Universidade Federal de Minas Gerais
Escola de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

Aumentando a Eficiência da Verificação Dinâmica de Propriedades em Sistemas Descritos em Alto Nível de Abstração por Meio da Utilização de Funções de Classificação Heurística Derivadas das Propriedades do Sistema

Alair Dias Júnior

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais como requisito parcial para obtenção do título de Doutor em Engenharia Elétrica.

Orientador: Prof. Diógenes C. da Silva Júnior

Belo Horizonte, Dezembro de 2012

Esta tese é dedicada à minha querida esposa Amália, que me apoiou com amor e paciência durante todo o meu período de pós-graduação.

Agradecimentos

Primeiramente, agradeço aos meus pais, Alair e Iêta, pelo apoio, incentivo, amor e, principalmente, por terem plantado a semente que me deu condições para chegar até este ponto de minha carreira acadêmica e profissional.

Agradeço ao meu irmão, Marcelo, por ouvir minhas ideias e pela paciência nos momentos quando estive ausente realizando esta pesquisa, as vezes impossibilitado de cumprir minhas obrigações de irmão mais velho.

Agradeço à minha esposa, Amália, a quem também dedico esta tese, que sempre me apoiou, mesmo quando meus estudos a privaram de minha companhia.

Agradeço a todos os meus familiares, em especial à madrinha Delaine, por indicar direções em momentos onde não encontrava as respostas certas sozinho.

Ao meu orientador e amigo, Diógenes, por estar sempre disposto a ouvir, por apontar os pontos fracos da pesquisa com pontaria certeira e por acreditar nas direções que tomei no desenvolvimento deste trabalho.

Aos meus amigos, que nunca se esqueceram de mim, mesmo quando não pude dar a todos a atenção que mereciam. Em especial ao Joel e ao Flávio Laper, que se dispuseram a ler trechos deste trabalho.

Por fim, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e à Fundação de Amparo a Pesquisa do Estado de Minas Gerais (FAPEMIG) pelo suporte financeiro para a conclusão desta pesquisa.

Resumo

A evolução das técnicas de projeto de circuitos integrados ocorrida nas duas últimas décadas permitiu um aumento expressivo na produtividade das equipes de projeto. O *gap* de produtividade, que foi durante muito tempo o principal obstáculo a ser vencido pela indústria de microeletrônica, deixou de ser o foco das atenções e o ciclo de desenvolvimento se deslocou de uma abordagem centrada no projeto para uma abordagem centrada na verificação. Hoje, mais de 70% dos recursos de um projeto de CI são gastos com a verificação. Diversas técnicas foram propostas para aumentar a eficiência e a eficácia do processo de verificação, entre elas, a verificação baseada em asserções tem ganhado uma posição de destaque. No entanto, métodos atuais de verificação baseada em asserções para descrições de alto nível de abstração dependem fortemente da estrutura interna do modelo, necessitando que todo o seu código fonte esteja disponível durante a verificação, ou ignoram completamente o sistema, não levando em conta informações importantes que podem ser inferidas a partir do seu comportamento. O método proposto neste trabalho combina as asserções com o modelo caixa-preta do sistema sob verificação de forma a criar funções heurísticas que podem ser utilizadas, juntamente com um algoritmo de otimização numérica, para gerar contraexemplos das propriedades do sistema. Diferentemente de outras abordagens de verificação dinâmica de propriedades, o método não se foca em aumentar a cobertura do conjunto de testes, mas realiza uma busca iterativa por vetores que violam a propriedade sob verificação, acelerando a busca por contraexemplos. Resultados experimentais mostram que a busca por contraexemplos utilizando o método proposto é ordens de magnitude mais eficiente em relação ao tempo do que a verificação baseada em vetores aleatórios.

Abstract

Over the last decade, the integrated circuit (IC) production flow has gradually shifted from *design centric* to *verification centric*, as the *design* methodologies, tools and techniques evolved. It is a well known fact that on the modern IC design cycle more than 70% of the total development resources are spent on verification. In recent years, several techniques have been proposed to address this issue. Among them, assertion-based verification has been regarded as the most promising approach. However, current assertion-based verification methods for high-level designs either heavily depend on the internal structure of the model, requiring the entire source code to be available during the verification, or completely ignore the system, disregarding important information that can be derived from its behaviour. The method proposed in this work conjoins the assertions with the black-box model of the system in order to derive heuristic functions that can be used by an optimization algorithm to generate counterexamples of the design properties. Differently from other dynamic property checking approaches, the method does not focus on increasing the coverage of the test set, but iteratively searches for property violations. Experiments show that our method is orders of magnitude more time efficient in finding property violations than random simulation.

Lista de Figuras

Lista de Tabelas

Lista de Definições e Teoremas

Lista de Abreviaturas e Siglas

ABV *Assertion-Based Verification*

AMS *Analog and Mixed Signal*

ASIC *Application Specific Integrated Circuit*

ASM *Abstract State Machine*

BDD *Binary Decision Diagram*

BMC *Bounded Model Checking*

CAD *Computer Aided Design*

CI Circuito Integrado

CMOS *Complementary Metal-Oxide Semiconductor*

CPU *Central Processing Unit*

DPC *Dynamic Property Checking*

DUV *Design Under Verification*

FCBA Função de Classificação Baseada em Asserção

FIFO *First In First Out*

FND Forma Normal Disjuntiva

FPGA *Field Programmable Gate Array*

FSM *Finite State Machine*

HDL *Hardware Description Language*

IC *Integrated Circuit*

IIR *Infinite Impulse Response*

IP *Intellectual Property*

LP Lógica Proposicional

LPE Lógica Proposicional Estendida

LTL *Lógica Temporal Linear*

LTLE Lógica Temporal Linear Estendida

MDC Máximo Divisor Comum

NA Não Aplicável

PSL *Property Specification Language*

RAM *Random Access Memory*

RISC *Reduced Instruction Set Computer*

RTL *Register Transfer Level*

SASS *Self Adaptive Stepsize Search*

SAT *Boolean Satisfiability*

SoC *System-on-Chip*

STL *Standard Template Library*

TLM *Transaction Level Modeling*

VHDL VHSIC *Hardware Description Language*

VHSIC *Very High Speed Integrated Circuits*

VLSI *Very Large Scale Integrated*

VM Verificação de Modelos

Lista de Símbolos

$\emptyset$	Conjunto Vazio
$\neg$	Conectivo Lógico da Negação
$\wedge$	Conectivo Lógico da Conjunção
$\vee$	Conectivo Lógico da Disjunção
$\rightarrow$	Conectivo Lógico da Condicional
$\leftrightarrow$	Conectivo Lógico de Bicondicional
$\forall$	Quantificador Universal
$\exists$	Quantificador Existencial
α	Fórmula da LP e da LPE
β	Fórmula da LP e da LPE
$\mathcal{A}_P$	Asserção da Propriedade P
Δ	Alfabeto de Saída de uma FSM
δ	Função de Transição de uma FSM
$\hat{\delta}$	Função de Transição Estendida de uma FSM
$\mathcal{D}$	Função Bijetora de Mapeamento Palavra Símbolo
d	Símbolo de um Alfabeto
η	Variável da LPE e da LTLE
ϵ	Constante Real
$\mathcal{E}$	Função de Avaliação das Variáveis
E	Conjunto de Estados de uma FSM
$\mathbf{F}$	Operador *eventualmente* da LTL
$f_{\mathcal{A}_P}$	FCBA da asserção $\mathcal{A}_P$
Γ	Conjunto de Variáveis
γ	Variável da LPE e da LTLE
$\mathbf{G}$	Operador *globalmente* da LTL
κ	Constante Real
λ	Palavra Vazia

μ	Constante Real
N	Constante Natural
$\mathbb{N}$	Conjunto dos Números Naturais
n	Constante Inteira
$\Omega_{\mathcal{A}_P}$	Conjunto dos Contraexemplos da Asserção $\mathcal{A}_P$
φ	Fórmula da LPE ou da LTLE
Ψ	Conjunto de Variáveis
$\mathbb{Q}$	Conjunto dos Números Racionais
$\mathbb{R}$	Conjunto dos Números Reais
Σ	Somatório ou um Alfabeto de uma Linguagem Formal
σ	Função de Saída de uma FSM
$\hat{\sigma}$	Função de Saída Estendida de uma FSM
$\mathcal{S}$	Função Bijetora de Mapeamento Palavra Símbolo
s	Símbolo de um Alfabeto
τ	Constante Real
$\mathbf{U}$	Operador *até* da LTL
$\mathcal{V}$	Função de Avaliação das Variáveis
$\mathbf{W}$	Operador *até fraco* da LTL
w	Palavra de uma Linguagem Formal
$\mathbf{X}$	Operador *próximo* da LTL
$\mathbb{Z}$	Conjunto dos Números Inteiros

Sumário

Capítulo 1

Introdução

A evolução das técnicas de projeto de circuitos integrados (CI) ocorrida nas duas últimas décadas, principalmente o advento do projeto em alto nível de abstração, o reúso de componentes por meio da utilização de blocos IP (*Intellectual Property*) e o desenvolvimento baseado em plataformas (CLAASEN, 2003; BERGAMASCHI; COHN, 2002; GAJSKI et al., 2000; BRICAUD, 1999; CHANG et al., 1999), permitiu um aumento expressivo na produtividade das equipes de projeto de CIs. O *gap* de produtividade, que foi durante muito tempo o principal obstáculo a ser vencido pela indústria de microeletrônica, deixou de ser o foco das atenções e o ciclo de desenvolvimento se deslocou de uma abordagem centrada no projeto para uma abordagem centrada na verificação. Garantir o funcionamento correto do sistema passa a ser o grande problema enfrentado no desenvolvimento de um SoC (*System-on-Chip*) moderno, consumindo mais de 70% dos recursos disponíveis (RANJAN; COELHO; SKALBERG, 2009; BERGERON, 2003).

Diversas técnicas foram propostas para aumentar a eficiência e a eficácia do processo de verificação destes sistemas complexos, entre elas, a verificação baseada em asserções (ABV[1]) tem ganhado uma posição de destaque. Nesta técnica, um conjunto de propriedades é desenvolvido para descrever o comportamento lógico e temporal desejado do DUV (*Design Under Verification*), sendo que cada propriedade fornece uma especificação parcial do sistema. Por ser uma especificação parcial, cada uma das propriedades é mais fácil de escrever, manter e testar do que um modelo de referência completo. Além disso, quando escritas utilizando linguagens de especificação de propriedades, as propriedades, na forma de asserções, podem ser verificadas utilizando tanto métodos estáticos (verificação de modelos[2]) quanto métodos dinâmicos (baseados em simulação).

[1]Do inglês *Assertion Based Verification*. (Tradução Nossa).
[2]Do inglês *Model Checking*. (Tradução Nossa).

Apesar dos recentes avanços nos métodos estáticos, a escalabilidade ainda é um problema fundamental no uso de verificação de modelos (METTA, 2011). O conhecido problema da explosão de estados dificulta a aplicação das técnicas de verificação de modelos (VM) no nível de sistema (FUJITA; GHOSH; PRASAD, 2008), restringindo esta abordagem ao nível de blocos (GROSSE; DRECHSLER, 2010). Além disso, a verificação por meio de VM não é tão facilmente automatizável quanto a validação por meio de simulação, usualmente requerendo intenso esforço manual (METTA, 2011). Escalabilidade, porém, não é a única dificuldade encontrada na aplicação de VM. Para se utilizar esta técnica, o DUV deve estar descrito por meio de modelos formais, como autômatos de Büchi, estruturas de Kripke (KROPF, 1999, p. 155), e outros modelos similares. Traduzir as descrições de alto nível para estes modelos formais é uma tarefa extremamente complexa, uma vez que as descrições de alto nível de abstração comumente utilizam construções não sintetizáveis. Ademais, estas descrições podem conter referências a bibliotecas de terceiros sem código fonte disponível, sem o qual é impossível se criar modelos formais.

Por todos estes motivos, a verificação baseada em simulação, também conhecida como validação, ainda é a abordagem mais adotada para a verificação de sistemas descritos em alto nível de abstração (FUJITA; GHOSH; PRASAD, 2008). Os métodos baseados em simulação escalam bem com o tamanho do DUV (WILE; GOSS; ROESNER, 2005) e podem ser aplicados de forma eficiente a qualquer projeto em quase todos os níveis de abstração (BHADRA et al., 2007). No entanto, garantir que uma implementação está correta por meio de simulações requer que todas as possíveis combinações de valores de entrada do sistema sejam aplicadas ao modelo, uma tarefa impraticável para a grande maioria dos projetos reais. Consequentemente, apenas um pequeno subconjunto dos vetores de entrada é aplicado durante a validação. Escolher vetores representativos que exercitem de forma eficaz o DUV ainda é um problema de grande relevância e muitas abordagens foram propostas para lidar com esta tarefa. Entre elas, a verificação dinâmica de propriedades (DPC[3]) dirigida por cobertura tem sido vista como uma abordagem promissora. Porém, ainda há muito a ser feito para que a geração de vetores de teste para a DPC de sistemas descritos em alto nível de abstração seja eficiente e efetiva. A geração de vetores de teste para a DPC de sistemas descritos em alto nível de abstração é o tema deste trabalho.

[3]Do inglês *Dynamic Property Checking*. (Tradução Nossa).

1.1 Motivação

Os métodos mais recentes para a geração de vetores de teste para a DPC de projetos de CIs podem ser classificados dentro de duas categorias, quando se considera a utilização do DUV durante a fase de geração de testes. Na primeira categoria, enquadram-se os métodos que ignoram completamente a descrição do modelo, utilizando somente a especificação do sistema para a geração dos vetores de teste. A desvantagem destes métodos é que, ignorando o DUV durante a validação, informações importantes que podem ser derivadas a partir da sua estrutura interna e de seu comportamento deixam de estar disponíveis, resultando na criação de testes irrelevantes ou redundantes para determinada implementação ou na criação de um conjunto de testes que não exercita a implementação de forma completa e eficaz. A segunda categoria é formada por métodos que operam sobre o código fonte do modelo, seja por meio de análise estática, ou pela inclusão de construções ao longo do código com o intuito de extrair informações durante a validação. Esta segunda categoria apresenta diversas desvantagens, sendo as principais a restrição ao estilo do código imposta pela ferramenta de análise estática, a necessidade da disponibilidade do código fonte durante a verificação, o que restringe o uso de bibliotecas de terceiros com código fonte fechado, e a dificuldade para gerar testes que revelem faltas de omissão, quando algum requisito do sistema não foi implementado no modelo.

Além destas limitações, os métodos atuais para a geração de vetores de teste para DPC, em geral, consideram que a propriedade sendo verificada é composta por literais que podem ser modelados na forma de *bits*. Apesar desta restrição ser aceitável em projetos descritos nos níveis de abstração mais baixos, como no RTL (*Register Transfer Level*), e também em sistemas dominados por controle, nos níveis mais altos de abstração e, principalmente, em sistemas dominados por dados, esta consideração pode ter como consequência direta a degradação tanto da eficiência quanto da eficácia apresentadas pelo método. Considere, por exemplo, a expressão $g(x) > 100$, onde x é um número inteiro (ou $x \in \mathbb{Z}$) e a função $g : \mathbb{Z} \mapsto \mathbb{Z}$ representa um sistema qualquer. Podem existir inúmeros valores de x que respeitem esta relação, assim como inúmeros valores que a violem. Definir apenas que a expressão deve ser verdadeira ou falsa durante os testes ainda deixa uma pergunta importante a ser respondida: quais valores de x são os melhores para validar o sistema?

1.2 Formulação do Problema e da Hipótese

Dado o contexto apresentado nas seções anteriores, pode-se formular uma pergunta norteadora para este trabalho:

> *Como gerar eficientemente vetores de teste para sistemas descritos em alto nível de abstração e sistemas dominados por dados sem que seja necessário recorrer ao código fonte do modelo?*

Acredita-se que as informações intrínsecas das asserções sendo verificadas podem ser combinadas com uma análise do comportamento do DUV durante a simulação para aprimorar, de modo iterativo, os vetores de testes, guiando de forma eficiente a simulação para traços que exponham faltas oriundas de defeitos de implementação ou de especificação, sem que seja necessário recorrer ao código fonte do modelo. Esta é a hipótese a ser testada neste trabalho.

1.3 Objetivos

O objetivo principal deste trabalho é o desenvolvimento de um método para aprimorar a busca por contraexemplos de asserções na DPC de sistemas descritos em alto nível de abstração, que utilize o DUV como uma caixa-preta, não recorrendo ao seu código fonte. Dado este objetivo principal, foram elaborados objetivos específicos, de forma a direcionar os esforços do desenvolvimento deste trabalho. Estes objetivos são listados a seguir:

- Criar uma definição formal para o método de DPC desenvolvido;
- Desenvolver um protótipo para uma ferramenta de *software* que implemente o método proposto em sistemas descritos em alto nível de abstração;
- Realizar um estudo de caso da aplicação da ferramenta desenvolvida;
- Indicar direções para trabalhos futuros para que o método proposto possa ser integrado de modo eficiente ao fluxo de projeto de circuitos integrados.

1.4 Justificativa

O desenvolvimento de um método que utilize o modelo caixa-preta do DUV para aprimorar a busca por contraexemplos de asserções se justifica por ser uma abordagem mais geral do que outras abordagens que se baseiam na análise do código fonte, podendo ser aplicada a qualquer modelo, independentemente do tipo de descrição empregada. Além disso, a utilização do comportamento do DUV para aprimorar, de modo iterativo, o conjunto de vetores de teste, potencialmente, aumenta a eficiência e a eficácia do método quando comparado às técnicas que utilizam apenas as asserções durante esta etapa.

Por sua vez, a formalização do método proposto é desejável por dois motivos principais. Primeiramente, uma definição formal, juntamente com um conjunto de teoremas que provem a fiabilidade da abordagem, permitem que o método seja aplicado com segurança à etapa de verificação, considerada a mais crítica no fluxo de projeto dos CIs modernos. Além disso, a formalização do método permite que este seja facilmente adaptado às diferentes linguagens de descrição de *hardware* e aos complexos ambientes de verificação utilizados na atualidade.

Mesmo com a formalização do método, o desenvolvimento de uma ferramenta que implemente as funcionalidades básicas para o emprego da abordagem é necessário para que seja oferecida uma prova de conceito de seu funcionamento. Uma vez implementada, a ferramenta será utilizada em um estudo de caso que exemplifique a sua aplicação em ambientes de validação.

Por fim, com o método e a ferramenta validados, é necessário definir como esta abordagem se encaixa de forma eficiente dentro do fluxo de projeto de circuitos integrados e, também, quais características adicionais são necessárias para que sua utilização seja realmente eficaz.

1.5 Metodologia

Para se definir o tema deste trabalho, foi feita uma análise das pesquisas mais recentes sobre verificação de sistemas descritos em alto nível de abstração. As principais fontes para esta análise foram artigos de conferências e periódicos, bem como dissertações e teses relacionadas ao tema. A análise revelou que grande parte dos trabalhos atuais

sobre verificação de sistemas descritos em alto nível de abstração aborda a verificação baseada em asserções, tanto na forma de verificação de modelos, quanto na forma de verificação dinâmica de propriedades. Uma vez que a simulação ainda é a abordagem predominante para a verificação de descrições de alto nível de abstração, optou-se por direcionar a presente pesquisa para a verificação dinâmica de propriedades.

Usualmente, os trabalhos sobre DPC concentram-se na criação de monitores para verificação das propriedades durante a execução do modelo. Nota-se que a geração de vetores de teste para a DPC de sistemas descritos em alto nível de abstração é abordada de modo simplista, pois, dentro dos limites da pesquisa realizada, não foram encontrados trabalhos que lidem diretamente com tipos de dados mais complexos. Todos os trabalhos direcionados para DPC que foram analisados consideram que os sinais envolvidos nas propriedades são do tipo *bit*, o que é uma simplificação exagerada dos sistemas reais. Desta forma, para sistemas descritos em alto nível de abstração cujas propriedades envolvam operadores relacionais sobre tipos de dados inteiros ou reais, a única alternativa disponível para a geração automática de vetores de teste é a simulação aleatória, e suas variações.

Detectada esta lacuna, passou-se à busca por uma solução. Foi verificado que o método apresentado em (DIAS JÚNIOR, 2008), originalmente proposto para gerar vetores de teste com o intuito de aumentar a cobertura de código, poderia ser adaptado para aumentar a eficiência da busca por contraexemplos de propriedades formais. O método original deveria ser reformulado, e desta forma, uma definição formal foi elaborada. Do modo como havia sido definido originalmente, o método não poderia ser aplicado a sistemas com memória. Esta limitação também foi removida neste trabalho.

Como prova de conceito, o método proposto foi concretizado por meio de uma ferramenta de *software*. Nesta etapa, considerou-se que o DUV estaria descrito utilizando a linguagem SystemC (IEEE, 2012). SystemC foi escolhida por ser o padrão *de-facto* para a descrição em alto nível de abstração de sistemas complexos. Além disto, por ser uma extensão da linguagem C++, SystemC permite que a a ferramenta seja facilmente integrada à descrição do modelo, desde que esta seja desenvolvida utilizando o padrão C++. Esta escolha não restringe de forma alguma a aplicabilidade do método proposto, que é genérico o bastante para ser aplicado a qualquer linguagem de descrição de *hardware* que permita a simulação do modelo.

Definido o método e criada a ferramenta, foi escolhido um estudo de caso para exemplificar sua utilização. A DPC de filtros digitais que empregam representação de números reais utilizando notação de ponto-fixo foi escolhida por três motivos: 1) trata-se de um problema de alta complexidade do ponto de vista de validação por simulação, pois os filtros digitais possuem um número de estados internos proibitivo para uma validação exaustiva e suas faltas não possuem características em comum que permitam sintetizá-las em um modelo eficaz para a geração de vetores de teste aleatórios; 2) filtros digitais são sistemas dominados por dados, muito comuns em sistemas embutidos, e que podem ser implementados tanto em *hardware* quanto em *software*, permitindo a validação da ferramenta em ambientes de simulação e emulação com diferentes características; e 3) filtros digitais são sistemas com memória, o que possibilita uma validação completa do método proposto.

Assim, a presente pesquisa pode ser classificada como *aplicada*, quanto à sua natureza, pois "objetiva gerar conhecimentos para aplicação prática e dirigidos à solução de problemas específicos"(SILVA; MENEZES, 2005). Quanto à abordagem, a pesquisa é tanto qualitativa quanto quantitativa. É qualitativa pois derivou da análise de parâmetros não mensuráveis, como a deficiência das técnicas atuais para validar sistemas dominados por dados e a sua restrição quanto à disponibilidade do código fonte do modelo, e é quantitativa nas comparações de desempenho entre o método proposto e outras abordagens. Quanto aos procedimentos, o trabalho se baseou em uma pesquisa bibliográfica para definir o estado da técnica e determinar a lacuna do processo de verificação não abordada pelos trabalhos atuais, bem como em um estudo de caso para exemplificar a aplicação do método proposto.

1.6 Contribuições

A contribuição principal deste trabalho é a proposição, formulação e validação de um método para a verificação dinâmica de propriedades direcionado para sistemas descritos em alto nível de abstração que utiliza o modelo caixa-preta do DUV para aprimorar a busca por contraexemplos. Além desta contribuição principal, também são contribuições deste trabalho:

1. A definição formal do conceito de sistemas sem memória e sistemas com memória, a partir da análise de máquinas de estados finitos;

2. A definição da sintaxe e da semântica para uma lógica proposicional estendida (LPE), utilizada para descrever o comportamento de sistemas sem memória modelados em alto nível de abstração;

3. A definição de uma extensão da sintaxe e da semântica da lógica temporal linear (LTL), que permite o emprego de operadores relacionais em suas fórmulas;

4. O mapeamento do problema de verificação de propriedades em sistemas sem memória para um problema de otimização numérica;

5. A definição de uma semântica alternativa para a LPE, que permite definir funções heurísticas para a classificação de vetores de teste com relação à sua proximidade em violar a descrição definida pela fórmula da LPE;

6. A proposição de um método para desdobrar o funcionamento de uma máquina de estados finitos ao longo do tempo que permite que esta seja tratada como um sistema sem memória;

7. O desenvolvimento de um protótipo para uma ferramenta que implementa o método proposto na verificação de sistemas descritos utilizando a linguagem SystemC;

8. Um estudo de caso sobre a detecção de condições de *overflow* em filtros digitais que empregam notação do tipo ponto fixo.

1.7 Estrutura do Texto

O texto deste trabalho está estruturado de forma a facilitar o entendimento da pesquisa realizada. Assim, no capítulo 2, é apresentado um panorama geral sobre projeto e verificação de CIs descritos em alto nível de abstração. Também é feita uma revisão dos trabalhos recentes relacionados à verificação baseada em asserções, com o intuito de oferecer uma visão geral do estado da técnica sobre verificação dinâmica de propriedades.

O capítulo 3 introduz os principais conceitos, definições e ferramentas empregadas no desenvolvimento da presente pesquisa, bem como são estabelecidos os significados

de algumas palavras chave utilizadas ao longo do texto. As contribuições deste trabalho se iniciam nas seções 3.2 e 3.3, onde são apresentadas as extensões para a lógica proposicional e para a lógica temporal linear.

Após esta revisão da literatura e da introdução dos conceitos preliminares, inicia-se a apresentação da contribuição principal desta tese. O capítulo 4 apresenta o mapeamento do problema da busca por contraexemplos de propriedades em projetos de CIs para um problema de otimização numérica. O método para a DPC heurística de sistemas sem memória também é definido neste capítulo. Finalizando a contribuição principal, no capítulo 5, o método apresentado no capítulo 4 é adaptado para que possa ser empregado em sistemas com memória.

No capítulo 6, é apresentada uma ferramenta que implementa o método proposto para a verificação de sistemas descritos em SystemC. A utilização da ferramenta é exemplificada por meio de um estudo de caso de detecção de condições de *overflow* em filtros digitais modelados usando aritmética de ponto fixo. Os resultados da aplicação da ferramenta são apresentados neste capítulo.

O trabalho finaliza no capítulo 7, com as considerações finais, onde são apresentadas discussões sobre o método proposto, sobre os resultados dos experimentos e sobre a ferramenta desenvolvida. Conclusões e direções para trabalhos futuros também são apresentadas neste capítulo.

Capítulo 2

Revisão da Literatura

Este capítulo apresenta uma revisão da literatura, concentrando-se em trabalhos recentes que abordam a verificação baseada em asserções de sistemas descritos em alto nível de abstração. Um maior enfoque é dado aos trabalhos que utilizam SystemC, uma vez que esta linguagem tem se tornado o padrão *de-facto* para a descrição em alto nível de abstração de sistemas complexos (GROSSE; LE; DRECHSLER, 2010).

Na seção 2.1, são estabelecidos os conceitos de defeitos, falhas, faltas e erros, para que estes termos sejam empregados de forma consistente ao longo do trabalho. A seção 2.2 apresenta uma visão geral sobre o ciclo de projeto de CIs e sobre o projeto em alto nível de abstração. Trabalhos recentes sobre verificação baseada em asserções são apresentados na seção 2.3.

2.1 Defeitos, Falhas, Faltas e Erros

No decorrer deste trabalho, utilizou-se as definições para os conceitos de defeitos, falhas, faltas e erros conforme apresentadas em (PARHAMI, 1997). Segundo Parhami, um sistema pode estar em qualquer dos estados da figura 2.1, adaptada de (PARHAMI, 1997), que representa um diagrama de estados de um modelo multinível de confiabilidade.

Resumidamente, um sistema pode possuir *defeitos*. Alguns estados do sistema podem expor tais *defeitos*, o que resulta no desenvolvimento de *faltas*, que são definidas como valores ou decisões incorretas dentro do sistema. Se uma *falta* é exercitada, ela pode contaminar os dados através do sistema, causando *erros*. Um *erro* pode levar a um *mau funcionamento* do sistema, dependendo do projeto do sistema ou da sua tolerância aos *erros*. Um *mau funcionamento* pode levar à *degradação* do funcionamento do sistema e esta *degradação* pode eventualmente levar à *falha* do sistema. A definição de *falha*,

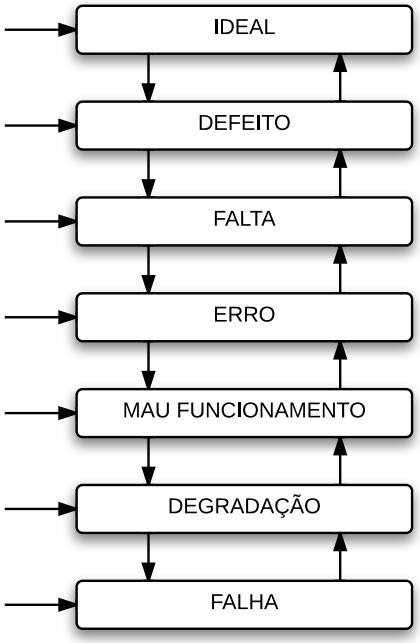


Figura 2.1: Estados e Transições no Modelo Multinível de Confiabilidade

portanto, está relacionada apenas ao propósito do sistema, ou seja, à sua capacidade de realizar a função para o qual foi concebido. A seguir, são dados dois exemplos destes conceitos em projeto de sistemas VLSI.

Exemplo 1

Um bloco IP desenvolvido para calcular o valor de 2^n, dada uma entrada n inteira, possui um *defeito*, pois não considera os valores de $n < 0$. Na utilização original deste bloco, isto não foi um problema, pois o sistema nunca atingiu valores de $n < 0$, não gerando uma *falta*. Se o valor calculado por este bloco for utilizado no caminho dos dados de um processador quando o valor da entrada n for menor do que 0, a *falta* contaminará o dado, resultando em um *erro*. Como não existe um sistema de correção de *erros* que atue sobre esta condição (trata-se de um defeito de projeto), o *erro* leva a um estado de *mau funcionamento* que resultará na *degradação* do desempenho do sistema, caso o estado possa ser contornado usando microcódigo, ou em uma *falha* completa do sistema, caso contrário.

Exemplo 2

Um bom modelo de faltas no nível de portas lógicas CMOS (*Complementary Metal-Oxide Semiconductor*) é o *Stuck-at*, que considera que uma entrada ou saída de uma porta lógica esteja fixa no valor lógico 0 ou 1. Este modelo de faltas está fortemente relacionado com *defeitos* de fabricação comuns, como circuitos abertos e curto circuitos. Simulando uma *falta* neste modelo, é possível verificar como estas se manifestam no comportamento do sistema, a partir dos *erros* propagados para as suas saídas.

2.2 Ciclo de Projeto de CI

Nas décadas de 1980 e 1990, a evolução dos processos de fabricação de circuitos integrados aumentou a área de silício disponível no *die*[1] a uma velocidade que não foi acompanhada pelas técnicas e ferramentas de CAD (*Computer Aided Design*). Este conhecido *gap* de produtividade (CHANG et al., 1999) levou a uma busca por novas metodologias de projeto que permitissem um aumento da eficiência das equipes de desenvolvimento. Uma das soluções encontradas foi o aumento no nível de abstração da descrição dos projetos de CIs.

A elevação do nível de abstração da descrição dos modelos, por meio de linguagens de descrição de *hardware*, permitiu que sistemas complexos pudessem ser projetados e verificados de forma mais ágil, aumentando a produtividade das equipes de projeto. Esta elevação do nível de abstração do projeto de CI aproximou a indústria de microeletrônica da indústria de *software*, processo que culminou no reúso de componentes por meio da utilização de blocos IP e no projeto baseado em plataformas (CLAASEN, 2003; BERGAMASCHI; COHN, 2002; GAJSKI et al., 2000; BRICAUD, 1999; CHANG et al., 1999), conceitos equivalentes às bibliotecas e aos *frameworks*[2] empregados no processo de desenvolvimento de *softwares* complexos. A figura 2.2, adaptada de (VAHID; GIVARGIS, 2002) mostra um paralelo entre a evolução do projeto de CI e de sistemas de *software*.

Uma consequência natural da elevação do nível de abstração é o emprego de refinamentos sucessivos no projeto do sistema. Inicialmente, o sistema é descrito em um nível de abstração elevado e, quando o modelo está suficientemente validado, o sistema

[1] Pastilha de material semicondutor na qual é impresso o circuito integrado.(Nota do Autor).

[2] Um *Framework* é "uma aplicação semi-completa, que pode ser especializada para criar aplicações customizadas."(FAYAD; SCHMIDT, 1997, Tradução Nossa).

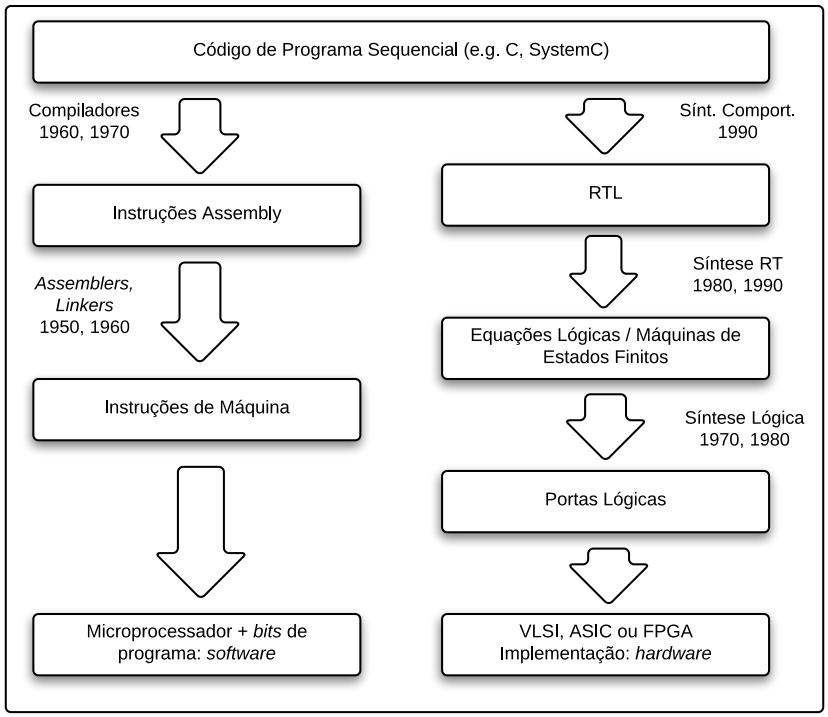


Figura 2.2: Paralelo entre a Evolução da Abstração em Sistemas de Software e Hardware

é refinado para um nível mais baixo, onde mais detalhes são incluídos. O processo de refinamento prossegue até que o sistema esteja em um nível adequado para a criação do circuito físico.

Para uma utilização mais efetiva do processo de refinamentos sucessivos, uma abordagem do tipo *top-down* com separação de interesses (*separation of concerns*), que mais tarde foi chamada por Keutzer et. al. (KEUTZER et al., 2000) de ortogonalização de interesses, é de fundamental importância e vários modelos foram propostos na literatura.

Em (GAJSKI; KUHN, 1983) é proposta uma divisão da representação do sistema em três domínios distintos: estrutural, funcional e geométrico. Cada um dos domínios pode ser representado por um eixo. O processo de refinamento é definido como um caminho sobre o mesmo eixo em direção à origem, enquanto o processo de síntese é definido como um caminho entre os eixos estrutural e geométrico. Neste modelo não há uma representação clara para as características de tempo.

O trabalho de (SILVA JR, 2001) apresenta uma representação do sistema em quatro domínios quase independentes: Fluxo-de-Dados; Tempo e Sequenciamento; Estrutural; e Físico. As ligações entre os eixos são definidas como ligações de operação ou ligações de realização. As ligações de operação interconectam os eixos de Fluxo-de-Dados,

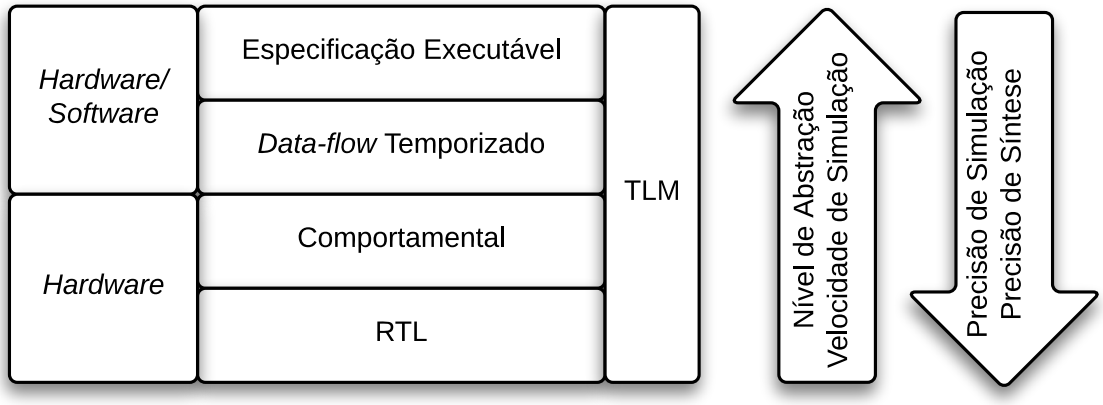


Figura 2.3: Níveis de abstração e velocidade de simulação

Tempo e Sequenciamento e Estrutural enquanto as ligações de realização interconectam os eixos Estrutural e Físico. Os refinamentos ocorrem nas ligações de operação, sendo que pode-se refinar qualquer um dos três aspectos desta ligação de forma quase independente, caminhando-se em direção à origem do eixo.

