



UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA
ELÉTRICA

APLICANDO VERIFICAÇÃO DE MODELOS BASEADA NAS
TEORIAS DO MÓDULO DA SATISFABILIDADE PARA O
PARTICIONAMENTO DE HARDWARE/SOFTWARE EM
SISTEMAS EMBARCADOS

ALESSANDRO BEZERRA TRINDADE

MANAUS

2015

UNIVERSIDADE FEDERAL DO AMAZONAS

FACULDADE DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ALESSANDRO BEZERRA TRINDADE

APLICANDO VERIFICAÇÃO DE MODELOS BASEADA NAS TEORIAS
DO MÓDULO DA SATISFABILIDADE PARA O PARTICIONAMENTO DE
HARDWARE/SOFTWARE EM SISTEMAS EMBARCADOS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Departamento de Eletrônica e Computação da Universidade Federal do Amazonas, como requisito para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração Controle e Automação de Sistemas.

Orientador: Prof. PhD. Lucas Carvalho Cordeiro

MANAUS

2015

Ficha Catalográfica elaborada por Suely O. Moraes - CRB 11/365

T833a Trindade, Alessandro Bezerra.

Aplicando verificação de modelos baseada nas teorias do módulo da satisfabilidade para o particionamento de hardware/software em sistemas embarcados. / Alessandro Bezerra Trindade. – Manaus: UFAM, 2015.

84 p.

Dissertação (Mestrado em Engenharia Elétrica), Universidade Federal do Amazonas, Faculdade de Tecnologia, Programa de Pós-Graduação em Engenharia Elétrica, 2015.

1. Hardware – Coprojeto. 2. Software. 3. Otimização. I. Cordeiro, Lucas Carvalho.
II. Título.

CDU 004.03

ALESSANDRO BEZERRA TRINDADE

APLICANDO VERIFICAÇÃO DE MODELOS BASEADA NAS TEORIAS DO
MÓDULO DA SATISFABILIDADE PARA O PARTICIONAMENTO DE
HARDWARE/ SOFTWARE EM SISTEMAS EM BARCADOS

Dissertação apresentada ao Programa de Pós-Graduação
em Engenharia Elétrica da Universidade Federal do
Amazonas, como requisito parcial para obtenção do
título de Mestre em Engenharia Elétrica na área de
concentração Controle e Automação de Sistemas.

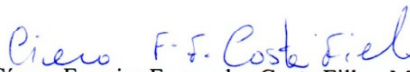
Aprovado em 09 de Fevereiro de 2015.

BANCA EXAMINADORA



Prof. Dr. Lucas Carvalho Cordeiro, Presidente

Universidade Federal do Amazonas- UFAM



Prof. Dr. Cicero Ferreira Fernandes Costa Filho, Membro

Universidade Federal do Amazonas- UFAM



Prof. Dra. Patrícia Nascimento Pena, Membro

Universidade Federal de Minas Gerais-UFMG

Resumo

Quando se realiza um coprojeto de hardware/software para sistemas embarcados, emerge o problema de se decidir qual função do sistema deve ser implementada em *hardware* (HW) ou em *software* (SW). Este tipo de problema recebe o nome de particionamento de HW/SW. Na última década, um esforço significativo de pesquisa tem sido empregado nesta área. Neste trabalho, são apresentadas duas novas abordagens para resolver o problema de particionamento de HW/SW usando técnicas de verificação formal baseadas nas teorias do módulo da satisfabilidade (SMT). São comparados os resultados obtidos com a tradicional técnica de programação linear inteira (ILP) e com o método moderno de otimização por algoritmo genético (GA). O objetivo é demonstrar, com os resultados empíricos, que as técnicas de verificação de modelos podem ser efetivas, em casos particulares, para encontrar a solução ótima do problema de particionamento de HW/SW usando um verificador de modelos baseado no solucionador SMT, quando comparado com técnicas tradicionais.

