,

No PCDVT, o processo de minimização é feito por meio do algoritmo *Greedy* combinado com estratégia de avaliação por identificação de testes insubstituíveis, chamado *Greedy eIrreplaceable*, proposto por Lin (LIN et al., 2014). Nesta estratégia, casos de testes são avaliados de acordo com sua cobertura de requisitos e tempo de execução. Testes com menores tempos de execução e maiores coberturas de requisitos recebem maior pontuação e, consequentemente, tendem a ser escolhidos. Quando um requisito é verificado por apenas um teste, então o teste verificando aquele requisito é classificado como insubstituível e selecionado para integrar a suíte de testes otimizada. Uma vez que a estratégia *Greedy eIrreplaceable* foi desenvolvida no contexto de testes de regressão, ele utiliza informações não disponíveis durante a criação do teste como tempo de execução. Por conta disso, o PCDVT propõe uma nova versão adaptada do algoritmo Greedy *eIrreplaceable* aqui denominado como eIrreplaceable for Early Tests(eIET) definida no Algoritmo 9, onde:

- eirreplaceability : Avaliação do quão insubstituível um teste é;
- r : requisito de teste;
- covNum : Número de casos de teste cobrindo o requisito r;
- Cost(t) : Custo de execução de um teste, que aqui é o número de passos existentes no caso de teste.

Algoritmo 9: Pseudocódigo para o algoritmo EIET

Entrada: t: Caso de teste

Saída: Avaliação do teste

```
1  início
2      eirreplaceability = 0
3      para cada r ∈ t faça
4          se r ∈ R então
5              covNum = número de testes cobrindo o requisito
6              para cada r ∈ t faça
7                  se covNum == 1 então
8                      eirreplaceability = ∞
9                      pare
10                 fim
11                 eirreplaceability += 1 / covNum
12             fim
13         fim
14     fim
15 fim
16 retorna eirreplaceability / Cost(t)
```

Cada propriedade LTL passa a ser tratada como um requisito de teste, visto que a estratégia de testes insubstituíveis é baseada em cobertura de requisitos e, ao encontrar múltiplos casos de testes verificando o mesmo requisito, aquele que tiver maior cobertura será selecionado para a *suite* de teste otimizada. Com todos os requisitos mapeados para todos os casos de teste, o algoritmo *greedy* é executado em combinação com a estratégia de identificação de testes insubstituíveis (LIN et al., 2014), minimizando a suíte de testes obtida.

A Figura 3.10 mostra a aplicação do Algoritmo 9, onde um dado conjunto de casos de teste é minimizado. Os testes 2 e 3 verificam requisitos que não são testados em nenhum outro dentro conjunto apresentado e, portanto, são classificados como insubstituíveis e recebem uma pontuação que garante que os mesmos sejam selecionados. Desta forma, os requisitos verificados por eles (*Req1*, *Req2*, *Req3* e *Req4*) são marcados como satisfeitos. Após este passo, restam os testes 1 e 4 cujo requisito *Req5* ainda não são verificados em nenhum teste já escolhido. Como o teste 4 apresenta um número de passos de teste menor do que o presente no teste 1, este é escolhido e o requisito *Req5* é marcado como satisfeito. Desta forma, a *suite* de testes minimizada passa ser composta pelos testes 2, 3 e 4.

3.1.5 Comparação com Trabalhos Relacionados

O trabalho mais próximo da técnica aqui proposta foi desenvolvido por Mohalik et al. (2014), onde é proposta uma técnica de geração automática de casos de teste para LPV por meio de *Model Checking*.

O trabalho de Mohalik et al. (2014) não descreve a forma como a análise MC/DC é feita, e tampouco informa a estratégia usada para gerar as propriedades LTL. Os autores não descrevem em seu artigo uso de padrões de especificação como proposto no PCDVT. Além disso, o trabalho em questão usa o *model checker* NuSMV, usando uma ferramenta de tradução embutida na LPV em um processo bastante distinto do aqui proposto.

Mohalik et al. (2014) não faz qualquer menção ao uso de minimização de testes em seu trabalho, sendo essa a maior diferença em relação à técnica PCDVT.

3.2 Considerações finais sobre esse Capítulo

Esse capítulo apresentou, em detalhes, o método PCDVT e todos os processos envolvidos na geração da suíte de testes minimizada. Além disso foram mostradas as principais diferenças desse trabalho em relação ao de Mohalik et al. 2014.

Foram apresentados todos os algoritmos envolvidos no processo do PCDVT e definidas cada uma das etapas:

- Análise MC/DC;
- Geração de Propriedades LTL;
- Tradução do modelo para PROMELA;
- Processo de *Model Checking*;
- Minimização.

O próximo capítulo apresenta a implementação método PCDVT, assim como duas avaliações experimentais realizadas.