Apesar de não proporem um conjunto de níveis de abstração, estes trabalhos levantam a possibilidade de uma classificação das descrições do sistema em níveis de abstração de acordo com características do modelo, uma linha de pensamento que continua sendo empregada até hoje. Vários outros trabalhos tentaram definir um conjunto de níveis de abstração para guiar o projeto de SoCs, muitos deles citados em (DIAS, 2007) e em (PAPA, 2006). No entanto, uma taxonomia para níveis de abstração não é universalmente aceita. A figura 2.3 apresenta uma divisão entre os níveis de abstração proposta por (DIAS, 2007) a partir da síntese do trabalho de vários outros autores. Nela, existem 4 níveis de abstração, dois onde a divisão entre *hardware* e *software* não existe e dois outros níveis onde essa divisão está bem caracterizada. O nível RTL é o último tratado no trabalho de (DIAS, 2007), pois este é o último nível suportado pelo SystemC, linguagem utilizada no trabalho.

A *Especificação Executável* é o mais alto nível proposto por (DIAS, 2007) e traduz diretamente os requisitos do consumidor para um modelo unicamente funcional. Neste modelo, não existe noção de temporização nem preocupação com a implementação final. Além de ser uma prova de conceito do produto, a especificação executável também é utilizada como referência para o comportamento que deve ser apresentado pelos demais níveis de abstração.

Abaixo da *Especificação Executável*, está o nível de *Data-flow Temporizado*. Nesse nível, inicia-se a modelagem dos atrasos para refletir as restrições de tempo da especificação, tempos de processamento e latências da implementação alvo. A partição entre *hardware* e *software* começa a ser analisada e arquiteturas são testadas para verificar o desempenho do sistema. Estruturas de dados e protocolos de comunicação ainda não são definidos nesse estágio.

Uma vez que a divisão entre *hardware* e *software* esteja definida, começa-se a refinar o modelo de *hardware*. No nível *comportamental* já existe precisão de pinos e uma definição da cadência de ciclos do sistema, mas as estruturas internas ainda não são as definitivas. As interações entre os diferentes módulos do sistema já podem ser verificadas e a velocidade das simulações ainda é maior do que no nível RTL.

O último nível da figura 2.3 é o RTL. Neste nível, o funcionamento do sistema é descrito por meio de máquinas de estados finitos e caminhos de dados. A descrição já possui precisão de pinos e do relógio (*clock*) e a estrutura do sistema reflete precisamente os registradores e lógicas combinatórias da implementação alvo. Apesar disso, esse nível ainda é independente da tecnologia de fabricação de circuitos integrados. Ferramentas para síntese a partir desse nível já estão bem consolidadas. Depois de obtido o modelo RTL, o circuito físico pode ser criado a partir de ferramentas de síntese ou de *layout* personalizado.

Como bem notado por (DIAS, 2007) e mostrado na figura 2.3, o modelo TLM (*Transaction Level Modeling*) não é definido exatamente como um nível de abstração da descrição do sistema. Na verdade, o TLM permite que o modelo de comunicação utilizado seja refinado de modo quase independente das descrições dos blocos funcionais. Esta abordagem permite um melhor reuso de blocos funcionais entre projetos de SoCs, pois torna a comunicação independente da funcionalidade.

O trabalho de Abrar e Thimmapuram (ABRAR; THIMMAPURAM, 2010) aponta que, atualmente, o refinamento suave dos modelos em projetos de SoCs está praticamente restrito às interfaces, por meio da utilização de TLM. A cada refinamento, existe pouco reuso de código para as funcionalidades do sistema, sendo que grande parte do código tem que ser reescrita para atender às características do próximo nível. Ou seja, apesar do modelamento em níveis de abstração guiar o refinamento do sistema, na prática o refinamento não preenche de uma única vez a lacuna semântica que existe entre dois

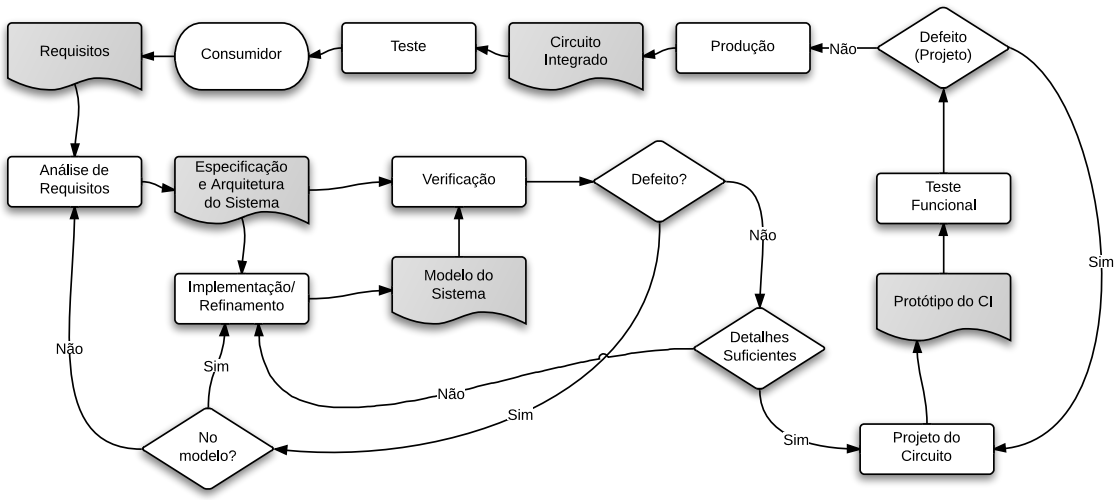


Figura 2.4: Ciclo de Projeto de SoC

níveis e, talvez por isso, nenhuma proposição de níveis de abstração tenha se tornado universalmente aceita. Características do modelo de computação, comunicação, estrutura de dados e temporização podem ser refinadas de forma individual e gradativa até que se atinja o conjunto de características requeridas pelo nível inferior.

Um ciclo de projeto de CI baseado na metodologia de refinamentos sucessivos é apresentado na figura 2.4, que é uma versão modificada da apresentada em (WILE; GOSS; ROESNER, 2005). A análise de requisitos dos clientes é o início do ciclo de projeto, onde são determinadas as características requeridas e desejáveis no produto final. Depois de analisados, os requisitos são traduzidos para a especificação e arquitetura do sistema, que guiarão o desenvolvimento durante todo o ciclo de projeto. A partir da especificação, constrói-se um modelo em alto nível de abstração que reflete toda a funcionalidade do sistema sem se preocupar, no entanto, com a implementação final em silício. Esse modelo é verificado e refinado gradativamente, até que se chegue ao nível do circuito físico. Caso não sejam detectados defeitos durante o teste, o circuito físico será produzido em larga escala e entregue para os clientes.

2.2.1 SystemC

A modelagem de sistemas por meio de linguagens de descrição de *hardware* não é novidade na indústria de microeletrônica. As linguagens de descrição de *hardware*, ou HDLs (*Hardware Description Language*), foram utilizadas, por muito tempo, como uma maneira de agilizar o processo de simulação, abstraindo-se das características físicas dos

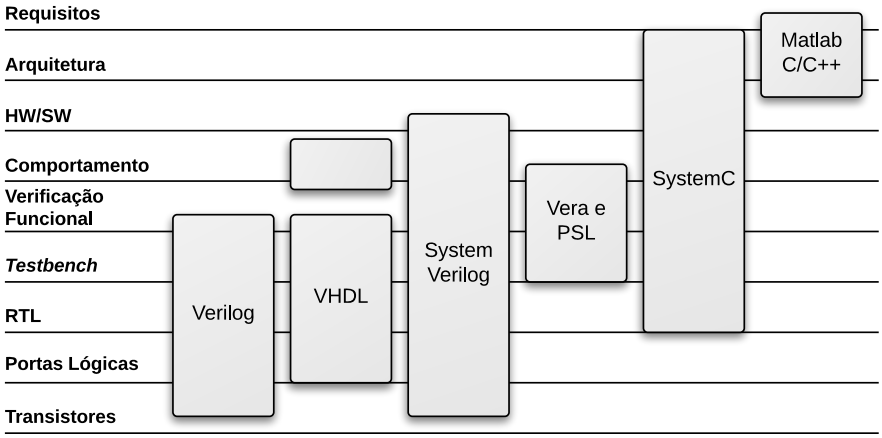


Figura 2.5: Comparação dos Domínios de Aplicação de Linguagens

semicondutores e concentrando-se no funcionamento lógico dos sistemas. A necessidade do aumento de produtividade pressionou o desenvolvimento de ferramentas de apoio ao projeto que permitissem a síntese de *hardware* a partir das HDLs e uma nova indústria de microeletrônica surgiu. Hoje, diversas linguagens são utilizadas para modelar tanto o *hardware* quanto o *software* dos sistemas computacionais e cada uma delas possui seus pontos fortes e fracos.

A figura 2.5 foi traduzida de (BLACK et al., 2009) e compara diversas linguagens utilizadas na descrição de SoCs, indicando os domínios nos quais essas linguagens são geralmente empregadas. Pode ser visto que o padrão SystemC (IEEE, 2012) apresenta uma grande abrangência de domínios, permitindo que a mesma linguagem seja utilizada durante quase todo o ciclo de projeto de SoCs. A abrangência do padrão e sua flexibilidade foram os principais motivos para a escolha do SystemC como linguagem base para o desenvolvimento deste trabalho.

A arquitetura interna do SystemC é apresentada na figura 2.6, que foi traduzida e adaptada de (BLACK et al., 2009). Nela, pode-se verificar que o SystemC, formalmente, não é uma linguagem de programação, mas uma extensão do padrão C++ direcionada para a descrição e simulação de sistemas digitais. Um núcleo de simulação baseado em eventos foi adicionado de forma a permitir a execução de processos concorrentes, necessários para simular o paralelismo de um circuito digital.

Aliado ao núcleo de simulação, está o núcleo da linguagem, que define o comportamento básico para os *Modules*[3], *Ports*, *Interfaces* e *Channels*. *Modules* são os blo-

[3]Para não ocorrerem ambiguidades, optou-se por não traduzir os nomes dos componentes do SystemC. (Nota do Autor).

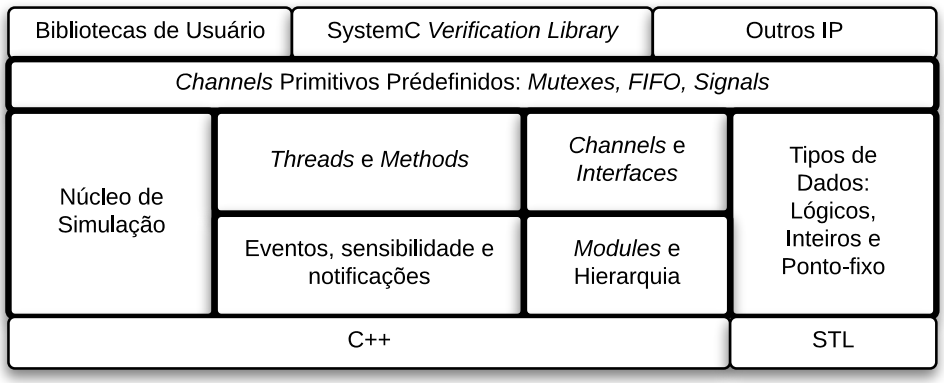


Figura 2.6: Arquitetura do SystemC

cos básicos do SystemC, que permitem dividir um sistema complexo em unidades mais simples. Cada *Module* pode possuir mais de um processo, indicando comportamentos paralelos internos e, inclusive, pode possuir outros *Modules*, formando uma hierarquia. Os *Modules* se comunicam com o ambiente externo por meio das *Ports*. As *Ports* são as instâncias, ou objetos reais, de *Channels*. Por sua vez, os *Channels* são as definições, ou implementações, das *Interfaces*. De uma forma simples, as *Interfaces* definem quais operações, como *read()* ou *write()*, podem ser efetuadas sobre as *Ports* e os *Channels* definem como essas operações devem ser efetuadas.

O SystemC define também tipos básicos de dados em adição aos tipos do padrão C++, incluindo bits, tipos de dados com quatro valores (0, 1, X, Z) para simular o comportamento de fios, inteiros de precisão arbitrária e números com representação em ponto fixo. Por utilizar o C++, tipos de dados definidos pelo usuário também pode ser criados. Alguns canais básicos são definidos pelo padrão SystemC. Esses canais são utilizados para implementar a comunicação entre módulos. É possível para o projetista definir outros canais mais complexos e bibliotecas específicas, conforme necessário.

A figura 2.7 mostra um fluxo de compilação simplificado de um código SystemC. Por se tratar, basicamente, de um código C++, pode-se utilizar o compilador GCC (GNU, 2010) para gerar o modelo executável do sistema. Na figura, as etapas internas do compilador GCC foram omitidas por simplicidade. O código fonte SystemC, desenvolvido pelo projetista, é compilado e ligado às bibliotecas do C++ e do SystemC para gerar um modelo executável do sistema com núcleo de simulação embutido.

Para uma descrição mais completa das funcionalidades do SystemC, pode-se consultar (GRÖTKER et al., 2002), (BLACK et al., 2009) e (IEEE, 2012).

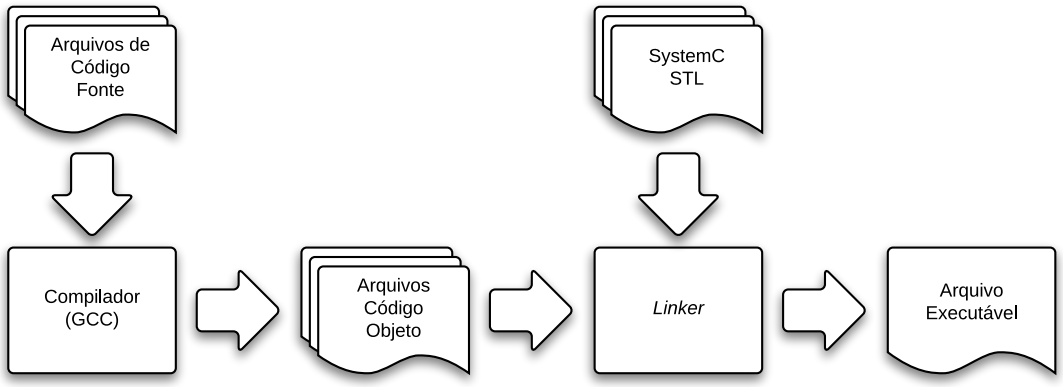


Figura 2.7: Fluxo de Compilação Simplificado do Código SystemC

2.3 Verificação Baseada em Asserções

A verificação é o processo que busca garantir que o dispositivo implementado atende à intenção do produto, antes que este seja efetivamente fabricado (CARTER; HEMMADY, 2007). Existem dois tipos básicos de verificação presentes no projeto de CIs: a verificação baseada em simulação e a verificação formal. Quando se considera que o processo de verificação será guiado por asserções, a verificação formal é realizada por meio da verificação de modelos, ou *model-checking*, enquanto a verificação baseada em simulação é realizada por meio da verificação dinâmica de propriedades, ou *dynamic property checking*.

Nesta seção, trabalhos recentes sobre verificação baseada em asserções são apresentados. Os trabalhos foram divididos em três diferentes categorias, de acordo com a abordagem adotada. A abordagem de verificação de modelos é tratada na subseção 2.3.1. Trabalhos recentes sobre verificação dinâmica de propriedades são descritos na subseção 2.3.2. Verificar as propriedades é somente um dos problemas relacionados à DPC, portanto, em 2.3.3, trabalhos recentes sobre a geração de vetores de teste para verificação dinâmica de propriedades são apresentados.

2.3.1 Verificação de Modelos

A verificação de modelos foi desenvolvida por Clarke e Emerson (CLARKE; EMERSON, 1982) e, independentemente, por Queille e Sifakis (QUEILLE; SIFAKIS, 1982). Basicamente, os algoritmos de verificação de modelos examinam estados alcançáveis do modelo do sistema, buscando por um estado que viola uma asserção de

uma propriedade formal. No entanto, em grande parte dos sistemas reais, o número de estados a serem verificados excede os limites práticos dos ambientes computacionais atuais no que se refere tanto à memória quanto ao tempo de execução. Este problema da explosão de estados dificulta a aplicação de VM no nível de sistema (FUJITA; GHOSH; PRASAD, 2008).

A verificação de modelos com profundidade limitada (BMC[4]) foi proposta por Biere et al. (BIERE et al., 1999) como uma técnica complementar à VM. BMC reduz consideravelmente o impacto da explosão de estados limitando o número de estados a serem explorados. Apenas os estados alcançáveis que estão dentro de um número limite de passos são verificados. Além disso, em vez de gerar estados e verificá-los contra a asserção da propriedade, o verificador BMC desdobra o sistema por um número determinado de passos e combina-o com o complemento da asserção a ser verificada, de forma a se obter uma fórmula proposicional. Esta fórmula é apresentada a um resolvedor de SAT (*Boolean Satisfiability*), que verifica se a fórmula é satisfazível. Se não o for, o sistema possui a propriedade, caso contrário, os assinalamentos que satisfazem a fórmula determinam um contraexemplo da asserção. Avanços recentes em resolvedores de SAT fizeram de BMC uma das técnicas formais mais comuns de verificação na indústria de microeletrônica. No entanto, mesmo com as evoluções nestas ferramentas, a verificação de modelos de sistemas simples ainda pode ser um desafio para as equipes de verificação (METTA, 2011).

A ferramenta CBMC (CLARKE; KROENING; LERDA, 2004) foi uma das primeiras ferramentas de BMC disponíveis. Em princípio, esta ferramenta foi desenvolvida para verificar programas na linguagem ANSI-C mas, atualmente, também suporta as linguagens C++, SMV, Verilog e netlists (Carnegie Mellon University, 2012). Existem planos para que a ferramenta ofereça suporte também para as linguagens SpecC e SystemC no futuro. O CBMC já foi utilizado para verificar tanto projetos de *hardware* (CLARKE; KROENING, 2003) quanto de *software* (METTA, 2011).

O trabalho apresentado em (GROSSE; LE; DRECHSLER, 2010) usa o CBMC como verificador base e permite verificar propriedades transacionais e de nível de sistema em descrições SystemC não temporizadas. O modelo SystemC original é traduzido automaticamente para um programa C, que é acrescido de uma versão abstrata e estática

[4]Diferentemente da sigla para *model checking*, optou-se por não traduzir a sigla para *bounded model checking*, uma vez que esta técnica é normalmente referida na literatura por sua sigla em inglês e sua tradução poderia dificultar o entendimento. (Nota do Autor).

do escalonador do SystemC. A lógica para a verificação da asserção também é adicionada ao código resultante antes do problema ser entregue ao verificador CBMC. O trabalho apresenta uma abordagem para a verificação baseada em indução de propriedades transacionais que aumenta o desempenho em relação ao tempo de execução da técnica baseada em BMC. Além disso, usando indução, a verificação de propriedades utilizando BMC se torna completa (i.e. as propriedades são verificadas sobre um número não limitado de passos). Apesar dos resultados alcançados pelo trabalho, a abordagem é muito invasiva, requerendo modificações severas no código fonte do sistema. Além disso, o trabalho utiliza um compilador de SystemC para C customizado e depende fortemente de uma versão abstraída do escalonador SystemC, o que compromete a generalidade da abordagem.

Um trabalho independente, mas similar, é apresentado em (CIMATTI et al., 2011a), onde duas abordagens para a verificação de propriedades para projetos SystemC são apresentadas. Na primeira abordagem, o modelo SystemC é traduzido para um programa ANSI-C, que é acrescido de uma versão customizada do escalonador SystemC. O código resultante é verificado utilizando um verificador de modelos para *software* desenvolvido pelos próprios autores do trabalho. A segunda abordagem consiste em traduzir cada processo SystemC em um programa ANSI-C sequencial e explorar cada programa de forma independente por meio de *lazy abstraction* (HENZINGER et al., 2002), enquanto a verificação do sistema como um todo é controlada pela execução da versão customizada do escalonador SystemC. Os resultados apresentados em (CIMATTI et al., 2011b) mostram que a segunda abordagem apresenta um melhor desempenho em relação ao tempo de execução quando comparada à primeira abordagem, uma vez que o escalonador SystemC não é incluído no problema de BMC. No entanto, este trabalho sofre dos mesmos problemas já atribuídos a (GROSSE; LE; DRECHSLER, 2010).

O método proposto por Ugarte e Sanchez (UGARTE; SANCHEZ, 2006) verifica asserções em descrições comportamentais utilizando uma técnica baseada em resolvedores não lineares. Depois de uma fase de análise estática do código fonte do DUV, o modelo é traduzido para um conjunto de inequações polinomiais inteiras que é alimentado em um resolvedor não linear comercial para verificar as asserções. Este método pode conseguir resultados melhores do que abordagens de BMC baseadas em BDD (*Binary Decision Diagram*) ou resolvedores de SAT quando a descrição do sistema envolve operadores aritméticos.

Um trabalho mais recente (CHOU et al., 2012) apresenta uma abordagem que utiliza VM simbólico em vez de BMC para a verificação de descrições SystemC. O problema da explosão de estados é prevenido por meio da utilização de verificação baseada em indução. Os resultados apresentados no trabalho mostram que este método possui um desempenho melhor do que o método apresentado em (CIMATTI et al., 2011a) no que se refere ao tempo de execução e à memória utilizada, para quase todos os *benchmarks* utilizados. Diferentemente de (GROSSE; LE; DRECHSLER, 2010) e (CIMATTI et al., 2011a), este método permite a verificação de descrições SystemC temporizadas. No entanto, os problemas relacionados à análise estática do código SystemC e à versão customizada do escalonador SystemC persistem.

A característica mais importante das ferramentas baseadas em verificação de modelos é a garantia da qualidade. O método verifica as propriedades em todos os estados alcançáveis e, por este motivo, é uma abordagem completa. A BMC, como originalmente definida, realiza uma exploração com profundidade limitada do espaço de estados, garantindo que o sistema está correto até este limite. No entanto, a verificação baseada em indução descrita em (GROSSE; LE; DRECHSLER, 2010) remove esta limitação, aumentando o desempenho no tempo da BMC, ao mesmo tempo que garante completude à abordagem. No entanto, o uso de métodos baseados em VM e BMC implicam em algumas desvantagens importantes:

- **Escalabilidade**: Apesar dos avanços recentes, escalabilidade é um problema importante no uso da verificação formal (METTA, 2011). A explosão do espaço de estados dificulta a aplicação de verificação formal no nível de sistema (FUJITA; GHOSH; PRASAD, 2008), restringindo esta técnica à verificação de blocos (GROSSE; DRECHSLER, 2010);
- **Intuitividade**: O uso de métodos formais não é tão intuitivo quanto a verificação baseada em simulação. Além disso, pode requerer intenso esforço manual (METTA, 2011);
- **Restrição ao Estilo de Código**: É necessário assumir que o código do DUV possui determinadas características. De outra maneira, citando Chou et al. "... verificar descrições SystemC é intratável na teoria."(CHOU et al., 2012, Tradução Nossa). Estas características criam restrições ao estilo de código permitido no desenvolvimento do sistema. Ferramentas de verificação

formal atuais apresentam limitações no tratamento de descrições que utilizam alocação dinâmica de estruturas de dados e concorrência baseada em variáveis compartilhadas, ou que utilizam propriedades do ambiente no qual o código é executado, como bibliotecas e outros programas (D'SILVA; KROENING; WEISSENBACHER, 2008);

- **Disponibilidade do Código Fonte**: As ferramentas de verificação formal precisam criar (ou ao menos ter acesso a) uma descrição formal do DUV. Em outras palavras, ou a ferramenta tem acesso ao código fonte completo do DUV ou o DUV deve estar descrito formalmente. Isto significa que ferramentas formais não podem ser utilizadas para verificar código com referência a bibliotecas de terceiros ou a blocos IP de terceiros sem código fonte disponível. Isto limita a aplicação de métodos formais em ambientes complexos de co-verificação *hardware/software*.

Enquanto que as desvantagens **Escalabilidade**, **Intuitividade** e **Restrição ao Estilo de Código** podem ser eliminadas por meio de intensa pesquisa na área, a desvantagem **Disponibilidade do Código Fonte** continuará a ser um problema para as abordagens formais no futuro.

2.3.2 Verificação Dinâmica de Propriedades

Apesar do crescente aumento da popularidade dos métodos formais, a verificação de sistemas descritos em alto nível de abstração ainda é fortemente baseada em simulação (FUJITA; GHOSH; PRASAD, 2008). As asserções têm sido utilizadas em ambientes de simulação já há algum tempo, mas os avanços recentes nas ferramentas de verificação e a padronização de linguagens de descrição de propriedades fizeram com que a importância da verificação dinâmica de propriedades aumentasse no contexto industrial.

A verificação dinâmica de propriedades pode ser dividida em dois passos básicos. Primeiramente, um conjunto de vetores de teste deve ser gerado para exercitar o DUV durante a simulação. Este problema é tratado na subseção 2.3.3. O segundo passo é verificar as asserções durante a simulação do modelo. Normalmente, isto é realizado por meio da criação de monitores para as propriedades e integrando-os ao modelo executável do sistema. Hoje, as linguagens de descrição de propriedades são utilizadas para

descrever o comportamento correto do sistema. Muitas ferramentas acadêmicas foram desenvolvidas para traduzir automaticamente as propriedades descritas nestas linguagens para monitores de propriedades.

O trabalho de Habibi et al. (HABIBI; GAWANMEH; TAHAR, 2004) traduz asserções escritas em PSL (IEEE, 2010) para máquinas de estados abstratas. As máquinas são, então, compiladas para a linguagem C# usando a ferramenta AsmL (Microsoft Coorporation, 2011). Uma vez que C++ é uma das linguagens suportadas pelo *framework* .NET da Microsoft, o monitor C# resultante pode ser integrado ao modelo SystemC para testar as asserções durante a simulação. Esta abordagem é desnecessariamente complicada, pois os mesmos autores apresentam uma forma de traduzir as asserções PSL diretamente para SystemC em (HABIBI; TAHAR, 2004). Além disso, todos os exemplos apresentados pelos autores possuem uma sinal de sincronização (*clock*). Portanto é razoável supor que o método não é aplicável em níveis de abstração mais altos.

Um método para geração de monitores de propriedades descritas em PSL baseado em autômatos é apresentado em (BOULÉ; ZILIC, 2008). O objetivo do trabalho é gerar monitores de asserções para serem utilizados em emulação, aceleração de simulação e depuração em silício. A propriedade PSL é inteiramente traduzida para um único autômato, permitindo que otimizações sejam realizadas no monitor resultante. Os autores apresentam técnicas para gerar monitores para um conjunto completo de operadores PSL. Uma vez que este trabalho é direcionado para a geração de monitores de *hardware*, ele só pode ser empregado no nível RTL.

Pierre e Ferro (FERRO; PIERRE, 2009; PIERRE; FERRO, 2008) abordam o problema de utilizar ABV no nível TLM (*Transaction Level Modelling*). Uma vez que não existe um sinal global de sincronização neste tipo de modelo, os autores adotam um conceito de sincronização local, no qual uma asserção é avaliada apenas quando um evento transacional envolvendo uma de suas variáveis ocorre. Quando uma asserção é avaliada, o seu ciclo corrente avança, permitindo a verificação de propriedades temporais. Esta abordagem permite verificar asserções que expressam propriedades relacionadas a eventos transacionais (comunicação) e requer poucas modificações no código fonte do DUV. Uma desvantagem de usar sincronização local é que as asserções devem ser revisadas para lidar com sincronização global à medida em que o modelo é refinado para níveis mais baixos de abstração. No entanto, esta desvantagem não tira o mérito do trabalho de Pierre e Ferro,

uma vez que ainda não foi desenvolvido outro modo de lidar com sincronização no nível TLM.

Várias ferramentas comerciais também oferecem suporte para a verificação dinâmica de propriedades, em especial usando PSL. Tanto o *Incisive Unified Simulator* da Cadence (Cadence Design Systems, 2012), quanto o ModelSim da Mentor Graphics (Mentor Graphics, 2012) oferecem suporte para DPC usando PSL, entre outras linguagens.

2.3.3 Geração de Vetores de Teste para Verificação Dinâmica de Propriedades

Provar que um sistema está correto utilizando simulação requer que todas as possíveis sequências dos símbolos de seu alfabeto de entrada sejam aplicadas durante a validação, uma tarefa impraticável, exceto para os modelos mais triviais. Por este motivo, apenas um subconjunto das combinações possíveis de valores de entrada do sistema é utilizado durante a validação. Selecionar um subconjunto de vetores de teste que efetivamente revela faltas de nível de projeto é um grande desafio. Tais faltas são difíceis de serem modeladas (TASIRAN; KEUTZER, 2001) pois os defeitos de projeto são menos localizados do que os defeitos de fabricação. Sem modelos formais para faltas de nível de projeto, a tarefa de medir a confiabilidade do sistema se torna difícil.

Apesar da falta de modelos formais para os faltas de nível de projeto, a intuição e estudos empíricos indicam uma relação entre cobertura de teste e confiabilidade (MALAIYA et al., 1994). Uma abordagem comumente utilizada na verificação baseada em simulação é exercitar o DUV com um conjunto de estímulos gerados aleatoriamente e medir a cobertura alcançada. Quando a cobertura atinge um dado valor objetivo, supostamente o DUV teria atingido o grau de confiabilidade desejado. O problema com esta abordagem é que estímulos aleatórios são eficientes nos primeiros estágios da validação, mas sua eficiência decresce à medida em que o processo de validação avança. Métodos de geração de estímulos de entrada dirigidos por cobertura, por outro lado, podem aumentar consideravelmente a eficiência da validação, guiando a geração de vetores de teste.

O sucesso da validação dirigida por cobertura está fortemente ligado à métrica de cobertura utilizada. Usualmente, as métricas de cobertura são divididas em estruturais e funcionais. Abordagens estruturais são baseadas em observação do tipo caixa-cinza ou caixa-branca, e a estrutura de controle interna do DUV é utilizada para guiar a geração dos

testes. Vários trabalhos utilizam testes estruturais, tanto para SoCs quanto para *Software*, como (LI; WEISS; YEE, 2006), (ANDREWS; O'FALLON; CHEN, 2004), (KATAYAMA; FURUKAWA; USHIJIMA, 1996), (VEMURI; KALYANARAMAN, 1995) e (NTAFOS, 1988). Apesar da efetividade do teste estrutural ter sido comprovada (MALAIYA et al., 1994), estas técnicas não são capazes de exercitar alguns tipos de faltas, como as faltas de omissão, quando alguma funcionalidade requerida do sistema não é implementada pela equipe de projeto. Infelizmente, este tipo de defeito é muito comum (GLASS, 1981) e usar apenas cobertura estrutural pode fazer com que eles não sejam descobertos.

As métricas de cobertura funcional, por outro lado, são modeladas sobre os atributos funcionais do sistema e, por este motivo, são mais próximas dos objetivos da verificação do que as métricas estruturais (CASTRO MÁRQUEZ et al., 2011). Métricas de cobertura funcional sempre foram consideradas difíceis de serem construídas, pois devem capturar o intuito do sistema, sendo, por isto, intimamente relacionadas ao projeto. Os avanços recentes em ABV diminuíram o fardo das equipes de verificação uma vez que as asserções podem ser utilizadas como intermediárias para determinar a cobertura funcional atingida durante a validação.

Habibi e Tahar (HABIBI; TAHAR, 2004) propuseram um sistema para aumentar a cobertura de asserções em modelos descritos em SystemC. Depois de uma fase de análise estática, uma versão abstrata do projeto é criada usando hipergrafos. Relações de dependência entre os sinais de entrada e as asserções sendo verificadas são extraídas dos hipergrafos resultantes para criar um grafo de dependência, que é utilizado para construir: 1) uma versão simplificada do sistema contendo apenas as unidades envolvidas nas asserções; e 2) uma estrutura de inicialização que serve de ponto inicial para o algoritmo de busca. Algoritmos genéticos são utilizados para aprimorar iterativamente a população de vetores de testes, utilizando cobertura de asserção como a função de aptidão. Os resultados apresentados no trabalho mostram que este método aumenta consideravelmente a cobertura de asserções quando comparado à geração puramente aleatória.

Em (BANERJEE et al., 2006), um método inteligente para a geração de vetores de teste a partir de especificações formais é apresentado. O método proposto gera somente vetores de teste não vazios e aplica o conceito da realizabilidade durante a geração, o que aumenta os valores de cobertura de asserções quando comparado aos testes aleatórios.

Alguns trabalhos ignoram completamente a descrição do DUV, utilizando apenas as asserções para gerar os vetores de teste. MYGEN (ODDOS et al., 2009), por exemplo, gera descrições HDL para geradores de vetores de teste a partir da descrição PSL das propriedades. O método adotado baseia-se na dualidade entre geradores e monitores. Enquanto os monitores reconhecem o complemento da linguagem definida pela propriedade, o gerador produz a linguagem gerada pela propriedade. A ferramenta MBAC (BOULÉ; ZILIC, 2008) é usada para gerar o automato que verifica a propriedade. Este autômato é, então, usado para criar a descrição HDL do gerador de testes. Apesar da fiabilidade da abordagem, os autores não apresentam resultados comparativos para se poder determinar o desempenho do método com maior exatidão.

Tong et al. (TONG; BOULÉ; ZILIC, 2010) aprimoram o trabalho apresentado em (ODDOS et al., 2009), abordando o conceito de cobertura. Os autores traduzem um conjunto de métricas de cobertura para autômatos finitos não determinísticos (AFN) que representam as asserções e o relaciona com objetivos específicos de cobertura. O trabalho assume que propriedades suficientes foram definidas para especificar de forma completa a funcionalidade do circuito. Baseado nesta forte suposição e usando a ferramenta MBAC, autômatos aceitadores e rejeitadores são usados para criar sequências de testes para tentar exercitar tanto o comportamento correto quanto incorreto do modelo e, desta forma, potencialmente aumentar a cobertura do conjunto de testes. A ferramenta proposta, Airwolf-TG, é capaz de gerar vetores de teste que alcançam cobertura total para um conjunto complexo de métricas de cobertura. O método, no entanto, é comparado somente com os resultados da ferramenta MYGEN, o que não permite que o desempenho geral do método seja avaliado.

2.4 Resumo do Capítulo

Neste capítulo, foi apresentada uma revisão dos principais trabalhos relacionados ao tema desta monografia. Foram apresentadas as limitações das técnicas de verificação de modelos e BMC, assim como as principais abordagens para a DPC de sistemas descritos em alto nível de abstração.

Foi verificada uma lacuna na geração de vetores de teste para a DPC destes sistemas, uma vez que os trabalhos analisados tratam a geração de estímulos sob uma

perspectiva simplista, considerando que os sinais envolvidos nas propriedades são do tipo *bit*, o que é uma simplificação exagerada dos sistemas reais. O método apresentado nos capítulos 4 e 5 foi desenvolvido especialmente para lidar com esta lacuna e com as outras limitações das técnicas de ABV atuais.