Palavras-chave: coprojeto hardware/software; sistemas embarcados; particionamento; programação linear inteira; algoritmo genético; verificação de modelos

Abstract

When performing hardware/software co-design for embedded systems, does emerge the problem of allocating properly which functions of the system should be implemented in hardware (HW) or in software (SW). This problem is known as HW/SW partitioning and in the last ten years, a significant research effort has been carried out in this area. In this proposed project, we present two new approaches to solve the HW/SW partitioning problem by using SMT-based verification techniques, and comparing the results using the traditional technique of Integer Linear Programming (ILP) and a modern method of optimization by Genetic Algorithm (GA). The goal is to show with experimental results that model checking techniques can be effective, in particular cases, to find the optimal solution of the HW/SW partitioning problem using a state-of-the-art model checker based on Satisfiability Modulo Theories (SMT) solvers, when compared to the traditional techniques.

Keywords: hardware/software co-design; embedded systems; partitioning; integer linear programming; genetic algorithm; model checking

Lista de Abreviaturas e Siglas

ASIC – *application-specific integrated circuit*, considerado como hardware neste trabalho

BMC – *bounded model checker* ou verificador de modelo limitado

BSD – *Berkeley software distribution*

CFG – *control flow graph* ou grafo de fluxo de controle

CBMC – *bounded model checker for ANSI-C*

CDCL – *conflict driven clause learning*, algoritmo de busca em árvore binária

CNF – *conjunctive normal form* ou forma normal conjuntiva

CPU – *central processing unit* ou unidade central de processamento de um microprocessador de propósito geral

CSP – *constraints satisfaction problems* ou problemas de satisfação de restrições

CTL – *computation tree logic* ou lógica de árvore computacional

DAC – *directed acyclic graph* ou grafo acíclico orientado

DFS – *depth first search*, ramificação em profundidade na busca em árvores binárias

DPLL – algoritmo Davis-Putnam-Logemann-Loveland de busca em árvore binária

E – matriz de incidência transposta do grafo que representa o sistema a ser particionado

ESBMC – *efficient SMT-based bounded model checker*

ESBMC-1 – algoritmo de ESBMC que usa modelagem matemática com variáveis auxiliares

ESBMC-2 – algoritmo de ESBMC que usa modelagem matemática sem variáveis auxiliares

FPGA – *field-programmable gate array*, considerado como hardware neste trabalho

GA – *genetic algorithm* ou algoritmo genético

HSCD – *hardware/software co-design* ou coprojeto de hardware/software

HW – hardware

IEEE – Institute of Electrical and Electronics Engineers

ILP – *integer linear programming* ou programação linear inteira

LP – *linear programming* ou programação linear

lb – *lower bound* ou limite inferior das variáveis de solução do algoritmo de GA

NP-hard – *non-deterministic polynomial-time hard*, na teoria da complexidade computacional, engloba uma classe de problemas que, informalmente, são tão difíceis de resolver quanto os mais difíceis de tempo polinomial não determinístico

PE – *processing engines* ou máquinas de processamento, podem ser CPUs, ASICs ou FPGAs

PL – *propositional logic* ou lógica proposicional

QBF – *quantified Boolean formulae* ou formulas booleanas quantificadas

SAT – satisfabilidade booleana ou proposicional

SMT – *satisfiability modulo theories* ou teorias do módulo da satisfabilidade

SOC – *system on chip* ou *system on a chip*

SSA – *static single assignment* ou designação única estática

SW – software

TL – *temporal logic* ou lógica temporal

TO – *time out* ou estouro do limite de tempo

ub – *upper bound* ou limite superior das variáveis de solução do algoritmo de GA

VC – *verification condition*, uma fórmula lógica pura cujo resultado de validação confere a correteza de um programa com relação a sua especificação