Figura 3.8 - Exemplo de tradução de modelo LPV para PROMELA

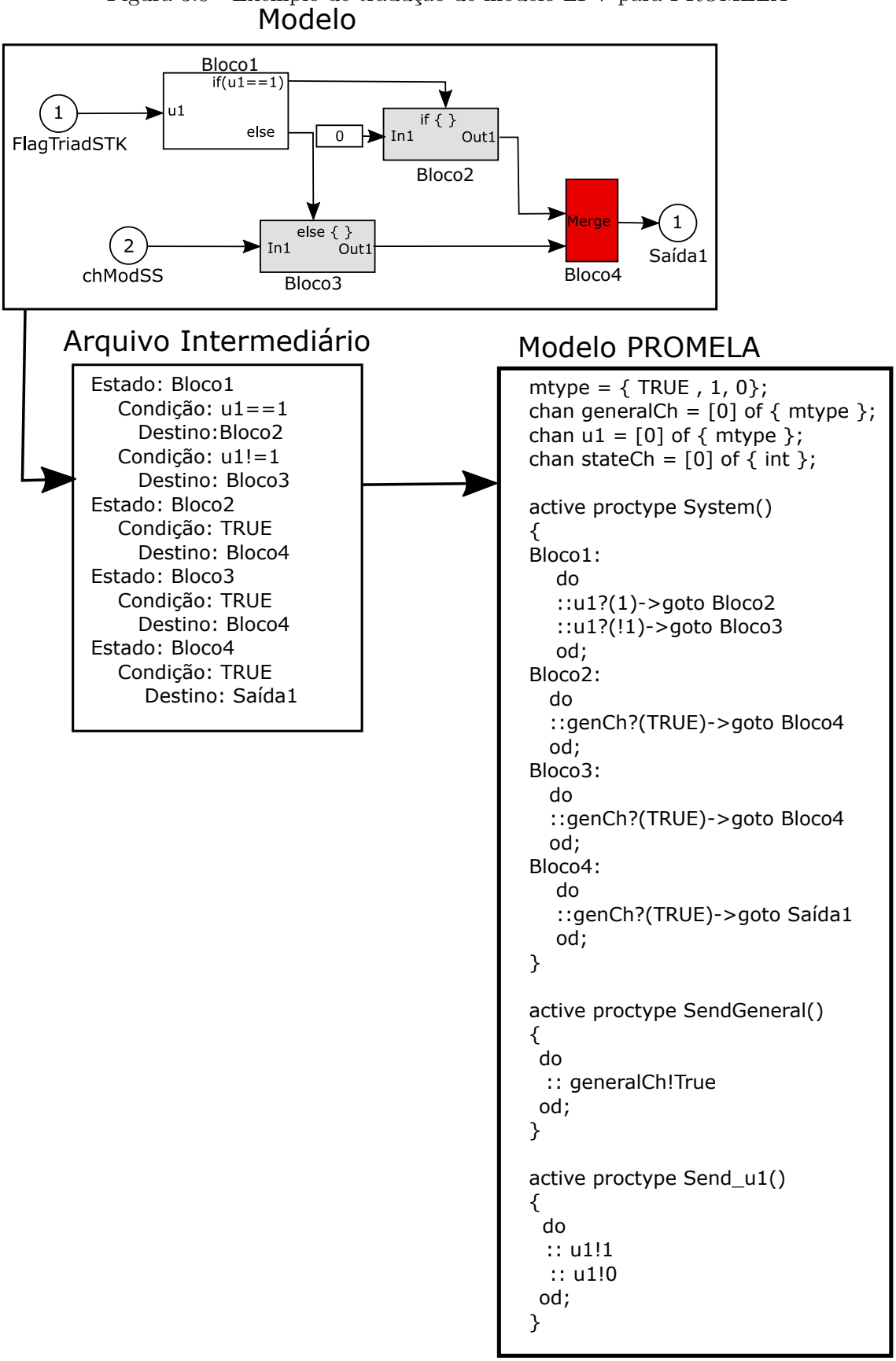


Figura 3.9 - Exemplo de análise de redundância

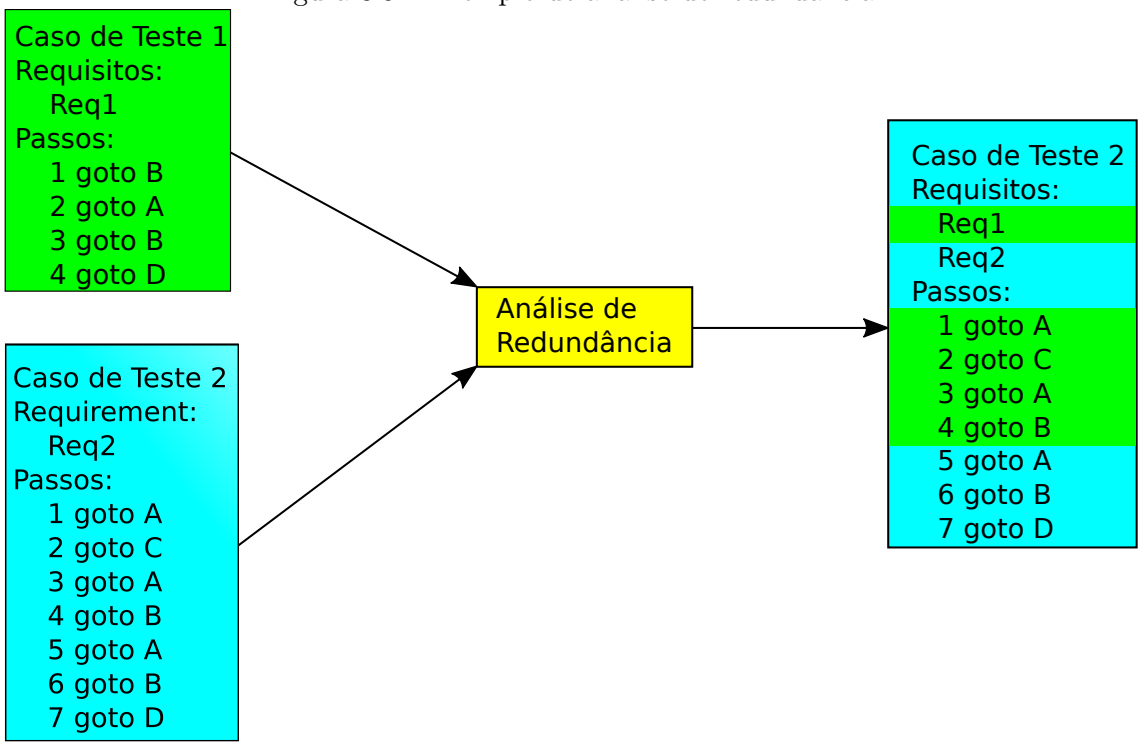
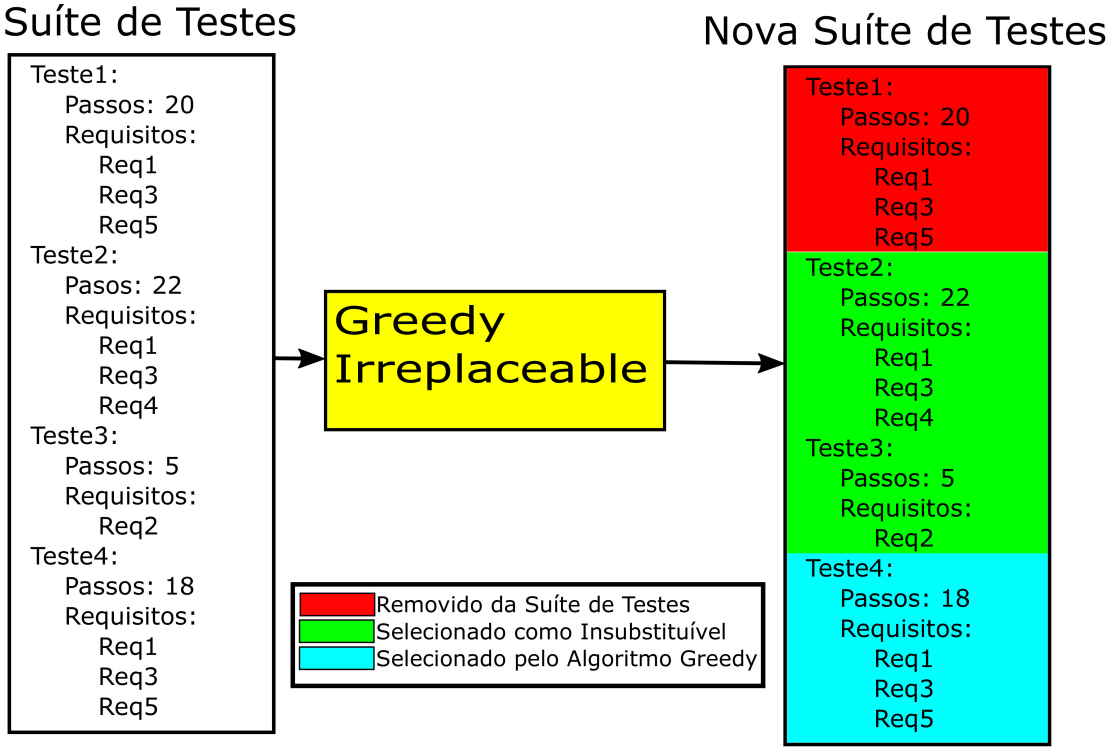


Figura 3.10 - Exemplo de minimização



4 IMPLEMENTAÇÃO DO PCDVT E AVALIAÇÕES EXPERIMENTAIS

Esse capítulo apresenta uma implementação do método PCDVT assim como os resultados de duas avaliações experimentais realizadas, considerando como estudos de caso duas modelagens *Simulink* de SCA similares ao do satélite Lattes-1 do INPE.

4.1 Implementação

A ferramenta PCDVT (mesmo nome do método) é uma instância do método apresentado no Capítulo 3, e a mesma foi desenvolvida usando a linguagem GO (DONOVAN; KERNIGHAN, 2015). Segundo Donovan e Kernighan (2015) GO é o "C do Século XXI" devido ao fato de herdar uma série de características da linguagem C, tais como sintaxe de expressões, fluxo de controle, tipos básicos de dados e ponteiros, compilação de código de máquina eficientemente, além de cooperar naturalmente com os sistemas operacionais atuais. Adicione-se a isso o fato de que GO fornece uma série de facilidades como *garbage collector*, apoio nativo a concorrência e importação de projetos *open source* disponíveis em repositórios de código na *internet*. Além disso, GO é uma linguagem orientada a objetos, favorecendo a modularização e o reuso. Essa linguagem foi escolhida devido à sua facilidade de programação, produtividade e bibliotecas disponíveis tendo em vista as características informadas.

A ferramenta PCDVT foi desenvolvida com o objetivo de ser extensível, reutilizável e configurável de forma que permita o processamento de diversas LPVs, que possa ser configurado para processar partes isoladas de modelos e que possa ser adaptado para outras técnicas. É possível configurar o processo para ser executado parcialmente ou completamente e as características da LPV. A ferramenta é configurada através de arquivos no formato JSON (JavaScript Object Notation) (BRAY, 2014), sendo permitidas as seguintes configurações:

- LPV: A semântica da linguagem LPV pode ser configurada através de um arquivo JSON com o nome da linguagem, onde deve ser definidos os caminhos em XPATH (*XML Path Language*) para se encontrar subsistemas, blocos, blocos de decisão, transições e símbolos lógicos "e"e "ou". Atualmente esse tipo de configuração está disponível somente para modelos definidos em linguagem XML *eXtended Markup Language*.

- Subsistemas: É possível informar quais subsistemas devem ser analisados para a geração dos casos de teste. Através de uma lista são definidos sub-