Capítulo 3

Base Teórica e Conceitual

Neste capítulo, são apresentados os conceitos e definições necessárias para o bom entendimento do presente trabalho. Algumas palavras merecem atenção especial, pois o seu significado varia na literatura, dependendo do contexto da sua aplicação. O termo *sinal*, por exemplo, neste trabalho é utilizado para referenciar uma conexão do sistema, no mesmo sentido em que a palavra *signal* é utilizada na linguagem SystemC, e *wire* é utilizada na linguagem Verilog. Considera-se que um *sistema* é uma entidade que opera sobre os valores de um ou mais *sinais de entrada*, gerando uma resposta cujo valor é atribuído a um ou mais *sinais de saída*. Outros conceitos necessitam de uma definição um pouco mais precisa e, por isto, são detalhados a seguir. As notações para as definições relativas às linguagens formais, máquinas de estados finitos e aos conectivos lógicos da lógica proposicional foram baseadas nas apresentadas por Vieira (2006). As notações para Lógica Temporal Linear e estruturas de Kripke foram baseadas nas apresentadas em (KROPF, 1999) e (IEEE, 2010). As contribuições originais deste trabalho se iniciam nas seções 3.2 e 3.3, onde são apresentadas as extensões para a lógica proposicional e para a lógica temporal linear.

3.1 Conceitos Preliminares

Esta seção apresenta conceitos e definições colhidos da literatura que formarão a base teórica para o desenvolvimento deste trabalho. São apresentados os conceitos de linguagens formais, máquinas de estados finitos, lógica proposicional, lógica temporal proposicional, propriedade, asserção e sistemas com memória e sem memória. Ao fim desta seção, também é descrita a notação utilizada nos algoritmos apresentados ao longo do texto.

3.1.1 Linguagens Formais

As linguagens formais são ferramentas úteis em várias áreas do conhecimento, como a engenharia, a física e, principalmente, a computação. Segundo Vieira (2006), são dois os fatores principais que diferem uma linguagem formal de uma linguagem natural:

a) uma linguagem formal tem uma sintaxe bem definida, de forma que, dada uma sentença, sempre seja possível saber se ela pertence ou não à linguagem;

b) uma linguagem formal tem uma semântica precisa, de modo que não contenha sentenças sem significado ou ambíguas.

Antes de se definir o que vem a ser uma linguagem formal, as definições 3.1 e 3.2 estabelecem, respectivamente, os conceitos de alfabeto e palavra.

Definição 3.1 (Alfabeto). *Um alfabeto é um conjunto finito não vazio de símbolos.* □

Definição 3.2 (Palavra). *Uma palavra sobre um alfabeto Σ é uma sequência finita de zero ou mais símbolos de Σ.* □

Uma palavra w possui um tamanho, $|w|$, que corresponde ao número de símbolos que a compõem. Em especial, pode-se definir uma palavra vazia, λ, constituída de zero símbolos do alfabeto. Por não conter símbolos, $|\lambda| = 0$. O *i-ésimo* símbolo de uma palavra $w = a_0a_1...a_i...a_n$ é definido como $w^i = a_i$.

Seja uma palavra $w = xyz$, onde x, y e z podem ou não ser λ. A palavra x é um prefixo de w, y é uma subpalavra de w e z é um sufixo de w. Em especial, λ é um prefixo, sufixo e uma subpalavra de qualquer palavra w. As definições 3.3, 3.4 e 3.5 estabelecem estes conceitos formalmente.

Definição 3.3 (Prefixo). *Seja uma palavra $w = a_0a_1...a_{i-1}a_ia_{i+1}...a_n$. Um prefixo de w é definido como $w^{..i} = a_0a_1...a_{i-1}$.* □

Definição 3.4 (Subpalavra). *Seja uma palavra $w = a_0a_1...a_i...a_{j-1}a_j...a_n$. Uma subpalavra de w é definida como $w^{i..j} = a_i...a_{j-1}$.* □

Definição 3.5 (Sufixo). *Seja uma palavra $w = a_0a_1...a_{i-1}a_ia_{i+1}...a_n$. Um sufixo de w é definido como $w^{i..} = a_ia_{i+1}...a_n$.* □

O conceito de linguagem formal é estabelecido a partir dos conceitos de alfabeto e palavra.

Definição 3.6 (Linguagem Formal). *Uma linguagem formal sobre um alfabeto Σ é um conjunto de palavras sobre Σ.* □

Por serem conjuntos, as operações binárias de união ($\cup$), intercessão ($\cap$) e diferença ($-$) de conjuntos podem ser aplicadas normalmente aos alfabetos e às linguagens. No entanto, duas operações adicionais podem ser úteis para a especificação de algumas linguagens. A primeira delas é a concatenação, cujo conceito é estabelecido na definição 3.7, para as palavras, e na definição 3.8, para as linguagens.

Definição 3.7 (Concatenação de Palavras). *O resultado da concatenação entre duas palavras $w_1 = a_1a_2...a_n$ e $w_2 = b_1b_2...b_m$, ou w_1w_2, é a palavra $a_1a_2...a_nb_1b_2...b_m$. Em especial, $\lambda w = w\lambda = w$ para qualquer palavra w.* □

Definição 3.8 (Concatenação de Linguagens). *Sejam L_1 e L_2 duas linguagens sobre os alfabetos Σ_1 e Σ_2, respectivamente. A concatenação de L_1 com L_2 é a linguagem L_3 sobre o alfabeto $\Sigma_1 \cup \Sigma_2$, dada por:*

$$L_3 = \{w_1w_2 | w_1 \in L_1 \text{ e } w_2 \in L_2\}$$

Em especial, $L\{\lambda\} = \{\lambda\}L = L$ e $L\emptyset = \emptyset L = \emptyset$ para qualquer linguagem L. □

A segunda operação útil para a especificação de algumas linguagens é o fecho de Kleene. O fecho de Kleene é uma operação unária sobre conjuntos, normalmente aplicada sobre um alfabeto ou sobre uma linguagem. A definição 3.9 define o fecho de Kleene recursivamente.

Definição 3.9 (Fecho de Kleene). *Seja o conjunto Υ um alfabeto ou uma linguagem. A operação fecho de Kleene sobre Υ, ou Υ^*, é definida recursivamente como:*

a) $\lambda \in \Upsilon^*$;

b) se $x \in \Upsilon^*$ *e* $y \in \Upsilon$, *então* $xy \in \Upsilon^*$. □

Informalmente, o fecho de Kleene sobre o conjunto Υ é o conjunto formado a partir da concatenação de zero ou mais elementos do próprio conjunto Υ.

As linguagens formais serão utilizadas principalmente para representar os estímulos e respostas do sistema sob verificação. Antes de relacionar o conceito de linguagens formais ao conceito de sistemas, serão apresentadas as máquinas de estados finitos, empregadas para modelar os sistemas estudados.

3.1.2 Máquinas de Estados Finitos

As Máquinas de estados finitos são abstrações matemáticas utilizadas para modelar uma grande quantidade de sistemas, entre eles os sistemas digitais.

> As máquinas de estados finitos são máquinas abstratas que capturam as partes essenciais de algumas máquinas concretas. Essas últimas vão desde máquinas de vender jornais e de vender refrigerantes, passando por relógios digitais e elevadores, até programas de computador, como alguns procedimentos de editores de textos e de compiladores. O próprio computador digital, se considerarmos que sua memória é limitada, pode ser modelado por meio de uma máquina de estados finitos (VIEIRA, 2006, p. 59).

Como apontado por Vieira, um computador digital, por ter memória finita e um número finito de sinais de entrada, pode ser modelado por meio de uma máquina de estados finitos. De modo mais geral, qualquer sistema digital, quando se considera suas limitações físicas, pode ser modelado por meio de uma máquina de estados finitos.

Proposição 3.1. *O comportamento de qualquer sistema simulado por um computador digital pode ser modelado por meio de uma máquina de estados finitos.*

Demonstração. Se o sistema é simulado por um computador digital e o funcionamento de um computador digital pode ser modelado por uma máquina de estados finitos então o comportamento do sistema pode ser modelado por uma máquina de estados finitos. □

O tema deste trabalho é a verificação dinâmica (por simulação) de sistemas descritos em alto nível de abstração. Dentro desta premissa, a proposição 3.1 estabelece que o DUV pode ser modelado por uma máquina de estados finitos independentemente de seu nível de abstração original. Note que a síntese da máquina de estados finitos que modela o DUV não é uma tarefa trivial para os níveis de abstração mais altos. No entanto, para os fins deste trabalho, a síntese da máquina de estados finitos não é necessária. A proposição 3.1 será utilizada somente como suporte para as definições e manipulações matemáticas apresentadas neste trabalho.

Para se continuar esta discussão, são introduzidas as definições de máquina de Moore e máquina de Mealy, que serão utilizadas para modelar os DUVs do ponto de vista matemático.

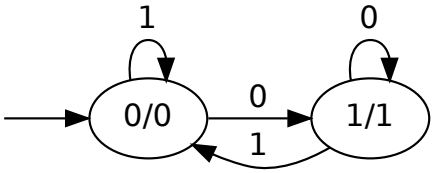


Figura 3.1: Máquina de Moore para Computar o Complemento da Palavra de Entrada

Definição 3.10 (Máquina de Moore). *Uma máquina de Moore é uma sêxtupla* $(E, \Sigma, \Delta, \delta, \sigma, i)$*, em que:*

- E *é um conjunto finito não vazio de elementos denominados estados;*
- Σ *é o alfabeto de entrada;*
- Δ *é o alfabeto de saída;*
- $\delta : E \times \Sigma \mapsto E$ *é a função de transição, que é uma função total;*
- $\sigma : E \mapsto \Delta$ *é a função de saída, que é uma função total;*
- $i \in E$ *é o estado inicial.* □

Informalmente, uma máquina de Moore é uma máquina de estados finitos com um símbolo de saída associado a cada estado. Sempre que um novo estado é atingido, o símbolo de saída associado àquele estado é concatenado à direita da palavra de saída. A figura 3.1 mostra um diagrama de estados para uma máquina de Moore cuja palavra de saída computada corresponde ao complemento da palavra de entrada aplicada à máquina. Nesta figura, cada estado é representado por uma elipse. Dentro da elipse que representa o estado, estão apresentados o nome do estado, antes da barra (/), e o símbolo de saída associado ao estado, depois da barra. As transições entre os estados são representadas por setas, indicando a direção da transição. Próximo a cada seta que representa uma transição está representado o símbolo do alfabeto de entrada sob o qual aquela transição deve ocorrer, dado o estado atual da máquina. O estado inicial é representado por uma seta que aponta para este estado.

A palavra de saída de uma máquina de Moore é computada por meio de sua função de saída estendida, $\hat{\sigma}$.

Definição 3.11 (Função de Saída Estendida da Máquina de Moore). *A função de saída estendida da máquina de Moore* $M = (E, \Sigma, \Delta, \delta, \sigma, i)$ *é a função* $\hat{\sigma} : E \times \Sigma^* \mapsto \Delta^*$*, definida recursivamente como:*

a) $\hat{\sigma}(e, \lambda) = \sigma(e)$*;*

b) $\hat{\sigma}(e, ay) = \sigma(e)\hat{\sigma}(\delta(e, a), y)$*, para todo* $a \in \Sigma$ *e* $y \in \Sigma^*$*.* □

Analogamente, pode-se definir o conceito de máquina de Mealy, que é uma máquina de estados finitos com um símbolo de saída associado à cada transição entre estados. Sempre que uma transição é executada, o símbolo de saída associado àquela transição é concatenado à direita da palavra de saída. A definição 3.12 estabelece formalmente o conceito de máquina de Mealy.

Definição 3.12 (Máquina de Mealy). *Uma máquina de Mealy é uma sêxtupla* $(E, \Sigma, \Delta, \delta, \sigma, i)$*, em que:*

- E *é um conjunto finito não vazio de elementos denominados estados;*
- Σ *é o alfabeto de entrada;*
- Δ *é o alfabeto de saída;*
- $\delta : E \times \Sigma \mapsto E$ *é a função de transição, que é uma função total;*
- $\sigma : E \times \Sigma \mapsto \Delta$ *é a função de saída, que é uma função total;*
- $i \in E$ *é o estado inicial.* □

A figura 3.2 mostra um diagrama de estados para uma máquina de Mealy cuja palavra de saída é o complemento da palavra de entrada. Nesta figura, elipses representam estados e nome do estado está representado dentro da elipse correspondente. Setas indicam as transições, e a condição para as transições é apresentada antes da barra (/), enquanto o símbolo de saída associado à transição é apresentado depois da barra. Esta máquina possui duas transições que estão representadas pela mesma seta, ambas partindo e retornando ao estado 0. O estado inicial é representado por uma seta que aponta para este estado.

Neste momento, será feito um abuso de notação para a definição da função de saída estendida da máquina de Mealy, utilizando o mesmo símbolo utilizado para a

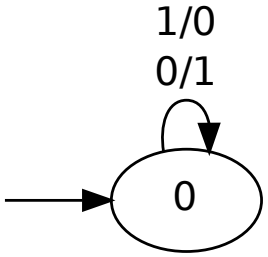


Figura 3.2: Máquina de Mealy para Computar o Complemento da Palavra de Entrada

máquina de Moore, $\hat{\sigma}$. O motivo para o abuso de notação ficará claro logo em seguida, na definição 3.14. A saída de uma máquina de Mealy é computada por meio de sua função de saída estendida, $\hat{\sigma}$.

Definição 3.13 (Função de Saída Estendida da Máquina de Mealy). *A função de saída estendida da máquina de Mealy* $M = (E, \Sigma, \Delta, \delta, \sigma, i)$ *é a função* $\hat{\sigma} : E \times \Sigma^* \mapsto \Delta^*$, *definida recursivamente como:*

a) $\hat{\sigma}(e, \lambda) = \lambda$;

b) $\hat{\sigma}(e, ay) = \sigma(e, a)\hat{\sigma}(\delta(e, a), y)$, *para todo* $a \in \Sigma$ *e* $y \in \Sigma^*$. □

Definição 3.14 (Saída Computada pela Máquina de Moore ou Mealy). *A saída computada por uma máquina de estados finitos (de Moore ou de Mealy)* $M = (E, \Sigma, \Delta, \delta, \sigma, i)$ *para a palavra* $w \in \Sigma^*$ *é* $\hat{\sigma}(i, w)$. □

É possível notar que a saída correspondente ao prefixo λ de uma palavra quando processada por uma máquina de Mealy é sempre λ, enquanto que, na máquina de Moore, a saída correspondente ao mesmo prefixo será o símbolo do alfabeto de saída associado ao estado inicial. No entanto, desconsiderando-se esta situação, as definições de máquina de Moore e Máquina de Mealy são equivalentes (VIEIRA, 2006). Assim, dadas uma máquina de Moore $M_1 = (E_1, \Sigma, \Delta, \delta_1, \sigma_1, i_1)$ e uma máquina de Mealy $M_2 = (E_2, \Sigma, \Delta, \delta_2, \sigma_2, i_2)$, diz-se que M_1 é equivalente a M_2 se, e somente se:

$$\hat{\sigma}_1(i_1, w) = \sigma_1(i_1)\hat{\sigma}_2(i_2, w), \forall w \in \Sigma^* \tag{3.1}$$

Para se determinar qual o estado atingido pela máquina de estados finitos após o processamento de uma palavra, w, define-se conceito de função de transição estendida, $\hat{\delta}$.

Definição 3.15 (Função de Transição Estendida). *Seja uma máquina de estados finitos $M = (E, \Sigma, \Delta, \delta, \sigma, i)$. A função de transição estendida para M, $\hat{\delta}$, é uma função de $E \times \Sigma^*$ para E, definida recursivamente como:*

a) $\hat{\delta}(e, \lambda) = e$;

b) $\hat{\delta}(e, ay) = \hat{\delta}(\delta(e, a), y)$, *para todo* $a \in \Sigma$ *e* $y \in \Sigma^*$. □

3.1.3 Lógica Proposicional

A lógica proposicional (LP) consiste de um sistema formal para compor afirmações por meio do emprego de conectivos lógicos com o intuito de representar proposições. Lógica proposicional tem sido usada em diversas áreas da ciência, em especial, em sistemas para prova automática de teoremas. No centro da lógica proposicional, estão os conectivos lógicos, que permitem combinar afirmações de modo gramaticalmente válido.

Definição 3.16 (Conectivo Lógico). *Um conectivo lógico é um símbolo ou uma palavra utilizados para modificar uma sentença, ou combinar duas sentenças, de modo gramaticalmente válido, para formar uma nova sentença cujo significado depende somente do significado das sentenças originais.* □

Os conectivos lógicos empregados em lógica proposicional são apresentados a seguir em duas notações. A primeira consiste de uma notação comumente utilizada em textos de lógica matemática e a segunda apresenta a forma na qual estes conectivos são utilizados em expressões em português.

- negação: $\neg$, não;
- conjunção: $\wedge$, e;
- disjunção: $\vee$, ou;
- condicional: $\rightarrow$, se ... então;
- bicondicional: $\leftrightarrow$, se, e somente se;

Tabela 3.1: Semântica da Negação

α	$\neg\alpha$
V	F
F	V

Tabela 3.2: Semântica da Conjunção

α	β	$\alpha \wedge \beta$
V	V	V
V	F	F
F	V	F
F	F	F

Tabela 3.3: Semântica da Disjunção

α	β	$\alpha \vee \beta$
V	V	V
V	F	V
F	V	V
F	F	F

Tabela 3.4: Semântica da Condicional

α	β	$\alpha \rightarrow \beta$
V	V	V
V	F	F
F	V	V
F	F	V

Tabela 3.5: Semântica da Bicondicional

α	β	$\alpha \leftrightarrow \beta$
V	V	V
V	F	F
F	V	F
F	F	V

As tabelas 3.1, 3.2, 3.3, 3.4 e 3.5 definem, respectivamente, a interpretação semântica dos conectivos lógicos negação, conjunção, disjunção, condicional e bicondicional. Nestas tabelas, as expressões α e β representam afirmativas quaisquer, que podem ser verdadeiras (V) ou falsas (F). A coluna da direita de cada uma das tabelas apresenta o resultado para a composição das afirmativas de acordo com o conectivo lógico empregado.

Definição 3.17 (Sintaxe das Fórmulas da Lógica Proposicional). *Seja Ψ um conjunto de variáveis, ou fórmulas atômicas. As fórmulas da lógica proposicional são definidas, recursivamente, como:*

1. *Todas as fórmulas atômicas de Ψ são fórmulas.*
2. *Se α e β são fórmulas, então também são fórmulas:*
   - *a)* $(\neg\alpha)$
   - *b)* $(\alpha \wedge \beta)$
   - *c)* $(\alpha \vee \beta)$
   - *d)* $(\alpha \rightarrow \beta)$
   - *e)* $(\alpha \leftrightarrow \beta)$

□

Pode-se derivar um valor para uma fórmula da lógica proposicional a partir da avaliação de suas variáveis. Uma avaliação das variáveis de Ψ é uma função $\mathcal{V} : \Psi \mapsto \{V, F\}$.

Definição 3.18 (Semântica da Fórmula da Lógica Proposicional). *Sejam α e β fórmulas de lógica proposicional, $\mathcal{V}$ uma função de avaliação das variáveis, e a interpretação semântica dos conectivos lógicos apresentada nas tabelas 3.1, 3.2, 3.3, 3.4 e 3.5. O valor assinalado às fórmulas da lógica proposicional é determinado usando a interpretação semântica $\mathcal{val}$, definida recursivamente como:*

a) $\mathcal{val}(\alpha) = \mathcal{V}(\alpha)$, *se* $\alpha \in \Psi$

b) $\mathcal{val}(\neg\alpha) = \neg\mathcal{val}(\alpha)$

c) $\mathcal{val}(\alpha \wedge \beta) = \mathcal{val}(\alpha) \wedge \mathcal{val}(\beta)$

d) $\mathcal{val}(\alpha \vee \beta) = \mathcal{val}(\alpha) \vee \mathcal{val}(\beta)$

e) $\mathcal{val}(\alpha \rightarrow \beta) = \mathcal{val}(\alpha) \rightarrow \mathcal{val}(\beta)$

f) $\mathcal{val}(\alpha \leftrightarrow \beta) = \mathcal{val}(\alpha) \leftrightarrow \mathcal{val}(\beta)$ □

A lógica proposicional serve como base para a definição de outros tipos de lógicas mais especializadas como, por exemplo, a lógica temporal proposicional.

3.1.3.1 Forma Normal Disjuntiva

A forma normal disjuntiva (FND), também conhecida como soma de produtos, é uma normalização para a apresentação de uma fórmula da lógica proposicional muito empregada para simplificação de projetos de circuitos lógicos e algoritmos para provas automáticas. Considera-se que uma fórmula da LP está na FND se, e somente se, ela for composta por uma disjunção (soma) de uma ou mais conjunções (produtos) de fórmulas atômicas. Cada fórmula atômica das cláusulas conjuntivas pode ser modificada pelo conectivo lógico da negação, ou seja, o conectivo lógico da negação pode preceder apenas uma variável da lógica proposicional. A fórmula da LP apresentada na equação 3.2 é um exemplo de fórmula escrita na FND, onde as variáveis são x_0, x_1 e x_2.

$$(x_0 \wedge x_1 \wedge x_2) \vee (x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge \ \neg x_1 \wedge x_2) \tag{3.2}$$

Por permitir que apenas os conectivos lógicos da conjunção, disjunção e negação sejam empregados, os demais conectivos lógicos apresentados na seção 3.1.3 devem ser reescritos utilizando estes três conectivos básicos. Para reescrever os conectivos lógicos da condicional e bicondicional utilizando apenas os conectivos permitidos na FND, utiliza-se as seguintes relações de equivalência:

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta \tag{3.3}$$

$$\alpha \leftrightarrow \beta \equiv (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta) \tag{3.4}$$

Como é amplamente conhecido, sempre é possível modificar uma forma da lógica proposicional para que ela esteja na forma normal disjuntiva. O problema de se obter a forma normal disjuntiva de mínimo custo a partir de uma fórmula da LP genérica é conhecido como *minimização em dois níveis* e já foi amplamente estudado, como apontado por Coudert (COUDERT, 1994).

3.1.4 Lógica Temporal Proposicional

A lógica proposicional, apresentada na subseção 3.1.3, não possui capacidade de expressão temporal, o que impede que seja utilizada para criar especificações de sistemas onde modelar a passagem do tempo é fundamental. O termo lógica temporal é utilizado, de forma geral, para descrever qualquer conjunto de regras desenvolvido para representar afirmações que envolvam a passagem do tempo.

> O termo lógica temporal tem sido utilizado para se referir, de forma geral, a diversas abordagens para representação de informação temporal em um determinado contexto, e também, mais especificamente, para se referir ao tipo de abordagem para a lógica modal introduzida por Arthur Prior por volta de 1960 (GALTON, 2008, tradução nossa).

Apesar de possuir aplicações em linguística, como na definição de tempos verbais em linguagens naturais, foi na inteligência artificial e na ciência da computação em geral que a lógica temporal proposicional encontrou um terreno fértil. Na verificação de sistemas, a lógica temporal proposicional é utilizada para especificar o comportamento lógico-temporal esperado do sistema em desenvolvimento, essencialmente definindo a linguagem formal sobre a qual o sistema deve operar.

O modelo escolhido para a representação do tempo afeta diretamente a semântica da lógica temporal. Segundo Kropf (KROPF, 1999), o tempo pode ser classificado de acordo com os seguintes critérios:

- *Tempo Linear* ou *Tempo Ramificado*: no tempo linear, assume-se que para cada instante de tempo existe exatamente um instante de tempo sucessor. Por outro lado, um tempo ramificado admite que um ponto no tempo seja seguido por vários sucessores, representando várias possibilidades de computação a partir de um determinado ponto;

- *Pontos no Tempo* ou *Intervalos de Tempo*: operadores temporais podem operar sobre pontos individuais no tempo ou sobre intervalos de tempo;

- *Tempo Discreto* ou *Tempo Contínuo*: usar tempo discreto para modelar a passagem do tempo implica que números inteiros podem ser utilizados para representar o tempo. Por outro lado, se o tempo contínuo for utilizado, o tempo geralmente é representado por números reais;

- *Passado* ou *Futuro*: os operadores temporais podem ser limitados a operar sobre instantes de tempos futuros ou podem cobrir também instantes passados.

Neste trabalho, será considerado um modelo de tempo *linear*, *discreto*, onde os operadores temporais operam sobre *pontos no tempo* e estão limitados a modificar instantes de tempo *futuros*. O modelo de tempo *linear* foi escolhido pois é mais indicado para métodos de verificação dinâmica de propriedades, nos quais uma única hipótese de execução (ou um traço) é considerada a cada momento. Optou-se pelo uso de um modelo de tempo *discreto* onde as avaliações são realizadas em *pontos no tempo* porque o trabalho aborda a verificação de sistemas digitais simulados por computador, onde as computações ocorrem em instantes de tempo discretos. Por fim, uma vez que os sistemas digitais geralmente possuem um estado inicial, a partir do qual as computações se iniciam, os operadores temporais precisam operar somente sobre instantes de tempo futuros, contados a partir deste estado inicial.

Normalmente, modelos de tempo discreto são formalizados por meio de estruturas baseadas em grafos de transição de estado, denominadas estruturas de Kripke (KROPF, 1999, p. 155).

Definição 3.19 (Estrutura de Kripke). *Uma estrutura de Kripke é uma quíntupla* $K = (E, E_0, R, \Psi, L)$*, em que:*

- E *é um conjunto finito não vazio de elementos denominados estados;*
- $E_0 \subset E$ *é um conjunto de estados iniciais;*
- $R \subset E \times E$ *é uma relação de transição, que é total;*
- Ψ *é um conjunto de variáveis proposicionais;*
- $L : E \mapsto \mathcal{P}(\Psi)$ *é uma função de rotulagem, que rotula cada estado com o subconjunto das variáveis de* Ψ *que são verdadeiras naquele estado.*

onde $\mathcal{P}(\Psi)$ *é o conjunto potência do conjunto* Ψ*.* □

Uma estrutura de Kripke determina o conjunto de computações de um sistema composto por variáveis proposicionais Booleanas. Uma estrutura de Kripke pode ser construída a partir de uma máquina de Mealy, usando-se a transformação descrita em (GROSSE; DRECHSLER, 2010, p. 22). Percorrendo-se as estruturas de Kripke, a partir de um estado inicial $e_0 \in E_0$, visitando, recursivamente, os estados sucessores do estado atual, gera-se uma árvore de computação com raiz no estado inicial e_0, com ramificações finitas mas com profundidade ilimitada, permitindo a representação de traços de execução com tamanho ilimitado. A figura 3.3, adaptada de (KROPF, 1999), mostra uma estrutura de Kripke e sua árvore de computação. Note que os símbolos dentro dos estados indicam quais variáveis são verdadeiras no determinado estado.

Uma vez que o modelo de tempo adotado neste trabalho é um modelo linear, cada estado da estrutura de Kripke pode possuir apenas um estado sucessor, ou seja, a árvore de computação gerada a partir da estrutura não possui ramificações. Além disso, por se tratar de um trabalho sobre verificação dinâmica de propriedades, a árvore de computação deve ter uma profundidade limitada, caso contrário não seria possível simular o DUV para a determinada árvore. Desta forma, no lugar das estruturas de Kripke, adotou-se um conceito de estruturas temporais que emprega palavras definidas sobre um alfabeto para modelar a passagem do tempo. Assim, a definição 3.20 estabelece o conceito das estruturas temporais lineares finitas utilizadas neste trabalho.

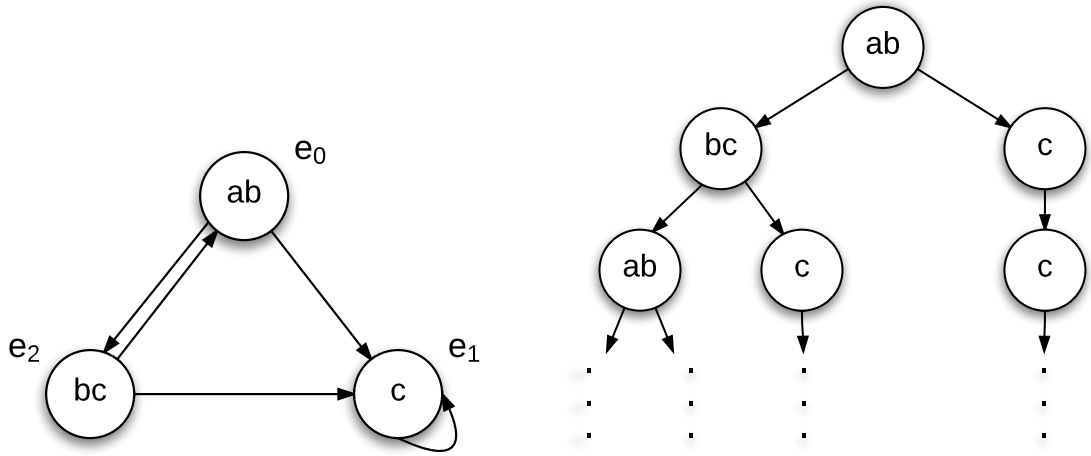


Figura 3.3: Estrutura de Kripke e Árvore de Computação Infinita

Definição 3.20 (Estrutura Temporal Linear Finita). *Uma estrutura temporal linear finita é uma quadrupla* $T = (\Sigma, \Psi, w, \mathcal{V})$*, em que:*

- Σ *é um alfabeto;*
- Ψ *é um conjunto de variáveis;*
- w *é uma palavra construída sobre o alfabeto* Σ*;*
- $\mathcal{V} : \Sigma \times \Psi \mapsto \mathbb{X}$ *é uma função de avaliação das variáveis, que é total.*

onde $\mathbb{X}$ *é o conjunto dos valores válidos para as variáveis de* Ψ*.* □

Informalmente, uma estrutura temporal linear finita permite que se defina uma avaliação para as variáveis de Ψ diferente para cada instante de tempo. Como a função de avaliação $\mathcal{V}$ relaciona um símbolo do alfabeto Σ e uma variável de Ψ a um valor do conjunto $\mathbb{X}$, percorrendo-se a palavra w, cada símbolo encontrado define o valor da avaliação para as variáveis de Ψ em um instante de tempo.

Para descrever comportamento temporal esperado do sistema sob verificação, pode-se empregar a lógica temporal linear (LTL). A LTL é um sistema formal para descrever afirmações temporais, usando o modelo de tempo linear, que utiliza, além dos conectivos lógicos da LP, operadores temporais. Neste trabalho, a LTL será empregada para definir as propriedades de sistemas com memória.

Os operadores **X** (*próximo*), **G** (*globalmente*), **F** (*Eventualmente*), **U** (*até*), e **W** (*até fraco*) são, normalmente, os operadores temporais utilizados na LTL[1]. Uma

[1]Optou-se por utilizar as letras originais para os operadores temporais, não modificando-as para o português, de forma a manter a compatibilidade com os trabalhos na língua inglesa. (Nota do Autor).

descrição textual para o funcionamento destes operadores é oferecida a seguir, antes de da definição da sintaxe e da semântica da LTL.

- **X** α significa que a expressão α deve ser verdadeira no próximo instante de tempo a partir do atual;
- **G** α indica que α deve ser verdadeira em todos os instantes de tempo;
- **F** α indica que α deve se tornar verdadeira em algum instante de tempo no futuro;
- α **U** β requer que α seja verdadeira até que β se torne verdadeira e que β se torne verdadeira em algum ponto no futuro;
- α **W** β significa o mesmo que o operador **U**, porém não requer que β se torne verdadeira no futuro.

A sintaxe para as fórmulas da LTL é apresentada na definição 3.21.

Definição 3.21 (Sintaxe da Fórmula da Lógica Temporal Linear). *Seja* Ψ *um conjunto de fórmulas atômicas, ou variáveis. As fórmulas da lógica temporal linear são definidas, recursivamente, como:*

1. *Todas as fórmulas atômicas de* Ψ *são fórmulas.*
2. *Se* α *e* β *são fórmulas, então também são fórmulas:*

   a) $(\neg\alpha)$

   b) $(\alpha \wedge \beta)$

   c) $(\alpha \vee \beta)$

   d) $(\alpha \rightarrow \beta)$

   e) $(\alpha \leftrightarrow \beta)$

   f) $(\mathbf{X}\ \alpha)$

   g) $(\mathbf{G}\ \alpha)$

   h) $(\mathbf{F}\ \alpha)$

   i) $(\alpha\ \mathbf{U}\ \beta)$

   j) $(\alpha\ \mathbf{W}\ \beta)$ □

No lugar da definição convencional para a semântica das fórmulas da LTL, usando estruturas de Kripke, será apresentada uma semântica que considera um número limitado de instantes de tempo. Desta forma, pode-se empregar as estruturas temporais lineares finitas, apresentadas na definição 3.20, para modelar a passagem do tempo. Uma descrição da semântica convencional da LTL pode ser encontrada em (KROPF, 1999, p. 156).

A semântica para as fórmulas da LTL considerando um intervalo de tempo finito é estabelecida na definição 3.22. A semântica apresentada foi criada usando como base as definições de (IEEE, 2010), (KROPF, 1999) e (BIERE et al., 1999). Nesta definição, os operadores básicos **X** e **U** foram empregados para definir o comportamento dos demais operadores temporais. Na verdade, todos os operadores temporais da LTL podem ser definidos a partir dos operadores básicos **X** e **U** (KROPF, 1999).

Definição 3.22 (Semântica de Tempo Finito para a Fórmula da LTL). *Sejam α e β fórmulas da LTL, $T = (\Sigma, \Psi, w, \mathcal{V})$ uma estrutura temporal linear, e a interpretação semântica dos conectivos lógicos apresentada nas tabelas 3.1, 3.2, 3.3, 3.4 e 3.5. O valor assinalado às fórmulas da LTL é determinado usando a interpretação semântica val, definida recursivamente como:*

a) $\mathit{val}(w, \alpha) = \mathcal{V}(w^0, \alpha)$, *se* $\alpha \in \Psi$

b) $\mathit{val}(w, \neg\alpha) = \neg\mathit{val}(w, \alpha)$

c) $\mathit{val}(w, \alpha \wedge \beta) = \mathit{val}(w, \alpha) \wedge \mathit{val}(w, \beta)$

d) $\mathit{val}(w, \alpha \vee \beta) = \mathit{val}(w, \alpha) \vee \mathit{val}(w, \beta)$

e) $\mathit{val}(w, \alpha \rightarrow \beta) = \mathit{val}(w, \alpha) \rightarrow \mathit{val}(w, \beta)$

f) $\mathit{val}(w, \alpha \leftrightarrow \beta) = \mathit{val}(w, \alpha) \leftrightarrow \mathit{val}(w, \beta)$

g) $\mathit{val}(w, \mathbf{X}\ \alpha) = \begin{cases} \text{V} & \text{, se } |w| > 1 \text{ e } \mathit{val}(w^{1\cdot\cdot}, \alpha) \\ \text{F} & \text{, caso contrário} \end{cases}$

h) $\mathit{val}(w, \alpha\ \mathbf{U}\ \beta) = \begin{cases} \text{V} & \text{, se } \exists k < |w| : \ \mathit{val}(w^{k\cdot\cdot}, \beta) \text{ e } \mathit{val}(w^{j\cdot\cdot}, \alpha),\ \forall j < k \\ \text{F} & \text{, caso contrário} \end{cases}$

i) $\mathit{val}(w, \mathbf{F}\ \alpha) = \mathit{val}(w, \text{V}\ \mathbf{U}\ \alpha)$

j) $\mathit{val}(w, \mathbf{G}\ \alpha) = \mathit{val}(w, \neg(\mathbf{F}\ (\neg\alpha)))$

k) $\mathit{val}(w, \alpha\ \mathbf{W}\ \beta) = \mathit{val}(w, (\alpha\ \mathbf{U}\ \beta) \vee (\mathbf{G}\ \alpha))$ □

3.1.5 Propriedades e Asserções

A definição de propriedade é um tema controverso do ponto de vista filosófico (SWOYER; ORILIA, 2011), porém, na verificação de sistemas digitais, o termo propriedade normalmente é empregado para se referir a uma invariante do sistema sob verificação. Por sua vez, uma propriedade *funcional* representa uma especificação parcial do *comportamento* desejado do sistema em desenvolvimento.

Uma vez que uma propriedade[2] determina uma especificação *parcial* do comportamento do sistema, dois sistemas podem apresentar comportamentos distintos e, mesmo assim, possuírem a determinada propriedade. Considere, por exemplo, a propriedade a seguir:

- **Propriedade P1:** Sempre que a entrada a do sistema estiver em nível lógico 1, no ciclo seguinte, a sua saída y deve estar em nível lógico 1.

Considerando que cada símbolo da palavra de entrada do sistema representa um ciclo e que a palavra de entrada seja $w_a = 00010$, um sistema que possua a propriedade $P1$, dentre outras possibilidades, poderia responder tanto com a palavra de saída 00011, quanto com a palavra de saída 11111. Assim, somente combinando-se todas as propriedades desejadas do sistema, uma especificação total de seu comportamento é obtida. A figura 3.4 representa a combinação de três propriedades, $P1$, $P2$ e $P3$, que determina uma especificação funcional, E, do sistema. Nesta figura, U representa todos os comportamentos possíveis para o sistema sob verificação.