VSIDS - *variable state independent decaying sum*

Lista de Ilustrações

Figura 1 - Uma árvore binária não ordenada.....	11
Figura 2 - Grafo direcionado	12
Figura 3 - Sistema de transições de uma máquina de vender bebidas.....	13
Figura 4 - Esquema típico de um verificador de modelo	19
Figura 5. Visão geral do BMC.....	21
Figura 6 - Exemplo de aplicação da técnica BMC	22
Figura 7 - Visão geral do ESBMC	24
Figura 8 - Etapas de conversão de um código em Linguagem C no ESBMC.....	26
Figura 9 - Restrições e propriedades a serem satisfeitas, geradas à partir do código da Figura 8	26
Figura 10 - Pseudocódigo do Algoritmo DPLL	30
Figura 11 - Árvore de busca resultante da enumeração de todos os possíveis estados das variáveis A, B e C.....	31
Figura 12 - Árvore de busca resultante da enumeração de todos os estados das variáveis A, B e C, explicitando designações que não são modelos para a CNF do exemplo.....	31
Figura 13 - Pseudocódigo do Algoritmo CDCL.....	34
Figura 14 - Comparativo de ramificação: DPLL e CDCL	35
Figura 15 - Arquitetura SOC com múltiplos núcleos.....	37
Figura 16 - Grafo direcionado do sistema a ser particionado.....	39
Figura 17 - Árvore binária para o exemplo de particionamento com 5 variáveis	43
Figura 18 - Pseudocódigo de um programa no MATLAB para resolver ILP	44
Figura 19 - População inicial de um problema a ser resolvido com GA.....	47
Figura 20 - Os três tipos de filhos da função GA do MATLAB	48
Figura 21 - Filhos resultantes da técnica de GA.....	49
Figura 22 - Ilustração das gerações da técnica de GA e sua aproximação a um valor ótimo ..	50
Figura 23 - Pseudocódigo do programa do MATLAB para o GA	51
Figura 24 - Pseudocódigo do ESBMC Algoritmo 1 (ESBMC-1)	54
Figura 25. Pseudocódigo do ESBMC Algoritmo 2 (ESBMC-2)	55

Lista de Tabelas

Tabela 1 - Comparativo resumido dos principais trabalhos relacionados	10
Tabela 2 - Resultados empíricos de S_0 com 10, 25, 50 e 100 nodos	57
Tabela 3 - Descrição dos <i>benchmarks</i> usados	60
Tabela 4 - Testes empíricos com os <i>benchmarks</i>	60

Sumário

1	Introdução.....	1
1.1	Descrição do problema	4
1.2	Objetivos.....	5
1.3	Contribuições	6
1.4	Trabalhos relacionados	6
1.4.1	Particionamento de hardware e software.....	7
1.4.2	Verificação de modelos	8
1.4.3	Quadro comparativo	9
1.5	Organização da dissertação.....	10
2	Fundamentação teórica.....	11
2.1	Definições Básicas	11
2.2	Otimização	14
2.3	Verificação de hardware e software.....	15
2.4	Verificação de modelos	18
2.5	BMC – Verificação de modelo limitado.....	20
2.6	ESBMC - Verificador de modelo eficiente baseado em teorias do módulo da satisfabilidade	23
2.7	CBMC x ESBMC	27
2.8	Solucionadores SAT	27
2.9	Solucionadores SMT.....	28
2.10	Algoritmo DPLL.....	29
2.11	Algoritmo CDCL.....	32
2.12	Modelagem matemática informal do coprojetado de primeira geração	35
2.13	Modelagem matemática formal do coprojetado de primeira geração	36
2.14	Coprojetado de segunda e terceira gerações.....	37
2.15	Resumo	37

3	Particionamento de hardware/software para sistemas embarcados.....	39
3.1	Aplicando o modelo formal a um exemplo	39
3.2	Algoritmo baseado em ILP: programação inteira binária.....	42
3.3	Algoritmo genético	45
3.4	Análise do problema de particionamento usando ESBMC	51
3.5	Resumo	55
4	Resultados experimentais.....	56
4.1	Configuração experimental.....	56
4.2	Mudando o valor de S_0	57
4.3	Testes de <i>benchmark</i>	59
4.4	Considerações sobre os resultados.....	61
4.5	Resumo	61
5	Conclusões	63
5.1	Trabalhos futuros	65
	Referências bibliográficas	66
	Apêndice A: Publicações.....	72

1 Introdução

A complexidade dos sistemas computacionais está ficando cada vez maior com o passar do tempo, devido principalmente ao aumento do número de funcionalidades que devem ser implementadas. A demanda para encurtar o tempo de um produto chegar ao mercado, aliada ao fato de que os produtos precisam ser funcionalmente corretos, baratos, rápidos e confiáveis representam problemas desafiadores para os projetistas de sistemas embarcados. Particularmente em sistema móveis, onde tamanho, dissipação de calor e consumo de energia representam restrições quando estão sendo projetados.

Sistema computacional neste contexto significa ter partes dos componentes implementados em hardware e outras partes em software. Um exemplo de hardware seria um ASIC (*application-specific integrated circuit*) ou qualquer outro processador de propósito único que é 100% customizado e que representa a implementação de um algoritmo completamente em hardware. Um exemplo de software seria um programa que executa em um processador de propósito geral, como os processadores x86, MIPS e ARM.