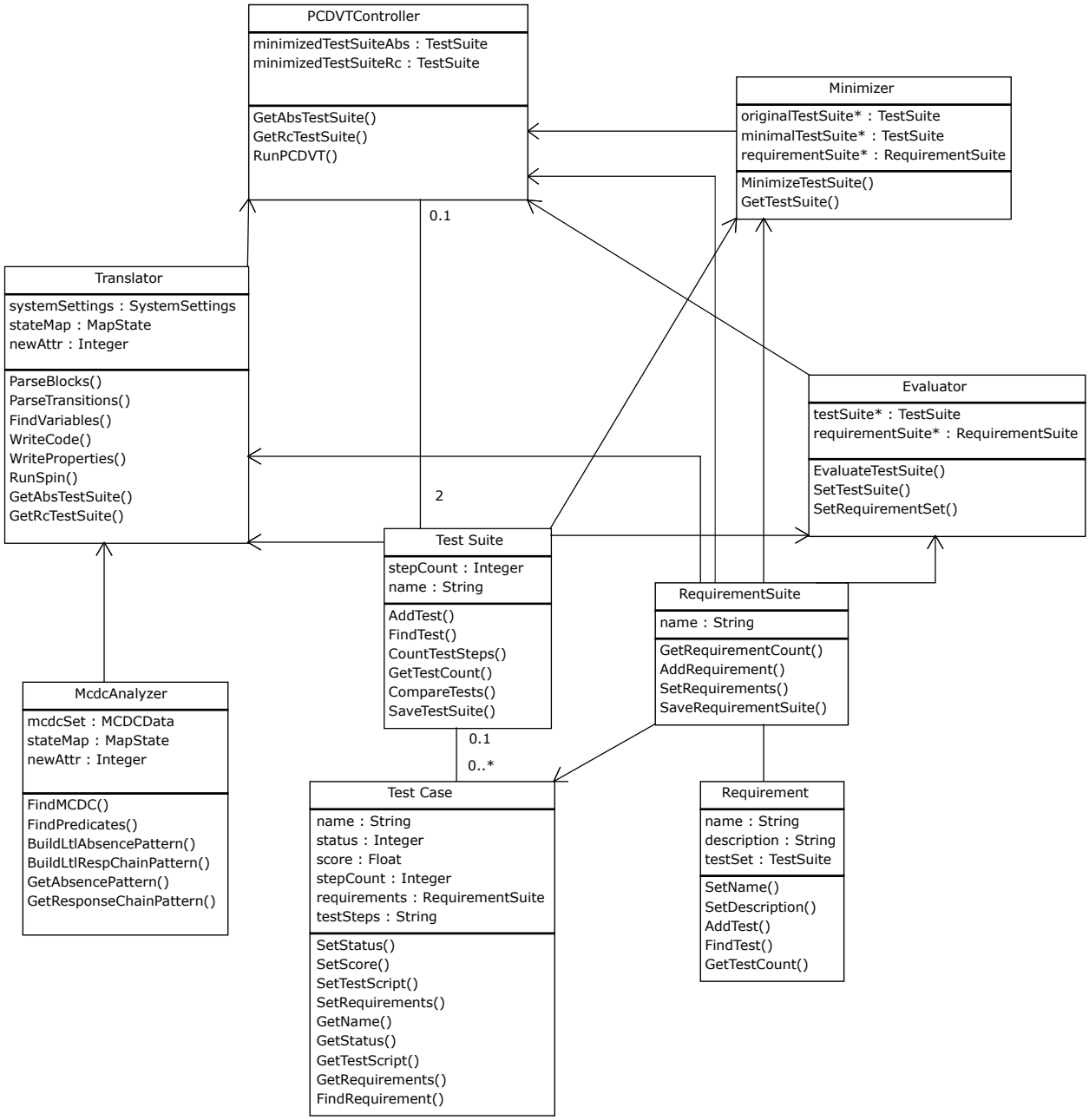
sistemas para os quais os testes devem ser gerados. Neste caso, transições direcionadas para blocos não pertencentes à lista de subsistemas testados são desconsideradas, bem como as variáveis presentes nestes blocos;

- Método de avaliação: Através desta configuração é definida a estratégia usada para obtenção das propriedades LTL, sendo as opções definidas no Capítulo 3 (Padrão de ausência em escopo entre Q e R ou padrão de resposta encadeada em escopo global);
- Minimização: O processo de minimização de testes pode ser desativado a critério do usuário. Quando ativo, o processo *eIET* definido no Capítulo 3 será executado.

A ferramenta foi desenvolvida seguindo o paradigma de programação orientada a objetos, sendo organizada em uma série de classes responsáveis por etapas específicas do processo de geração de casos de teste, como mostra a Figura 4.1. O detalhamento das responsabilidades dessas classes é dado a seguir:

- PCDVTController: Responsável por organizar e executar as demais etapas do processo. Todas as configurações para a execução do método aqui proposto deve ser definida nessa classe.Nesse nível, por exemplo, é definido se a geração de testes será feita para o modelo completo ou parcial;
- Translator: Responsável pela etapa de tradução de LPV para PROMELA e execução do SPIN. Foi desenvolvido de forma a ser extensível para diferentes sintaxes de modelos;
- McdcAnalyzer: Responsável pelo processo de análise MC/DC e geração de propriedades LTL. Utiliza dos dados fornecidos pela Classe Translator alimenta esta com as propriedades LTL identificadas;
- TestCase: Responsável por armazenar as informações básicas de casos de testes e executar as operações mínimas do mesmo. Todo contraexemplo é armazenado nessa classe incluindo passos de teste e propriedades LTL verificadas, aqui tratadas como requisitos. Os identificadores dos requisitos são armazenados em uma de lista que faz parte desta estrutura de dados, devendo estar de acordo com os dados encontrados no objeto requirement suite.;
- TestSuite: Responsável por agrupar casos de teste e executar operações em conjunto sobre eles;

Figura 4.1 - Diagrama de Classes do PCDVT



- Requirement: Responsável por armazenar as informações básicas de um requisito. Nesta estrutura é armazenado a lista de identificadores de casos de testes associados ao requisito. A inclusão do identificador do teste na lista de testes associados ao requisitos faz parte do processo de associação de um requisito a um teste e considerando que em uma das etapas do algoritmo *eIET* é necessário consultar tal informação, esta associação reduz

o esforço de busca;

- RequirementSuite: Responsável por agrupar requisito e executar operações em conjunto sobre eles;
- Evaluator: Responsável pelo processo de avaliação de casos de teste, incluindo também a análise de similaridade entre casos de teste. Implementa o algoritmo *eIET* Foi desenvolvido para ser flexível, permitindo que sejam implementadas diferentes abordagens para avaliação de testes;
- Minimizer: Responsável por executar o processo de minimização propriamente dito. Nesta classe os métodos implementam algoritmos de seleção de testes, neste caso o *Greedy Algorithm*.

A ferramenta foi criada de forma modular, permitindo que seja estendida para executar novos algoritmos de seleção, minimização e analisar diferentes tipos de LPV. Além disso, as classes favorecem a reutilização para o desenvolvimento de outras ferramentas de teste.

Processo de operação começa com o processo de interpretação do modelo LPV. A ferramenta PCDVT é capaz de interpretar somente LPVs convertidas para XML ou baseadas em XML. Caso o formato original do modelo seja binário (como no Simulink, por exemplo), o usuário deve gerar a versão XML através das ferramentas oferecidas pela LPV e então configurar esta ferramenta para processar o modelo compatível com XML.

O processo de interpretação do modelo é feito através de Xpath, onde buscam se todos os blocos e transições de acordo com os algoritmos definidos no Capítulo 3. Uma vez encontrados, os dados são carregados em memória e posteriormente salvos em uma série de arquivos JSON (BRAY, 2014) com o objetivo de organizar as informações e melhorar a desempenho ao liberar da memória informações que em um dado momento não são necessárias e consultar no arquivo quando forem requisitadas. O formato JSON foi escolhido devido à sua simplicidade, alta desempenho quando comparado ao XML (NURSEITOV et al., 2009), facilidade de leitura por pessoas e ao fato de que existe uma biblioteca nativa em GO para ler e escrever arquivos, convertendo-os diretamente para objetos. Durante esse processo, é executada a análise MC/DC e efetuada a tradução do modelo para PROMELA. Os arquivos principais gerados contém:

a) Definições de estados: Contém informações sobre o tipo de estado, a qual

subsistema pertence, quais as decisões, possíveis saídas existentes no estado e identificadores do estado;

b) Definições de transições: Contém definições de estado origem, estado destino e a qual das possíveis saídas do estado a transição está associada;

c) Definições de subsistemas: Contém os identificadores de estados que são subsistemas.

Em seguida é efetuado o processo de *Model Checking*, onde o SPIN é executado e sua saída gravada em um arquivo de texto. Esse arquivo contém o contraexemplo obtido sendo composto por uma sequência de estados e valores de variáveis feitos de acordo com os valores obtidos no processo de tradução do modelo LPV. Essa é a etapa mais demorada do processo, devido às limitações de desempenho do SPIN. Nesta etapa, o SPIN é chamado como um processo externo, onde o mesmo executa o processo de *Model Checking* para cada propriedade LTL disponível. Devido à grande diferença de desempenho, o processo do PCDVT permanece parado até que todas as propriedades LTL sejam analisadas pelo SPIN, visto que as etapas seguintes como o processo de minimização exigem que acesso a todos os contraexemplos obtidos.

Após concluído o processo de *Model Checking*, os arquivos de texto gerados pelo SPIN são analisados, sendo extraídos os passos de teste e a propriedade LTL usada em cada contraexemplo. Os passos do contraexemplo são carregados como casos de teste enquanto as propriedades LTL são convertidas em requisitos.

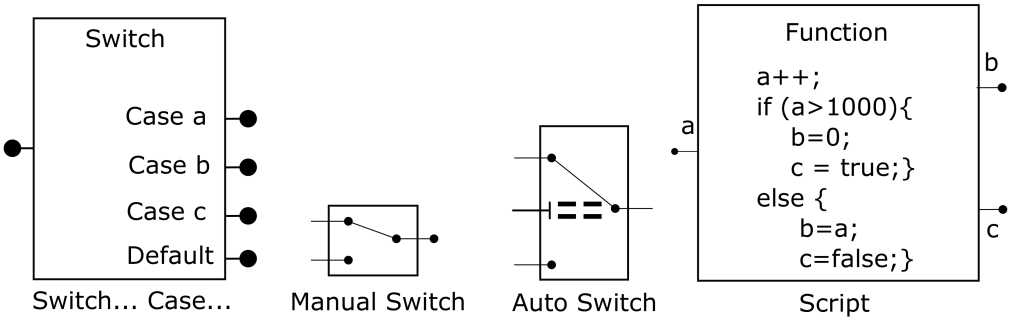
Então inicia-se o processo de avaliação dos casos de teste obtidos, onde em um primeiro momento é feita a análise de similaridade entre os casos de teste. Nesta etapa, os requisitos vão sendo adicionados aos casos de teste de acordo com o resultado das análises, isto é, contraexemplos similares têm seus requisitos associados a outros testes conforme a análise. Após isso, os casos de teste são avaliados através do algoritmo EIET, recebendo então pontuações de acordo com o resultado obtido.