Dado um sistema representado pela máquina de estados finitos $M = (E, \Sigma, \Delta, \delta, \sigma, i)$, o conjunto U, representado na figura 3.4, é definido como $U = \Sigma^* \times \Delta^*$. Informalmente, U é o conjunto de todas as tuplas $<$*estímulo*$/$*resposta*$>$ definidas a partir da combinação de uma palavra de entrada com uma palavra de saída, construídas, respectivamente, a partir dos alfabetos de entrada e de saída do sistema. A partir desta definição do conjunto U, pode-se estabelecer formalmente o conceito de propriedade, que é apresentado na definição 3.23.

Definição 3.23 (Propriedade Funcional). *Seja o sistema S, modelado pela máquina de estados finitos $M = (E, \Sigma, \Delta, \delta, \sigma, i)$. Uma propriedade funcional do sistema S é uma*

[2] A partir deste ponto, o termo propriedade será empregado para se referir a uma propriedade *funcional* do sistema. (Nota do Autor).

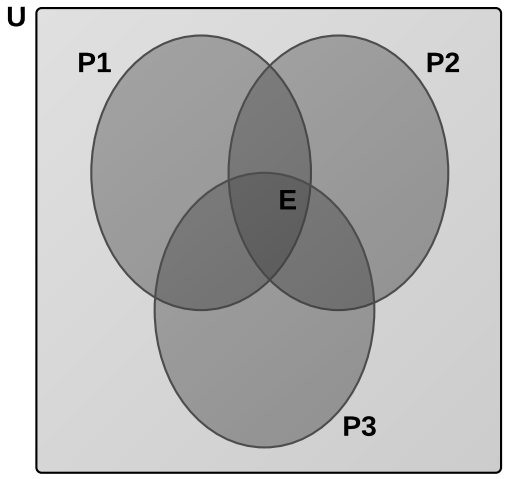


Figura 3.4: Especificação do Sistema como uma Intercessão das Propriedades.

linguagem formal que representa uma dada especificação parcial do sistema, definida sobre a relação entre os conjuntos Σ^* *e* Δ^**. Mais especificamente, a propriedade funcional* P *é o subconjunto de* $\Sigma^* \times \Delta^*$ *que contém todas as tuplas* (s, d) *que não violem a especificação parcial do sistema, onde* $s \in \Sigma^*$ *e* $d \in \Delta^*$*.* □

Uma *asserção* da propriedade P para o sistema S é uma afirmação de que o sistema S possui a propriedade P. Asserções são muito comumente utilizadas na verificação de sistemas.

Definição 3.24 (Asserção da Propriedade). *Seja o sistema,* S*, modelado pela máquina de estados finitos* $M = (E, \Sigma, \Delta, \delta, \sigma, i)$*. Uma asserção da propriedade* P *para este sistema é a função* $\mathcal{A}_P : \Sigma^* \mapsto \{V, F\}$*, definida como:*

$$\mathcal{A}_P(w) = \begin{cases} V, \text{ se } (w, \hat{\sigma}(w)) \in P \\ F, \text{ caso contrário} \end{cases}$$

onde $w \in \Sigma^*$ □

Uma asserção de uma propriedade para um sistema pode ser válida ou não. A definição 3.25 define o conceito de asserção válida.

Definição 3.25 (Asserção Válida). *Seja o sistema,* S*, modelado pela máquina de estados finitos* $M = (E, \Sigma, \Delta, \delta, \sigma, i)$*. Uma asserção,* $\mathcal{A}_P$*, da propriedade* P *para o sistema* S *é válida quando for uma tautologia, ou seja, quando* $\mathcal{A}_P(w) = V, \ \forall w \in \Sigma^*$ □

Obviamente, listar todas as tuplas (s, d), onde $s \in \Sigma^*$ e $d \in \Delta^*$, que pertencem a uma propriedade P do sistema é uma tarefa impraticável. Por este motivo, uma propriedade é normalmente descrita por meio de uma formulação mais sucinta que descreve uma linguagem formal. Formulações comumente utilizadas são a lógica proposicional e sua contrapartida temporal, a lógica temporal proposicional.

3.1.6 Sistemas com Memória e Sistemas sem Memória

Nesta seção, são abordados os conceitos de sistemas com memória e sistemas sem memória. Esta definição se faz necessária porque os conceitos de circuitos combinatórios e circuitos sequenciais, normalmente empregados em projetos de CI, não são consistentes com o comportamento de sistemas descritos em alto nível de abstração. Considere um sistema para calcular a multiplicação entre dois inteiros, por exemplo. Este sistema pode ser implementado tanto por meio de um circuito combinatório quanto por um circuito sequencial. Assim, os conceitos de sistemas com memória e sistemas sem memória são definidos por meio da análise do comportamento temporal do sistema.

Como demonstrado na proposição 3.1, os sistemas tratados neste trabalho podem ser modelados no nível de abstração original por meio de máquinas de estados finitos. Por este motivo, optou-se por considerá-los como máquinas de estados finitos que operam sobre linguagens formais. Antes de definir sistemas sem memória e sistemas com memória, os conceitos normalmente empregados na descrição de sistemas digitais são, portanto, relacionados aos conceitos de linguagens formais. As definições 3.26 e 3.27 estabelecem os conceitos de símbolo de entrada e símbolo de saída.

Definição 3.26 (Símbolo de Entrada). *Um símbolo de entrada do sistema é uma n-upla $(s_1, s_2, .., s_n)$, onde n é o número de sinais de entrada do sistema e s_i representa um estado possível para a sua i-ésima entrada.* □

Definição 3.27 (Símbolo de Saída). *Um símbolo de saída do sistema é uma n-upla $(s_1, s_2, .., s_n)$, onde n é o número de sinais de saída do sistema e s_i representa um estado possível para a sua i-ésima saída.* □

Implicitamente, os símbolos de entrada e de saída definem uma função de avaliação dos sinais do sistema. Para simplificar as notações empregadas neste trabalho,

se γ é um símbolo de entrada ou de saída de um sistema, será utilizada a notação $\gamma(s_1)$ para denotar a avaliação do sinal s_1 dentro do símbolo γ.

A partir dos conceitos de *Símbolo de Entrada* e *Símbolo de saída*, definem-se os conceitos de *Alfabeto de Entrada* e *Alfabeto de Saída* de um sistema.

Definição 3.28 (Alfabeto de Entrada). *O alfabeto de entrada de um sistema é o conjunto de todos os símbolos de entrada do sistema.* □

Definição 3.29 (Alfabeto de Saída). *O alfabeto de saída de um sistema é o conjunto de todos os símbolos de saída do sistema.* □

As palavras de entrada e saída do sistema representam a evolução de seus sinais de entrada e de saída com o passar do tempo. Neste sentido as palavras de entrada e saída podem ser associadas às estruturas temporais lineares finitas, apresentadas na definição 3.20, onde cada símbolo indica, respectivamente, o estado dos sinais de entrada e saída em um determinado instante de tempo. Em sistemas síncronos, os instantes de tempo são determinados pelo sinal de sincronização que é, em geral, o sinal de *clock*. Nos sistemas assíncronos, os instantes de tempo são marcados por eventos discretos, como uma mudança no valor de um sinal, por exemplo. Neste modelo de evolução do tempo, o tamanho da palavra indica quantos instantes de tempo ocorreram, mas não indica o tempo exato em que a avaliação dos sinais ocorreu. Nos casos em que o valor do tempo seja explicitamente necessário para a computação realizada pelo sistema, este pode ser adicionado aos símbolos de entrada e saída, comportando-se como um sinal adicional.

Segundo Haykin e Veen (HAYKIN; VEEN, 2002), um sistema possui memória quando o valor atual de seu sinal de saída depende de valores passados do sinal de entrada[3]. Unindo-se este conceito ao modelo de evolução do tempo adotado, pode-se definir formalmente o conceito de sistema sem memória, que é estabelecido na definição 3.30 utilizando-se a máquina de Mealy[4].

[3]Na verdade, Haykin e Veen definem um sistema com memória como aquele cujo valor atual de seu sinal de saída depende de valores passados ou futuros do sinal de entrada. Como todos os sistemas tratados neste trabalho são *causais* (não dependem de valores futuros), a definição de sistema com memória pode ser simplificada. (Nota do Autor).

[4]Como a condição de equivalência entre as Máquinas de Mealy e de Moore foi apresentada na seção 3.1.2, a partir deste ponto a máquina de Mealy será utilizada como a máquina padrão, uma vez que esta é a preferida para a representação de sistemas digitais na literatura. (Nota do Autor).

Definição 3.30 (Sistema sem Memória). *Seja o sistema S, modelado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$. O sistema S é um sistema sem memória se, e somente se:*

$$\forall(x, y \in \Sigma^* \text{ e } a \in \Sigma)\text{: } \sigma(\hat{\delta}(i, x), a) = \sigma(\hat{\delta}(i, y), a) \qquad \square$$

Informalmente, a definição 3.30 estabelece que a saída computada por um sistema sem memória quando recebe o símbolo de entrada a depois de ter processado a palavra x deve ser a mesma saída computada pelo sistema quando recebe o mesmo símbolo a depois de ter processado a palavra y, para todas as palavras x e y construídas sobre o alfabeto de entrada e todo símbolo a do alfabeto de entrada.

Apesar da definição de sistema com memória decorrer diretamente da definição 3.30, formaliza-se este conceito na definição 3.31.

Definição 3.31 (Sistema com Memória). *Seja o sistema S, modelado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$. O sistema S é um sistema com memória se, e somente se:*

$$\exists(x, y \in \Sigma^* \ e \ a \in \Sigma)\text{: } \sigma(\hat{\delta}(i, x), a) \neq \sigma(\hat{\delta}(i, y), a) \qquad \square$$

3.1.7 Notação dos Algoritmos

Os algoritmos utilizados no decorrer deste trabalho são descritos por meio de pseudocódigo. Portanto, na maior parte dos casos, não é necessária uma descrição da sintaxe e da semântica empregadas, pois as palavras chave possuem o mesmo significado do português. No entanto, para o caso das construções dos algoritmos multitarefa, é necessário uma explicação mais detalhada. Neste caso, quatro palavras chave são utilizadas para descrever comportamento concorrente: *PROCESSO*, *COMITÊ*, *sincronize* e *aborte*. Estas palavras são empregadas com o mesmo sentido que as palavras *spawn*, *inlet*, *sync* e *abort* são, respectivamente, definidas na linguagem Cilk (Massachusetts Institute of Technology, 2012).

A palavra chave *PROCESSO* sempre precede uma chamada de função e é utilizada para criar um novo processo para tratar a função disparada, liberando o processo pai para continuar a execução da função original. A palavra chave *sincronize* é utilizada dentro do código executado pelo processo pai para que este aguarde a finalização da execução

```
1  FUNCAO fib
2    ENTRADA
3      Um numero inteiro n
4    SAIDA
5      O valor do n-esimo numero da sequencia de Fibonacci
6  INICIO
7    x ← 0
8
9    COMITE somador
10     ENTRADAS
11       Um numero inteiro resultado
12   INICIO
13       x ← x + resultado
14   FIM
15
16   se (n < 2) entao
17     retorne n
18   senao
19     somador(PROCESSO fib(n − 1))
20     somador(PROCESSO fib(n − 2))
21   fim se
22   sincronize
23   retorne x
24 FIM
```

Listagem 3.1: Algoritmo Multitarefa para o Cálculo da Sequência de Fibonacci

de todos os seus processos filhos. Sempre que encontrada uma chamada *sincronize*, a execução do processo pai ficará suspensa até que todos os filhos tenham terminado suas tarefas.

De forma a permitir um tratamento mais elaborado sobre a execução dos processos filhos, pode-se utilizar a palavra chave *COMITÊ*, que define uma função interna à função principal, que é empregada para tratar o retorno de funções que estão sendo executadas por processos filhos. Por fim, a palavra chave *aborte* pode ser utilizada dentro de uma função *COMITÊ* para terminar a execução de todos os processos filhos de um determinado processo. O emprego destas palavras chave, exceto a palavra *aborte*, é exemplificado no algoritmo da Listagem 3.1, que calcula o n-ésimo número da sequência de Fibonacci.

3.2 Proposta de uma Extensão para a LP

É interessante, para os fins deste trabalho, estender a sintaxe e a semântica da lógica proposicional de forma a incluir operadores relacionais sobre variáveis e constan-

tes do tipo real, ou algum de seus subconjuntos próprios. Do ponto de vista semântico, esta extensão não aumenta a capacidade de abstração da LP, pois as fórmulas atômicas (ou variáveis) da LP já poderiam incluir expressões construídas usando operadores relacionais, uma vez que tais operadores se enquadram na definição da função de avaliação das fórmulas atômicas, $\mathcal{V}$, apresentada anteriormente. No entanto, uma definição mais precisa das fórmulas atômicas é necessária para que, no capítulo 4, uma fórmula da lógica proposicional possa ser traduzida para as heurísticas empregadas no método proposto neste trabalho. Assim, nesta seção, define-se a sintaxe e a semântica para esta lógica proposicional estendida (LPE).

Definição 3.32 (Sintaxe da Fórmula da Lógica Proposicional Estendida). *Seja* Γ *um conjunto de variáveis, ou fórmulas atômicas. As fórmulas da lógica proposicional estendida são definidas, recursivamente, como:*

1. *Se* γ *e* η *são fórmulas atômicas de* Γ*, então são fórmulas relacionais:*
   - *a)* $(\gamma = \eta)$
   - *b)* $(\gamma \neq \eta)$
   - *c)* $(\gamma > \eta)$
   - *d)* $(\gamma \geq \eta)$
   - *e)* $(\gamma < \eta)$
   - *f)* $(\gamma \leq \eta)$
2. *Todas as fórmulas relacionais da LPE são fórmulas da LPE.*
3. *Se* α *e* β *são fórmulas da LPE, então também são fórmulas da LPE:*
   - *a)* $(\neg\alpha)$
   - *b)* $(\alpha \wedge \beta)$
   - *c)* $(\alpha \vee \beta)$
   - *d)* $(\alpha \rightarrow \beta)$
   - *e)* $(\alpha \leftrightarrow \beta)$

□

Da mesma forma que na LP, pode-se derivar um valor para uma fórmula da lógica proposicional estendida a partir da avaliação de suas variáveis. Uma avaliação das variáveis de Γ é uma função $\mathcal{E} : \Gamma \mapsto \mathbb{X}$, onde $\mathbb{X} \subseteq \mathbb{R}$. Primeiro, define-se a semântica

das fórmulas relacionais da LPE. Em seguida, a semântica para as fórmulas da LPE é apresentada.

Definição 3.33 (Semântica da Fórmula Relacional da LPE). *Sejam γ e η fórmulas atômicas da lógica proposicional estendida, $\mathcal{E}$ uma função de avaliação das variáveis e a interpretação semântica do conectivo lógico negação, apresentada na tabela 3.1. O valor assinalado às fórmulas relacionais da lógica proposicional estendida é determinado usando a interpretação semântica $\mathit{val}_{\mathcal{R}}$, definida recursivamente como:*

1. $\mathit{val}_{\mathcal{R}}(\gamma = \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{E}(\gamma) \text{ for igual a } \mathcal{E}(\eta) \\ \text{F}, & \text{caso contrário} \end{cases}$
2. $\mathit{val}_{\mathcal{R}}(\gamma > \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{E}(\gamma) \text{ for maior que } \mathcal{E}(\eta) \\ \text{F}, & \text{caso contrário} \end{cases}$
3. $\mathit{val}_{\mathcal{R}}(\gamma \geq \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{E}(\gamma) \text{ for maior ou igual a } \mathcal{E}(\eta) \\ \text{F}, & \text{caso contrário} \end{cases}$
4. $\mathit{val}_{\mathcal{R}}(\gamma \neq \eta) = \neg \mathit{val}_{\mathcal{R}}(\gamma = \eta)$
5. $\mathit{val}_{\mathcal{R}}(\gamma < \eta) = \neg \mathit{val}_{\mathcal{R}}(\gamma \geq \eta)$
6. $\mathit{val}_{\mathcal{R}}(\gamma \leq \eta) = \neg \mathit{val}_{\mathcal{R}}(\gamma > \eta)$

Definição 3.34 (Semântica da Fórmula da Lógica Proposicional Estendida). *Sejam α e β fórmulas da lógica proposicional estendida, a interpretação semântica dos conectivos lógicos apresentada nas tabelas 3.1, 3.2, 3.3, 3.4 e 3.5 e a semântica das fórmulas relacionais da lógica proposicional estendida, apresentada na definição 3.33. O valor assinalado às fórmulas da lógica proposicional estendida é determinado usando a interpretação semântica val, definida recursivamente como:*

1. $\mathit{val}(\alpha) = \mathit{val}_{\mathcal{R}}(\alpha)$, *se α for uma fórmula relacional da LPE*
2. $\mathit{val}(\neg\alpha) = \neg \mathit{val}(\alpha)$
3. $\mathit{val}(\alpha \wedge \beta) = \mathit{val}(\alpha) \wedge \mathit{val}(\beta)$
4. $\mathit{val}(\alpha \vee \beta) = \mathit{val}(\alpha) \vee \mathit{val}(\beta)$
5. $\mathit{val}(\alpha \rightarrow \beta) = \mathit{val}(\alpha) \rightarrow \mathit{val}(\beta)$
6. $\mathit{val}(\alpha \leftrightarrow \beta) = \mathit{val}(\alpha) \leftrightarrow \mathit{val}(\beta)$ □

3.3 Proposta de uma Extensão para a LTL

Da mesma forma que foi feito para a LP, pode-se definir uma extensão para a LTL, permitindo a utilização de operadores relacionais. Nesta seção, é definida esta LTL estendida (LTLE).

Definição 3.35 (Sintaxe da Fórmula da Lógica Temporal Linear Estendida). *Seja* Γ *um conjunto de fórmulas atômicas, ou variáveis. As fórmulas da lógica temporal linear estendida são definidas, recursivamente, como:*

1. *Se* γ *e* η *são fórmulas atômicas de* Γ*, então são fórmulas relacionais:*
   - *a)* $(\gamma = \eta)$
   - *b)* $(\gamma \neq \eta)$
   - *c)* $(\gamma > \eta)$
   - *d)* $(\gamma \geq \eta)$
   - *e)* $(\gamma < \eta)$
   - *f)* $(\gamma \leq \eta)$
2. *Todas as fórmulas relacionais da LTLE são fórmulas da LTLE.*
3. *Se* α *e* β *são fórmulas, então também são fórmulas:*
   - *a)* $(\neg\alpha)$
   - *b)* $(\alpha \wedge \beta)$
   - *c)* $(\alpha \vee \beta)$
   - *d)* $(\alpha \rightarrow \beta)$
   - *e)* $(\alpha \leftrightarrow \beta)$
   - *f)* $(\mathbf{X}\ \alpha)$
   - *g)* $(\mathbf{G}\ \alpha)$
   - *h)* $(\mathbf{F}\ \alpha)$
   - *i)* $(\alpha\ \mathbf{U}\ \beta)$
   - *j)* $(\alpha\ \mathbf{W}\ \beta)$

□

Pode-se derivar um valor para uma fórmula da lógica temporal linear estendida a partir da avaliação de suas variáveis, usando uma estrutura temporal linear, conforme a definição 3.20. Primeiro, define-se a semântica das fórmulas relacionais da LTLE. Em seguida, a semântica para as fórmulas da LTLE é apresentada.

Definição 3.36 (Semântica da Fórmula Relacional da LTLE). *Sejam γ e η fórmulas atômicas da lógica temporal linear estendida, $T = (\Sigma, \Psi, w, \mathcal{V})$ uma estrutura temporal linear, e a interpretação semântica do conectivo lógico negação, apresentada na tabela 3.1. O valor assinalado às fórmulas relacionais da lógica temporal linear estendida é determinado usando a interpretação semântica $\mathit{val}_{\mathcal{R}}$, definida recursivamente como:*

1. $\mathit{val}_{\mathcal{R}}(w, \gamma = \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{V}(w^0, \gamma) \text{ for igual a } \mathcal{V}(w^0, \eta) \\ \text{F}, & \text{caso contrário} \end{cases}$

2. $\mathit{val}_{\mathcal{R}}(w, \gamma > \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{V}(w^0, \gamma) \text{ for maior que } \mathcal{V}(w^0, \eta) \\ \text{F}, & \text{caso contrário} \end{cases}$

3. $\mathit{val}_{\mathcal{R}}(w, \gamma \geq \eta) = \begin{cases} \text{V}, & \text{se } \mathcal{V}(w^0, \gamma) \text{ for maior ou igual a } \mathcal{V}(w^0, \eta) \\ \text{F}, & \text{caso contrário} \end{cases}$

4. $\mathit{val}_{\mathcal{R}}(w, \gamma \neq \eta) = \neg \mathit{val}_{\mathcal{R}}(w, \gamma = \eta)$

5. $\mathit{val}_{\mathcal{R}}(w, \gamma < \eta) = \neg \mathit{val}_{\mathcal{R}}(w, \gamma \geq \eta)$

6. $\mathit{val}_{\mathcal{R}}(w, \gamma \leq \eta) = \neg \mathit{val}_{\mathcal{R}}(w, \gamma > \eta)$

Definição 3.37 (Semântica de Tempo Finito para a Fórmula da LTLE). *Sejam α e β fórmulas da LTLE, $T = (\Sigma, \Psi, w, \mathcal{V})$ uma estrutura temporal linear, a interpretação semântica dos conectivos lógicos apresentada nas tabelas 3.1, 3.2, 3.3, 3.4 e 3.5, e a interpretação semântica das fórmulas relacionais da LTLE, apresentada na definição 3.36. O valor assinalado às fórmulas da LTLE é determinado usando a interpretação semântica val, definida recursivamente como:*

a) $\mathit{val}(w, \alpha) = \mathit{val}_{\mathcal{R}}(w, \alpha)$, *se α é uma fórmula relacional da LTLE*

b) $\mathit{val}(w, \neg\alpha) = \neg \mathit{val}(w, \alpha)$

c) $\mathit{val}(w, \alpha \wedge \beta) = \mathit{val}(w, \alpha) \wedge \mathit{val}(w, \beta)$

d) $\mathit{val}(w, \alpha \vee \beta) = \mathit{val}(w, \alpha) \vee \mathit{val}(w, \beta)$

e) $\mathit{val}(w, \alpha \rightarrow \beta) = \mathit{val}(w, \alpha) \rightarrow \mathit{val}(w, \beta)$

f) $\mathit{val}(w, \alpha \leftrightarrow \beta) = \mathit{val}(w, \alpha) \leftrightarrow \mathit{val}(w, \beta)$

$$g)\ \mathit{val}(w, \mathbf{X}\ \alpha) = \begin{cases} \text{V} & \text{, se } |w| > 1 \text{ e } \mathit{val}(w^{1\cdot\cdot}, \alpha) \\ \text{F} & \text{, caso contrário} \end{cases}$$

$$h)\ \mathit{val}(w, \alpha\ \mathbf{U}\ \beta) = \begin{cases} \text{V} & \text{, se } \exists k < |w| :\ \mathit{val}(w^{k\cdot\cdot}, \beta) \text{ e } \mathit{val}(w^{j\cdot\cdot}, \alpha),\ \forall j < k \\ \text{F} & \text{, caso contrário} \end{cases}$$

$$i)\ \mathit{val}(w, \mathbf{F}\ \alpha) = \mathit{val}(w, \text{V}\ \mathbf{U}\ \alpha)$$

$$j)\ \mathit{val}(w, \mathbf{G}\ \alpha) = \mathit{val}(w, \neg(\mathbf{F}\ (\neg\alpha)))$$

$$k)\ \mathit{val}(w, \alpha\ \mathbf{W}\ \beta) = \mathit{val}(w, (\alpha\ \mathbf{U}\ \beta) \vee (\mathbf{G}\ \alpha))$$ □

3.4 Resumo do Capítulo

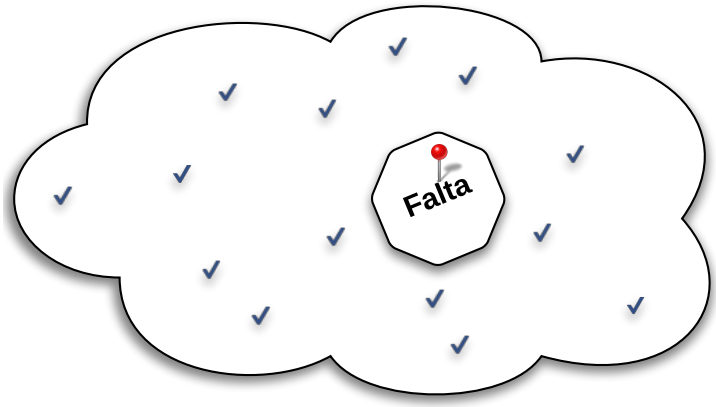
Neste capítulo, foram apresentados os principais conceitos e notações que serão utilizados no decorrer dos próximos capítulos. Além de conceitos já bem estabelecidos na literatura, como os relacionados às linguagens formais, máquinas de estados finitos, lógica proposicional e proposicional temporal, também foram propostas extensões para a lógica proposicional e para a lógica temporal linear que oferecem suporte explícito a operadores aritméticos relacionais em suas fórmulas. Estas duas contribuições originais deste trabalho, apresentadas respectivamente nas seções 3.2 e 3.3, serão utilizadas como suporte para as demais contribuições deste trabalho, que serão apresentadas nos capítulos seguintes.

Capítulo 4

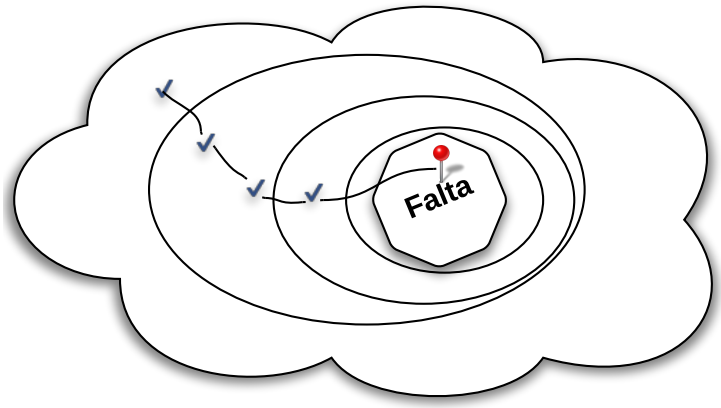
Mapeamento da Verificação Dinâmica de Propriedades para um Problema de Otimização em Sistemas sem Memória

Como foi apresentado no capítulo 2, os métodos de verificação de propriedades para sistemas descritos em alto nível de abstração baseados em verificação de modelos garantem completude da verificação às custas de uma escalabilidade limitada. Além disso, estas abordagens baseiam-se em uma análise estática do código fonte, o que (atualmente) implica na perda de generalidade da técnica. Por outro lado, os métodos dinâmicos para verificação de propriedades trocam completude por escalabilidade, mas apresentam uma desvantagem importante: estes métodos utilizam a estrutura interna do DUV, o que resulta na perda de generalidade do método, ou ignoram o DUV completamente, renunciando a informações importantes que podem ser derivadas a partir de seu comportamento durante a validação.

O método de DPC proposto neste trabalho aborda a verificação baseada em asserções de modelos descritos em alto nível de abstração sob uma perspectiva diferente. As fórmulas proposicionais que representam as propriedades são traduzidas para funções heurísticas capazes de avaliar pares $<estímulo/resposta>$ com relação à sua proximidade aos contraexemplos das asserções. Para isto, as funções heurísticas são definidas de tal forma que seus ótimos globais correspondem às violações da especificação parcial representada pela propriedade, se estas violações existirem. Diferentemente das estratégias atuais para a geração de vetores de teste para a DPC, onde as faltas são exercitadas apenas de forma incidental, como mostrado na figura 4.1a, o método proposto neste trabalho



(a) Geração Convencional de Vetores de Teste



(b) Geração de Estímulos Usando DPC Heurística

Figura 4.1: Representação Ilustrativa da Exploração do Espaço de Estados

direciona a simulação para os contraexemplos, usando algoritmos de otimização guiados pelas funções heurísticas. A figura 4.1b apresenta uma representação ilustrativa deste processo.

Neste capítulo, é apresentado o método para a DPC heurística de sistemas sem memória. No capítulo 5, este método é expandido para os sistemas com memória.

4.1 Modelando a Busca por Contraexemplos como um problema de Otimização

Considere um sistema sem memória S (conforme a definição 3.30) modelado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$, uma propriedade P, e uma asserção $\mathcal{A}_P$ que afirma que o sistema S possui a propriedade P. Pode-se definir o conjunto dos símbolos do alfabeto de entrada Σ que, quando aplicados à máquina M, produzem um símbolo

de saída que viola a asserção $\mathcal{A}_P$. Este conjunto dos contraexemplos da asserção $\mathcal{A}_P$, ou $\Omega_{\mathcal{A}_P}$, é definido conforme apresentado na equação 4.1.

$$\Omega_{\mathcal{A}_P} = \{s \in \Sigma \mid \mathcal{A}_P(s) = \mathrm{F}\} \tag{4.1}$$

A partir da definição do conjunto $\Omega_{\mathcal{A}_P}$, pode-se definir o conceito de função de classificação baseada em asserção (FCBA), que atribui um valor real a cada símbolo de entrada do sistema, de forma que, dados dois símbolos de Σ, se apenas um deles pertencer ao conjunto dos contraexemplos, o valor atribuído a este símbolo deve ser menor do que o valor atribuído ao outro símbolo que não pertence ao conjunto dos contraexemplos. O conceito para a FCBA é estabelecido na definição 4.1.

Definição 4.1 (Função de Classificação Baseada em Asserção). *Uma função de classificação baseada em asserção é uma função $f_{\mathcal{A}_P} : \Sigma \mapsto \mathbb{R}$, tal que:*

$$\forall s_1 \in \Omega_{\mathcal{A}_P} \text{ e } \forall s_2 \notin \Omega_{\mathcal{A}_P} : f_{\mathcal{A}_P}(s_1) < f_{\mathcal{A}_P}(s_2)$$

onde $s_1, s_2 \in \Sigma$. □

Apesar de simples, este conceito de FCBA é suficiente para modelar o problema de busca por contraexemplos como um problema de otimização. Este mapeamento é introduzido pelo teorema 4.1 e pelo seu corolário 4.1, apresentados a seguir.

Teorema 4.1 (Mapeamento Validação-Otimização). *Seja o sistema sem memória S, representado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$, e uma propriedade P. Seja a função de classificação baseada em asserção $f_{\mathcal{A}_P}$ criada a partir da asserção $\mathcal{A}_P$, que afirma que o sistema S possui a propriedade P. Se um símbolo de entrada, s, do alfabeto Σ minimiza a função $f_{\mathcal{A}_P}$, pode-se concluir que s pertence ao conjunto dos contraexemplos da asserção $\mathcal{A}_P$ ou então que este conjunto é vazio:*

$$s \in \underset{x \in \Sigma}{argmin}\ f_{\mathcal{A}_P}(x) \rightarrow s \in \Omega_{\mathcal{A}_P} \vee \Omega_{\mathcal{A}_P} = \emptyset \tag{4.2}$$

Demonstração. A prova do teorema 4.1 é realizada pela contrapositiva. Suponha a negação do consequente da equação 4.2:

$$\neg(s \in \Omega_{\mathcal{A}_P} \vee \Omega_{\mathcal{A}_P} = \emptyset) \tag{4.3}$$

Aplicando-se o teorema de DeMorgan em (4.3):

$$s \notin \Omega_{\mathcal{A}_P} \wedge \Omega_{\mathcal{A}_P} \neq \emptyset \tag{4.4}$$

Mas, se $s \notin \Omega_{\mathcal{A}_P}$ e $\Omega_{\mathcal{A}_P} \neq \emptyset$, deve existir um elemento $z \neq s$ tal que $z \in \Omega_{\mathcal{A}_P}$. Usando a definição de função de classificação baseada em asserção (definição 4.1), se $z \in \Omega_{\mathcal{A}_P}$ e $s \notin \Omega_{\mathcal{A}_P}$, então $f_{\mathcal{A}_P}(z) < f_{\mathcal{A}_P}(s)$, ou:

$$\neg(s \in \Omega_{\mathcal{A}_P} \vee \Omega_{\mathcal{A}_P} = \emptyset) \rightarrow \neg(s \in \underset{x \in \Sigma}{argmin}\ f_{\mathcal{A}_P}(x)) \tag{4.5}$$

Portanto, se $s \in \underset{x \in \Sigma}{argmin}\ f_{\mathcal{A}_P}(x)$, então $s \in \Omega_{\mathcal{A}_P}$ ou $\Omega_{\mathcal{A}_P} = \emptyset$ □

Corolário 4.1. *Seja o sistema sem memória S, representado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$. Se um símbolo do alfabeto de entrada, s, que minimiza a função $f_{\mathcal{A}_P}$ for encontrado, ele pode ser testado contra a asserção $\mathcal{A}_P$. Caso $\mathcal{A}_P(s) = V$, então a asserção $\mathcal{A}_P$ é uma asserção válida, conforme a definição 3.25.*

Demonstração. A partir do teorema 4.1, usando prova direta para a condicional:

$$\frac{\begin{array}{c} s \in \underset{x \in \Sigma}{argmin}\ f_{\mathcal{A}_P}(x) \rightarrow s \in \Omega_{\mathcal{A}_P} \ \vee\ \Omega_{\mathcal{A}_P} = \emptyset \\ s \in \underset{x \in \Sigma}{argmin}\ f_{\mathcal{A}_P}(x) \end{array}}{s \in \Omega_{\mathcal{A}_P} \ \vee\ \Omega_{\mathcal{A}_P} = \emptyset}$$

Aplicando-se silogismo disjuntivo a este resultado:

$$\frac{\begin{array}{c} s \in \Omega_{\mathcal{A}_P} \ \vee\ \Omega_{\mathcal{A}_P} = \emptyset \\ \neg(s \in \Omega_{\mathcal{A}_P}) \end{array}}{\Omega_{\mathcal{A}_P} = \emptyset}$$

Mas, se $\Omega_{\mathcal{A}_P} = \emptyset$, então a asserção $\mathcal{A}_P$ é válida pela definição 3.25, pois, pela equação 4.1, $\mathcal{A}_P(s) = V$, para todo $s \in \Sigma$. □

Embora o teorema 4.1 e o corolário 4.1 mapeiem o problema da busca por contraexemplos de uma asserção para um problema de otimização, o conceito de FCBA é demasiadamente fraco para ser empregado diretamente. Por exemplo, uma FCBA ingênua para a asserção $\mathcal{A}_P$ pode ser criada diretamente a partir do conceito de asserção, estabelecido na definição 3.24. A equação 4.6 apresenta esta FCBA ingênua, $g_{\mathcal{A}_P}$, que é facilmente obtida a partir da simulação do sistema, porém de pouca utilidade para a busca por con-

traexemplos, pois não acrescenta nenhuma informação que possa ser utilizada durante a busca.

$$g_{\mathcal{A}_P}(s) = \begin{cases} 0, & \text{se } \mathcal{A}_P(s) = \text{F} \\ 1, & \text{se } \mathcal{A}_P(s) = \text{V} \end{cases} \tag{4.6}$$

Como o operador *argmin*, empregado pelo teorema e pelo corolário para definir os símbolos de entrada que minimizam a função $f_{\mathcal{A}_P}$ é, geralmente, concretizado por um algoritmo de otimização numérica, se a FCBA não oferecer nenhuma informação que possa ser utilizada durante a execução do algoritmo, a busca por contraexemplos se torna uma validação exaustiva, tarefa impraticável para a maioria dos sistemas reais. Na próxima seção, é apresentada uma FCBA heurística que pode ser utilizada para este propósito.