Por um lado, componentes em hardware são usualmente muito mais rápidos do que os de software, entretanto são significativamente mais caros. Os componentes de software, por outro lado, são mais baratos de criar e manter, mas são mais lentos.

Durante o coprojeto de hardware/software (HSCD) muitas questões são tratadas, tais como modelagem, cossíntese e cossimulação, mas o passo mais crítico do HSCD é o particionamento. Nesta fase, o desafio é escolher quais componentes devem ser colocados em hardware e quais em software, para atendimento das restrições de desempenho e preço.

Na chamada primeira geração de coprojeto, a plataforma que predominou até o início dos anos 2000 foi a de uma única unidade central de processamento (CPU) e de um único ASIC com ambos se comunicando por um barramento e usando memória compartilhada ou *buffers* para implementação.

Nos anos seguintes, em virtude do limite no aumento da frequência de relógio dos processadores, novos tipos de arquitetura foram propostas com mais de um processador de propósito geral, mais de um processador de propósito único e/ou processadores de múltiplos

núcleos demandando multiprogramação e multiprocessamento. Com estes novos tipos de arquitetura, chega-se na segunda geração de coprojetos.

Inicialmente, o particionamento era feito manualmente, entretanto os sistemas a serem projetados estão se tornando cada vez mais complexos. Portanto fazê-lo manualmente é impraticável. Como resultado, muita pesquisa tem sido focada no particionamento automatizado.

Neste trabalho, um problema de particionamento será apresentado, modelado e resolvido por três técnicas distintas para cada coprojetos de primeira e segunda geração. A ideia principal é aplicar um método de verificação de modelos baseado em teorias do módulo da satisfabilidade (*satisfiability modulo theories* ou SMT, em inglês), que tem sua finalidade usual como verificador de códigos de programas, para resolver o particionamento de hardware e software. Busca-se ainda comparar a técnica proposta neste trabalho com os resultados obtidos através do emprego de técnicas tradicionais de otimização, como a programação linear inteira (ILP) e o algoritmo genético (GA).

Muitos dos trabalhos relacionados comparam os resultados de particionamento entre diferentes técnicas de otimização, podendo ser uma clássica e alguma heurística, como visto em Arató et al (2003) e Mann et al (2007). O objetivo é sempre comparar a corretude e o desempenho entre as diferentes soluções das técnicas. Aqui neste trabalho foram escolhidas as técnicas ILP e GA para comparação de resultados devido ao fato de haver mais dados empíricos e *benchmarks* disponíveis para estas técnicas do que para outras.

Em qualquer projeto de hardware e software de sistemas complexos, mais tempo é gasto em verificação do que na construção propriamente dita, conforme mostrado em Baier e Katoen (2008). Métodos formais baseados em verificação de modelos oferecem grande potencial para se obter uma verificação mais efetiva e rápida no processo de projeto. Matemática aplicada para modelagem e análise de sistemas de tecnologia da informação e comunicação caracteriza os métodos formais, cujo objetivo é estabelecer a corretude do sistema através do rigor matemático.

Nas décadas mais recentes, as pesquisas em métodos formais têm levado ao desenvolvimento de técnicas de verificação muito promissoras que facilitam a detecção precoce de defeitos.

Técnicas de verificação baseadas em modelos descrevem o possível comportamento do sistema por meio de uma matemática precisa e não ambígua. Portanto, problemas tais como incompletude, ambiguidade e inconsistência, que normalmente só são detectados em estágios avançados, podem ser detectados precocemente. Os modelos de sistema são acompanhados por algoritmos que sistematicamente exploram todos os estados do modelo de sistema. De acordo com Clarke et al (1999), os verificadores de modelos originalmente exploravam os estados de um sistema através da força bruta. Como resultado, apenas com a recente disponibilidade de computadores rápidos e com grande capacidade de armazenamento é que foi possível a adoção de verificação de modelos. Entretanto, pesquisas levaram a verificadores de modelos que evitam explorar todos os estados do sistema, mas garantindo a verificação da corretude do mesmo. Estes utilizam fórmulas lógicas booleanas para representarem as transições do sistema e utilizam a satisfabilidade booleana (SAT) para verificar a violação de propriedades.