Posteriormente inicia-se o processo de seleção, onde os casos de teste selecionados são então copiados para uma nova suíte de testes de acordo com as pontuações. A nova suíte de testes é re-analisada, extraindo-se os valores de número total de passos de teste, entre outras informações, e então a suíte de testes é salva no formato JSON.

Uma limitação presente nesta versão da ferramenta PCDVT consiste em uma série de blocos de decisão que não são suportados pelo processo de tradução proposto(ver Figura 4.2) sendo, portanto, tratados como blocos sem decisão (ESDs). Assim, esses

blocos não são analisados em busca das condições que cubram o critério MC/DC. Isso implica que a cobertura MC/DC pode ficar aquém do esperado dependendo do número de blocos de decisão não suportados adotados no modelo LPV. No modelo utilizado como estudo de caso, existem 131 blocos não suportados, o que representa 6,33% do total de blocos presentes no sistema. Dentre os blocos não suportados, os blocos de *scripts* são os mais comuns, representando 71,74% do total.

Figura 4.2 - Exemplos de blocos contendo decisões não suportados pela ferramenta PCDVT



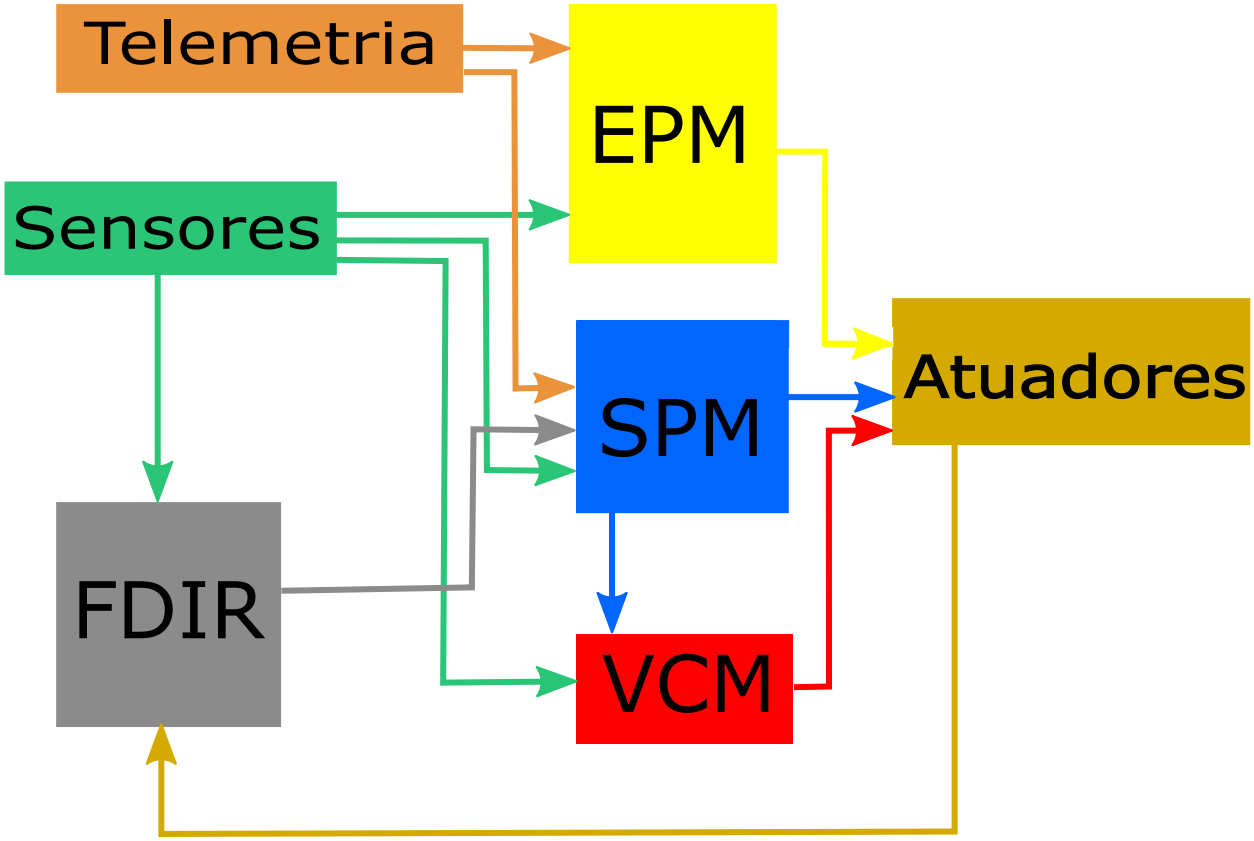
Além disso não é possível analisar decisões que sejam compostas por múltiplos níveis de verificação tais como $((A \wedge B) \vee (C \vee A) \wedge D)$ sendo, portanto, necessário que as decisões estejam em um único nível como, por exemplo, em $A \wedge B \vee C \wedge A \wedge D$. Essa limitação não afeta o modelo em questão visto que nenhuma decisão presente em blocos do tipo está presente no modelo usado como estudo de caso.

4.2 Avaliações Experimentais

4.2.1 Estudo de Caso: Modelo de SCA similar ao do satélite Lattes-1

Como estudo de caso, foi utilizado um modelo LPV de um SCA, em duas dimensões (controle em um eixo), similar ao do satélite Lattes-1 (ARAUJO, 2012; ALARCÓN et al., 2014). O sistema, mostrado na Figura 4.3, apresenta 3 modos de operação, sendo eles *Sun Pointing Mode* (SPM), *Earth Pointing Mode* (EPM) e *Velocity Control Mode* (VCM). Além disso, apresenta um módulo de detecção, isolamento e recuperação de falhas *Fault Detection, Isolation, and Recovery* (FDIR), que é capaz de disparar o modo de *Sun Pointing*. O SCA foi desenvolvido para ser tolerante a falhas simples e combinadas, e o FDIR deve provocar a transição para o modo SPM sempre que houverem falhas compostas, isto é dois ou mais componentes do mesmo tipo em falha, simultaneamente.

Figura 4.3 - Arquitetura do SCA



Neste modelo, são considerados os seguintes sensores e atuadores:

- Sensor Solar: Responsável pela identificação da direção do Sol em relação ao satélite. Existem, ao todo, 7 sensores, sendo sempre 3 iluminados pelo Sol, independentemente da orientação do satélite;
- Magnetômetro: Responsável pela medição do campo magnético da Terra. Existem 2 magnetômetros no satélite;
- Giroscópio: Responsável pela medição da velocidade angular do satélite, havendo 2 giroscópios no satélite para fins de redundância;
- Sensor de estrelas: Sensor responsável pela determinação da atitude e velocidade angular com alta precisão, existindo 2 destes no satélite. Este sensor é utilizado somente no modo EPM;
- Roda de Reação: Atuador responsável pela execução das manobras de atitude do satélite. Existem 2 rodas de reação para fins de redundância,

entretanto ambas são utilizadas nas manobras em condições normais de operação.

4.2.1.1 Sun Pointing Mode

É considerado como o modo inicial do SCA e também considerado como modo de contingência. É acionado por telecomandos ou pelo FDIR. É o modo de operação mais simples, aonde o satélite trava os painéis solares e segue a direção do Sol. Nesse modo a missão do satélite não opera e somente os sistemas de telemetria e controle de atitude permanecem ativos. O controle do satélite é feito utilizando somente os sensores solares e magnetômetros para determinação da atitude.

4.2.1.2 Earth Pointing Mode

É o modo em que o satélite executa sua missão, sendo sempre disparado por telecomandos vindos das estações de controle e rastreio de satélites. Neste modo todos os componentes estão ativos e o satélite mantém apontamento preciso para o centro da Terra por padrão, mas podem ser configurados outros pontos através de telecomandos.

4.2.1.3 Velocity Control Mode

É o modo em que satélite opera quando está em rotação e precisa se estabilizar. Neste modo o processo de estabilização do satélite é feito de maneira que não danifique nenhum componente. Esse é um modo que espera-se que seja usado somente no início da missão, visto que, uma vez estabilizado, o satélite deve manter seu apontamento de acordo com os requisitos da missão.

4.3 Detalhamento dos Experimentos

A fim de verificar a viabilidade da estratégia PCDVT, foram desenvolvidas duas avaliações experimentais, uma utilizando um modelo de complexidade intermediária do SCA do satélite Lattes-1 (Experimento 1), e outra usando um modelo completo do SCA (Experimento 2).