4.2 Uma FCBA Heurística

A FCBA ingênua, definida na equação 4.6, é de pouca utilidade para a busca por contraexemplos por não fornecer informação suficiente para agilizar o processo de busca. Isto é, dados dois símbolos s_1 e s_2 quaisquer do alfabeto de entrada, se nenhum dos símbolos for um contraexemplo da asserção, a FCBA não oferece nenhuma informação sobre qual dos dois símbolos está mais próximo de violar a asserção sendo testada. É possível imaginar uma FCBA ideal para a qual, dados dois símbolos quaisquer do alfabeto de entrada, o valor atribuído pela FCBA ao símbolo que está mais próximo de violar a asserção é sempre menor do que o valor atribuído ao símbolo mais distante de violar a asserção, de acordo com um critério de proximidade definido. Caso este critério de proximidade seja a distância euclidiana entre os dois símbolos, o processo de busca utilizando esta FCBA ideal como guia se torna tão trivial que um simples algoritmo *Hill-Climbing* (RUSSELL; NORVIG, 2002, p. 111) pode ser utilizado para validar o sistema facilmente. A figura 4.2 apresenta uma ilustração de uma FCBA ideal, onde ω_1 e ω_2 são os contraexemplos e as curvas de nível indicam a direção na a qual o valor da FCBA aumenta.

Derivar uma FCBA ideal para um par *sistema/propriedade*, no entanto, não é uma tarefa viável na maioria dos casos, pois requer um conhecimento minucioso do comportamento do modelo. Porém, é possível definir uma FCBA prática a partir da uti-

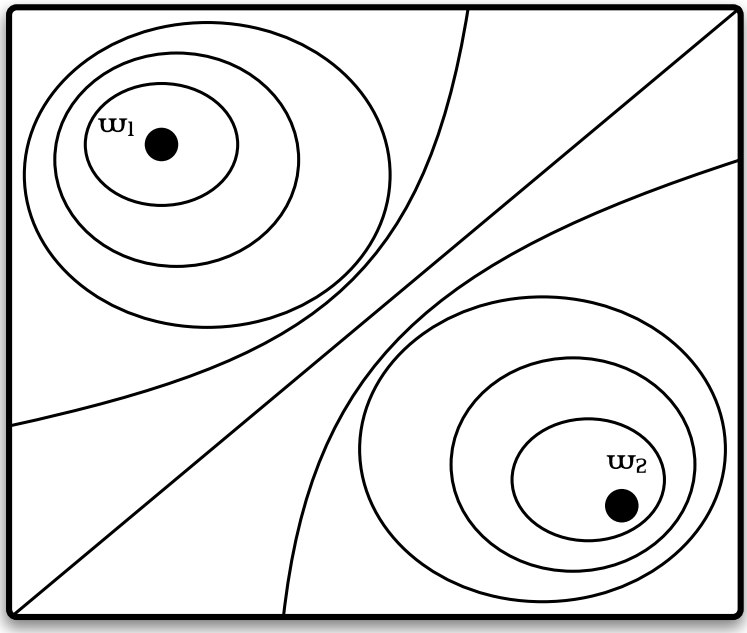


Figura 4.2: Representação Ilustrativa de uma FCBA Ideal

lização da técnica de função de penalidade (NOCEDAL; WRIGHT, 1999; GALLAGHER; NARASIMHAN, 1997), normalmente empregada para transformar um problema de otimização sujeito a restrições em um problema de otimização sem restrições. Nesta técnica, uma função de penalidade é associada a cada tipo de restrição (*igual*, *maior*, *maior ou igual*, *menor*, *menor ou igual* ou *diferente*) de forma que o valor da função seja sempre positivo, mas diminua na medida em que a restrição esteja próxima de ser satisfeita.

Para definir esta FCBA prática, é oferecida uma interpretação semântica alternativa para as fórmulas da lógica proposicional estendida (LPE), apresentada na seção 3.2. No restante desta discussão, assume-se que a propriedade P do sistema está descrita na forma de uma fórmula da LPE e que o critério de proximidade utilizado é a distância euclidiana. Para as interpretações semânticas a seguir, considere que h_{eq}, h_{gt}, h_{ge} e h_{ne} são, respectivamente, as funções de penalidade para os operadores $\{=, >, \geq \ e \ \neq\}$. A formulação para tais funções de penalidade é apresentada nas subseções 4.2.1 e 4.2.2.

Definição 4.2 (Semântica Alternativa da Fórmula Relacional da LPE). *Sejam γ e η fórmulas atômicas, ou variáveis, da lógica proposicional estendida e $\mathcal{E} : \Gamma \mapsto \mathbb{X}$, uma função de avaliação de suas variáveis, onde $\mathbb{X} \subseteq \mathbb{R}$ e Γ é o conjunto das variáveis da LPE. O valor assinalado às fórmulas relacionais da lógica proposicional estendida, determinado pela interpretação semântica alternativa $heu_{\mathcal{R}}$, é definido como:*

1. $\mathcal{heu}_{\mathcal{R}}(\gamma = \eta) = h_{eq}(\mathcal{E}(\gamma), \mathcal{E}(\eta))$
2. $\mathcal{heu}_{\mathcal{R}}(\gamma > \eta) = h_{gt}(\mathcal{E}(\gamma), \mathcal{E}(\eta))$
3. $\mathcal{heu}_{\mathcal{R}}(\gamma \geq \eta) = h_{ge}(\mathcal{E}(\gamma), \mathcal{E}(\eta))$
4. $\mathcal{heu}_{\mathcal{R}}(\gamma \neq \eta) = h_{ne}(\mathcal{E}(\gamma), \mathcal{E}(\eta))$
5. $\mathcal{heu}_{\mathcal{R}}(\gamma < \eta) = \mathcal{heu}_{\mathcal{R}}(\eta > \gamma)$
6. $\mathcal{heu}_{\mathcal{R}}(\gamma \leq \eta) = \mathcal{heu}_{\mathcal{R}}(\eta \geq \gamma)$
7. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma = \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma \neq \eta)$
8. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma > \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma \leq \eta)$
9. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma \geq \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma < \eta)$
10. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma \neq \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma = \eta)$
11. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma < \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma \geq \eta)$
12. $\mathcal{heu}_{\mathcal{R}}(\neg(\gamma \leq \eta)) = \mathcal{heu}_{\mathcal{R}}(\gamma > \eta)$ □

As equações 4.7 e 4.8, a seguir, estabelecem as equivalências que foram utilizadas para se construir os itens 5 e 6 da definição 4.2. Os itens de 7 a 12 da mesma definição foram obtidos substituindo-se o operador relacional da fórmula por seu operador complementar.

$$(\gamma < \eta) = (\eta > \gamma) \tag{4.7}$$

$$(\gamma \leq \eta) = (\eta \geq \gamma) \tag{4.8}$$

Definição 4.3 (Semântica Alternativa da Fórmula da LPE). *Sejam α e β fórmulas da lógica proposicional estendida, min(x,y) uma função que retorna o menor valor entre x e y, com $x, y \in \mathbb{X}$ e $\mathbb{X} \subseteq \mathbb{R}$, e a semântica das fórmulas relacionais da lógica proposicional estendida, apresentada na definição 4.2. O valor assinalado às fórmulas da lógica proposicional estendida, determinado usando a interpretação semântica alternativa $\mathcal{heu}$, é definido recursivamente como:*

1. $\mathcal{heu}(\alpha) = \mathcal{heu}_{\mathcal{R}}(\alpha)$, *se* α *for uma fórmula relacional da LPE*
2. $\mathcal{heu}(\neg\alpha) = \mathcal{heu}_{\mathcal{R}}(\neg\alpha)$, *se* α *for uma fórmula relacional da LPE*
3. $\mathcal{heu}(\alpha \wedge \beta) = \mathcal{heu}(\alpha) + \mathcal{heu}(\beta)$
4. $\mathcal{heu}(\alpha \vee \beta) = min(\mathcal{heu}(\alpha), \mathcal{heu}(\beta))$
5. $\mathcal{heu}(\neg(\alpha \wedge \beta)) = \mathcal{heu}(\neg\alpha \vee \neg\beta)$
6. $\mathcal{heu}(\neg(\alpha \vee \beta)) = \mathcal{heu}(\neg\alpha \wedge \neg\beta)$
7. $\mathcal{heu}(\alpha \rightarrow \beta) = \mathcal{heu}(\neg\alpha \vee \beta)$
8. $\mathcal{heu}(\neg(\alpha \rightarrow \beta)) = \mathcal{heu}(\alpha \wedge \neg\beta)$
9. $\mathcal{heu}(\alpha \leftrightarrow \beta) = \mathcal{heu}((\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta))$
10. $\mathcal{heu}(\neg(\alpha \leftrightarrow \beta)) = \mathcal{heu}(\neg(\alpha \wedge \beta) \wedge \neg(\neg\alpha \wedge \neg\beta))$ □

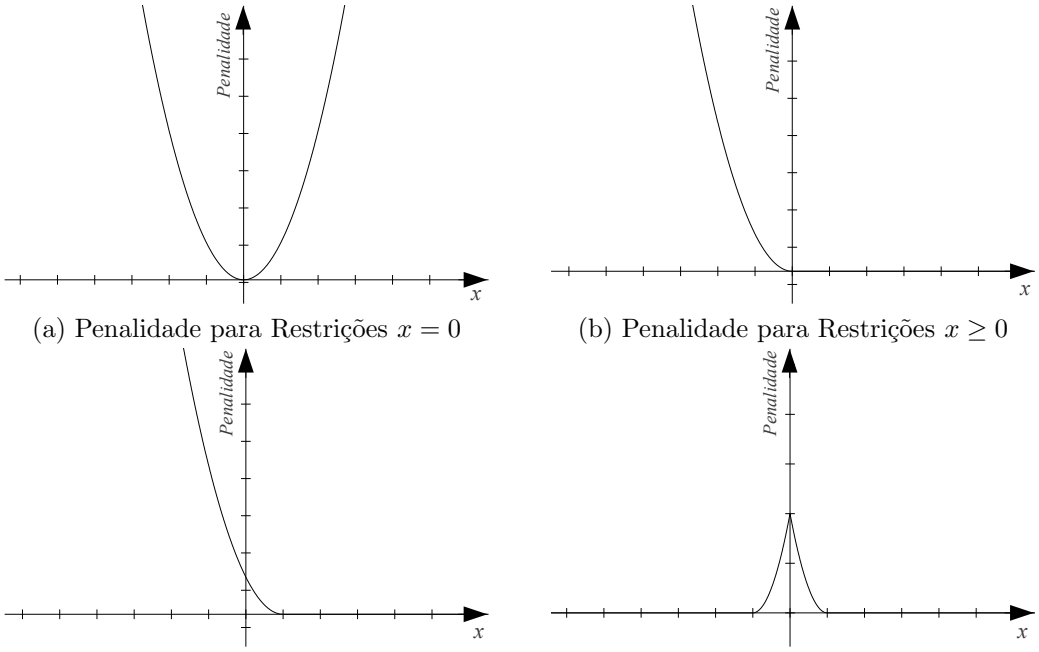
As transformações utilizadas nos itens 3 e 4 da definição 4.3 foram propostas e demonstradas em (GALLAGHER; NARASIMHAN, 1997). Os itens de 5 a 10 da mesma definição foram obtidos aplicando-se o teorema de DeMorgan (VIEIRA, 2006, p.7).

Em seguida, são apresentadas duas alternativas para o cálculo das funções de penalidade h_{eq}, h_{gt}, h_{ge} e h_{ne}. Primeiramente, é apresentada uma função de penalidade quadrática e, logo após, uma função de penalidade exponencial. Para facilitar o entendimento desta seção, na subseção 4.2.3, é apresentado um exemplo de derivação de uma FCBA a partir de uma propriedade descrita na LPE.

4.2.1 Função de Penalidade Quadrática

Nesta seção, é apresentada uma das possíveis formulações para as funções de penalidade h_{eq}, h_{ge}, h_{gt} e h_{ne} usando funções de penalidade quadráticas (NOCEDAL; WRIGHT, 1999). As equações 4.9, 4.10, 4.11 e 4.12[1] apresentam estas formulações. Nestas equações, $\mu > 0$ é um parâmetro de penalidade e $\epsilon > 0$ é uma constante pequena. Quanto mais próximo de 0 (zero) for o valor de μ, mais severa será a penalidade sobre as violações da restrição. A constante ϵ é utilizada para que a origem não seja penalizada

[1] Obtida a partir de $(x > y) \vee (x < y)$. (Nota do Autor).



(a) Penalidade para Restrições $x = 0$ (b) Penalidade para Restrições $x \geq 0$

(c) Penalidade para Restrições $x > 0$ (d) Penalidade para Restrições $x \neq 0$

Figura 4.3: Funções de Penalidade Quadráticas

nas restrições do tipo $x > y$ e $x < y$. Na prática, o valor de ϵ é definido como o menor valor positivo passível de ser representado no tipo de dados utilizado na restrição.

$$h_{eq}(x, y) = \frac{1}{2\mu}(x - y)^2 \tag{4.9}$$

$$h_{ge}(x, y) = \begin{cases} \frac{1}{2\mu}(x - y)^2 & , \text{ se } x < y \\ 0 & , \text{ caso contrário} \end{cases} \tag{4.10}$$

$$h_{gt}(x, y) = h_{ge}(x, y + \epsilon) \tag{4.11}$$

$$h_{ne}(x, y) = min(h_{gt}(x, y), h_{gt}(y, x)) \tag{4.12}$$

A figura 4.3 mostra as curvas das funções de penalidade para restrições do tipo $x = y$, $x \geq y$, $x > y$ e $x \neq y$, com $y = 0$, $\mu = 1/2$ e $\epsilon = 1$. É possível perceber que o valor da função diminui na medida em que a restrição está próxima de ser satisfeita, finalmente atingindo o valor 0 (zero) nos pontos onde a restrição não é mais violada[2].

4.2.2 Função de Penalidade Exponencial

Funções de penalidade do tipo exponencial são utilizadas em (GALLAGHER; NARASIMHAN, 1997), e suas fórmulas são apresentadas nas equações 4.13, 4.14, 4.15

[2]Em restrições do tipo $x > y$ e $x \neq y$, para que a restrição seja considerada satisfeita, deve-se observar o valor de ϵ. (Nota do Autor).

e 4.16. A variação do valor da penalidade imposta em função da proximidade entre ponto de avaliação e o ponto em que a restrição é satisfeita é maior neste tipo de função do que nas funções quadráticas. Apesar desta característica ser uma vantagem, quando se considera que pequenas variações em direção à satisfação da restrição resultam em grandes variações no valor da penalidade, esta maior variação pode resultar em situações de *overflow* com muito mais facilidade do que nas funções de penalidade quadráticas, apresentadas anteriormente. Por este motivo, as funções de penalidade quadrática são mais indicadas para serem utilizadas em conjunto com o método proposto neste trabalho e as funções de penalidade exponencial são descritas apenas por razões históricas, uma vez que o método aqui proposto inicialmente empregava este tipo de função.

Outra desvantagem das funções de penalidade exponencial é o número de parâmetros que devem ser definidos pelo usuário. Nas equações 4.13, 4.14, 4.15 e 4.16, $\kappa > 0$ é um parâmetro de penalidade que controla a penalidade imposta às violações das restrições, $\epsilon > 0$ é uma constante utilizada para que a origem não seja penalizada nas restrições do tipo $x \geq y$ e $x \leq y$, e $\tau > 0$ é uma constante de uniformidade empregada para manter a uniformidade de magnitude entre as restrições do tipo $x \neq y$ e as outras restrições.

$$h_{eq}(x, y) = e^{\kappa|x-y|} - 1 \tag{4.13}$$

$$h_{ge}(x, y) = e^{-\kappa(x-y+\epsilon)} \tag{4.14}$$

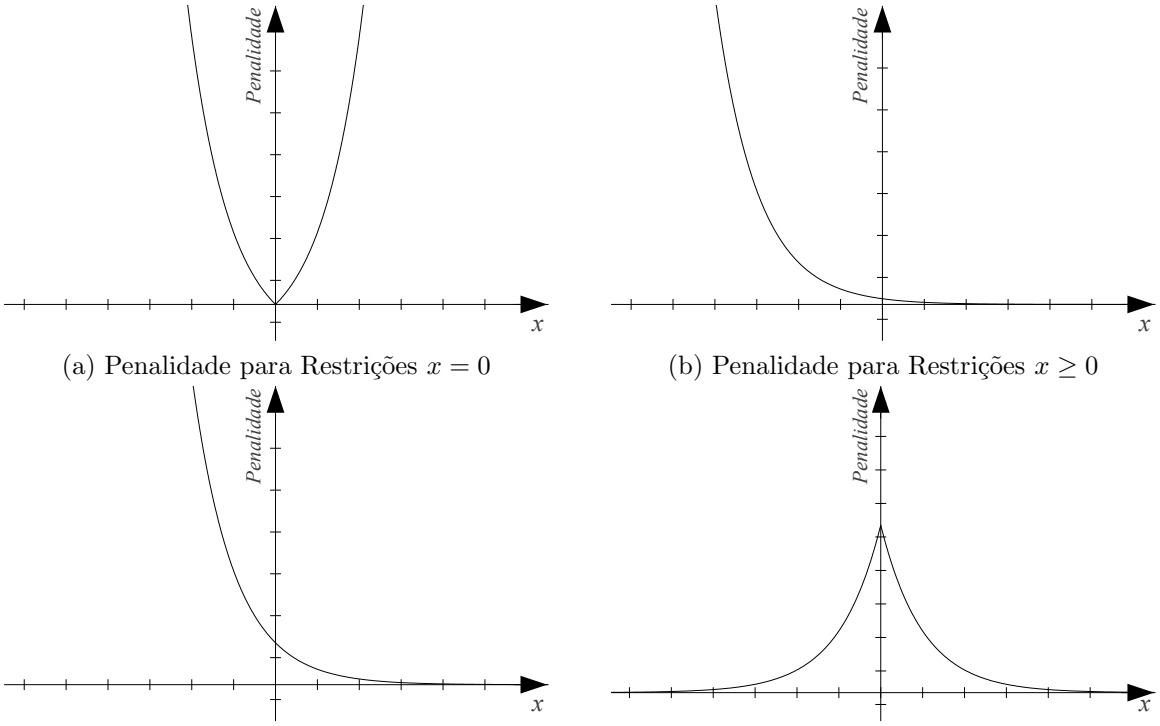
$$h_{gt}(x, y) = e^{-\kappa(x-y)} \tag{4.15}$$

$$h_{ne}(x, y) = \tau.e^{-\kappa|x-y|} \tag{4.16}$$

A figura 4.4 mostra as curvas para as funções de penalidade exponencial. Assim como nas funções quadráticas, é possível perceber que o valor da função diminui na medida em que a restrição está próxima de ser satisfeita, finalmente se aproximando de 0 (zero) nos pontos onde a restrição não é mais violada. Para a construção destas curvas, considerou-se $y = 0$, $\kappa = 1$, $\tau = 1$ e $\epsilon = 2$.

4.2.3 Exemplo de FCBA Heurística

Como exemplo ilustrativo, seja o sistema sem memória, S, cujo comportamento é descrito pela a função $g : \mathbb{F}_n \times \mathbb{F}_n \mapsto \mathbb{F}_n$, apresentada na equação 4.17, onde $\mathbb{F}_n \subset \mathbb{Q}$ é o



(a) Penalidade para Restrições $x = 0$

(b) Penalidade para Restrições $x \geq 0$

(c) Penalidade para Restrições $x > 0$

(d) Penalidade para Restrições $x \neq 0$

Figura 4.4: Funções de Penalidade Exponenciais

conjunto dos números passíveis de serem representados por uma notação ponto flutuante de n bits. Considere que este sistema seja fielmente modelado pela máquina de Mealy $M = (E, \Sigma, \Delta, \delta, \sigma, i)$, onde $\Sigma = \mathbb{F}_n \times \mathbb{F}_n$ e $\Delta = \mathbb{F}_n$. Um símbolo do alfabeto de entrada da máquina, $s \in \Sigma$, representa a dupla de sinais (s_1, s_2), onde s_1 e s_2 são sinais do tipo ponto flutuante[3], e um símbolo do alfabeto de saída de M, $d \in \Delta$, representa uma 1-upla (d_1) de um sinal também do tipo ponto flutuante.

$$g(s_1, s_2) = (s_1)^2 + 2s_1 + (s_2)^2 + 3s_2 + 4 \tag{4.17}$$

Considere a propriedade P, apresentada na equação 4.18, onde $s = (s_1, s_2)$, e uma asserção $\mathcal{A}_P$ que afirma que M possui a propriedade P. Analisando-se cuidadosamente esta asserção, chega-se à conclusão de que não se trata de uma asserção válida, pois existem símbolos do alfabeto de entrada para os quais $\mathcal{A}_P(s) = \text{F}$. Um destes contraexemplos é o símbolo $s = (1,38, -2,03)$, cujo símbolo de saída é $d = (6,6953)$.

$$P \equiv ((s_1 > 1,0) \rightarrow (d_1 > 7,0)) \tag{4.18}$$

[3]Por ter um número de bits finito para criar representações, o conjunto $\mathbb{F}_n$ é um conjunto finito. Se $\mathbb{F}_n$ é um conjunto finito, então também o é $\Sigma = \mathbb{F}_n \times \mathbb{F}_n$, não violando a definição 3.1. (Nota do Autor).

Para se determinar uma FCBA que permita avaliar os símbolos do alfabeto de entrada, Σ, em relação a sua proximidade aos contraexemplos da asserção $\mathcal{A}_P$, pode-se utilizar a interpretação semântica alternativa das fórmulas da LPE, apresentada na definição 4.3. Para isto, primeiramente nega-se a propriedade P de forma a se obter a fórmula da LPE que determina os contraexemplos. Então, aplica-se a interpretação semântica heu da definição 4.3, conforme apresentado a seguir, onde $\alpha = (s_1 > 1, 0)$ e $\beta = (d_1 > 7, 0)$:

$$\begin{aligned}
\mathit{heu}(\neg(\alpha \rightarrow \beta)) &= \mathit{heu}(\alpha \wedge \neg\beta) && \text{(por 4.3, item 8)} \\
&= \mathit{heu}(\alpha) + \mathit{heu}(\neg\beta) && \text{(definição 4.3, item 3)} \\
&= \mathit{heu}_{\mathcal{R}}(\alpha) + \mathit{heu}_{\mathcal{R}}(\neg\beta) && \text{(definição 4.3, itens 1 e 2)}
\end{aligned}$$

A partir deste resultado, aplica-se a interpretação semântica $\mathit{heu}_{\mathcal{R}}$, apresentada na definição 4.2. A interpretação semântica do termo $\mathit{heu}_{\mathcal{R}}(\alpha)$ é apresentada a seguir:

$$\begin{aligned}
\mathit{heu}_{\mathcal{R}}(s_1 > 1, 0) &= h_{gt}(\mathcal{E}(s_1),\ \mathcal{E}(1, 0)) && \text{(definição 4.2, item 2)} \\
&= h_{gt}(\mathcal{E}(s_1),\ 1, 0)
\end{aligned}$$

A função de avaliação das variáveis, $\mathcal{E}$, é empregada para se determinar o valor do sinal s_1. Como apresentado na subseção 3.1.6, um símbolo de entrada ou de saída do sistema define uma avaliação das variáveis e pode ser usado para avaliar o valor de s_1. Assim, $\mathcal{E}(s_1) = s(s_1)$.

A interpretação do segundo termo, $\mathit{heu}_{\mathcal{R}}(\neg\beta)$, é definida como:

$$\begin{aligned}
\mathit{heu}_{\mathcal{R}}(\neg(d_1 > 7, 0)) &= \mathit{heu}_{\mathcal{R}}(d_1 \leq 7, 0)) && \text{(definição 4.2, item 8)} \\
&= \mathit{heu}_{\mathcal{R}}(7, 0 \geq d_1)) && \text{(definição 4.2, item 6)} \\
&= h_{ge}(\mathcal{E}(7, 0), \mathcal{E}(d_1)) && \text{(definição 4.2, item 3)} \\
&= h_{ge}(7, 0,\ \mathcal{E}(d_1))
\end{aligned}$$

Assim, combinando-se as interpretações semânticas dos termos, obtém-se a FCBA heurística apresentada na equação 4.19. Verifica-se que, para calcular o valor da

função $f_{\mathcal{A}_P}(s)$, é preciso computar a saída da máquina de Mealy que modela o sistema. Este requisito é representado na equação 4.19 pelo termo d_1, que é obtido a partir da função de saída da máquina de Mealy, ou $\sigma(i, s)$. Para se calcular o valor da FCBA, pode-se empregar tanto as funções de penalidade quadráticas quanto as exponenciais, ou qualquer outro tipo de função de penalidade para os respectivos operadores relacionais.

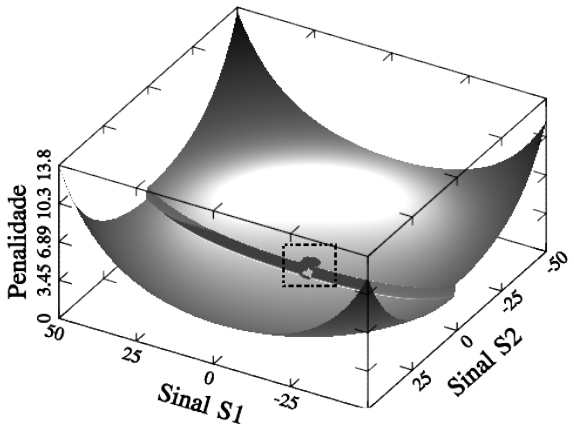
$$f_{\mathcal{A}_P}(s) = h_{gt}(s(s_1),\ 1, 0) + h_{ge}(7, 0,\ d(d_1)) \tag{4.19}$$

A figura 4.5 mostra a topografia da função apresentada na equação 4.19, criada utilizando as funções de penalidade exponenciais. Para se construir este gráfico, utilizou-se $\kappa = 5 \times 10^{-4}$ e $\epsilon = 0,1$, pois estes valores permitem uma boa visualização da topologia da função. Próximo ao centro da figura 4.5a, marcado pela caixa pontilhada, está a região que contém os contraexemplos da propriedade P. Os detalhes desta região podem ser verificados na figura 4.5b, que é uma ampliação da área que contém os mínimos globais da FCBA derivada.

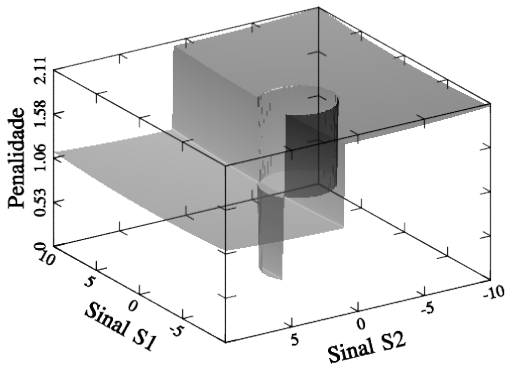
4.3 Método para a DPC Heurística de Sistemas sem Memória

O método proposto neste trabalho utiliza diretamente a FCBA heurística definida nas seções anteriores. Nesta seção, o algoritmo para a aplicação do método será apresentado em detalhes. Para isto, este foi dividido em pequenas unidades com o intuito de facilitar o entendimento. Primeiramente, considere o algoritmo da Listagem 4.1, que apresenta a função de busca por um candidato a contraexemplo da asserção sendo verificada.

A função *busca_candidato* recebe como entrada uma fórmula da LPE, $\overline{\varphi}$, já negada, um alfabeto de entrada, Σ, uma função de saída, σ, e o estado inicial, i, da máquina de Mealy que modela o sistema. Depois de realizar as computações, a função retorna um símbolo, s, do alfabeto de entrada que é um candidato a contraexemplo da asserção. Na linha 10 da Listagem 4.1, a FCBA é definida, utilizando-se a interpretação semântica alternativa ***heu*** apresentada na definição 4.3. Uma vez definida a FCBA, emprega-se um algoritmo de otimização numérica para encontrar um símbolo do alfabeto de entrada que



(a) Visão Geral



(b) Detalhe da Região Contendo os Contraexemplos

Figura 4.5: Topografia da FCBA Heurística do Exemplo

minimiza a FCBA. Este algoritmo é representado na Listagem 4.1 pela chamada à função *minimize*.

A função *minimize* utiliza a função de saída, σ, da máquina de Mealy que modela o sistema para determinar, de modo iterativo, qual o símbolo do alfabeto de entrada que minimiza a FCBA. O fluxo geral da função *minimize* é apresentado na figura 4.6[4]. A cada iteração do *Algoritmo de Otimização*, um *Símbolo de Entrada* é aplicado à *Função de Saída* da máquina que modela o sistema. O *Valor da Penalidade* para o *Símbolo de Entrada* aplicado é calculado pela FCBA, utilizando o próprio símbolo de entrada e o *Símbolo de Saída* determinado. O *Algoritmo de Otimização* utiliza este *Valor*

[4]O algoritmo da função *minimize* não foi apresentado, uma vez que a forma deste algoritmo pode variar consideravelmente dependendo do método de otimização empregado. Assim, preferiu-se apresentar um diagrama geral, apenas para facilitar o entendimento do processo. (Nota do Autor).

```
1 FUNCAO busca_candidato
2   ENTRADAS
3     Uma formula φ̄ da LPE representando a negacao da propriedade P
4     Um alfabeto de entrada Σ
5     Uma funcao de saida σ
6     Um estado inicial i
7   SAIDA
8     Um simbolo, s, de Σ
9 INICIO
10  f_A ← heu(φ̄)
11  s ← minimize(f_A,Σ,σ,i)
12  retorne s
13 FIM
```

Listagem 4.1: Algoritmo para a Busca por Candidato a Contraexemplo

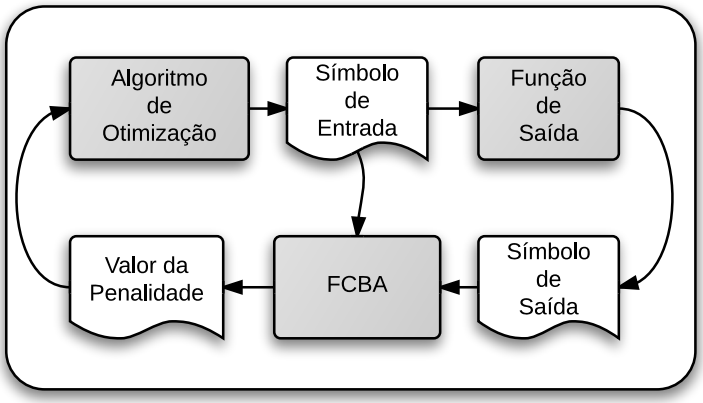


Figura 4.6: Fluxo Geral da Função Minimize

da Penalidade para determinar qual o próximo *Símbolo de Entrada* que será gerado. Este processo continua até que um *Símbolo de Entrada* que minimize o *Valor da Penalidade* seja determinado.

A Listagem 4.2 mostra o algoritmo para a DPC heurística de sistemas sem memória. Este algoritmo se inicia com a negação da fórmula da LPE que representa a propriedade P para se obter a fórmula geral dos contraexemplos. A função *busca_candidato*, mostrada na Listagem 4.1, é chamada para se determinar o símbolo s que é um candidato a contraexemplo. Uma vez determinado o símbolo candidato a contraexemplo, aplica-se o corolário 4.1 para determinar se a asserção sendo verificada é válida. Os assinalamentos para os sinais de entrada do sistema, determinados pelo símbolo s, são utilizados para se avaliar a fórmula da LPE, utilizando a semântica original, estabelecida na definição 3.34. Se a avaliação for igual a F, a asserção de P foi violada e o símbolo s é um contraexemplo. Caso contrário, a asserção de P é uma asserção válida.

```
1  FUNCAO dpc_heuristica_sem_memoria
2    ENTRADAS
3      Uma formula φ da LPE representando a propriedade P
4      Um alfabeto de entrada Σ
5      Uma funcao de saida σ
6      Um estado inicial i
7    SAIDA
8      Um contraexemplo ou sucesso
9  INICIO
10   φ̄ ← ¬φ
11   s ← busca_candidato(φ̄,Σ,σ,i)
12   se avalia(φ,s) = F entao
13     retorne s
14   senao
15     retorne sucesso
16   fim se
17 FIM
```

Listagem 4.2: Algoritmo para a DPC Heurística de Sistemas sem Memória

É possível transformar o algoritmo do método proposto para que ele possa ser executado por várias tarefas independentes. Esta versão multitarefa da DPC heurística é apresentada na Listagem 4.3. Para possibilitar a computação em paralelo, a fórmula da LPE que representa a propriedade é transformada para a forma normal disjuntiva (FND), pela função *fnd*. Por se tratar de uma disjunção de conjunções, qualquer conjunção que seja satisfeita, fará com que a fórmula na FND seja satisfeita. Assim, qualquer símbolo do alfabeto de entrada que satisfaça uma das conjunções da fórmula $\overline{\varphi}$, é um contraexemplo da asserção original. A busca por candidatos a contraexemplos, então, é executada sobre cada uma das conjunções da fórmula $\overline{\varphi}$. Um processo é disparado para cada uma destas buscas. Para cada processo que retornar um símbolo s, o comitê, referido como *decisor*, verificará se o símbolo encontrado é um contraexemplo. Caso seja, todos os processos podem ser abortados, pois a asserção não é válida. Caso todos os processos retornem sem que um contraexemplo seja encontrado, a asserção de P é válida.

4.4 Considerações sobre o Método para Sistemas sem Memória

Os resultados apresentados neste capítulo são importantes, do ponto de vista teórico. Foi definido um mapeamento entre o problema da busca por contraexemplos de

```
1 FUNCAO dpc_heuristica_sem_memoria_multitarefa
2   ENTRADAS
3     Uma formula φ da LPE representando a propriedade P
4     Um alfabeto de entrada Σ
5     Uma funcao de saida σ
6     Um estado inicial i
7   SAIDA
8     Um contraexemplo ou sucesso
9 INICIO
10  resultado ← sucesso
11
12  COMITE decisor
13    ENTRADAS
14      Um simbolo s de Σ
15  INICIO
16    se avalia(φ, s) = F entao
17      resultado ← s
18      aborte
19    fim se
20  FIM
21  φ̄ ← fnd(¬φ)
22  para cada conjuncao C em φ̄ faca
23    decisor(PROCESSO busca_candidato(C,Σ,σ,i))
24  fim para
25  sincronize
26  retorne resultado
27 FIM
```

Listagem 4.3: Algoritmo Multitarefa para a DPC Heurística de Sistemas sem Memória

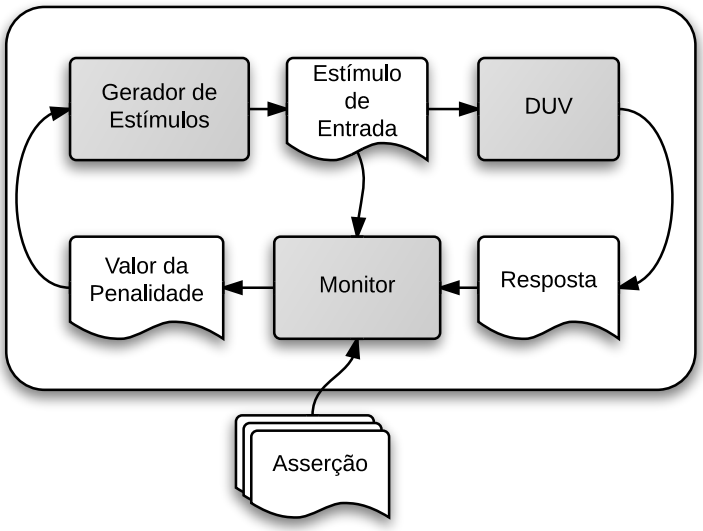


Figura 4.7: Fluxo Geral do Método Proposto

asserções e um problema de otimização, permitindo que algoritmos de otimização sejam empregados na validação de sistemas sem memória. Apesar de, neste mapeamento, o sistema ser modelado por uma máquina de Mealy, é importante ressaltar que a síntese propriamente dita desta máquina não é necessária, pois esta foi utilizada somente como suporte para as definições apresentadas.

O fluxo prático do método proposto é mostrado na figura 4.7, que é bem similar ao apresentado na figura 4.6. O funcionamento do método pode ser dividido entre dois blocos básicos: o *Gerador de Estímulos* e o *Monitor*. O *Gerador de Estímulos* determina qual estímulo deve ser aplicado ao DUV. Após o processamento deste estímulo, a resposta do DUV é analisada pelo *Monitor*, que calcula o valor da penalidade, utilizando a asserção para criar a FCBA. Assim, *Gerador de Estímulos* implementa o algoritmo de otimização, enquanto o DUV, juntamente com o *Monitor*, determina a função objetivo a ser minimizada.

Do ponto de vista prático, o método apresentado neste capítulo possui uma limitação: somente pode ser aplicado a sistemas sem memória. No próximo capítulo, um método para contornar esta limitação será proposto.