Similar a um programa de computador para jogar xadrez que checa todos os possíveis movimentos para dar uma resposta, um verificador de modelo, a ferramenta computacional que checa se um modelo satisfaz a uma dada propriedade, examina todas as possibilidades do cenário por meio de um procedimento sistemático.

Trazendo essa ideia para o particionamento de hardware/software, um verificador de modelos poderia analisar um código formalmente especificado em uma dada linguagem onde cada componente do sistema, especificado por meio de objetivos claros (tal como minimizar o custo de hardware) e requisitos (tal como o custo de software), poderia ser explorado em todos os seus estados até uma solução ótima ser encontrada. Salvo melhor juízo ou convicção, a proposta aqui apresentada é o primeiro trabalho na qual se usa a técnica de verificação de modelos baseada em teorias do módulo da satisfabilidade para resolver um problema de particionamento. A proposta é implementada com a ferramenta ESBMC (*efficient SMT-based bounded model checker*) que é construído no *front-end* do verificador de modelos para ANSI-C chamado CBMC (*bounded model checker for ANSI-C*). A utilização de SMT, no lugar da satisfabilidade booleana (SAT) dos verificadores BMC originais, vem como uma alternativa para contornar limitações da modelagem dos sistemas, principalmente considerando-se que a complexidade destes vem aumentando cada vez mais e o método SMT possui teorias de alto nível e mais ricas que a SAT para representar os modelos.

Como maior contribuição, este projeto visa mostrar que é possível aproveitar algumas características das ferramentas de verificação formal para resolver um problema de otimização em ANSI-C, especificamente o particionamento de hardware/software, sem que nenhuma adaptação se faça na ferramenta. Uma contribuição secundária reside na avaliação do valor inicial do custo de software na definição dos requisitos da modelagem matemática do particionamento HW/SW, o que não foi feito em trabalhos similares. Busca-se definir um valor que possibilite melhorar o desempenho das técnicas, buscando diminuição do tempo de execução e da complexidade do problema.

1.1 Descrição do problema

Os sistemas embarcados hoje estão presentes em todo lugar, nos equipamentos eletrônicos em geral, em brinquedos, nos eletrodomésticos, em equipamentos médicos, em equipamentos industriais, em aviões e nos automóveis. A produção mundial desse tipo de dispositivo já passa a casa de bilhão de unidades por ano. Esse crescimento, acentuado nos anos mais recentes, tem grande impacto na funcionalidade e na redução do custo final dos produtos.

O mercado consumidor de sistemas embarcados possui algumas características marcantes: é altamente competitivo, é um mercado heterogêneo, exige produtos específicos e sempre em constante evolução. A competitividade é marcante devido ao grande número de empresas que desenvolvem equipamentos que utilizam sistemas digitais embarcados; isto exige o lançamento constante de novos produtos em um tempo cada vez menor. A heterogeneidade do mercado consumidor implica em sistemas embarcados que devem atender a requisitos bastante diferentes entre si: os sistemas embarcados de um avião possuem especificações bem diferentes de um brinquedo, por exemplo, especificações de segurança e de confiabilidade. Além de possuírem funcionalidades diferentes, as aplicações muitas vezes impõem restrições temporais e de consumo de energia aos sistemas embarcados.

Desta maneira o projetista de sistemas embarcados deve lidar com projetos de natureza bastante distintas que atendam a um conjunto de requisitos variados. Além disto, restrições de tempo real exigem dos sistemas digitais embarcados um desempenho que muitas vezes não pode ser obtido pela simples utilização de componentes de uso geral.

Os produtos que utilizam sistemas embarcados estão evoluindo cada vez mais, o que faz com que o sistema embarcado também deva ser flexível para permitir a rápida evolução

dos equipamentos que o utilizam. Finalmente o uso dos sistemas embarcados tem tornado o custo da eletrônica embarcada um fator importante no custo total dos equipamentos atualmente desenvolvidos. Como resultado, reduções no custo de desenvolvimento e produção de sistemas embarcados em escala provocam uma redução razoável no preço final dos equipamentos que os usam.

Em síntese, o projetista de sistemas digitais embarcados precisa desenvolver sistemas que sejam ao mesmo tempo confiáveis, apresentem alto desempenho e baixo consumo de energia, sejam flexíveis, de baixo custo e tudo isso em um tempo de projeto cada vez menor, o que é altamente desafiador.