As métricas usadas nessas avaliações experimentais foram custo e efetividade. Custo é definido como a quantidade total de passos de testes de todos os casos de teste da *suite* de teste. A explicação para a escolha dessa definição de custo é que um caso de teste é formado por um conjunto de passos de teste. Um passo de teste é uma atividade atômica para preparar ou estimular o SST. O estímulo pode conter

Tabela 4.1 - Configuração do computador usado nos experimentos

Componente	Características
Processador	Intel Core I5-5200U CPU 2x 2.20 GHZ
Memória RAM	8GB DDR3 L
Armazenamento	HDD 1TB 5200 RPM
Placa de Vídeo	NVIDIA GeForce 920M 2GB VRAM
Sistema Operacional	Windows 10

os dados de entrada de teste e os resultados esperados. Em outras palavras:

$$ct = \{pt_i \quad | \quad i \in \mathbb{N} \backslash \{0\}\} \tag{4.1}$$

onde ct = caso de teste e pt_i = passo de teste i. Entretanto, um caso de teste, ct_1, pode ter associado somente 1 passo de teste e, por exemplo, um segundo caso de teste, ct_2, pode ser composto de 50 passos de teste. Então, comparar o custo de duas *suites* de teste considerando a quantidade de casos de teste não é muito adequado. Desse modo, se for considerado um tempo de execução uniforme para cada passo de teste (i.e. um passo de teste pode demandar, em média, α unidades de tempo para ser executado), a quantidade de passos de teste fornece uma indicação do custo para executar a *suite* de teste. Portanto, quanto menor a quantidade total de passos de teste que uma *suite* de teste gera, melhor.

Efetividade é definida como a cobertura MC/DC acumulada dos modelos pela execução das *suites* de teste. Desse modo, quanto maior a cobertura MC/DC acumulada devido a uma *suite* de teste, melhor.

Todos os experimentos foram executados em um computador pessoal com a configuração mostrada na Tabela 4.1.

4.3.1 Experimento 1

No Experimento 1, modelo de complexidade intermediária, a ferramenta foi configurada para executar o processo de geração de casos de teste somente para os blocos de referentes aos modos de operação EPM e SPM, o FDIR somente para as rodas de reação e desconsiderando os sensores, sendo analisados 1348 blocos. Este experimento efetua a execução dos testes para os mesmos blocos verificados pelos testes de Alarcón (2013).

Duas *suites* de testes foram geradas para os padrões de especificação definidos ante-

riormente. O conjunto de testes gerado por meio do padrão de ausência com escopo entre Q e R será denominado PCDVT-A, e aquele gerado pelo padrão de resposta encadeada com escopo global será chamado de de PCDVT-RC. Os resultados obtidos são confrontados com os resultados dos testes manuais desenvolvidos para o modelo (ALARCON, 2013).

Foram geradas, automaticamente, 1348 propriedades LTL para o modelo intermediário via o método PCDVT de acordo com a análise MC/DC, resultando em 1348 casos de teste na suíte de testes, sendo um teste para cada bloco do modelo. Sem a aplicação do algoritmo de minimização Greedy eIET, o número casos de teste gerados automaticamente foi bastante superior ao especificado manualmente no projeto (ver Tabela 4.2). Ao passar pelo algoritmo Greedy eIET, a *suite* PCDVT-A foi reduzida a 1 caso de teste (ver Tabela 4.3) enquanto a gerada via PCDVT-RC foi reduzida a 2 casos de teste (ver Tabela 4.4). Desse modo, houve uma redução de aproximadamente 99% no número de passos de teste para a suíte PCDVT-A e de aproximadamente 97% para a suíte PCDVT-RC, com a aplicação do algoritmo de minimização do PCDVT.

As Tabelas 4.2, 4.3 e 4.4, a linha **Total**, mostram os resultados obtidos para cada uma das *suites* de teste consideradas, sendo a Tabela 4.2 referente aos testes manuais, a Tabela 4.3 referente à *suite* PCDVT-A e a Tabela 4.3 referente à PCDVT-RC. Para cada uma dessas tabelas a primeira coluna mostra o caso de teste, a segunda coluna indica o número de passos de teste e a terceira coluna mostra a cobertura MC/DC obtida para o caso de teste. A última linha representa o total, onde na segunda coluna é apresentado o número total de passos de teste para a *suite* de testes em questão e a cobertura MC/DC acumulada, isso é, a cobertura obtida em conjunto por todos os casos de teste.

A Figura 4.4 mostra os unifica os resultados obtidos, onde as barras azuis representam o número total de passos de teste de cada uma das *suites*, referente ao eixo da esquerda. A linha laranja representa a cobertura MC/DC acumulada para cada um dos conjuntos de teste, referentes à coluna direita.

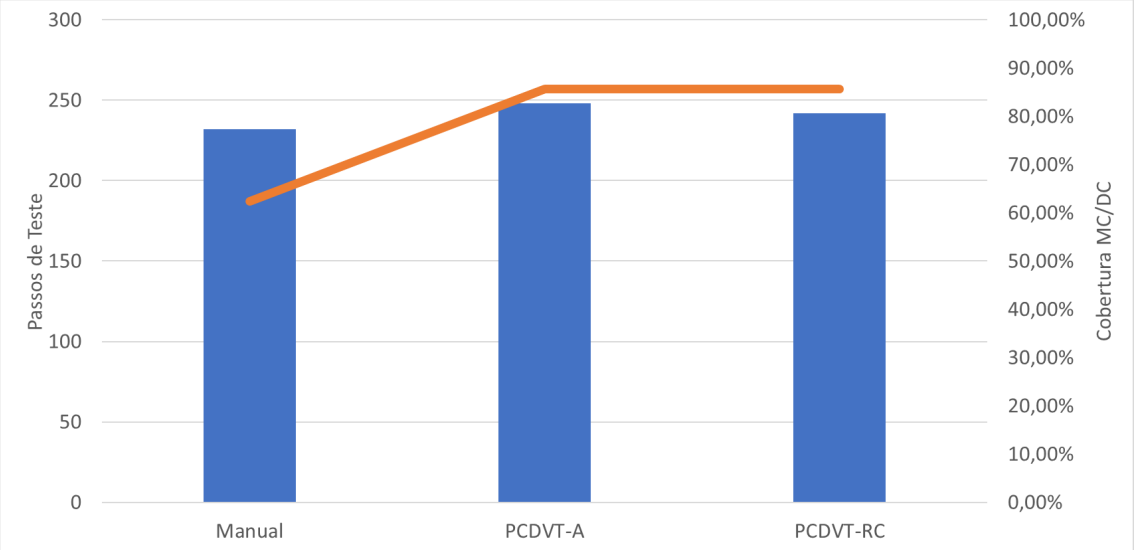
Levando em consideração, portanto, as *suites* de teste geradas através das duas instâncias do PCDVT e com a aplicação do algoritmo de minimização, percebe-se que o custo das 3 abordagens é similar (ver Figura 4.4) , com uma leve vantagem da abordagem manual (ver Tabela 4.2), onde a mesma apresenta 232 passos de teste no total, em relação aos casos de teste gerados via PCDVT-A (Ver Tabela 4.3) e PCDVT-RC (4.3), onde ambos contém 248 passos de teste no total.

Conforme observado na Figura 4.4, percebe-se uma substancial melhoria de efetividade dos conjuntos de casos de teste gerados via PCDVT-A (85.66%) e PCDVT-RC (85.67%) quando comparados com a abordagem manual (62.44%).

O processo de geração das suítes de teste via a ferramenta PCDVT para as duas instâncias em conjunto levou 48 horas, sendo aproximadamente 98% desse tempo (47 horas) dedicado à etapa de *Model Checking*. A execução de cada suíte de teste (PCDVT-A, PCDVT-RC) levou aproximadamente 72 horas em média, que foi aproxidamente o mesmo tempo requerido pela abordagem *ad-hoc*.

Portanto, os resultados apresentados nessa subseção mostram que as duas instâncias do PCDVT, PCDVT-A e PCDVT-RC, possuem efetividade (cobertura MC/DC do modelo) significativamente maior mas mantém custo similar (quantidade de passos de teste), se comparadas a uma abordagem manual e *ad hoc* feita no contexto do projeto (ALARCON, 2013). Isso é um indício de que o método PCDVT pode ser de extrema valia para melhorar a qualidade de modelos LPV de aplicações críticas, como os satélites que o INPE desenvolve.

Figura 4.4 - Experimento 1: Visão unificada das métricas



4.3.2 Experimento 2

Para o Experimento 2, foram considerados todos os blocos existentes no modelo SCA, ou seja, o modelo completo possuindo 2068 blocos.