Capítulo 5

Mapeamento da Verificação Dinâmica de Propriedades para um Problema de Otimização em Sistemas com Memória

O método proposto no capítulo 4 pode ser aplicado somente a sistemas sem memória. Grande parte dos sistemas digitais, no entanto, se enquadra dentro do conceito de sistemas com memória, apresentado na definição 3.31. Para que o método proposto possa ser aplicado a esta classe de sistemas, a limitação do capítulo 4 deve ser contornada. Para isto, é preciso resolver dois problemas. O primeiro é permitir que o teorema 4.1 e o corolário 4.1 sejam empregados também em sistemas com memória e, desta forma, mapear o problema da busca por contraexemplos nestes sistemas para um problema de otimização. O segundo é desenvolver um procedimento que permita derivar uma FCBA a partir da fórmula da LTLE que descreve uma propriedade de um sistema com memória.

A estratégia adotada para resolver o primeiro problema foi demonstrar que, se o tamanho da palavra de entrada for fixo, é possível construir uma máquina de Mealy que representa um sistema sem memória a partir da qual pode-se computar a saída de uma máquina de Mealy que representa um sistema com memória. Com isto, é possível aplicar o teorema 4.1 e o corolário 4.1 em sistemas com memória, desde que o tamanho da palavra de entrada seja fixo.

Para o segundo problema, a abordagem utilizada foi transformar uma fórmula da LTLE em uma fórmula da LPE, fixando-se também o tamanho da palavra da estrutura temporal linear da definição 3.20. Com esta transformação, a semântica alternativa da definição 4.3 pode ser empregada em fórmulas da LTLE, desde que o número de intervalos

de tempo seja fixo. Em (BIERE et al., 1999), é apresentado um método para transformar fórmulas da LTL definidas sobre um intervalo de tempo limitado em fórmulas da LP. Como as fórmulas da LTLE e da LPE são extensões da LTL e da LP, respectivamente, a transformação apresentada por Biere et al. (1999) pode ser utilizada como base para a definição da transformação das fórmulas da LTLE e da LPE. Esta transformação das fórmulas da LTLE para fórmulas da LPE é proposta neste capítulo.

Na próxima seção, será apresentado o conceito de máquina de Melay N-equivalente, que mostra que o teorema 4.1 e seu corolário podem ser aplicados a sistemas com memória cujo tamanho da palavra tenha sido fixado. Em seguida, na seção 5.2, será apresentado o desdobramento da fórmula da LTLE em uma fórmula da LPE. Uma generalização do método apresentado no capítulo 4, empregando as transformações deduzidas neste capítulo, será proposta na seção 5.3.

5.1 Equivalência entre Sistema sem Memória e Sistema com Memória em Linguagens com Palavras de Tamanho Fixo

Seja S_1 um sistema com memória, conforme a definição 3.31. O teorema 4.1, proposto no capítulo 4, não pode ser aplicado diretamente ao sistema S_1, pois este considera que o sistema não possui memória. A estratégia adotada para contornar esta limitação é definir um tamanho fixo, N, para a palavra de entrada (e, consequentemente para a palavra de saída). Com esta premissa, é possível construir um sistema sem memória, S_2, que modele o funcionamento de S_1 para todas as palavras de tamanho N, permitindo a aplicação do teorema 4.1 para este subconjunto da linguagem de entrada, composto das palavras de tamanho N.

A definição 5.1 apresenta o conceito de máquina de Mealy N-equivalente que captura de forma parcial o comportamento de uma máquina de Mealy que representa um sistema com memória, em uma máquina de Mealy que representa um sistema sem memória. Este conceito será empregado para demonstrar que o teorema 4.1 pode ser aplicado aos sistemas com memória, dadas algumas restrições.

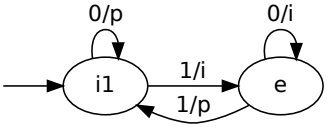


Figura 5.1: Máquina M_1 para Exemplo da Operação de Desdobramento

Definição 5.1 (Máquina de Mealy *N*-equivalente). *Seja a máquina de Mealy* $M_1 = (E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$. *A máquina de Mealy N-equivalente a* M_1 *é a máquina de Mealy* $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$, *em que:*

- $E_2 = \{i_2\}$
- $\Sigma_2 = \prod^{N} \Sigma_1$
- $\Delta_2 = \prod^{N} \Delta_1$
- $\delta_2(i_2, a) = i_2, \ \forall a \in \Sigma_2$
- $\sigma_2(i_2, a) = \mathcal{D}(\hat{\sigma}_1(\mathcal{S}^{-1}(a))), \ \forall a \in \Sigma_2$

onde $\mathcal{S} : \Sigma_1^N \mapsto \Sigma_2$ *é uma função bijetora, denominada de função de mapeamento palavra-símbolo, que mapeia as palavras de tamanho N criadas sobre o alfabeto de entrada da máquina* M_1 *para os símbolos do alfabeto de entrada da máquina* M_2, $\mathcal{D} : \Delta_1^N \mapsto \Delta_2$ *é também uma função bijetora de mapeamento palavra-símbolo, que mapeia as palavras de tamanho N criadas sobre o alfabeto de saída da máquina* M_1 *para os símbolos do alfabeto de saída da máquina* M_2, Σ_1^N *é o conjunto formado por N concatenações do conjunto* Σ_1, Δ_1^N *é o conjunto formado por N concatenações do conjunto* Δ_1, *e para qualquer função bijetora* f, f^{-1} *representa a função inversa de* f. □

Uma máquina de Mealy M_2 N-equivalente à máquina de Mealy M_1 é uma máquina de Mealy que representa um sistema combinatório que captura o funcionamento de M_1 por um número de ciclos definido, N. Isto é, a máquina M_2 simula o funcionamento de M_1 por um número fixo, N, de ciclos.

Como exemplo, considere a máquina M_1, apresentada na figura 5.1. Esta máquina de Mealy emite um símbolo p sempre que a palavra de entrada, até o momento, apresente um número par de símbolos 1, e emite um símbolo i caso contrário.

Tabela 5.1: Exemplo de Mapeamento Palavra-Símbolo da Entrada

$\mathcal{S}$	
Σ_1^3	Σ_2
000	a
001	b
010	c
011	d
100	e
101	f
110	d
111	h

Tabela 5.2: Exemplo de Mapeamento Palavra-Símbolo da Saída

$\mathcal{D}$	
Δ_1^3	Δ_2
ppp	s
ppi	t
pip	u
pii	v
ipp	w
ipi	x
iip	y
iii	z

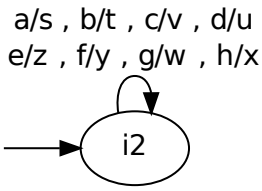


Figura 5.2: Máquina M_1 para Exemplo da Operação de Desdobramento

Deseja-se obter a máquina M_2 que seja 3-equivalente à M_1, ou $M_1 \overset{3}{\rightsquigarrow} M_2$. As tabelas 5.1 e 5.2 apresentam as funções de mapeamento palavra-símbolo das palavras da máquina M_1 para o alfabeto de entrada ($\mathcal{S} : \Sigma_1^3 \mapsto \Sigma_2$) e para o alfabeto de saída ($\mathcal{D} : \Delta_1^3 \mapsto \Delta_2$) da máquina M_2. Na primeira coluna destas tabelas está a palavra formada a partir da concatenação de três símbolos do alfabeto da máquina M_1, e na segunda coluna está o símbolo correspondente no alfabeto da máquina M_2. Utilizando a a definição 5.1 e as tabelas 5.1 e 5.2 para construir a função de saída (σ_2), obtém-se a máquina M_2 3-equivalente à máquina M_1 da figura 5.1. Esta máquina está representada na figura 5.2, onde a seta indica as oito transições do estado i_2 para o próprio estado i_2.

Proposição 5.1 (Memória da Máquina N-Equivalente). *A máquina Mealy $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$ N-equivalente a máquina de Mealy $M_1 = (E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$, ou $M_1 \overset{N}{\rightsquigarrow} M_2$, representa um sistema sem memória.*

Demonstração. Pela definição, se $M_1 \overset{N}{\rightsquigarrow} M_2$, então $E_2 = \{i_2\}$ e $\delta_2(i_2, a) = i_2$ para todo a do alfabeto de entrada Σ_2. Assim, a definição 5.1 estabelece que a máquina M_2 possui

apenas um estado, i_2. Portanto, provando-se que uma máquina de Mealy com apenas um estado sempre modela um sistema sem memória, prova-se a proposição.

Usando a definição 3.15 da página 51, é facilmente verificado que, para qualquer máquina de Mealy $M = (\{i\}, \Sigma, \Delta, \delta, \sigma, i)$ com apenas um estado:

$$\hat{\delta}(i, w_1) = \hat{\delta}(i, w_2) = i, \ \forall w_1, w_2 \in \Sigma^* \tag{5.1}$$

Para que o sistema modelado pela máquina de Mealy M seja considerado sem memória, pela definição 3.30:

$$\forall (w_1, w_2 \in \Sigma^* \text{ e } a \in \Sigma) : \ \sigma(\hat{\delta}(i, w_1), a) = \sigma(\hat{\delta}(i, w_2), a) \tag{5.2}$$

Substituindo-se (5.1) em (5.2), obtém-se:

$$\forall a \in \Sigma : \ \sigma(i, a) = \sigma(i, a) \tag{5.3}$$

A equação 5.3 é verdadeira por definição. Portanto qualquer máquina de Mealy M que possua apenas um estado, representa um sistema sem memória. Uma vez que se $M_1 \overset{N}{\rightsquigarrow} M_2$, então M_2 possui apenas um estado, consequentemente M_2 representa um sistema sem memória. □

A partir da proposição 5.1, verifica-se que o teorema 4.1 e o corolário 4.1 se aplicam a qualquer máquina de Mealy N-equivalente. É possível provar que para duas máquinas de Mealy $M_1(E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$ e $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$, se $M_1 \overset{N}{\rightsquigarrow} M_2$, então:

1. Dada uma linguagem L_1, construída sobre o alfabeto Σ_1, e cujas palavras têm tamanho N, é possível simular o comportamento de M_1 para as palavras de L_1 utilizando-se a máquina M_2;

2. Dada uma linguagem L_2, construída sobre o alfabeto Σ_2, é possível simular o comportamento de M_2 para as palavras de L_2 utilizando-se a máquina M_1;

3. Se as duas afirmações anteriores forem verdadeiras, então o teorema 4.1 e o corolário 4.1 se aplicam a sistemas com memória, dadas as condições de contorno apresentadas.

A prova para o item 1 é oferecida no lema 5.1 e a demonstração do item 2 é apresentada pelo lema 5.2. O item 3 é provado no teorema 5.1.

Lema 5.1 (Simulação da Máquina M a partir da Máquina N-Equivalente). *Sejam duas máquinas de Mealy $M_1 = (E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$ e $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$, tais que $M_1 \overset{N}{\rightsquigarrow} M_2$, onde $N \in \mathbb{N}$. Seja a linguagem formal $L_1 = \Sigma_1^N$ construída a partir da concatenação de N elementos de Σ_1. A saída computada pela máquina M_1 para a linguagem L_1 pode ser computada a partir da saída computada pela máquina M_2 para a linguagem $L_2 = \Sigma_2$.*

Demonstração. Usando a definição 5.1, a saída da máquina de Mealy M_1 para a palavra $w \in L_1$ pode ser computada usando a função de saída da máquina M_2 da seguinte forma:

$$\hat{\sigma_1}(i_1, w) = \mathcal{D}^{-1}(\sigma_2(i_2, \mathcal{S}(w)))$$

Portanto é possível computar a saída da máquina M_1 para a linguagem L_1 a partir da saída computada pela máquina M_2 para a linguagem L_2. □

Lema 5.2 (Simulação da Máquina N-Equivalente a partir da Máquina M). *Sejam duas máquinas de estados finitos $M_1 = (E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$ e $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$, tais que $M_1 \overset{N}{\rightsquigarrow} M_2$, onde $N \in \mathbb{N}$. Seja a linguagem formal $L_1 = \Sigma_1^N$ construída a partir da concatenação de N elementos de Σ_1. A saída computada pela máquina M_2 para a linguagem $L_2 = \Sigma_2^*$ pode ser computada a partir da saída computada pela máquina M_1 para a linguagem L_1.*

Demonstração. Usando a definição 5.1 e a definição 3.13, a saída da máquina de Mealy M_2 para a palavra $w \in L_2$ pode ser computada usando a função de saída estendida da máquina M_1, recursivamente, como:

- $\hat{\sigma_2}(i_2, \lambda) = \lambda$
- $\hat{\sigma_2}(i_2, aw) = \mathcal{D}(\hat{\sigma_1}(i_1, \mathcal{S}^{-1}(a)))\hat{\sigma_2}(i_2, w)$, onde $a \in \Sigma_2$ e $w \in \Sigma_2^*$

Portanto é possível computar a saída da máquina M_2 para a linguagem L_2 a partir da saída computada pela máquina M_1 para a linguagem L_1. □

Teorema 5.1 (Mapeamento Validação-Otimização para Sistemas com Memória). *O mapeamento Validação-Otimização, apresentado no Teorema 4.1, se aplica aos sistemas com memória para linguagens de entrada cujas palavras possuam um tamanho N, onde $N \in \mathbb{N}$.*

Demonstração. Sejam duas máquinas de estados finitos $M_1 = (E_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, i_1)$ e $M_2 = (E_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, i_2)$, tais que $M_1 \overset{N}{\rightsquigarrow} M_2$, onde $N \in \mathbb{N}$. Considere as seguintes afirmações, representadas pelas variáveis da LP:

- **a** = O teorema 4.1 se aplica à máquina M_2;
- **b** = O teorema 4.1 se aplica à máquina M_1;
- **c** = A máquina M_2 representa um sistema sem memória;
- **d** = $M_1 \overset{N}{\rightsquigarrow} M_2$;
- **e** = A máquina M_2 simula a máquina M_1;
- **f** = As palavras da linguagem de entrada possuem tamanho N;
- **g** = A máquina M_2 pode substituir a máquina M_1.

O teorema 5.1 será provado em seis passos. Para os passos 4 e 5, assume-se que, se um sistema S_1 apresenta o mesmo comportamento do sistema S_2, então para validar o sistema S_2, pode-se utilizar o sistema S_1. Este princípio é o mesmo utilizado nas técnicas de verificação por equivalência e em alguns tipos de testes de programas (MILNER, 1971). Esta premissa é referida como premissa φ.

PASSO 1: A máquina M_2 representa um sistema sem memória.

$$\frac{\begin{array}{ll}\mathbf{d} \rightarrow \mathbf{c} & \text{(proposição 5.1)} \\ \mathbf{d} & \text{(premissa da prova)}\end{array}}{\mathbf{c}}$$

PASSO 2: O teorema 4.1 se aplica à máquina M_2.

$$\frac{\begin{array}{ll}\mathbf{c} \rightarrow \mathbf{a} & \text{(premissa do teorema 4.1)} \\ \mathbf{c} & \text{(Passo 1)}\end{array}}{\mathbf{a}}$$

PASSO 3: A máquina M_2 simula a máquina M_1.

$$\frac{\begin{array}{ll}\mathbf{f} \rightarrow \mathbf{e} & \text{(Lema 5.1)} \\ \mathbf{f} & \text{(premissa do teorema 5.1)}\end{array}}{\mathbf{e}}$$

PASSO 4: A máquina M_2 pode substituir a máquina M_1.

$$\frac{\begin{array}{ll}\mathbf{e} \rightarrow \mathbf{g} & \text{(premissa } \varphi\text{)} \\ \mathbf{e} & \text{(Passo 3)}\end{array}}{\mathbf{g}}$$

PASSO 5: Se o teorema 4.1 se aplica à máquina M_2, então o teorema 4.1 se aplica à máquina M_1.

$$\frac{\begin{array}{ll}\mathbf{g} \rightarrow (\mathbf{a} \rightarrow \mathbf{b}) & \text{(premissa } \varphi\text{)} \\ \mathbf{g} & \text{(Passo 4)}\end{array}}{\mathbf{a} \rightarrow \mathbf{b}}$$

PASSO 6: O teorema 4.1 se aplica à máquina M_1.

$$\frac{\begin{array}{ll}\mathbf{a} \rightarrow \mathbf{b} & \text{(Passo 5)} \\ \mathbf{a} & \text{(Passo 2)}\end{array}}{\mathbf{b}}$$

Portanto, para linguagens de entrada cujas palavras possuam um tamanho N, onde $N \in \mathbb{N}$, o mapeamento Validação-Otimização, apresentado no Teorema 4.1, se aplica aos sistemas com memória. □

O desdobramento da máquina M_1 em uma máquina N-equivalente M_2 não é apenas um artifício matemático. Essencialmente, a função de saída da máquina M_2 é obtida a partir da simulação da máquina M_1. Sendo assim, é possível definir um algoritmo para computar a máquina M_2 a partir da máquina M_1. Este algoritmo é apresentado na Listagem 5.1.

Uma análise do algoritmo revela que, para a computação da máquina de Mealy N-equivalente, é preciso executar exaustivamente a máquina de Mealy original para todas as palavras de tamanho N, o que é inviável na prática. Felizmente, como é mostrado na

```
1  FUNCAO N-equivalente
2    ENTRADAS
3      Uma maquina de Mealy M_1 = (E_1, Σ_1, Δ_1, δ_1, σ_1, i_1)
4      Um inteiro N representando o numero de passos do desdobramento
5    SAIDA
6      Uma maquina de Mealy M_2 = (E_2, Σ_2, Δ_2, δ_2, σ_2, i_2) N-equivalente a M_1
7  INICIO
8    E_2 ← {i_2}
9    Σ_2 ← ∏_N Σ_1
10   Δ_2 ← ∏ Δ_1
11   para cada N-upla a = (a_1, a_2, ..., a_N) ∈ Σ_2 faca
12     δ_2(i_2, a) ← i_2
13     d ← uma nova N-upla (d_1, d_2, ..., d_N)
14     e ← i_1
15     para i de 0 ate N − 1 faca
16       d_i ← σ_1(e, a_i)
17       e ← δ_1(e, a_i)
18     fim para
19     σ_2(i_2, a) ← d
20   fim para
21   retorne M_2
22 FIM
```

Listagem 5.1: Algoritmo para Computar Máquina N-equivalente

seção 5.3, não é necessário computar a máquina completamente para se aplicar o método proposto neste trabalho em sistemas com memória. Somente as linhas entre 14 e 19 do algoritmo da Listagem 5.1 são utilizadas para a aplicação do método.

5.2 Desdobramento da Propriedade ao Longo do Tempo

Para que seja possível derivar uma FCBA a partir de uma fórmula da LTLE, foi adotado um procedimento semelhante ao apresentado em (BIERE et al., 1999), onde uma fórmula da LTL é transformada em uma fórmula da LP. Nesta seção, é apresentado um procedimento para desdobrar uma fórmula da LTLE em uma fórmula da LPE, permitindo a aplicação da semântica alternativa apresentada na definição 4.3. Com isto, é possível derivar uma FCBA que pode ser utilizada para avaliar palavras de entrada, que tenham um tamanho máximo N, em relação a sua proximidade aos contraexemplos da asserção da propriedade sendo validada, quando esta propriedade está escrita na LTLE.

Na definição 5.2, considere que a operação de desdobramento sobre uma fórmula, α, da LTLE é representada pelo operador $\|\alpha\|_k^i$, que simboliza que a fórmula α deve ser desdobrada entre os instantes de tempo i e k.

Definição 5.2 (Desdobramento da Fórmula da LTLE). *Sejam γ e η fórmulas atômicas da LTLE, e α e β fórmulas da LTLE. O desdobramento de uma fórmula da LTLE em uma fórmula da LPE é definido recursivamente como se segue:*

1. $\|\mathrm{V}\|_k^i = \mathrm{V}$
2. $\|(\gamma > \eta)\|_k^i = (\gamma[i] > \eta[i])$
3. $\|(\gamma \geq \eta)\|_k^i = (\gamma[i] \geq \eta[i])$
4. $\|(\gamma = \eta)\|_k^i = (\gamma[i] = \eta[i])$
5. $\|(\gamma \leq \eta)\|_k^i = (\gamma[i] \leq \eta[i])$
6. $\|(\gamma < \eta)\|_k^i = (\gamma[i] < \eta[i])$
7. $\|(\gamma \neq \eta)\|_k^i = (\gamma[i] \neq \eta[i])$
8. $\|(\neg\alpha)\|_k^i = \neg\|(\alpha)\|_k^i$
9. $\|(\alpha \wedge \beta)\|_k^i = \|(\alpha)\|_k^i \wedge \|(\beta)\|_k^i$
10. $\|(\alpha \vee \beta)\|_k^i = \|(\alpha)\|_k^i \vee \|(\beta)\|_k^i$
11. $\|(\alpha \rightarrow \beta)\|_k^i = \|(\alpha)\|_k^i \rightarrow \|(\beta)\|_k^i$
12. $\|(\alpha \leftrightarrow \beta)\|_k^i = \|(\alpha)\|_k^i \leftrightarrow \|(\beta)\|_k^i$
13. $\|(\mathbf{X}\ \alpha)\|_k^i = \begin{cases} \|(\alpha)\|_k^{i+1} & , \text{ se } i < k \\ \mathrm{F} & , \text{ caso contrário} \end{cases}$
14. $\|(\alpha\ \mathbf{U}\ \beta)\|_k^i = \bigvee_{j=i}^{k} \left(\|\beta\|_k^j \ \wedge\ \bigwedge_{n=i}^{j-1} \|\alpha\|_k^n \right)$
15. $\|(\mathbf{F}\ \alpha)\|_k^i = \|(\mathrm{V}\ \mathbf{U}\ \alpha)\|_k^i$
16. $\|(\mathbf{G}\ \alpha)\|_k^i = \|(\neg(\mathbf{F}\ (\neg\alpha)))\|_k^i$
17. $\|(\alpha\ \mathbf{W}\ \beta)\|_k^i = \|((\alpha\ \mathbf{U}\ \beta) \vee (\mathbf{G}\ \alpha))\|_k^i$ □

Aplicando-se o desdobramento para as formulas da LTLE, apresentado na definição 5.2, obtém-se uma formula na LPE, onde $\gamma[i]$ e $\eta[i]$ são utilizados, respectivamente, para referenciar os valores das variáveis γ e η no instante de tempo i.

Como exemplo, considere a propriedade temporal, P, sobre as variáveis γ e η, representada na equação 5.4.

$$P \equiv (\mathbf{G}((\gamma > 3) \rightarrow (\eta \geq 5))) \tag{5.4}$$

O operador temporal **G** indica que a fórmula entre parêntesis deve ser válida para todos os instantes de tempo. Deseja-se desdobrar esta propriedade entre os instantes de tempo $i = 0$ e $k = 2$. Para isto, aplica-se a definição 5.2, conforme a seguir, onde $\alpha = (\gamma > 3) \rightarrow (\eta \geq 5)$:

$$\begin{aligned}
\|\mathbf{G}(\alpha)\|_2^0 &= \|(\neg(\mathbf{F}(\neg\alpha)))\|_2^0 && \text{(definição 5.2, item 16)}\\
&= \neg\|(\mathbf{F}(\neg\alpha))\|_2^0 && \text{(definição 5.2, item 8)}\\
&= \neg\|(\mathrm{V}\ \mathbf{U}\ (\neg\alpha))\|_2^0 && \text{(definição 5.2, item 15)}\\
&= \neg\left(\bigvee_{j=0}^{2}\left(\|(\neg\alpha)\|_2^j \wedge \bigwedge_{n=i}^{j-1}\|\mathrm{V}\|_2^n\right)\right) && \text{(definição 5.2, item 14)}\\
&= \neg\left(\bigvee_{j=0}^{2}(\|(\neg\alpha)\|_2^j)\right) && \text{(identidade multiplicativa)}\\
&= \neg\left(\bigvee_{j=0}^{2}(\neg\|\alpha\|_2^j)\right) && \text{(definição 5.2, item 8)}\\
&= \bigwedge_{j=0}^{2}(\|\alpha\|_2^j) && \text{teorema de DeMorgan}\\
&= \bigwedge_{j=0}^{2}((\gamma[j] > 3) \rightarrow (\eta[j] \geq 5))
\end{aligned}$$

A fórmula $\bigwedge_{j=0}^{2}((\gamma[j] > 3) \rightarrow (\eta[j] \geq 5))$ é uma fórmula da LPE, conforme a sintaxe estabelecida na definição 3.32. Assim, a semântica alternativa da definição 4.3 pode ser aplicada para se derivar uma FCBA para esta fórmula.

```
1  FUNCAO computa_saida
2    ENTRADAS
3      Um simbolo s[] do alfabeto de entrada
4      A funcao de saida σ da maquina original
5      A funcao de transferencia δ da maquina original
6      O estado inicial i da maquina original
7      Um inteiro N representando o numero de instantes de tempo
8    SAIDA
9      Um simbolo d[] do alfabeto de saida
10 INICIO
11   e ← i
12   para k de 0 ate N − 1 faca
13     d[k] ← σ(e, s[k])
14     e ← δ(e, s[k])
15   fim para
16   retorne d[]
17 FIM
```

Listagem 5.2: Algoritmo para Computar a Saída do Sistema N-equivalente

5.3 Generalização do Método para Sistemas com Memória

Nesta seção, é definido o método de DPC heurística para sistemas com memória. Os algoritmos apresentados na seção 4.3 são adaptados para lidar com sistemas com memória, aplicando-se a transformação da máquina M_1 original em uma máquina M_2 N-equivalente a M_1. Em vez de aplicar o algoritmo para a transformação completa da máquina, apresentado na Listagem 5.1, a transformação somente é realizada para os símbolos utilizados durante a busca. O algoritmo para computar a saída do sistema sem memória equivalente ao sistema com memória é apresentado na Listagem 5.2.

Na função *computa_saida*, o símbolo do alfabeto de entrada é tratado como um vetor, $s[i]$, onde cada índice i representa um instante de tempo. Essencialmente, os símbolos são considerados como estruturas temporais lineares, conforme a definição 3.20. O algoritmo computa a saída da máquina original, um símbolo de cada vez, a partir do estado inicial i. A cada símbolo, o estado da máquina é atualizado e o vetor que representa o símbolo de saída, $d[]$, da máquina de Mealy sem memória equivalente é atualizado. Por fim, o símbolo de saída da máquina sem memória equivalente é retornado.

Da mesma forma que na versão para sistemas sem memória, a função *busca_candidato_com_memoria* recebe como entrada uma fórmula da LPE, $\overline{\varphi}$, já negada, um alfabeto de entrada, Σ, uma função de saída, σ, e o estado inicial, i, da máquina de

```
1  FUNCAO busca_candidato_com_memoria
2    ENTRADAS
3      Uma formula φ̄ da LPE representando a negacao da propriedade P
4      Um alfabeto de entrada Σ
5      Uma funcao de saida σ
6      Um estado inicial i
7      Um inteiro N representando o numero de instantes de tempo
8    SAIDA
9      Um simbolo, s, de Σ
10 INICIO
11   f_A ← heu(φ̄)
12   s[]← minimize_com_memoria(f,Σ,σ,i,N)
13   retorne s
14 FIM
```

Listagem 5.3: Algoritmo para a Busca por Candidato a Contraexemplo em Sistemas com Memória

Mealy que modela o sistema com memória original. Um parâmetro adicional, no entanto, é incluído nas entradas da função. Este parâmetro, N, representa o número de instantes de tempo nos quais a busca deve ser efetuada. Depois de realizar as computações, a função retorna um símbolo, s, do alfabeto de entrada que é um candidato a contraexemplo da asserção. Na linha 11 da Listagem 5.3, a FCBA é definida, utilizando-se a interpretação semântica alternativa heu apresentada na definição 4.3. Uma vez definida a FCBA, emprega-se um algoritmo de otimização numérica para encontrar um símbolo do alfabeto de entrada que minimiza a FCBA. Este algoritmo é representado na Listagem 5.3 pela chamada à função *minimize_com_memoria*.

A função *minimize_com_memoria* utiliza a função *computa_saida*, da Listagem 5.2, para determinar, de modo iterativo, qual o símbolo do alfabeto de entrada que minimiza a FCBA. O fluxo geral da função *minimize_com_memoria* é apresentado na figura 5.3. A cada iteração do *Algoritmo de Otimização*, um *Vetor de Símbolos de Entradas* é aplicado à função *Computa Saída* da máquina N-equivalente. Para o *Algoritmo de Otimização*, o *Vetor de Símbolos de Entrada* é tratado como um único símbolo, composto de N vezes mais sinais do que o símbolo da máquina original. O *Valor da Penalidade* para o *Vetor de Símbolos de Entrada* aplicado é calculado pela FCBA, utilizando o próprio vetor de símbolos de entrada e o *Vetor de Símbolos de Saída* determinado. O *Algoritmo de Otimização* utiliza este *Valor da Penalidade* para determinar qual o próximo *Vetor de Símbolos de Entrada* que será gerado. Este processo continua até que um *Vetor de Símbolos de Entrada* que minimize o *Valor da Penalidade* seja determinado.

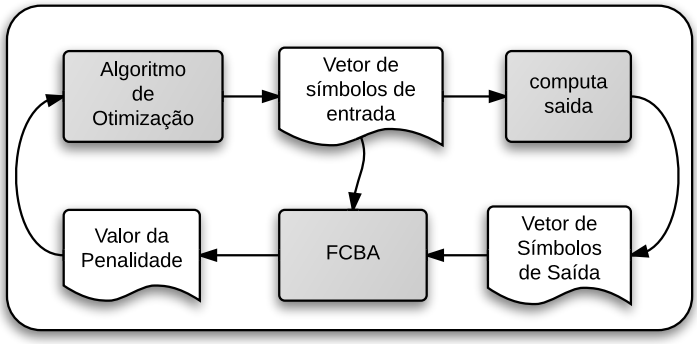


Figura 5.3: Fluxo Geral da Função Minimize para Sistemas com Memória

A Listagem 5.4 mostra o algoritmo para a DPC heurística de sistemas com memória. O algoritmo valida o sistema para cada palavra de tamanho $k+1$, com k variando de 0 a $N-1$. O laço se inicia com a negação da fórmula da LTLE que representa a propriedade P para se obter a fórmula geral dos contraexemplos. A fórmula da LTLE negada é desdobrada em uma fórmula da LPE, utilizando-se a definição 5.2. A função *busca_candidato_com_memoria*, mostrada na Listagem 5.3, é chamada para se determinar o símbolo $s[\,]$ que é um candidato a contraexemplo. Uma vez determinado o símbolo candidato a contraexemplo, aplica-se o corolário 4.1 para determinar se a asserção sendo verificada é válida. Os assinalamentos para os sinais de entrada do sistema, determinados pelo símbolo $s[\,]$, são utilizados para se avaliar a fórmula da LPE, utilizando a semântica original, estabelecida na definição 3.34. Se a avaliação for igual a F, a asserção de P foi violada e o símbolo $s[\,]$ é um contraexemplo. Caso contrário, o tamanho da palavra é incrementado e a busca reinicia. Caso a palavra tenha atingido o tamanho máximo N e nenhum contraexemplo tenha sido encontrado, a asserção de P é uma asserção válida para palavras de tamanho N.

5.4 Resumo do Capítulo

No capítulo 4, foi apresentado um método para transformar o problema da busca por contraexemplos em sistemas descritos em alto nível de abstração para um problema de otimização. O método proposto naquele capítulo, no entanto, é aplicável somente em sistemas sem memória, estando restrito a um pequeno conjunto de sistemas. Neste capítulo, o método apresentado no capítulo 4 foi expandido para lidar com sistemas com memória. A estratégia adotada foi transformar o sistema com memória em um

```
1  FUNCAO dpc_heuristica_com_memoria
2    ENTRADAS
3      Uma formula φ da LTLE representando a propriedade P
4      Um alfabeto de entrada Σ
5      Uma funcao de saida σ
6      Um estado inicial i
7      Um inteiro N representando o numero de instantes de tempo
8    SAIDA
9      Um contraexemplo s[] ou sucesso
10 INICIO
11   para k de 0 a N − 1
12     φ̄ ← ‖¬φ‖_k^0
13     s[]← busca_candidato_com_memoria(φ̄, Σ, σ, i, k + 1)
14     se avalia(φ, s[]) = F entao
15       retorne s[]
16     fim se
17   fim para
18   retorne sucesso
19 FIM
```

Listagem 5.4: Algoritmo para a DPC Heurística de Sistemas com Memória

sistema sem memória que captura o funcionamento do sistema com memória sob verificação por um número fixo de ciclos. Desdobrando-se o sistema original em um sistema N-equivalente, foi provado que é possível aplicar o método do capítulo 4 em sistemas com memória, desde que seja considerado um número fixo, N, de instantes de tempo.

No próximo capítulo, é apresentada uma ferramenta que implementa os algoritmos projetados neste capítulo e no capítulo 4. A aplicação desta ferramenta em exemplos ilustrativos e em um estudo de caso é descrita como uma prova de conceito do método de DPC heurística proposto neste trabalho.

Capítulo 6

Ferramenta ProHChecker e Resultados

Neste capítulo, são apresentados resultados da aplicação do método que foi definido formalmente nos capítulos 4 e 5. Inicialmente, na seção 6.1, é descrita a ferramenta ProHChecker, que implementa o método proposto para a validação de sistemas descritos em SystemC. Dois exemplos ilustrativos da utilização da ferramenta são apresentados na seção 6.2, juntamente com os resultados obtidos.

Na seção 6.3, é apresentado um estudo de caso da aplicação do método proposto para a detecção de condições de *overflow* em filtros IIR (*Infinite Impulse Response*). Os resultados deste estudo de caso mostram que o método pode ser ordens de magnitude mais eficiente na busca por contraexemplos do que a verificação baseada em simulação aleatória, além de encontrar contraexemplos mais rapidamente do que BMC. Como mais uma motivação para a sua utilização, é apresentado um experimento onde o método é executado diretamente no *hardware* de um sistema embutido cuja memória RAM é de apenas 512 *Bytes*.

6.1 Descrição da Ferramenta ProHChecker

Como prova de conceito para o método proposto nos capítulos 4 e 5, foi desenvolvido um protótipo de uma ferramenta que implementa os algoritmos descritos neste trabalho. A ferramenta ProHChecker (*Properties Heuristic Checker*) mostra a viabilidade da utilização da DPC heurística apresentada para a validação de DUVs reais. ProHChecker foi desenvolvida na linguagem C++, objetivando sua aplicação em descrições que utilizem a linguagem SystemC. SystemC foi escolhida como linguagem base para a des-

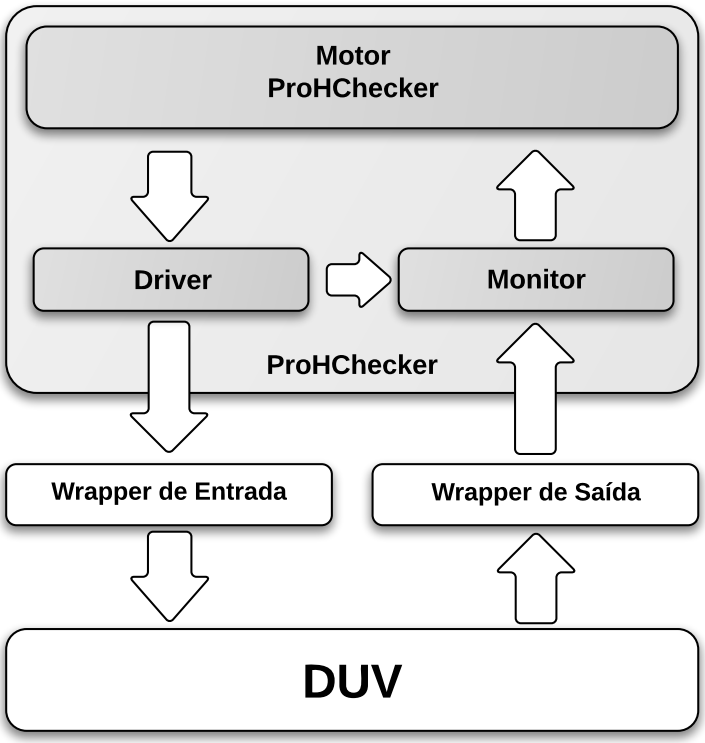


Figura 6.1: Diagrama de Blocos da Ferramenta ProHChecker

crição do DUV por dois motivos: 1) SystemC é o padrão *de-facto* para a descrição em alto nível de abstração de sistemas complexos; e 2) SystemC é baseado em C++, facilitando a integração do DUV com a ferramenta ProHChecker. É importante ressaltar que, apesar da ferramenta ter sido desenvolvida para a validação de sistemas descritos em SystemC, esta não é uma limitação imposta pelo método. O método proposto é genérico o bastante para ser aplicado em sistemas descritos em outras HDLs, desde que a linguagem permita a simulação do modelo.