O problema de coprojeto consiste de vários componentes, conforme definido por Gupta e De Micheli (1997): da especificação de um sistema, usualmente na forma comportamental, em uma representação que seja adequada para descrevê-la tanto em hardware quanto em software; do particionamento do sistema em hardware ou software; do escalonamento da execução das tarefas do sistema para atender as restrições de tempo e de consumo de energia; da modelagem do sistema por todo o processo de projeto para validar a solução e garantir que ele atinja seus objetivos e metas originais.

Mais importante ainda, como um dos principais objetivos do projetista de sistemas digitais é desenvolver projetos no menor tempo possível, esta metodologia deve ser completamente automatizada.

O objetivo do particionamento, que é uma das etapas do coprojeto e foco deste trabalho, é de determinar, de forma automática, quais elementos do sistema digital devem ser implementados em hardware e quais devem ser implementados em software.

1.2 Objetivos

O objetivo geral deste trabalho é demonstrar que é possível se utilizar a técnica de verificação de modelos baseada em teorias do módulo da satisfabilidade, usando o ESBMC (*efficient SMT-based bounded model checker*), para resolver problemas de particionamento de hardware/software de sistemas embarcados.

Os objetivos específicos são listados a seguir:

- Modelar matematicamente os problemas de coprojeto de primeira geração;

- Desenvolver algoritmos em ANSI-C para resolver o problema de particionamento com verificação de modelos, usando a ferramenta ESBMC;
- Utilizar algoritmos de programação linear inteira e genético para comparar os resultados empíricos;
- Realizar testes empíricos, usando vetores de teste próprios e *benchmarks* de terceiros, para comparar o resultado do particionamento de hardware/software por meio das técnicas ILP, GA e de verificação de modelos;
- Avaliar o impacto da escolha do valor do custo inicial de software no desempenho para obtenção da solução do particionamento de hardware/software, em todas as técnicas testadas neste trabalho.

1.3 Contribuições

O presente trabalho aborda a adoção de verificação formal, mais especificamente um método de verificação baseado em SMT para resolver o particionamento de hardware e software. Entretanto, para se chegar nesse objetivo, várias etapas são realizadas:

- Investigar e desenvolver algoritmos usando a ferramenta ESBMC para resolver o problema de particionamento;
- Realizar testes comparativos entre as técnicas de particionamento de hardware e software;
- Encontrar limites para o uso de cada técnica de particionamento de hardware e software;
- Avaliar o impacto do valor inicial do custo de software na definição dos requisitos da modelagem matemática do particionamento de hardware e software, buscando melhorar o desempenho das técnicas e, com isso, diminuição do tempo de execução e da complexidade do problema.

1.4 Trabalhos relacionados

Esta seção menciona alguns trabalhos relevantes que possuem objetivos semelhantes aos apresentados pelo método proposto neste trabalho, com o objetivo de fornecer uma noção sobre o estado da arte das técnicas atualmente utilizadas. A seção está dividida em três subseções, uma dedicada à solução do problema de particionamento de hardware e software,

outra dedicada à verificação de modelos baseada nas teorias do módulo da satisfabilidade e a terceira com uma tabela sumarizando e comparando os trabalhos que foram descritos.

1.4.1 Particionamento de hardware e software

Quando se fala em particionamento de hardware/software, as mais relevantes iniciativas são apresentadas por Arató et al (2003), Mann et al (2007) e Arató et al (2005), onde o problema é formalizado apropriadamente e alguns algoritmos são empregados para resolvê-lo. A partir da 2ª metade da década de 2000, três caminhos principais foram trilhados para resolver a otimização do particionamento de hardware/software, seja para encontrar a solução exata, o uso de heurísticas para obter respostas mais rápidas, ou soluções híbridas.