Tabela 4.2 - Experimento 1: *Suite* de teste manual gerada no contexto do projeto

Caso de Teste	Passos de Teste	Cobertura MC/DC
1	3	45%
2	5	59%
3	8	51%
4	8	51%
5	8	52%
6	8	51%
7	8	53%
8	8	52%
9	8	53%
10	8	52%
11	8	53%
12	8	52%
13	8	53%
14	8	53%
15	8	52%
16	8	53%
17	8	52%
18	8	51%
19	8	52%
20	8	53%
21	8	52%
22	8	52%
23	8	53%
24	8	53%
25	8	52%
26	8	53%
27	8	51%
28	8	53%
29	8	53%
30	8	52%
Total	232	62,44 %

Fonte: Adaptado de Alarcon(2013)

Tabela 4.3 - Experimento 1: Suíte de teste gerada via PCDVT-A

Caso de Teste	Passos de Teste	Cobertura MC/DC
1	248	85,66%
Total	248	85,66%

Tabela 4.4 - Experimento 1: Suíte de teste gerada via PCDVT-RC

Caso de Teste	Passos de Teste	Cobertura MC/DC
1	6	53%
2	242	85,66%
Total	248	85,67 %

Porém, não foi possível executar as *suites* de teste devido ao tempo necessário para a execução das mesmas e, assim, não foi realizada medida de efetividade nesse experimento. Além disso, em termos de custo, não foi possível comparar os conjuntos de casos de teste do PCDVT-A e o PCDVT-RC com a abordagem manual descrita por Alarcón (2013), visto que estes testes não contemplam o modelo completo.

Tabelas 4.5 e 4.6 mostram os resultados obtidos para esse experimento, onde, na primeira coluna é mostrado o caso de teste, na segunda e terceira colunas o número de passos de teste obtidos para cada um dos testes. Na terceira coluna, o traço (-") denota que o teste foi removido no processo de minimização. A última linha sumariza os resultados obtidos, mostrando na segunda coluna o número total de passos de teste sem o processo de minimização e na terceira coluna o número total de passos de teste obtidos após a minimização.

A Figura 4.5 mostra os resultados obtidos, estando o gráfico em escala logarítmica. Cada coluna representa uma *suite* de teste, estando as obtidas através do mesmo padrão de especificação lado a lado.

Foram geradas, automaticamente, 2068 propriedades LTL para o modelo completo via o método PCDVT, resultando em 2068 testes na suíte de testes, para cada um dos padrões, sendo um teste para cada bloco do modelo. Conforme visto nas Tabelas 4.5 e 4.6, a suíte de testes PCDVT-A continha 512864 passos de teste enquanto a PCDVT-RC tinha 13096 passos de teste, sem a aplicação do algoritmo de minimização Greedy eIET.

Ao passar pelo processo de minimização, a suíte PCDVT-A foi reduzida a dois casos de teste com 572 passos de teste no total (Tabela 4.5). Já a suíte de testes PCDVT-RC foi reduzida a quatro casos de teste (Tabela 4.6) apresentando no total 639 passos de teste.

A *suite* PCDVT-A apresentou um número significativamente maior de passos de teste em relação à PCDVT-RC antes do processo de minimização. Entretanto, após a minimização o número de passos de teste da *suite* PCDVT-A foi inferior, indicando que a que esta apresenta um índice de redundância significativamente maior do que a PCDVT-RC e tornando-a inviável de ser utilizada sem o processo de minimização para modelos com alto nível de detalhamento. Esses resultados também mostram o alto desempenho do algoritmo de minimização Greedy eIET que atingiu índices de minimização superiores a 99%.

Tabela 4.5 - Experimento 2: Suíte de testes via PCDVT-A

Caso de Teste	Passos de Teste - sem EIET	Passos de Teste - com EIET
1	248	248
2	326	326
3	326	-
4	127	-
5	6	-
⋮	⋮	⋮
Total	512864	572

Tabela 4.6 - Experimento 2: Suíte de testes via PCDVT-A

Caso de Teste	Passos de Teste - sem EIET	Passos de Teste - com EIET
1	6	6
2	513	513
3	48	48
4	72	72
5	6	-
⋮	⋮	⋮
Total	13096	639

O processo de geração das suítes de teste por meio da ferramenta PCDVT, para as duas instâncias em conjunto, levou 87 horas, sendo aproximadamente 96% desse tempo (84 horas) dedicado à etapa de *Model Checking*.

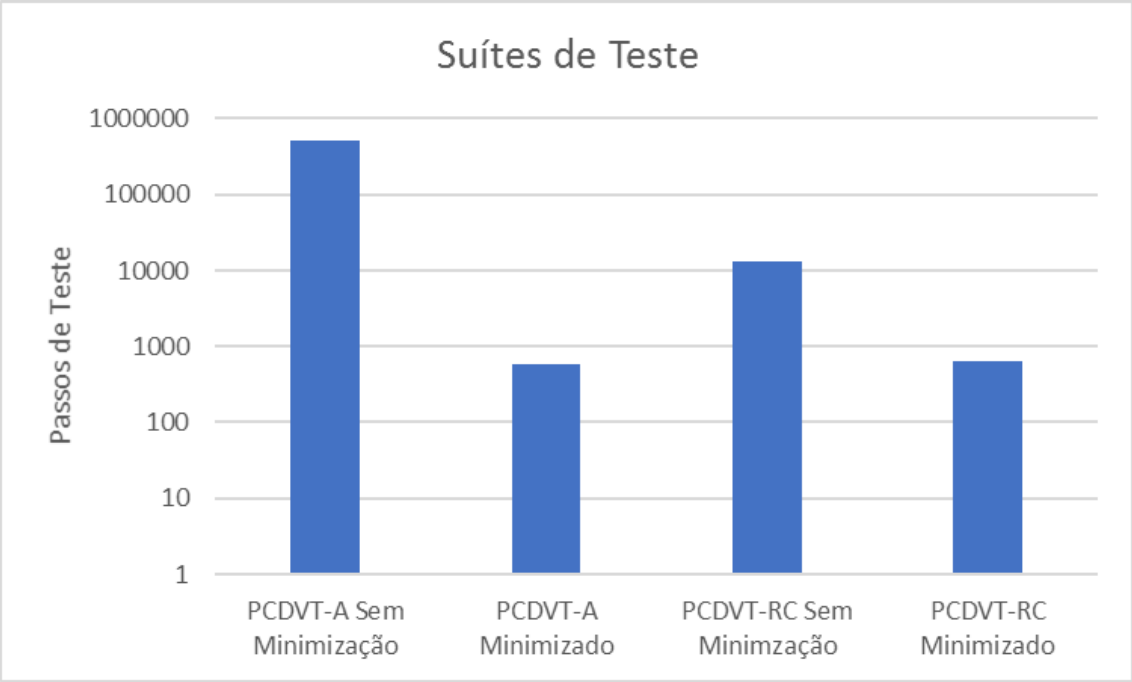
Uma observação final sobre experimentação deve ser salientada. Conforme mencionado na Seção 3.1.5, o trabalho de Mohalik et al. (2014) é o que mais se assemelha a esse trabalho de pesquisa proposto (PCDVT). No entanto, as ferramentas usadas e desenvolvidas no trabalho de Mohalik et al. (2014) não estão livremente disponíveis para uso e, assim, não foi possível comparar, via experimentação, o método PCDVT com tal trabalho.

4.4 Considerações finais sobre esse Capítulo

Esse capítulo apresentou a ferramenta PCDVT, uma instância do método PCDVT implementada em linguagem GO. Também foi apresentado todo o processo de execução da ferramenta, suas características e limitações.

O estudo de caso foi apresentado em maiores detalhes, mostrando as principais características do SCA do satélite Lattes-1.

Figura 4.5 - Experimento 2: Custo



Os resultados dos experimentos executados foram apresentados, mostrando que a técnica teve alta efetividade e foi bem sucedida no processo de minimização, apresentando um tamanho similar à *suite* de testes manual. Quando aplicado ao modelo completo, o método apresentou um percentual de minimização bastante alto.

O próximo capítulo apresenta as conclusões dessa dissertação de mestrado assim como os trabalhos futuros.

5 CONCLUSÃO

O processo de desenvolvimento de sistemas espaciais, devido ao alto custo e impossibilidade de manutenção, demanda ferramentas que assegurem a qualidade dos sistemas desenvolvidos, reduzindo assim o risco de falhas. Por conta disso, é comum o aplicação do desenvolvimento orientado a modelos, que permite trazer a etapa de Verificação e Validação para os períodos iniciais do projeto. Além disso, é necessário o uso de ferramentas e técnicas de teste que assegurem a maior cobertura de código possível, reduzindo o risco de falhas durante a operação.

Este trabalho propôs uma nova metodologia para geração automática de casos de testes à partir de LPVs de forma que apresentem uma alta cobertura MC/DC e, simultaneamente, sejam viáveis de ser executados durante o processo de desenvolvimento de um sistema.

Essa pesquisa foi aplicada a estudos de caso que são modelos LPVs para SCA similar ao do satélite Lattes-1, onde foram realizados dois experimentos. No primeiro experimento comparou-se os resultados obtidos por meio de duas instâncias do método/ferramenta PCDVT, uma usando padrão de Ausência em Escopo entre Q e R, e a outra usando Padrão de Resposta Encadeada em Escopo Global, com os testes manuais *ad-hoc* desenvolvidos para o sistema em termos de custo, número total de passos de teste, e efetividade, cobertura MC/DC do modelo. No segundo experimento, foi comparada a eficiência do processo de minimização de testes aplicada no PCDVT, onde as duas instâncias da ferramenta desenvolvida foram avaliadas em relação à quantidade de casos de teste gerados com e sem minimização.