A ferramenta ProHChecker é composta dos cinco blocos principais apresentados na figura 6.1 e descritos a seguir:

1. O **Wrapper de Saída** converte o padrão de sinalização de saída do DUV para o padrão de entrada da ferramenta. Este bloco deve ser gerado manualmente pelo projetista, pois o protocolo de sinalização do DUV é específico do projeto;

2. O **Monitor** calcula o valor da penalidade para a asserção da propriedade sendo verificada, a partir dos valores da entrada aplicada e da saída computada pelo DUV. Este bloco pode ser gerado automaticamente, usando as asserções informadas pelo usuário e as interfaces do *Wrapper de Entrada* e do *Wrapper*

de Saída. Atualmente, este bloco é gerado manualmente, mas um compilador para a linguagem PSL está sendo construído para gerá-lo de forma automática;

3. O **Motor ProHChecker** emprega um algoritmo de otimização numérica para buscar por um vetor de entrada que minimiza o valor da penalidade calculada pelo *Monitor*. O algoritmo de otimização é configurável, permitindo que se escolha o mais adequado para o DUV sendo verificado. Atualmente, os algoritmos suportados por este bloco são: 1) uma versão do algoritmo de Quasi-Newton com atualização BFGS, LBFGS (OKAZAKI, 2010); 2) a busca binária de Knowles e Hughes (2005); 3) o algoritmo SASS (*Self Adaptive Stepsize Search*) (NOLLE; BLAND, 2012); e 4) o algoritmo *Hill-Climbing* (RUSSELL; NORVIG, 2002). Este bloco é genérico e pode ser reutilizado para qualquer DUV;

4. O **Driver** traduz os tipos de dado utilizados pelo *Motor ProHChecker* para os tipos de entrada do DUV. O *driver* pode ser gerado automaticamente utilizando a interface fornecida pelo *Wrapper de Entrada*. Atualmente, este bloco está sendo gerado manualmente;

5. O **Wrapper de Entrada** converte o padrão de sinalização da ferramenta para o padrão utilizado pelo DUV. Este bloco deve ser gerado manualmente pelo projetista, uma vez que o protocolo do DUV é específico do projeto.

Sempre que um novo conjunto de valores para as entradas do DUV é fornecido pelo *Motor ProHChecker*, o bloco *Driver* sinaliza para o *Wrapper de Entrada*, por meio de uma FIFO (*sc_fifo*), que um novo valor está disponível. O *Wrapper de Entrada*, então, coleta este novo conjunto de valores e apresenta ao DUV. Da mesma forma, sempre que for disponibilizar a resposta computada pelo DUV à ferramenta, o *Wrapper de Saída* deve sinalizar para o bloco *Monitor*, por meio de uma FIFO. A execução da ferramenta fica suspensa durante os períodos em que o DUV computa sua saída. Após receber a resposta do DUV, a ferramenta computa o próximo valor a ser apresentado ao DUV em apenas um *ciclo delta*[1] da máquina de simulação do SystemC, permitindo a atualização instantânea das entradas, considerando-se o tempo de simulação.

[1]Um ciclo delta é, segundo Black et al. (2009, p. 29, Tradução Nossa) "Uma avaliação, seguida de uma atualização" dos sinais do sistema. O tempo de simulação não avança durante um ciclo delta. (Nota do Autor).

Por se tratar de uma prova de conceito, esta versão da ferramenta ProHChecker não implementa os algoritmos multitarefa apresentados nos capítulos 4 e 5.

6.2 Exemplos Ilustrativos

De modo a ilustrar a aplicação da ferramenta ProHChecker, dois módulos de exemplo foram desenvolvidos. O primeiro representa um sistema que implementa a equação 4.17, descrita na subseção 4.2.3. Neste exemplo, o valor da saída do DUV depende apenas dos valores atuais das entradas e, portanto, pode-se aplicar o método para sistemas sem memória apresentado no capítulo 4. O segundo exemplo é um módulo que implementa uma versão comportamental do máximo divisor comum (MDC) de Euclides, no qual a leitura das entradas é realizada apenas no primeiro ciclo de computação. O módulo processa esta entrada por um número variável de ciclos e o resultado da computação é disponibilizado no último ciclo. Para este módulo do MDC de Euclides, todo o tratamento temporal foi realizado pelos *Wrappers* de entrada e de saída, fazendo com que o módulo se comportasse como um sistema sem memória para a ferramenta. Portanto, nestes dois exemplos, a ferramenta foi utilizada de acordo com a formulação oferecida no capítulo 4.

Em ambos exemplos, foram utilizadas as funções de penalidade quadráticas, apresentadas na subseção 4.2.1. No primeiro exemplo, a etapa de otimização, realizada pelo *Motor ProHChecker*, foi implementada utilizando a biblioteca LBFGS (OKAZAKI, 2010), que fornece uma versão do método de otimização Quasi-Newton. Métodos de Quasi-Newton usam o gradiente da função para realizar a otimização e, em geral, apresentam bom desempenho mesmo para superfícies que não sejam suaves. A função de penalidade quadrática foi escolhida para se evitar situações de *overflow* no cálculo das penalidades.

Os resultados da ferramenta foram comparados com a geração aleatória de vetores de teste, uma vez que técnicas de geração de vetores de teste para a validação de sistemas descritos em alto nível de abstração que empregam operadores relacionais, em geral, utilizam geração aleatória em algum nível. Além disso, qualquer abordagem que aumente a eficiência de métodos puramente aleatórios a partir da restrição do espaço de busca, como as técnicas de satisfação de restrições, afetará positivamente a eficiência do método proposto nesta tese. As técnicas convencionais para a DPC dirigida por cobertura

não foram consideradas para a comparação por não oferecerem suporte para operadores relacionais, permitindo apenas a validação de sistema cujos sinais de entrada são do tipo *bit*.

Como o método proposto é, em sua essência, um método de verificação dinâmica de propriedades, mesmo se um contraexemplo não for encontrado durante a validação, não se pode afirmar que o sistema não possui defeitos. Por este motivo, não se pode avaliar o desempenho do método em relação ao tempo necessário para provar que o sistema está correto, mas sim pelo tempo necessário para encontrar um contraexemplo de uma determinada propriedade e pelo número de avaliações do sistema sob verificação durante esta tarefa. Para avaliar o desempenho do método, portanto, um erro artificial foi introduzido em cada um dos exemplos.

Todos os exemplos foram executados em um computador equipado com um processador Intel Core i7, com 6GB de memória RAM, executando Linux. Os valores médios para os tempos de execução e para o número de avaliações do DUV foram calculados a partir da execução de 50 experimentos independentes.

6.2.1 Sistema Simples sem Memória

Neste primeiro exemplo, um módulo SystemC implementando o exemplo ilustrativo apresentado na equação 4.17 foi validado contra a asserção da propriedade da equação 4.18. Por conveniência, estas duas equações são novamente apresentadas nas equações 6.1 e 6.2, respectivamente.

$$g(s_1, s_2) = (s_1)^2 + 2s_1 + (s_2)^2 + 3s_2 + 4 \tag{6.1}$$

$$P \equiv ((s_1 > 1, 0) \rightarrow (d_1 > 7, 0)) \tag{6.2}$$

O módulo apresenta duas entradas ponto flutuante de precisão dupla, s_1 e s_2, e uma saída ponto flutuante de precisão dupla, d_1, representando, respectivamente, os operandos e o resultado do cálculo da função da equação 6.1. O resultado, neste caso, depende somente dos valores atuais das entradas, o que caracteriza um sistema sem memória. Por este motivo, a aplicação do método proposto neste tipo de módulo é direta, como foi apresentado no capítulo 4.

Dois métodos de geração de testes aleatórios com distribuição uniforme foram usados durante a simulação. No primeiro conjunto de testes, as entradas (s_1 e s_2) foram limitadas à faixa $[-4,0 \times 10^9,\ 4,0 \times 10^9]$. No segundo conjunto de testes, a entrada s_1 foi restrita à faixa $(1,\ 4,0 \times 10^9]$, em outras palavras, o antecedente da propriedade P foi sempre verdadeiro durante a aplicação deste segundo conjunto de testes. Ambos cenários executaram por mais de meia hora e, aproximadamente, 3 bilhões de avaliações foram realizadas durante cada simulação. Nenhum dos métodos aleatórios de geração de vetores de teste foi capaz de encontrar uma violação da propriedade sendo testada.

O sistema foi, então, manualmente traduzido para um programa ANSI-C e verificado utilizando-se a ferramenta CBMC. Inesperadamente, a ferramenta CBMC não foi capaz de encontrar um contraexemplo para a asserção da propriedade sendo verificada, apesar do sistema não possuir a propriedade P da equação 6.2. Uma análise mais detalhada do experimento revelou que a causa do problema foi a representação interna utilizada pelo CBMC para modelar números do tipo ponto flutuante. Todos os números ponto flutuante são modelados com notação do tipo ponto fixo (Carnegie Mellon University, 2012), o que impediu que o verificador BMC derivasse um contraexemplo para este problema específico. Para que o CBMC pudesse derivar um contraexemplo, foi preciso restringir a faixa de operação do ponto flutuante para $[-9.000, 9.000]$. Neste caso, o CBMC encontrou um contraexemplo em 483 milissegundos, em média.

Diferentemente da simulação aleatória e da ferramenta CBMC, a ferramenta ProHChecker, que implementa o método de DPC heurística proposto neste trabalho, foi capaz de direcionar a simulação de forma eficiente para os contraexemplos da asserção da propriedade verificada utilizando a faixa original das entradas. A ferramenta precisou de apenas 12 milissegundos e 3.700 avaliações do DUV (em média) para revelar um contraexemplo, um ganho de aproximadamente 40 vezes em relação ao CBMC, que realizou a verificação utilizando uma faixa reduzida de valores válidos para as entradas.

6.2.2 Máximo Divisor Comum de Euclides

Neste segundo exemplo, foi realizada a validação de um módulo implementando uma versão comportamental do algoritmo para cálculo do MDC de Euclides. O módulo possui duas entradas inteiras (a e b), representando os operados do MDC, uma entrada booleana (*início*) que permite a sinalização do início da computação, uma entrada de

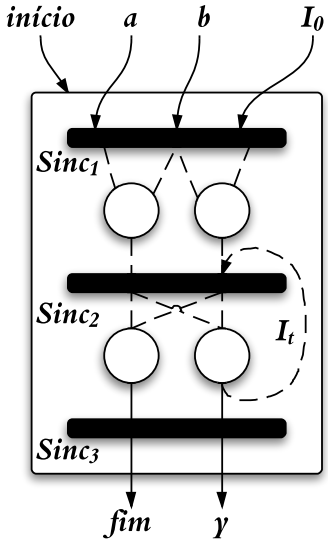


Figura 6.2: Comportamento Geral do Módulo para Cálculo do MDC

relógio (*clock*), uma saída inteira (y), representando o resultado da operação, e uma saída booleana (*fim*) que indica quando a computação do módulo está concluída. A figura 6.2 mostra o comportamento geral do módulo implementado.

O sistema possui um único ponto para a sincronização das entradas ($Sinc_1$), onde o valor das entradas é coletado e o estado interno, I, recebe o valor inicial, I_0. Depois disso, o módulo entra em um laço, onde um novo valor do estado interno é calculado a cada ciclo. O início de um novo ciclo é marcado pelo ponto de sincronização $Sinc_2$. Quando a computação termina, o sinal da saída *fim* é ativado, indicando que o valor na saída y é válido.

Foram verificadas duas propriedades neste exemplo:

- *Propriedade* P_1: Dados dois números, a e b, se $a = b$, então o MDC entre a e b é igual a b;

- *Propriedade* P_2: Dados dois números, a e b, se b é diferente de 0 (zero) e a é múltiplo de b, então o MDC entre a e b é igual a b.

Os *Wrappers de Entrada* e *de Saída* foram implementados de forma que o DUV se comportasse como um sistema sem memória para a ferramenta. O *Wrapper de Entrada* recebe os valores a serem aplicados às entradas do DUV e ativa o sinal de *início* para que o DUV comece a computação. Enquanto isto, o *Wrapper de Saída* aguarda o sinal de *fim*, só enviando o valor da saída y para o bloco *Monitor* após este sinal de fim de computação ser ativado. Como a execução da ferramenta fica suspensa entre o envio dos

sinais de entrada para o *Driver* e o recebimento do valor do sinal de saída, proveniente do *Monitor*, a definição de sistema sem memória (definição 3.30) é obedecida. Com isto, pode-se escrever as propriedades P_1 e P_2 usando a LPE, sem se recorrer a operadores temporais. Estas propriedades estão descritas nas equações 6.3 e 6.4.

$$P_1 \equiv ((a = b) \rightarrow (y = a)) \tag{6.3}$$

$$P_2 \equiv (((b \neq 0) \ \wedge \ (a \ \mathbf{mod} \ b = 0)) \rightarrow y = b) \tag{6.4}$$

Aplicando-se a interpretação semântica alternativa, apresentada na definição 4.3, pode-se determinar as FCBAs para as asserções das propriedades P_1 e P_2, que são apresentadas, respectivamente, nas equações 6.5 e 6.6.

$$f_{\mathcal{A}_1}(a, b, y) = h_{eq}(a, b) + h_{ne}(y, a) \tag{6.5}$$

$$f_{\mathcal{A}_2}(a, b, y) = h_{ne}(b, 0) + h_{eq}(a \ \mathbf{mod} \ b, 0) + h_{ne}(y, b) \tag{6.6}$$

Neste exemplo, as entradas foram limitadas à faixa $[-2^{31}, 2^{31} - 1]$ (i.e. 2^{32} valores). Portanto o número de combinações de entrada é de 2^{64} valores. As asserções das propriedades P_1 e P_2 foram validadas utilizando simulação aleatória, BMC e a ferramenta ProHChecker empregando os quatro algoritmos de otimização suportados.

Os resultados dos experimentos são apresentados na tabela 6.1, e mostram que a ferramenta ProHChecker foi muito mais eficiente na busca por contraexemplos do que a simulação aleatória. Para a asserção da propriedade P_1, o ganho em desempenho foi de aproximadamente 22 mil vezes, em média, para o algoritmo LBFGS (OKAZAKI, 2010), e 121 vezes para o algoritmo SASS (NOLLE; BLAND, 2012). Para a asserção da propriedade P_2 os resultados não foram da mesma ordem de grandeza, porém o ganho em tempo ainda foi expressivo. O número de avaliações do DUV quando utilizando a ferramenta ProHChecker foi reduzido na mesma ordem de grandeza do que o tempo da busca. Isto mostra que os resultados do emprego da ferramenta em módulos cuja simulação seja demorada devem ser ainda mais favoráveis. Os algoritmos de busca binária de Knowles e Hughes (2005) e o *Hill-Climbing* não foram capazes de derivar contraexemplos.

Tabela 6.1: Resultados para o Exemplo do MDC

	Propriedade P_1		Propriedade P_2	
	Tempo	Avaliações	Tempo	Avaliações
ProHChecker (LBFGS)	0,022 s	1.359	0.0186 s	1.181
ProHChecker (SASS)	4,124 s	359.628	0.4729 s	129.813
ProHChecker (*Hill-Climbing*)	-	-	-	-
ProHChecker (*Binário*)	-	-	-	-
Aleatório	500 s	39 milhões	15,2 s	1,2 milhões
Ganho (LBFGS)	22.727	28.697	817	1016
Ganho (SASS)	121	108	32	9
CBMC	0,0365 s	NA	0,0412 s	NA
Ganho (LBFGS)	1,66	NA	2,22	NA
Ganho (SASS)	0,009	NA	0,087	NA
Ganho (*Hill-Climbing*)	-	-	-	-
Ganho (*Binário*)	-	-	-	-

Quando comparado aos resultados atingidos pela ferramenta CBMC, o ganho em tempo de execução da ferramenta ProHChecker, no caso do algoritmo LBFGS, não foi expressivo e, no caso do algoritmo SASS, a ferramenta obteve resultados desfavoráveis. É importante ressaltar, no entanto, que a ferramenta não recorreu ao código fonte do DUV e foi aplicada sobre a versão original do módulo do MDC de Euclides. Para ser verificado pelo CBMC, o DUV teve que ser traduzido manualmente para ANSI-C, ou seja, o CBMC somente verificou a implementação do algoritmo do MDC de Euclides, não o módulo completo. É razoável afirmar que o ganho da ferramenta em relação ao CBMC seria maior caso o escalonador do SystemC fosse incluído na verificação, como ocorre nas abordagens propostas em (CHOU et al., 2012), (CIMATTI et al., 2011a) e (GROSSE; LE; DRECHSLER, 2010).

6.3 Estudo de Caso: validação de filtros IIR

Neste estudo de caso, a ferramenta ProHChecker foi utilizada para validar um filtro IIR (*Infinite Impulse Respose*). O filtro IIR é um sistema com memória, ou seja, sua saída atual depende de valores passados de suas entradas, requerendo a aplicação do método proposto no capítulo 5.

Filtros digitais são utilizados em vários tipos de aplicação, desde sistemas de áudio, até sistemas de monitoramento de sinais vitais. Sendo componentes tão comuns,

erros em filtros digitais podem levar a resultados catastróficos. Normalmente, as ferramentas para projetos de filtros digitais usam aritmética de ponto flutuante para calcular os parâmetros do filtro. No entanto, devido a restrições de desempenho, é comum que a implementação de filtros digitais em sistemas embutidos seja feita por meio de aritmética de ponto fixo, o que pode comprometer a robustez dos algoritmos e levar a resultados indesejáveis, como instabilidades e condições de *overflow*.

Na maior parte das vezes, a validação dos filtros digitais é feita por meio da aplicação de um conjunto extenso de estímulos de entrada. Porém, validar um filtro de forma completa por meio de testes é uma tarefa quase impossível, pois o número de estados a serem verificados é proibitivo. O trabalho de Cox, Sankaranarayanan e Chang (2012) mostrou que testes aleatórios e dirigidos não são capazes de derivar as condições de *overflow* em filtros digitais causadas pelo emprego de notação do tipo ponto fixo.

A técnica de BMC oferece uma alternativa interessante para o teste aleatório. No entanto, a verificação de filtros digitais utilizando BMC sofre, além dos problemas mencionados no capítulo 3, de outros problemas específicos. Primeiramente, para aplicar BMC, o algoritmo do filtro deve ser modificado para que o verificador BMC possa analisá-lo corretamente. Algumas destas modificações não são triviais para os que não estão acostumados a utilizar este tipo de ferramenta. Além disso, a necessidade de acesso ao código fonte do modelo é um problema mais difícil de ser contornado nos casos de sistemas embutidos, uma vez que o algoritmo de BMC não pode garantir que o filtro digital irá funcionar corretamente na plataforma de *hardware* do sistema, a menos que toda a plataforma possa ser descrita formalmente. Assim, a utilização de técnicas de DPC se torna uma alternativa interessante para a validação deste tipo de sistema.

Esta seção é divida em três partes. Primeiramente, é feita uma descrição do filtro IIR utilizado no exemplo. Em seguida, é apresentado o experimento no qual a ferramenta ProHChecker foi empregada para validar o filtro em relação a condições de *overflow* de sua saída. Finalmente, na subseção 6.3.3, o mesmo filtro é validado utilizando o método proposto neste trabalho dentro de um sistema embutido com severas restrições de memória. A motivação para este último exemplo é mostrar que o método proposto pode ser empregado para validar o sistema dentro de uma plataforma de *hardware* com recursos computacionais limitados, onde técnicas mais complexas, como o BMC, não podem ser empregadas e em situações onde a validação aleatória apresenta resultados ineficientes.

Tabela 6.2: Coeficientes do Filtro IIR

Coeficiente	Valor Ponto Flutuante	Valor Ponto Fixo
b_0	0,2066	0,21875
b_1	0,4131	0,40625
b_2	0,2066	0,21875
a_1	-0,3695	-0,375
a_2	0,1958	0,1875

6.3.1 Descrição do Filtro IIR

O filtro utilizado neste exemplo foi reproduzido de um dos experimentos apresentados em (COX; SANKARANARAYANAN; CHANG, 2012). O objetivo foi determinar se a ferramenta ProHChecker seria capaz de derivar um contraexemplo para uma condição de *overflow*, aumentando a eficácia da validação. O filtro desenvolvido implementa a fórmula para a forma direta I de um filtro IIR, que é apresentada na equação 6.7, onde a_j e b_i são os coeficientes do filtro, y_0 é a constante que representa o estado inicial do filtro, e x é a sequência de entrada apresentada ao filtro.

$$y(t) = \begin{cases} \sum_{i=0}^{N} b_i x(t-i) - \sum_{j=1}^{M} a_j y(t-j) & , \; se\ t \geq 0 \\ y_0 & , \; se\ t < 0 \end{cases} \tag{6.7}$$

Foi escolhido um filtro IIR Butterworth passa-baixas, com frequência de corte de 9600 Hz e frequência de amostragem de 48 KHz. Os coeficientes do filtro são mostrados na tabela 6.2, que também apresenta a versão quantizada para estes coeficientes, utilizando notação ponto fixo Q2.5[2], que permite a representação de números na faixa $[-2,\ 1,96875]$. Os valores de entrada do filtro foram restritos à faixa $[-1,6,\ 1,6]$ de forma a restringir os valores de saída à faixa $[-2,2]$ (COX; SANKARANARAYANAN; CHANG, 2012).

O módulo do sistema possui duas entradas (a entrada do filtro, x, e uma entrada de *reset*). A entrada de *reset* leva o filtro para o estado inicial, enquanto a saída y é calculada utilizando a equação 6.7, usando os coeficientes da terceira coluna da tabela 6.2. O sistema não possui uma saída de *overflow*, uma vez que uma condição de *overflow* não é esperada de um sistema sem defeitos.

[2]Notação usando o formato QI.F, onde I representa o número de bits inteiros e F o números de bits fracionários. (Nota do Autor).

A propriedade a ser verificada neste exemplo é:

- *Propriedade P*: Se as entradas do filtro permanecerem dentro da faixa válida, uma condição de *overflow* na saída nunca deve ocorrer.

Para escrever esta propriedade mais formalmente, é preciso definir a faixa válida para os valores, em termos da representação Q2.5. Usando esta representação, a faixa válida para os valores de entrada é traduzida para $[-51, 51]$, e os valores de saída devem permanecer na faixa $[-64, 63]$.

É possível perceber que a propriedade P deste exemplo não pode ser descrita utilizando a LPE. Neste caso, é preciso utilizar a LTLE, apresentada na seção 3.3, com seus operadores temporais. A equação 6.8 descreve a propriedade P utilizando a LTLE.

$$P \equiv (\mathbf{G}((x \geq -51) \wedge (x \leq 51)) \rightarrow (\mathbf{G}((y \geq -64) \wedge (y \leq 63))) \tag{6.8}$$

Aplicando-se as transformações apresentadas no capítulo 5, obtém-se a fórmula da LPE equivalente a partir do desdobramento da propriedade P. Esta fórmula é apresentada na equação 6.9, que mostra o desdobramento da propriedade por N pontos no tempo.

$$\begin{aligned} P^N \equiv & \left(\left(\bigwedge_{j=1}^{N} (x[j] \geq -51) \right) \wedge \left(\bigwedge_{j=1}^{N} (x[j] \leq 51) \right) \right) \rightarrow \\ & \left(\left(\bigwedge_{j=1}^{N} (y[j] \geq -64) \right) \wedge \left(\bigwedge_{j=1}^{N} (y \leq 63) \right) \right) \end{aligned} \tag{6.9}$$

Utilizando-se da interpretação semântica alternativa, apresentada na definição 4.3, pode-se determinar a FCBA para a asserção da propriedade P, que é apresentada na equação 6.10.

$$\begin{aligned} f_{\mathcal{A}_{P^N}} = & \sum_{j=1}^{N} h_{ge}(x[j], -51) + \sum_{j=1}^{N} h_{ge}(51, x[j]) + \\ & min\left(Min_{j=1}^{N}(h_{gt}(-64, y[j])), Min_{j=1}^{N}(h_{gt}(y[j], 63))\right) \end{aligned} \tag{6.10}$$

A FCBA da equação 6.10 foi utilizada para validar filtros digitais em dois experimentos. No primeiro, foi utilizada a ferramenta ProHChecker, apresentada no início deste capítulo. Este experimento é descrito na seção 6.3.2. No segundo experimento, o método proposto neste trabalho foi utilizado para construir um *framework* para buscar por condições de *overflow* em filtros sendo executados dentro sistemas embutidos, com desempenho limitado. Este experimento é descrito na seção 6.3.3.

6.3.2 Validação do Filtro IIR usando ProHChecker

Cox, Sankaranarayanan e Chang (2012) verificaram condições de *overflow* em várias configurações de filtros IIR utilizando aritmética de ponto fixo. Os autores utilizaram três métodos de geração aleatória de estímulos para validar os sistemas: 1) geração com distribuição uniforme; 2) valores máximos de entrada até que a saída estabilizasse, seguido do valor mínimo de entrada; 3) valor mínimo de entrada até a estabilização da saída, seguido do valor máximo. Os resultados do trabalho mostraram que a simulação não foi capaz de gerar condições de *overflow*, apesar dos sistemas possuírem defeitos de projeto.

Neste experimento, dois métodos de verificação foram comparados com os resultados da ferramenta ProHChecker: geração aleatória e verificação de modelos com profundidade limitada (BMC). Novamente, as técnicas convencionais para a DPC dirigida por cobertura não foram consideradas para a comparação por não oferecerem suporte para operadores relacionais, permitindo apenas a validação de sistema cujos sinais de entrada são do tipo *bit*. Na geração aleatória, foi utilizada uma distribuição uniforme de probabilidades, enquanto a ferramenta de BMC utilizada foi, novamente, o CBMC. Além destes métodos, um teste exaustivo também foi conduzido para se determinar quantas das sequências de entrada (de até seis instantes de tempo de comprimento) levavam a uma condição de *overflow*.

O teste exaustivo mostrou que apenas 26 sequências (de um total de 1200 bilhões de sequências de exatamente seis instantes de tempo de comprimento) eram capazes de disparar uma condição de *overflow* no filtro sob verificação. Usando estes números, combinados com o número de avaliações por segundo do algoritmo do filtro, foi possível determinar o tempo médio requerido para encontrar um contraexemplo utilizando geração aleatória, utilizando a equação 6.11, onde E é o número de avaliações por segundo do

algoritmo. Para os demais métodos, o tempo médio foi calculado sobre 50 execuções independentes.

$$t_{avg} = \frac{1.200 \times 10^9}{26} \times \frac{1}{E} \tag{6.11}$$

Para a geração aleatória, foram utilizados três métodos distintos. Primeiramente, uma sequência única de 1×10^9 vetores de entrada foi aplicada ao sistema. Em seguida, 10 milhões de sequências de 100 instantes de tempo de comprimento foram aplicadas. Por fim, 170 milhões de sequências de seis vetores de comprimento foram apresentadas ao DUV. Nenhuma das sequências conseguiu revelar uma condição de *overflow*. Cada um destes métodos foi executado por mais de meia hora.

No caso da ferramenta ProHChecker, foram utilizadas as funções de penalidade quadrática, para se evitar as situações de *overflow*. Todos os quatro métodos de otimização suportados pelo *Motor ProHChecker* foram executados durante o experimento, porém o algoritmo LBFGS não foi capaz de derivar contraexemplos dentro do limite de tempo estipulado de meia hora.

Os resultados do experimento são mostrados na tabela 6.3. Estes resultados mostram que a ferramenta ProHChecker consegue detectar as condições de *overflow* mais eficientemente do que a geração aleatória para todos os algoritmos de otimização empregados, exceto o LBFGS. Mesmo em relação ao CBMC, os resultados do ProHChecker mostram-se relevantes, principalmente quando se considera que a ferramenta validou o módulo original, descrito em SystemC, enquanto o CBMC verificou uma versão ANSI-C do módulo, que não inclui o escalonador do SystemC.

No Apêndice A, é apresentado o código fonte, em SystemC, do bloco *Monitor* deste exemplo. Naquele trecho de código fonte, pode-se perceber que o bloco monitor é facilmente criado a partir da equação que descreve a FCBA da propriedade sendo verificada. Todo o procedimento de transformação da propriedade sendo verificada para equação da FCBA, descrito nos capítulos 4 e 5, e da equação da FCBA para o código fonte do bloco *Monitor* pode ser automatizado facilmente, eliminando-se a possibilidade de inserção de defeitos de implementação nos blocos da ferramenta ProHChecker.

Tabela 6.3: Resultados da Validação do Filtro IIR usando ProHChecker

	Propriedade P	
	Tempo	Avaliações
ProHChecker (LBFGS)	-	-
ProHChecker (SASS)	0,24 s	6.435
ProHChecker (Binário)	0,22 s	7.423
ProHChecker (*Hill-Climbing*)	1,69 s	127.583
Aleatório[a]	1,07 horas	46 milhões
Ganho (LBFGS)	-	-
Ganho (SASS)	16.050	7.148
Ganho (Binário)	17.509	6.196
Ganho (*Hill-Climbing*)	2.279	360
CBMC	2,39 s	NA
Ganho (LBFGS)	-	-
Ganho (SASS)	9,96	NA
Ganho (Binário)	10,86	NA
Ganho (*Hill-Climbing*)	1,41	NA

[a]Calculado com base no número de execuções do filtro por segundo.

6.3.3 Validação do Filtro IIR dentro de um Sistema Embutido

Neste experimento, o método apresentado nos capítulos 4 e 5 foi adaptado para ser executado dentro de uma plataforma típica de sistema embutido. Para isto, foi criado um *framework* para a busca por condições de *overflow* em filtros digitais. O *framework* foi implementado utilizando a linguagem ANSI-C, uma vez que esta linguagem é muito comum para o desenvolvimento de sistemas embutidos. O algoritmo de otimização empregado neste experimento foi o SASS. SASS foi escolhido por três motivos: 1) é um algoritmo de otimização quase sem parâmetros e, desta forma, requer poucos ajustes antes de se realizar a otimização; 2) o tamanho do passo automaticamente e dinamicamente ajustado deste algoritmo o torna interessante para ser utilizado em problemas com vários mínimos locais (i.e. multimodais) e onde não é possível se extrair boas informações do gradiente da função; e, mais importante, 3) O SASS requer pouca memória para ser executado, diferentemente do LBFGS e do algoritmo de busca binária de Knowles e Hughes (2005).

A plataforma de *hardware* utilizada foi o kit de desenvolvimento da Texas Instruments para a linha Launchpad (Texas Instruments, 2012), que foi equipado com o

```
1 fcfLongInt filter(fcfInt x,
2                   fcfInt xHist[],
3                   fcfLongInt yHist[])
4 {
5  xHist[2] = xHist[1]; xHist[1] = xHist[0];
6  yHist[2] = yHist[1]; yHist[1] = yHist[0];
7  xHist[0] = x;
8
9  fcfLongInt y = mult(xHist[0],b0, 5)
10               + mult(xHist[1],b1, 5)
11               + mult(xHist[2],b2, 5)
12               + mult(yHist[1],-a1,5)
13               + mult(yHist[2],-a2,5);
14  yHist[0] = y;
15  return y;
16 }
```

Listagem 6.1: Código Fonte do Filtro IIR Digital

microcontrolador MSP430G2553. Este microcontrolador executa a 16 MHz e possui uma memória Flash de 16 KBytes, utilizados para armazenar todo o programa do usuário. A característica mais importante deste microcontrolador, no entanto, é que a sua memória RAM possui apenas 512 *Bytes*. Isto mostra que, apesar dos resultados atingidos, o método proposto no presente trabalho requer muito pouca memória durante a execução.

O mesmo filtro IIR utilizado na subseção 6.3.2 foi implementado para a plataforma de *hardware* deste exemplo e verificado utilizando o *framework* desenvolvido. O código fonte deste filtro é apresentado na Listagem 6.1. Nesta Listagem, *fcfInt* e *fcfLongInt* representam, respectivamente, os tipos de dados *Inteiro* e *Inteiro Longo* da plataforma alvo. Os arranjos *xHist* e *yHist* armazenam os valores históricos da entrada e da saída durante as execuções do algoritmo. A macro *mult* executa a multiplicação dos dois primeiros parâmetros, utilizando notação de ponto-fixo, utilizando o terceiro parâmetro como o número de *bits* fracionários.

Os resultados experimentais para este exemplo são mostrados na tabela 6.4. Assim como no experimento anterior, o tempo de execução médio para a validação baseada em vetores aleatórios foi calculado utilizando a equação 6.11. Verifica-se, a partir dos resultados apresentados, que a utilização do método proposto neste trabalho é a única alternativa viável para a validação do filtro IIR dentro da plataforma de *hardware* alvo.

Tabela 6.4: Resultados da Validação do Filtro IIR dentro de um Sistema Embutido

Teste aleatório[a]	*Framework*
276 dias	144 s

[a]Calculado com base no número de execuções do filtro por segundo.

6.4 Resumo do Capítulo

Neste capítulo, foi apresentada uma ferramenta para aplicação do método proposto neste trabalho. A ferramenta ProHChecker foi desenvolvida em C++, direcionada para a aplicação em sistemas descritos em SystemC. Dois exemplos ilustrativos da aplicação da ferramenta foram apresentados. Primeiramente, um módulo que implementa uma função matemática simples foi validado contra a asserção de uma propriedade artificial, criada especificamente para não ser válida para o módulo desenvolvido. Em seguida, um módulo implementando uma versão comportamental do algoritmo para o cálculo do MDC de Euclides foi validado usando a ferramenta. Em ambos os exemplos, os resultados apresentados pela ferramenta foram superiores aos das técnicas usadas para comparação, que foram a simulação aleatória e o BMC.

Um estudo de caso da aplicação do método também foi descrito neste capítulo. A ferramenta ProHChecker foi aplicada para determinar se um filtro IIR digital que emprega aritmética de ponto fixo possuía estados que geravam condições de *overflow* da saída. Este estudo de caso foi inspirado no trabalho de Cox, Sankaranarayanan e Chang (2012), que mostrou que os métodos de validação convencionais não eram capazes de revelar as condições de *overflow*. Novamente, o método proposto foi comparado com simulação aleatória e BMC. Os resultados foram relevantes, mostrando que, apesar da dificuldade em exercitar estas condições de *overflow*, o método foi capaz de derivar contraexemplos mais rapidamente do que as duas técnicas utilizadas para comparação.

O método também foi empregado para validar um filtro digital sendo executado dentro de um sistema embutido com severas restrições de memória. Este experimentou mostra que, diferentemente das técnicas de verificação de modelo, o método requer muito pouca memória para ser executado, dependendo do algoritmo de otimização utilizado.

No próximo capítulo, são apresentadas discussões sobre o método, a ferramenta e os resultados obtidos nos experimentos, bem como as conclusões deste trabalho.

Capítulo 7

Considerações Finais

Neste capítulo, são oferecidas as considerações finais sobre o presente trabalho. Primeiramente, na seção 7.1, é apresentada uma discussão sobre o método proposto nos capítulos 4 e sobre a ferramenta ProHChecker, além de uma análise mais detalhada acerca dos resultados obtidos nos experimentos do capítulo 6. Na seção 7.2, desdobramentos da presente pesquisa são apresentados na forma de sugestões para trabalhos futuros. O capítulo é finalizado com as conclusões deste trabalho.

7.1 Discussões

Esta seção apresenta uma análise crítica a respeito dos resultados obtidos com a presente pesquisa. Na subseção 7.1.1, são apresentadas discussões sobre o método proposto, discorrendo-se sobre suas vantagens, desvantagens e limitações. Na subseção 7.1.2, é realizada uma análise mais detalhada sobre a ferramenta ProHChecker, indicando as funcionalidades que devem ser acrescentadas para que ela possa ser integrada de modo eficaz ao ciclo de projeto de CIs. A subseção 7.1.3 apresenta uma discussão mais detalhada acerca dos resultados obtidos nos experimentos do capítulo 6.

7.1.1 Sobre o Método

O método proposto neste trabalho, que foi apresentado em detalhes nos capítulos 4 e 5, permite modelar o problema da busca por contraexemplos como um problema de otimização utilizando funções heurísticas criadas a partir das asserções das propriedades do sistema. A principal vantagem apresentada por este método é que ele incorpora, dentro de sua formulação, os operadores relacionais normalmente empregados

em sistemas de alto nível de abstração. As técnicas atuais de DPC dirigida por cobertura consideram que os sinais envolvidos nas propriedades do sistema são do tipo *bit*, simplificando demasiadamente a interface dos sistemas sob verificação. Esta limitação se torna evidente quando se analisa as linguagens de descrição de propriedades utilizadas nos ambientes de verificação atuais. Estas linguagens admitem a utilização de operadores relacionais de forma implícita em suas construções, e até mesmo oferecerem suporte para a criação de monitores para propriedades que os envolvam. Porém o uso de operadores relacionais não é previsto diretamente em sua semântica, dificultando a criação de geradores de estímulos direcionados para a validação de propriedades que os envolvam. O método proposto neste trabalho, por sua vez, admite tais operadores explicitamente na sintaxe de suas propriedades e permite a criação de geradores de estímulos que levam em conta a semântica destes operadores.

A segunda vantagem da abordagem apresentada na presente pesquisa é sua independência em relação à disponibilidade do código fonte do modelo. O DUV é tratado como uma caixa-preta durante a validação, analisando-se somente o comportamento de suas saídas em relação aos estímulos de entrada aplicados. Desta forma, o método pode ser empregado mesmo quando o código fonte do DUV não está completamente disponível, como ocorre em sistemas que utilizam bibliotecas de terceiros ou blocos IP fechados. Mais do que isto, o método proposto é aplicável, na sua forma original, em ambientes de validação complexos, que sejam baseados em emulação ou que empreguem aceleradores por *hardware*. Em resumo, a abordagem apresentada abrange todos os domínios da figura 2.5, apresentada na página 32, desde que se tenha um modelo executável do DUV.

Outra vantagem do método proposto é que este requer pouca memória para ser executado, como ficou evidente no estudo de caso apresentado na subseção 6.3.3. Esta característica permite que ferramentas baseadas no método sejam utilizadas para validar aplicações sendo executadas dentro de sistemas embutidos com restrições de memória e de desempenho, não estando limitado apenas aos ambientes de desenvolvimento. Além disso, como verificado nos experimentos apresentados no capítulo 6, o método apresenta um desempenho muito superior aos métodos de validação baseados em geração aleatória de estímulos e, superando até mesmo o desempenho da técnica de BMC, no que diz respeito ao tempo de busca.

A maior desvantagem do método de DPC heurística proposto é a sua dificuldade em provar que o DUV está correto. Apesar do teorema 4.1 e do corolário 4.1 mostrarem que é possível realizar esta prova na teoria, a aplicação do corolário 4.1 na prática esbarra no problema de determinar o mínimo global da FCBA. Assim, na maior parte dos sistemas, caso não seja encontrado um contraexemplo durante a busca, não é possível afirmar que o DUV esteja correto. Apesar de relevante, esta mesma desvantagem é atribuída a todos os métodos de DPC, uma vez que para se validar completamente um sistema por meio de simulação, é preciso que todas as entradas possíveis sejam aplicadas ao modelo. Esta desvantagem não pode ser eliminada, uma vez que é inerente da abordagem de simulação empregada. É importante ressaltar, no entanto, que diferentemente de outras abordagens de DPC, o método proposto também permite que esta limitação seja parcialmente contornada de duas formas: 1) utilizando a heurística como guia para realizar uma busca completa pelo espaço de estados do sistema, priorizando estados mais próximos de violar as propriedades do sistema; e 2) derivando-se um método para estimar a probabilidade de existir um contraexemplo da propriedade sendo validada, dado que este não foi encontrado durante a busca. Estas duas abordagens são oferecidas como propostas para trabalhos futuros na seção 7.2.

Uma característica do método, que não chega a ser uma desvantagem, pois é inerente a todos os métodos de DPC, é que o modelo do sistema deve ser executado ao menos uma vez a cada iteração do algoritmo de otimização. Caso a execução do modelo seja demorada, o desempenho do método pode ser afetado de modo desfavorável. No entanto, como o método aprimora iterativamente os vetores de teste, esta característica pode se tornar uma vantagem do método proposto em relação a outras abordagens de DPC, pois normalmente o método requer menos vetores de testes para derivar um contraexemplo da propriedade (i.e. menos avaliações do modelo), como foi mostrado nos experimentos do capítulo 6.

Por fim, o método proposto apresenta uma limitação quanto à forma das propriedades que podem ser verificadas. As FCBAs definidas para algumas propriedades possuem uma topologia plana, atribuindo o mesmo valor a todos os símbolos de entrada, com exceção dos contraexemplos, que possuem valor zero. Devido a esta topologia plana, não é possível classificar os vetores de teste, pois o algoritmo de otimização não é capaz de definir a direção da busca, que se degenera para uma validação aleatória. Neste trabalho,

não foi possível derivar uma forma geral para as propriedades que são desinteressantes para a aplicação do método. No entanto, é possível determinar, usando poucas iterações, se a informação fornecida pela FCBA derivada da propriedade permite acelerar a busca pelos contraexemplos. Nos casos em que esta informação for insuficiente, a ferramenta que implementa o método aborta o processo e informa ao usuário que a propriedade não pode utilizada para acelerar a busca.

7.1.2 Sobre a Ferramenta ProHChecker

A ferramenta ProHChecker foi desenvolvida como prova de conceito para a aplicação do método proposto neste trabalho. As vantagens e desvantagens da ferramenta estão intimamente ligadas às vantagens e desvantagens do próprio método, como apresentado na subseção 7.1.1. No entanto, algumas limitações são impostas somente pela versão atual da ferramenta, e são descritas nesta subseção.

Primeiramente, os blocos do *Driver* e do *Monitor*, apresentados na figura 6.1 da página 105, não são gerados automaticamente nesta versão da ferramenta, devendo ser implementados pelo usuário. A implementação do *Driver* é simples, uma vez que este bloco nada mais é do que um tradutor, que traduz o tipo de dados empregado pelo *Motor ProHChecker* para os tipos de dados utilizados pelo DUV. Na maior parte das vezes, a própria linguagem de programação utilizada oferece suporte para a tradução dos tipos de dado, sendo necessário somente criar as atribuições dos sinais adequadamente. O *Monitor*, no entanto, por definir a FCBA que deve ser otimizada pelo *Motor PoHChecker*, precisa incluir o cálculo do valor desta função em seu código. A tradução das propriedades sendo verificadas para um código em C++ que permita o cálculo da FCBA é trabalhosa e pode se revelar complexa, dependendo da propriedade sendo verificada. Desta forma é necessário criar um compilador que permita gerar o bloco *Monitor* a partir da propriedade sendo verificada e da interface fornecida pelos *Wrappers* do DUV. Esta necessidade de intervenção manual é a principal limitação da ferramenta atualmente.

Apesar desta limitação, a ferramenta é totalmente funcional e pode ser utilizada para a validação de descrições em SystemC em qualquer nível de abstração, sem que seja imposta qualquer limitação ao estilo de código utilizado pelo usuário, nem mesmo às construções do C++ que podem ser empregadas. Ponteiros, alocação dinâmica de memória, estruturas de dados definidas pelo usuário, todas estas funcionalidades podem

ser utilizadas sem restrições no modelo sob verificação sem que seja necessário alterar o comportamento do *Motor ProHChecker*, que implementa a essência do método apresentado neste trabalho.

Atualmente, a ferramenta pode ser configurada para utilizar tanto funções de penalidade quadrática, quanto exponenciais, e quatro diferentes algoritmos de otimização estão disponíveis, podendo ser escolhidos pelo usuário. São eles: 1) a adaptação do algoritmo de Quasi-Newton, LBFGS (OKAZAKI, 2010); 2) a busca binária desenvolvida por Knowles e Hughes (2005); 3) o algoritmo SASS (NOLLE; BLAND, 2012); e 4) o algoritmo *Hill-Climbing* (RUSSELL; NORVIG, 2002, p. 111). Dentre estes, o que apresenta resultados melhores, em geral, é o SASS. O algoritmo de busca binária de Knowles e Hughes (2005) é um algoritmo exaustivo que também apresenta resultados muito bons, mas que requer muita memória para ser executado.

Algumas sugestões para trabalhos futuros que envolvam a ferramenta ProHChecker são apresentados na seção 7.2.

7.1.3 Sobre os Resultados dos Experimentos

Os resultados da aplicação do método, apresentados no capítulo 6 foram relevantes do ponto de vista do desempenho do método. No primeiro experimento, onde um sistema simples sem memória foi validado, foi mostrado que a ferramenta ProHChecker e, consequentemente, o método proposto neste trabalho, podem ser aplicados facilmente a sistemas onde a validação aleatória comum, e mesmo o CBMC, falham. Apesar de ter sido escolhido apenas por seu caráter ilustrativo, este experimento apresentou um grau de dificuldade inesperado para as outras abordagens de verificação, mas foi facilmente resolvido pelo ProHChecker, que apresentou um ganho de 40 vezes no tempo para derivação do contraexemplo, quando comparado ao CBMC, que só conseguiu verificar o DUV quando a faixa de valores de entrada foi extremamente reduzida.

O exemplo do MDC de Euclides, do segundo experimento, foi utilizado para mostrar que alguns sistemas que, a princípio, são considerados sistemas com memória, podem ser transformados em sistemas sem memória a partir de uma simples manipulação de seus sinais. Neste experimento, a diferença de eficiência entre a ferramenta ProHChecker e a simulação aleatória ficou evidente. O ganho mínimo da utilização do ProHChecker em relação à simulação aleatória para as asserções validadas foi de quase mil vezes, compor-

tamento que não se repetiu na comparação do ProHChecker com o CBMC, que atingiram resultados equiparáveis. No entanto, o CBMC não verificou o código original do DUV, mas sim uma versão traduzida manualmente para ANSI-C. Assim, é de se esperar que estes resultados pendam em direção ao método proposto caso o CBMC seja utilizado para verificar o módulo original pois, como sugerem os trabalhos de Chou et al. (2012), Cimatti et al. (2011a) e Grosse, Le e Drechsler (2010), o escalonador do SystemC deve ser incluído na instância do problema de BMC, o que invariavelmente resultará na perda de desempenho da verificação por meio do CBMC.

O estudo de caso do filtro IIR digital, apresentado na seção 6.3, mostrou o real potencial do método e da ferramenta descritos neste trabalho. Apesar do algoritmo do filtro não ser complexo do ponto de vista de linhas de código, seu número de estados que revelam faltas é muito pequeno (26) quando comparado ao tamanho do espaço de estados, que é de 1200 bilhões, considerando somente as sequências de comprimento seis. Estudos anteriores, apresentados por Cox, Sankaranarayanan e Chang (2012) mostraram que métodos baseados em simulação não são capazes de encontrar estas condições de falta de modo eficiente, uma vez que elas são muito localizadas. Neste caso, o método se mostrou eficiente, derivando contraexemplos mais rapidamente do que a geração aleatória e o CBMC. Da mesma forma que no exemplo do MDC, o CBMC verificou somente uma versão do filtro digital traduzida manualmente para ANSI-C e, desta forma, espera-se que seus números piorem se o escalonador do SystemC seja incluído na instância do problema de BMC.

O método proposto neste trabalho também se revelou comedido em sua utilização de memória. Em um segundo experimento do mesmo estudo de caso, o filtro digital foi implementado dentro de um sistema embutidos cuja unidade de computação possuía apenas 512 *Bytes* de memória RAM disponíveis, um período de *clock* reduzido e nenhum suporte de *hardware* para multiplicações e divisões. Mesmo com estas restrições, o *framework* desenvolvido a partir do método foi capaz de derivar um contraexemplo em 144 segundos, em média, enquanto a simulação aleatória teve um tempo médio estimado de 276 dias. Assim, para a validação *in-loco*, o método oferecido neste trabalho passa a ser a única alternativa viável.

7.2 Trabalhos Futuros

Considerando a discussão elaborada na seção 7.1, pode-se enumerar trabalhos futuros, elaborados a partir da presente pesquisa. Para uma melhor perspectiva sobre as contribuições de cada trabalho, estes foram divididos em três grupos. O primeiro engloba os trabalhos com potencial para gerar teses de doutoramento, que envolvem uma análise mais profunda acerca das soluções necessárias. O segundo grupo corresponde aos trabalhos com potencial para serem pesquisados por estudantes de nível de mestrado, que apresentam uma aplicação mais direta dos resultados desta tese, porém ainda inovadora e relevante. O último grupo envolve os trabalhos que requerem um estudo preliminar para analisar sua viabilidade, antes de poderem ser considerados potenciais trabalhos futuros.

7.2.1 Trabalhos com Potencial para Nível de Doutorado

O primeiro trabalho com potencial para nível de doutoramento e, a princípio, o que apresentaria os resultados mais relevantes é a derivação de um método para determinar a probabilidade de existir um contraexemplo da propriedade sendo validada, dado que nenhum contraexemplo foi encontrado durante a busca realizada pelo método de DPC heurística proposto. Esta é uma pergunta relevante, pois permite que o método seja integrado de modo eficiente dentro do ciclo de projeto de CIs. A hipótese a ser testada neste trabalho futuro é se é possível inferir esta probabilidade a partir da topologia da FCBA, que seria analisada durante a busca. Uma possível linha de trabalho seria empregar métodos de otimização baseados em enxame, como é o caso do SASS, analisando-se a dispersão das partículas após a busca, assim como os valores encontrados por cada uma das partículas. Referência preliminares para esta pesquisa são (SCHUTTE; HAFTKA; FREGLY, 2007), (KRYZHANOVSKY; MAGOMEDOV; FONAREV, 2006) e (FINCH; MENDELL; THODE HENRY C., 1989).

Um segundo trabalho neste grupo é modificar a ferramenta ProHChecker para que seja realizada uma busca exaustiva do espaço de estados do sistema utilizando a heurística derivada no capítulo 4 para priorizar a direção da busca. Apesar do tempo para se finalizar uma busca completa em problemas reais ser, em geral, proibitivo, uma busca heurística poderia priorizar os estados mais próximos de violar as prioridades, potencialmente, validando estes estados primeiro. Assim, um método baseado nesta estratégia

tem potencial para derivar contraexemplos, caso eles existam, em um tempo médio menor quando comparado aos métodos aleatórios. Além disso, como este método valida regiões com maior potencial de violar as propriedades primeiro, é de se esperar que N avaliações empregando esta técnica sejam mais eficazes do que as mesmas N avaliações usando geração aleatória de estímulos. Uma pergunta que pode ser respondida ao fim desta pesquisa é se é possível derivar métricas de cobertura de teste que se baseiem na heurística definida no capítulo 4. Problemas a serem enfrentados incluem manter uma estrutura de dados que permita armazenar de forma eficiente, do ponto de vista de tempo e espaço, os estados visitados durante a busca, para evitar que estes sejam repetidos.

7.2.2 Trabalhos com Potencial para Nível de Mestrado

Trabalhos futuros de nível de mestrado desdobram-se diretamente das limitações da ferramenta e do método proposto neste trabalho. O primeiro deles é finalizar a implementação da ferramenta ProHChecker, adicionando as capacidades para a geração automática do *Driver* e do *Monitor*. Dentro deste trabalho inclui-se o desenvolvimento de uma ferramenta que permita traduzir uma linguagem de descrição de propriedades, como a PSL, para os monitores necessários para a ferramenta. Assim, uma análise detalhada da sintaxe e da semântica da linguagem se faz necessária para que possam ser relacionadas à sintaxe e semântica das fórmulas da LTLE, empregadas neste trabalho.

Um segundo trabalho futuro seria aplicar a ferramenta ProHChecker em um sistema de nível industrial. Apesar de parecer muito direto, este trabalho esbarra em uma limitação encontrada no desenvolvimento desta tese: os *benchmarks* encontrados na literatura são, basicamente, sistemas dominados por controle, onde o potencial do método proposto não é totalmente aproveitado. Assim, em uma fase inicial deste trabalho futuro, seria necessário buscar *benchmarks* em diferentes áreas, como em processamento de sinais ou, talvez, dentro da área de AMS (*Analog and Mixed Signal*), que não foi explorada na presente pesquisa. Dentro deste trabalho também estaria incluído a implementação dos algoritmos multitarefa na ferramenta ProHChecker, para comparação de desempenho com os algoritmos atuais.

7.2.3 Temas de Pesquisa que Podem Gerar Trabalhos Futuros

Outros temas de pesquisa que podem ser desdobrados deste trabalho incluem a validação de sistemas já fabricados, ou produzidos por meio de prototipagem rápida em FPGAs (*Field Programmable Gate Array*), substituindo-se o DUV simulado por um DUV físico; e o emprego do método proposto nesta tese para a validação em ambientes puramente de *software*.

7.3 Conclusão

Neste trabalho, foi apresentado um método para a DPC heurística de sistemas descritos em alto nível de abstração que combina funções heurísticas derivadas das asserções de forma a aprimorar iterativamente os vetores de entrada com relação à sua proximidade a contraexemplos das propriedades do sistema. A derivação formal das funções heurísticas foi dada, juntamente com um teorema que mapeia o problema de validação de um sistema para um problema de otimização. Este teorema forma a base na qual o método é construído.

Um protótipo de ferramenta para automatizar a aplicação do método proposto foi também apresentado. Apesar de ainda não estar completamente implementada, os resultados apresentados foram relevantes. Usando dois algoritmos de otimização diferentes, um algoritmo baseado no método de Quasi-Newton e o algoritmo SASS, a ferramenta foi capaz de encontrar contraexemplos para asserções das propriedades em DUVs com diferentes características, de forma mais eficiente do que simulação aleatória e verificação de modelos com profundidade limitada.

Uma conclusão importante deste trabalho é que o método proposto, a princípio, não substitui a simulação aleatória nem a verificação de modelos em um ciclo de projeto de CIs. A verificação de modelos é um método completo por natureza e, desta forma, deve ser sempre adotado quando suas limitações práticas permitirem. Por outro lado, uma análise crítica da literatura e da história dos métodos de validação de circuitos integrados mostra que o custo-benefício da geração aleatória de estímulos dificilmente será superado por outra técnica de validação, principalmente nos estágios iniciais da verificação. Assim, o método proposto se encaixa no ciclo de projeto como um complemento a estas duas técnicas. Duas possibilidades para a sua aplicação devem ser consideradas. O método

pode ser aplicado no nível de sistema, depois da verificação formal dos blocos e antes antes da simulação aleatória dirigida por cobertura. A intenção é verificar a integração entre os blocos e derivar contraexemplos rapidamente, devido à sua eficiência nesta tarefa. A segunda possibilidade é que o método seja aplicado após a validação aleatória, quando a eficiência dos vetores de testes aleatórios começa a decair. A escolha dentre estas duas abordagem requer um maior número de experimentos em sistemas de porte industrial.

Assim, a partir da análise dos resultados obtidos, juntamente com as discussões oferecidas nas seções 7.1 e 7.2, conclui-se que os resultados deste trabalho oferecem uma contribuição relevante para a área de verificação de CIs, resolvendo problemas atuais e fomentando a geração de trabalhos futuros.

Referências Bibliográficas

ABRAR, S.; THIMMAPURAM, A. Functional refinement: A generic methodology for managing esl abstractions. In: . [S.l.: s.n.], 2010. p. 122 –127. ISSN 1063-9667.

ANDREWS, A.; O'FALLON, A.; CHEN, T. Rubastem: A method for testing vhdl behavioral models. *hase*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 187–196, 2004. ISSN 1530-2059.

BANERJEE, A. et al. Test generation games from formal specifications. In: *Proceedings of the 43rd annual Design Automation Conference*. New York, NY, USA: ACM, 2006. (DAC '06), p. 827–832. ISBN 1-59593-381-6. Disponível em: <http://doi.acm.org/10.1145/1146909.1147120>.

BERGAMASCHI, R. A.; COHN, J. The a to z of socs. In: *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM Press, 2002. p. 790–798. ISBN 0-7803-7607-2.

BERGERON, J. *Writing Testbenches: Functional Verification of HDL Models*. [S.l.]: Kluwer Academic Publishers, 2003.

BHADRA, J. et al. A survey of hybrid techniques for functional verification. *Design Test of Computers, IEEE*, v. 24, n. 2, p. 112 –122, march-april 2007. ISSN 0740-7475.

BIERE, A. et al. Symbolic model checking without bdds. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK, UK: Springer-Verlag, 1999. (TACAS '99), p. 193–207. ISBN 3-540-65703-7. Disponível em: <http://dl.acm.org/citation.cfm?id=646483.691738>.

BLACK, D. et al. *SystemC: From the Ground Up, Second Edition*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2009. ISBN 9780387699578.

BOULÉ, M.; ZILIC, Z. Automata-based assertion-checker synthesis of psl properties. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 13, n. 1, p. 4:1–4:21, fev. 2008. ISSN 1084-4309. Disponível em: <http://doi.acm.org/10.1145/1297666.1297670>.

BRICAUD, P. J. Ip reuse creation for system-on-a-chip design. In: *Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999*. New York, NY, USA: IEEE Computer Society, 1999. p. 395–401. ISBN 0-7803-5443-5.

Cadence Design Systems. *Cadence Design Systems*. ago. 2012. Disponível em: <http://www.cadence.com/>.

Carnegie Mellon University. *The CBMC Homepage*. 2012. Disponível em: <http://www.cs.cmu.edu/ modelcheck/cbmc>.

CARTER, H. B.; HEMMADY, S. G. *Metric Driven Design Verification: an engineer's and executive's guide to first pass success*. San Jose, CA, USA: Springer, 2007. ISBN 978-0-384-38151-0.

CASTRO MÁRQUEZ, C. I. et al. A functional verification methodology based on parameter domains for efficient input stimuli generation and coverage modeling. *J. Electron. Test.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 27, n. 4, p. 485–503, ago. 2011. ISSN 0923-8174. Disponível em: <http://dx.doi.org/10.1007/s10836-011-5225-8>.

CHANG, H. et al. *Surviving the SoC Revolution - A Guide to Platform-Based Design*. [S.l.]: Kluwer Academic Publishers, 1999.

CHOU, C.-N. et al. Symbolic model checking on systemc designs. In: *Proceedings of the 49th Annual Design Automation Conference*. New York, NY, USA: ACM, 2012. (DAC '12), p. 327–333. ISBN 978-1-4503-1199-1. Disponível em: <http://doi.acm.org/10.1145/2228360.2228421>.

CIMATTI, A. et al. Kratos - a software model checker for systemc. In: GOPALAKRISHNAN, G.; QADEER, S. (Ed.). *CAV*. Springer, 2011. (Lecture Notes in Computer Science, v. 6806), p. 310–316. ISBN 978-3-642-22109-5. Disponível em: <http://dblp.uni-trier.de/db/conf/cav/cav2011.html#CimattiGMNR11>.

CIMATTI, A. et al. *Kratos - A Software Model Checker for SystemC*. [S.l.], 2011. Disponível em: <https://es.fbk.eu/tools/kratos>.

CLAASEN, T. A. System on a chip: Changing ic design today and in the future. *IEEE Micro*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 3, p. 20–26, 2003. ISSN 0272-1732.

CLARKE, E.; EMERSON, E. Design and synthesis of synchronization skeletons using branching time temporal logic. In: KOZEN, D. (Ed.). *Logics of Programs*. Springer Berlin / Heidelberg, 1982, (Lecture Notes in Computer Science, v. 131). p. 52–71. ISBN 978-3-540-11212-9. 10.1007/BFb0025774. Disponível em: <http://dx.doi.org/10.1007/BFb0025774>.

CLARKE, E.; KROENING, D. Hardware verification using ANSI-C programs as a reference. In: *Proceedings of ASP-DAC 2003*. [S.l.]: IEEE Computer Society Press, 2003. p. 308–311. ISBN 0-7803-7659-5.

CLARKE, E.; KROENING, D.; LERDA, F. A tool for checking ANSI-C programs. In: JENSEN, K.; PODELSKI, A. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 2988), p. 168–176. ISBN 3-540-21299-X.

COUDERT, O. Two-level logic minimization: an overview. *Integr. VLSI J.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 17, n. 2, p. 97–140, out. 1994. ISSN 0167-9260. Disponível em: <http://dx.doi.org/10.1016/0167-9260(94)00007-7>.

COX, A.; SANKARANARAYANAN, S.; CHANG, B.-Y. E. A bit too precise? bounded verification of quantized digital filters. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. [S.l.: s.n.], 2012.

DIAS JÚNIOR, A. *Análise de Cobertura e Geração de Vetores de Teste para Módulos Descritos em SystemC*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, 2008.

DIAS, S. R. *SRD: Uma Ferramenta de Apoio ao Projetista de Sistemas de Hardware Utilizando a Linguagem SystemC*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, 2007.

D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. A survey of automated techniques for formal software verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, IEEE Press, Piscataway, NJ, USA, v. 27, n. 7, p. 1165–1178, jul. 2008. ISSN 0278-0070. Disponível em: <http://dx.doi.org/10.1109/TCAD.2008.923410>.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/262793.262798>.

FERRO, L.; PIERRE, L. Isis: Runtime verification of tlm platforms. In: *Specification Design Languages, 2009. FDL 2009. Forum on*. [S.l.: s.n.], 2009. p. 1 –6. ISSN 1636-9874.

FINCH, S. J.; MENDELL, N. R.; THODE HENRY C., J. Probabilistic measures of adequacy of a numerical search for a global maximum. *Journal of the American Statistical Association*, American Statistical Association, v. 84, n. 408, p. pp. 1020–1023, 1989. ISSN 01621459. Disponível em: <http://www.jstor.org/stable/2290078>.

FUJITA, M.; GHOSH, I.; PRASAD, M. *Verification Techniques for System-Level Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123706165, 9780123706164.

GAJSKI, D.; KUHN, R. New vlsi tools. *Computer*, v. 16, n. 12, p. 11 –14, dec. 1983. ISSN 0018-9162.

GAJSKI, D. D. et al. Embedded tutorial: essential issues for ip reuse. In: *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*. New York, NY, USA: ACM Press, 2000. p. 37–42. ISBN 0-7803-5974-7.

GALLAGHER, M. J.; NARASIMHAN, V. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 8, p. 473–484, 1997. ISSN 0098-5589.

GALTON, A. Temporal logic. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Fall 2008. [s.n.], 2008. Disponível em: <http://plato.stanford.edu/archives/fall2008/entries/logic-temporal/>.

GLASS, R. L. Persistent software errors. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 7, n. 2, p. 162–168, 1981.

GNU. *GCC, the GNU Compiler Collection*. 2010. Disponível em: <http://gcc.gnu.org/>.

GROSSE, D.; DRECHSLER. *Quality-Driven SystemC Designs*. [S.l.]: Springer, 2010. Hardcover.

GROSSE, D.; LE, H.; DRECHSLER, R. Proving transaction and system-level properties of untimed systemc tlm designs. In: *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*. [S.l.: s.n.], 2010. p. 113 –122.

GRÖTKER, T. et al. *System Design with SystemC*. [S.l.]: Kluwer Academic Publishers, 2002.

HABIBI, A.; GAWANMEH, A.; TAHAR, S. Assertion based verification of psl for systemc designs. In: *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*. [S.l.: s.n.], 2004. p. 177 – 180.

HABIBI, A.; TAHAR, S. Towards an efficient assertion based verification of systemc designs. In: *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*. [S.l.: s.n.], 2004. p. 19 – 22. ISSN 1552-6674.

HAYKIN, S.; VEEN, B. V. *Signals and Systems*. [S.l.]: John Wiley & Sons, 2002. ISBN 9780471164746.

HENZINGER, T. A. et al. Lazy abstraction. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2002. (POPL '02), p. 58–70. ISBN 1-58113-450-9. Disponível em: <http://doi.acm.org/10.1145/503272.503279>.

IEEE. IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std1850-2005)*, p. 1 –188, 6 2010.

IEEE. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, p. 1–638, 9 2012.

KATAYAMA, T.; FURUKAWA, Z.; USHIJIMA, K. A method for structural testing of ada concurrent programs using the event interactions graph. In: *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 1996. p. 355. ISBN 0-8186-7638-8.

KEUTZER, K. et al. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 19, n. 12, p. 1523 –1543, dec. 2000. ISSN 0278-0070.

KNOWLES, J.; HUGHES, E. J. Multiobjective optimization on a budget of 250 evaluations. In: *Evolutionary Multi-Criterion Optimization. Third International Conference, EMO 2005*. [S.l.]: Springer, 2005. p. 176–190.

KROPF, T. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. 1st. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN 3540654453.

KRYZHANOVSKY, B.; MAGOMEDOV, B.; FONAREV, A. On the probability of finding local minima in optimization problems. In: *Neural Networks, 2006. IJCNN '06. International Joint Conference on*. [S.l.: s.n.], 2006. p. 3243 –3248.

LI, J. J.; WEISS, D.; YEE, H. Code-coverage guided prioritized test generation. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 48, n. 12, p. 1187–1198, 2006. ISSN 0950-5849.

MALAIYA, Y. K. et al. The relationship between test coverage and reliability. In: *Proceedings of 1994 International Symposium On Software Reliability Engineering*. [S.l.: s.n.], 1994. p. 186–195.

Massachusetts Institute of Technology. *The Cilk Project*. out. 2012. Disponível em: <http://supertech.csail.mit.edu/cilk/>.

Mentor Graphics. *The "EDA" Technology Leader*. ago. 2012. Disponível em: <http://www.mentor.com/>.

METTA, R. Verifying code and its optimizations: An experience report. In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2011. (ICSTW '11), p. 578–583. ISBN 978-0-7695-4345-1. Disponível em: <http://dx.doi.org/10.1109/ICSTW.2011.44>.

Microsoft Coorporation. *AsmL: Abstract State Machine Language*. 2011. Disponível em: <http://research.microsoft.com/en-us/projects/asml/>.

MILNER, R. *An algebraic definition of simulation between programs*. Stanford, CA, USA, 1971.

NOCEDAL, J.; WRIGHT, S. J. *Numerical Optimization*. [S.l.]: Springer, 1999. Hardcover.

NOLLE, L.; BLAND, J. Self-adaptive stepsize search for automatic optimal design. *Knowledge-Based Systems*, v. 29, n. 0, p. 75 – 82, 2012. ISSN 0950-7051.

NTAFOS, S. C. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 14, n. 6, p. 868–874, 1988. ISSN 0098-5589.

ODDOS, Y. et al. Mygen: automata-based on-line test generator for assertion-based verification. In: *Proceedings of the 19th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2009. (GLSVLSI '09), p. 75–80. ISBN 978-1-60558-522-2. Disponível em: <http://doi.acm.org/10.1145/1531542.1531563>.

OKAZAKI, N. *libLBFGS: a library of Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)*. 2010. Disponível em: <http://www.chokkan.org/software/liblbfgs/>.

PAPA, R. N. *Geração de Especificações Executáveis para Projeto de Módulos para Sistemas em C̈hips¨*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, 2006.

PARHAMI, B. Defect, fault, error,..., or failure? *IEEE Transactions on Reliability*, Lafayette, CO, USA, v. 46, n. 4, p. 450–451, 1997.

PIERRE, L.; FERRO, L. A tractable and fast method for monitoring systemc tlm specifications. *Computers, IEEE Transactions on*, v. 57, n. 10, p. 1346 –1356, oct. 2008. ISSN 0018-9340.

QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in cesar. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1982. p. 337–351. ISBN 3-540-11494-7. Disponível em: <http://dl.acm.org/citation.cfm?id=647325.721668>.

RANJAN, R. K.; COELHO, C.; SKALBERG, S. Beyond verification: leveraging formal for debugging. In: *Proceedings of the 46th Annual Design Automation Conference*. New York, NY, USA: ACM, 2009. (DAC '09), p. 648–651. ISBN 978-1-60558-497-3.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002. Hardcover. ISBN 0137903952. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0137903952>.

SCHUTTE, J. F.; HAFTKA, R. T.; FREGLY, B. J. Improved global convergence probability using multiple independent optimizations. *International Journal for Numerical Methods in Engineering*, John Wiley & Sons, Ltd., v. 71, n. 6, p. 678–702, 2007. ISSN 1097-0207. Disponível em: <http://dx.doi.org/10.1002/nme.1960>.

SILVA, E. L. da; MENEZES, E. M. *Metodologia da pesquisa e elaboração de dissertação*. 4a.. ed. Florianópolis, SC, Brasil: Universidade Federal de Santa Catarina, 2005.

SILVA JR, D. C. *Compreehensive Framework for the Specification of Hardware/Software Systems*. Tese (Doutorado) — University of Southern California, 2001.

SWOYER, C.; ORILIA, F. Properties. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Winter 2011. [s.n.], 2011. Disponível em: <http://plato.stanford.edu/archives/win2011/entries/properties/>.

TASIRAN, S.; KEUTZER, K. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 18, n. 4, p. 36–45, 2001. ISSN 0740-7475.

Texas Instruments. *MSP430 Launchpad - TI E2E Community*. jun. 2012. Disponível em: <http://e2e.ti.com/group/msp430launchpad/w/default.aspx>.

TONG, J. G.; BOULÉ, M.; ZILIC, Z. Defining and providing coverage for assertion-based dynamic verification. *J. Electron. Test.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 26, n. 2, p. 211–225, abr. 2010. ISSN 0923-8174. Disponível em: <http://dx.doi.org/10.1007/s10836-010-5148-9>.

UGARTE, I.; SANCHEZ, P. Assertion-based verification of behavioral descriptions with non-linear solver. In: *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*. [S.l.: s.n.], 2006. p. 61 –68. ISSN 1552-6674.

VAHID, F.; GIVARGIS, T. *Embedded System Design: a unified hardware/software introduction*. Danvers, MA, USA: John Wiley & Sons,Inc., 2002. ISBN 978-0-471-38678-0.

VEMURI, R.; KALYANARAMAN, R. Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming. *IEEE Trans. Very Large Scale Integr. Syst.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 3, n. 2, p. 201–214, 1995. ISSN 1063-8210.

VIEIRA, N. J. *Introdução aos Fundamentos da Computação: linguagens e máquinas*. 1. ed. São Paulo, SP, Brasil: Pioneira Thomson Learning, 2006. ISBN 978-788522105083.

WILE, B.; GOSS, J.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0127518037.

Apêndice A

Código Fonte do Bloco Monitor do Estudo de Caso do Filtro IIR

```
1  /*************************************************************
2  File: npPropMonitor.cpp
3  Title: npPropMonitor
4  Objective: Property Monitor of IIR filter
5  *************************************************************/
6
7  // Lib Includes
8  #include "npPropMonitor.h"
9  #include <systemc.h>
10 #include "pcPenaltyControl.h"
11 #include "pcExponentialPenalty.h"
12 #include "pcLinearPenalty.h"
13 #include "pcPolynomialPenalty.h"
14
15 // Constants
16 #define HIGH_LIMIT 64
17 #define LOW_LIMIT -65
18 #define INPUT_HIGH_LIMIT 51
19 #define INPUT_LOW_LIMIT -51
20
21 /*************************************************************
22 Function: compute()
23 Description: Execute the inputWrapper
24 *************************************************************/
25 void npPMon::compute()
26 {
27
28         pcPolynomialPenalty ep;
29
30         while (true)
31         {
32                 np_input_strobe.read();
33                 np_basefp penalty = 0.0;
34                 sc_int<7>* inputX = np_input_x.read();
35                 sc_int<8>* inputY = np_input_y.read();
36
37                 np_basefp temp = 0.0;
38
39
40
```

```
41                  // SUM i from 0 to N - 1
42                  for (unsigned i = 0; i < IIR_CYCLES; ++i)
43                  {
44                          // hge(x[i], -51)
45                          temp += ep.funcGE(inputX[i], INPUT_LOW_LIMIT, 2);
46                          // hge(51,x[i])
47                          temp += ep.funcGE(INPUT_HIGH_LIMIT, inputX[i], 2);
48                  }
49                  penalty += temp;
50
51                  temp = NP_MAX_FP_VALUE;
52                  // MIN
53                  for (unsigned i = 0; i < IIR_CYCLES; ++i)
54                  {
55                          // MIN (temp, hgt(-64,y[i]))
56                          temp = std::min(temp,
57                                 ep.funcGT(LOW_LIMIT, inputY[i], 2));
58                          // MIN (temp, hgt(y[i],63))
59                          temp = std::min(temp,
60                                 ep.funcGT( inputY[i], HIGH_LIMIT, 2));
61                  }
62
63                  penalty += temp;
64
65                  np_penalty.write(penalty);
66
67          }
68
69  }
```