Os resultados do primeiro experimento mostraram que ambas as instâncias do método PCDVT apresentam efetividade consideravelmente maior do que a obtida através da abordagem manual *ad-hoc* mantendo um custo de execução bastante próximo desta, sendo uma diferença de 16 passos de teste a mais para as *suites* geradas através do método proposto. O processo de minimização de casos de teste mostrou-se bastante eficiente contornando um problema comum na maioria das técnicas de geração automática de casos de teste que é a explosão combinatorial. O número de passos de teste obtidos pela técnica foi similar ao especificado manualmente de forma que a técnica se mostra viável para a geração de testes em modelos com alto nível de detalhamento. Além disso, houve um salto de 62% para 85% de cobertura MC/DC, sendo um ganho bastante expressivo.

No segundo experimento, onde foi considerado o modelo completo, o processo de

minimização de testes levou a uma redução próxima a 99% do número de passos de teste, mantendo o tamanho das *suites* de teste a aproximadamente 600 passos de teste.

Considera-se que o objetivo do trabalho foi alcançado, onde propôs-se um novo método para geração de casos de teste para modelos elaborados via LPV, com maior efetividade e com custo similar se comparado a uma abordagem ad hoc manual. Em termos de minimização, pode-se afirmar, portanto, que as suítes de teste geradas via o método PCDVT são competitivas em termos de custo. O método proposto foi completamente automatizado de forma que tornou possível sua incorporação ao processo completo de geração de testes.

Este trabalho mostra-se importante para projetos do INPE, apresentando uma alternativa para a geração automatizada de casos de teste que garante alta cobertura de modelos descritos em Linguagens de Programação Visual. Desse modo, o método proposto tem grande potencial para ser aplicado a projetos espacias futuros do INPE.

5.1 Trabalhos Futuros

Em etapas futuras, considera-se executar os testes gerados por meio do PCDVT para o modelo completo do SCA similar ao do satélite Lattes-1 (experimento 2), para comparar a efetividade da técnica em tal situação. Além disso considera-se efetuar outras avaliações experimentais com diferentes métricas de efetividade, incluindo análise de mutantes e escore de mutação. O resultado das análises de efetividade poderão, realmente, encontrar defeitos críticos nos modelos LPVs do SCA.

Além disso, considera-se aplicar o método/ferramenta PCDVT a outros estudos de caso do INPE e da área espacial como um todo, e integrar a ferramenta aos ambientes que apoiam o desenvolvimento baseado em modelos que o INPE utiliza.

Outro ponto importante é que este trabalho não considerou a análise de cobertura de blocos contendo *scripts*, devido à dificuldade de interpretação do mesmo e à necessidade de se anexar características de diversas LPVs. Além disso, a ferramenta PCDVT ainda não é capaz de analisar alguns tipos de decisões diferentes dos blocos "Se... então.."tais como blocos "*Switch... Case...*"e diversos outros tipos de blocos "*switches*"presentes em diversas LPVs. Esses consistem em pontos importantes de melhoria para essa pesquisa.

REFERÊNCIAS BIBLIOGRÁFICAS

ALARCÓN, P. N. d. S.; CARVALHO, F. G. M. d.; SIMÕES, A. R. Geração automática de casos de teste aplicada ao projeto de aocs de satélites artificiais. In: CONGRESSO BRASILEIRO DE AUTOMÁTICA., 20 set., 2014, Belo Horizonte. **Anais...** [S.l.], 2014. Access in: 20 jan. 2018. 2, 19, 44

ALARCON, P. N. d. S.; SANTIAGO JÚNIOR, V. A. d. Test case/step minimization for visual programming language models and its application to space systems. In: GERVASI, O.; MURGANTE, B.; MISRA, S.; BORRUSO, G.; TORRE, C. M.; ROCHA, A. M. A. C.; TANJAR, D.; APDUHAN, B. O.; STANKOVA, E.; CUZZOCREA, A. (Ed.). **Computational Science and Its Applications – ICCSA 2017**. [S.l.]: Springer, 2017. p. 160–175. ISBN 978-331962406-8. Access in: 20 jan. 2018. 3, 19

ALARCON, P. N. S. **Test Specification and Procedures for AOCS**. Coordenação de Engenharia e Tecnologia Espacial - Instituto Nacional de Pesquisas Espaciais. São José dos Campos, 2013. 47, 48, 49, 50, 51

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. 2. ed. [S.l.]: Cambridge University Press, 2017. 1

ARAUJO, H. A. B. **AOCS Design Specification**. Coordenação de Engenharia e Tecnologia Espacial - Instituto Nacional de Pesquisas Espaciais. São José dos Campos, 2012. 44

AWEDIKIAN, Z.; AYARI, K.; ANTONIOL, G. Mc/dc automatic test input data generation. In: 11TH ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION. **Proceedings...** [S.l.], 2009. p. 1657–1664. 8

AYAV, T. Prioritizing mcdc test cases by spectral analysis of boolean functions. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 27, n. 7, 2017. 11

BAIER, C.; KATOEN, J. **Principles of model checking**. USA: MIT Press, 2008. 3, 9, 12, 13, 14, 27

BALERA, J. M.; SANTIAGO JÚNIOR, V. A. d. T-tuple reallocation: an algorithm to create mixed-level covering arrays to support software test case generation. In: **Lecture Notes in Computer Science**. Springer International Publishing, 2015. v. 9158, p. 503–517. ISBN 9783319214092. Available from:

<http://link.springer.com/10.1007/978-3-319-21410-8_39>. Access in: 20 jan. 2018. 2

BERNARD, E.; LEGEARD, B.; LUCK, X.; PEUREUX, F. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. **Software: Practice and Experience**, Wiley Online Library, v. 34, n. 10, p. 915–948, 2004. 2

BRAY, T. The javascript object notation (json) data interchange format. **Internet Engineering Task Force**, 2014. Access in: 20 jan. 2018. 39, 42

CAMPOS, J.; ABREU, R. Encoding test requirements as constraints for test suite minimization. In: TENTH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY. **Proceedings...** [S.l.], 2013. p. 317–322. 2, 9, 10

CIMATTI, A.; CLARKE, E.; GIUNCHIGLIA, E.; GIUNCHIGLIA, F.; PISTORE, M.; ROVERI, M.; SEBASTIANI, R.; TACCHELLA, A. Nusmv 2: An opensource tool for symbolic model checking. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION. **Proceedings...** [S.l.]: Springer, 2002. p. 359–364. 18

DABNEY, J. B.; HARMAN, T. L. **Mastering simulink**. [S.l.]: Pearson, 2004. 21

DANDAN, G.; TIANTIAN, W.; XIAOHONG, S.; PEIJUN, M. A test-suite reduction approach to improving fault-localization effectiveness. **Computer Languages, Systems & Structures**, Elsevier, v. 39, n. 3, p. 95–108, 2013. 9, 10, 11

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de Software**. Brasil: Elsevier, 2007. 1, 2

DIAS NETO, A. C. **Seleção de técnicas de teste baseado em modelos**. 2009. 220 p. PhD Thesis (Thesis (PhD in Computing and Systems Engineering)) — Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil, 2009. 2

DONOVAN, A. A.; KERNIGHAN, B. W. **The Go programming language**. [S.l.]: Addison-Wesley Professional, 2015. 39

DORF, R. C.; BISHOP, R. H. **Modern control systems**. [S.l.]: Pearson, 2011. 19

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Property specification patterns for finite-state verification. In: SECOND WORKSHOP ON FORMAL

METHODS IN SOFTWARE PRACTICE. **Proceedings...** [S.l.], 1998. p. 7–15. 3, 14, 15, 23, 24, 26

EDELKAMP, S. Promela planning. **Model Checking Software**, Springer, p. 197–213, 2003. 19

ENOIU, E. P.; CAUSEVIC, A.; OSTRAND, T. J.; WEYUKER, E. J.; SUNDMARK, D.; PETTERSSON, P. Automated test generation using model checking: an industrial evaluation. **International Journal of Software Tools and Technology Transfer**, 2014. 9

ERWIG, M.; SMELTZER, K.; WANG, X. What is a visual language? **Journal of Visual Languages & Computing**, Elsevier, v. 38, p. 9–17, 2017. 17

FERRANTE, O.; FERRARI, A.; MARAZZA, M. Model based generation of high coverage test suites for embedded systems. **19th IEEE European Test Symposium**, v. 99, n. 2, p. 335–337, 2014. 2, 9

FERRANTE, O.; MARAZZA, M.; FERRARI, A. Formal specs verifier atg: a tool for model-based generation of high coverage test suites. **19th IEEE European Test Symposium**, v. 99, n. 2, p. 335–337, 2014. 9

FRASER, G.; WOTAWA, F. Mutant minimization for model-checker based test-case generation. In: ACADEMIC AND INDUSTRIAL CONFERENCE PRACTICE AND RESEARCH TECHNIQUES-MUTATION. **Testing:**. [S.l.], 2007. p. 161–168. 2, 9, 10, 11, 33

FRASER, G.; WOTAWA, F.; AMMANN, P. E. Testing with model checkers: a survey. **Software Testing, Verification and Reliability**, v. 19, n. 3, p. 215–261, 2009. 9

GENT, K.; HSIAO, M. S. Functional test generation at the rtl using swarm intelligence and bounded model checking. In: 22ND ASIAN TEST SYMPOSIUM. **Proceedings...** [S.l.], 2013. p. 233–238. 9

HANSELMAN, D. C.; LITTLEFIELD, B. **Mastering matlab 7**. [S.l.]: Pearson/Prentice Hall, 2005. 2, 17

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, p. 231–274, 1987. 2

HARMAN, M.; JIA, Y.; ZHANG, Y. Achievements, open problems and challenges for search based software testing. In: 2015 IEEE INTERNATIONAL

CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION. **Proceedings...** [S.l.]: IEEE, 2015. p. 1–12. 2

HELKE, S.; NEUSTUPNY, T.; SANTEN, T. Automating test case generation from z specifications with isabelle. In: INTERNATIONAL CONFERENCE OF Z USERS. **Proceedings...** [S.l.], 1997. p. 52–71. 2

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LüTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using formal specifications to support testing. **ACM Computing Surveys**, v. 41, n. 2, p. 1–76, 2009. 2

HOLLOWAY, C. M. Towards understanding the do-178c/ed-12c assurance case. **System Safety, incorporating the Cyber Security Conference 2012, 7th IET International Conference on**, IET, p. 1–6, 2012. 5, 19

HOLZMANN, G. J. The model checker spin. **IEEE Transactions on software engineering**, IEEE, v. 23, n. 5, p. 279–295, 1997. 19, 27

JANÍK, Z.; ŽÁKOVÁ, K. Online design of matlab/simulink and scilab/xcos block schemes. **Interactive Collaborative Learning (ICL), 2011 14th International Conference on**, p. 241–247, 2011. 2, 17, 21

KAPOOR, K.; BOWEN, J. Experimental evaluation of the variation in effectiveness for dc, fpc and mc/dc test criteria. In: 2003 INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING. **Proceedings...** [S.l.], 2003. 3, 8

KATRAGADDA, R.; TIRUMALA, S. S.; NANDIGAM, D. Etl tools for data warehousing: an empirical study of open source talend studio versus microsoft ssis. Institute of Electrical and Electronics Engineers (IEEE), 2015. 17

KUHN, D. R.; KACKER, R. N.; LEI, Y. **Introduction to combinatorial testing**. [S.l.]: Chapman and Hall/CRC, 2013. 2

KULKARNI, T. D.; KOHLI, P.; TENENBAUM, J. B.; MANSINGHKA, V. Picture: A probabilistic programming language for scene perception. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION. **Proceedings...** [S.l.], 2015. p. 4390–4399. 17

LI, B.-L.; LI, Z.-s.; QING, L.; CHEN, Y.-H. Test case automate generation from uml sequence diagram and ocl expression. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND SECURITY. **Proceedings...** [S.l.], 2007. p. 1048–1052. 2

LIN, C.; TANG, K.; KAPFHAMMER, G. M. Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests. **Information and Software Technology**, v. 56, p. 1322–1344, 2014. 2, 3, 4, 9, 10, 11, 34, 35

MACLAURIN, M. B. **The design of Kodu: A tiny visual programming language for children on the Xbox 360**. 2011. 241–246 p. 17

MALONEY, J.; RESNICK, M.; RUSK, N.; SILVERMAN, B.; EASTMOND, E. The scratch programming language and environment. **ACM Transactions on Computing Education (TOCE)**, ACM, v. 10, n. 4, p. 16, 2010. 17

MATHUR, A. P. **Foundations of software testing**. Delhi, India: Dorling Kindersley (India), Pearson Education in South Asia, 2008. 689 p. 1

MEENAKSHI, B.; BHATNAGAR, A.; ROY, S. Tool for translating simulink models into input language of a model checker. **Formal Methods and Software Engineering**, Springer, p. 606–620, 2006. 18

MILLER, S. P. Bridging the gap between model-based development and model checking. In: SPRINGER. **TACAS**. [S.l.], 2009. v. 2009, p. 443–453. 18

MOHALIK, S.; GADKARI, A. A.; YEOLEKAR, A.; SHASHIDHAR, K.; RAMESH, S. Automatic test case generation from simulink/stateflow models using model checking. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 24, n. 2, p. 155–180, 2014. 9, 18, 36, 52

MYERS, G. F. **The Art of Software Testing**. [S.l.]: John Wiley Sons, Inc., 2004. 2

NURSEITOV, N.; PAULSON, M.; REYNOLDS, R.; IZURIETA, C. Comparison of json and xml data interchange formats: a case study. **Caine**, v. 2009, p. 157–162, 2009. 42

OUIMET, M.; LUNDQVIST, K. Formal software verification: Model checking and theorem proving. **Embeded Systems Laboratory, Massachusetts Institute of Technology, Tech. Rep**, 2007. 12, 13

PACHECO, C.; LAHIRI, S. K.; ERNST, M. D.; BALL, T. Feedback-directed random test generation. In: 29TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** [S.l.], 2007. p. 75–84. 2

PETKE, J.; COHEN, M. B.; HARMAN, M.; YOO, S. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 9, p. 901–924, 2015. 2

PISACANE, V. L. **Fundamentals of space systems**. [S.l.]: Oxford University Press, USA, 2005. 1

PNUELI, A. The temporal logic of programs. In: 18TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE. **Proceedings...** [S.l.], 1977. p. 46–57. 13

PRADHAN, D.; WANG, S.; ALI, S.; YUE, T.; LIAAEN, M. Cbga-es: A cluster-based genetic algorithm with elitist selection for supporting multi-objective test optimization. In: 2017 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION. **Proceedings...** [S.l.], 2017. p. 367–378. 10

PRESSMAN, R. S. **Engenharia de Software: Uma Abordagem Profissional**. [S.l.]: AMGH, 2011. 7

QIU, D.; LI, B.; JI, S.; LEUNG, H. Regression testing of web service: a systematic mapping study. **ACM Computing Surveys (CSUR)**, ACM, v. 47, n. 2, p. 21, 2015. 11

SANTIAGO JÚNIOR, V. A.; SILVA, W. P.; VIJAYKUMAR, N. L. Shortening test case execution time for embedded software. In: IEEE INTERNATIONAL CONFERENCE ON SECURE SYSTEM INTEGRATION AND RELIABILITY IMPROVEMENT, 2., Yokohama. **Proceedings...** [S.l.]: IEEE Computer Society, 2008. p. 81–88. ISBN 9780769532. Access in: 20 jan. 2018. 1

SANTIAGO JÚNIOR, V. A. d. **SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications**. 264 p. PhD Thesis (PhD) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011-12-12 2011. Available from: <http://urlib.net/sid.inpe.br/mtc-m19/2011/11.07.23.30>. Access in: 20 jan. 2018. 1, 2

SANTIAGO JÚNIOR, V. A. d.; SILVA, F. E. C. d. From statecharts into model checking: A hierarchy-based translation and specification patterns properties to generate test cases. In: BRAZILIAN SYMPOSIUM ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING, 2. (SAST), 18-19 Sept., Fortaleza, CE. **Proceedings...** [S.l.], 2017. ISBN 978-145035302-1. Access in: 31 jan. 2018. 9

SINGH, R.; SANTOSH, M. Test case minimization techniques: A review. **International Journal of Engineering Research & Technology (IJERT)**, v. 2, n. 12, 2013. 10, 11

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing**: A tools approach. Waltham, MA, USA: Morgan Kaufmann Publishers, 2007. 456 p. 2, 9

WANG, S.; ALI, S.; GOTLIEB, A. Minimizing test suites in software product lines using weight-based genetic algorithms. **Proceedings of the 15th annual conference on Genetic and evolutionary computation**, ACM, p. 1493–1500, 2013. 2, 10

WERTZ, J. R.; LARSON, W. J. **Space Mission Analysis and Design**. Hawthorne, USA: Microcosm Press, 1999. 976 p. 1

WIESE, R.; EIGLSPERGER, M.; KAUFMANN, M. yfiles—visualization and automatic layout of graphs. In: **Graph Drawing Software**. [S.l.]: Springer, 2004. p. 173–191. 21

YAMADA, C.; MILLER, D. M. Using spin to check simulink stateflow models. **International Journal of Networked and Distributed Computing**, v. 4, n. 1, p. 65–74, 2016. 27, 28

YEOLEKAR, A.; UNADKAT, D.; AGARWAL, V.; KUMAR, S.; VENKATESH, R. Scaling model checking for test generation using dynamic inference. In: SIXTH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION. **Proceedings...** [S.l.]: IEEE, 2013. p. 184–191. 9

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 22, n. 2, p. 67–120, 2012. 2

ZENG, H.; MIAO, H.; LIU, J. Specification-based test generation and optimization using model checking. **Symposium on Theoretical Aspects of Software Engineering(TASE'07)**, 2007. 9

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.