No primeiro grupo, a solução exata do problema de particionamento de hardware/software é encontrada. Aqui deve ser citado o trabalho relevante de Mann et al (2007), que desenvolveu um algoritmo *branch-and-bound* para acelerar a obtenção da solução, basicamente aprimorando a escolha das variáveis de decisão e o caminho a ser adotado para se chegar no particionamento. Wang e Zhang (2010) apresentaram um algoritmo guloso que também obtém a solução ótima global através do acréscimo de uma função que guia a busca durante o particionamento e permite analisar vários requisitos ao mesmo tempo. Sapienza et al (2013) usam múltiplos critérios para análise da decisão em uma das fases finais do projeto de sistemas embarcados, efetuando uma otimização que permite contemplar requisitos da aplicação, restrições de sistema e de projeto. O uso da verificação baseada em SMT deste trabalho faz parte desta categoria, porque encontra a solução exata do problema de particionamento. A diferença está baseada na técnica empregada para resolver o problema, na forma como as variáveis de decisão são escolhidas durante a otimização e como as soluções que atendem aos requisitos podem ser descartadas ou escolhidas como ótimas do problema, com consequências no tempo para se obter essa solução.

Outro caminho que foi desenvolvido por trabalhos similares, e tem sido o que mais apresenta contribuições, está na criação de uma heurística para acelerar o tempo de obtenção da solução. Essas técnicas estão baseadas principalmente em métodos modernos de otimização. Os trabalhos relevantes incluem os de Arató et al (2003) que usa algoritmo genético para resolver o problema de otimização. Em Bhattacharya et al (2008) a solução é obtida por meio de um algoritmo de enxame de partículas. Wang et al (2006) usaram um algoritmo de otimização por colônia de formigas. Luo et al (2010) usaram um algoritmo que mistura enxame de partículas e clonagem imune. Jianliang e Manman (2011) criaram um

algoritmo modificado de enxame de partículas. Jiang et al (2012) usaram um algoritmo genético adicionado de recozimento simulado. Jigang et al (2010) propuseram um algoritmo 1D para diminuir a complexidade do problema. As diferenças entre as soluções que usam heurísticas e a solução baseada em SMT estão em dois fatores: apenas o método ESBMC garante encontrar a solução exata, entretanto as heurísticas são mais rápidas quando a complexidade do problema aumenta.

Finalmente, existem pesquisas híbridas que juntam heurística com solução exata. A ideia, por trás destas técnicas, é usar uma heurística para acelerar alguma fase da ferramenta de solução exata. Entretanto, deve-se destacar que a solução final não é necessariamente a solução ótima global. Em contrapartida, o tempo de execução é bem menor do que as alternativas que encontram a solução exata e, dependendo da implementação, são até mais rápidos que os puramente heurísticos. Nesta categoria de soluções mistas, está a de Arató et al (2005) que emprega uma heurística de biparticionamento gráfico e que indica ainda o quão longe a solução está da ótima. Em Eimuri e Salehi (2010) um algoritmo de *branch-and-bound* é modificado com otimização por enxame de partículas. Em Huong e Binh (2012) uma otimização usando Pareto é modificada com algoritmo genético. Em Li et al (2013) um modelo de programação linear inteira mista é modificado com uma heurística criada especialmente para resolver particionamento. Comparando esta categoria com a metodologia de verificação baseada em SMT chega-se no mesmo caso do grupo anterior, ou seja, apenas a verificação baseada em SMT garante encontrar a solução exata, mas os algoritmos híbridos são mais rápidos quando a complexidade aumenta.

1.4.2 Verificação de modelos

Considerando-se a ferramenta mais conhecida de verificação de modelos, o CBMC, é notório seu destaque através de prêmios internacionais recebidos em competições devido ao seu desempenho para validar programas em ANSI-C e C++. Nos trabalhos relacionados ao CBMC, grande parte da pesquisa está focada na melhoria da ferramenta e no seu desempenho nas competições, sem endereçar, no entanto, a questão de particionamento de hardware/software ou ainda de otimização, conforme pode ser visto em Kroening e Tautschnig (2014).

Especificamente, quanto à verificação baseada em teorias do módulo da satisfabilidade (SMT), todos os trabalhos relacionados estão restritos à apresentação do modelo, suas modificações para a linguagem C/C++ e a aplicação para algoritmos multitarefa ou para

checar a corretude de sistemas embarcados. Ramalho et al (2013) apresenta um verificador de modelos limitado para programas escritos em C++, que é a evolução da versão original em ANSI-C. Cordeiro et al (2012) usam um verificador de modelos ESBMC para software embarcado em ANSI-C e Cordeiro (2011) propõe o uso de ESBMC para verificação de modelo de software multitarefa de sistemas embarcados. Entretanto, nenhum dos trabalhos relacionados focou no particionamento de hardware/software ou usou a técnica baseada em SMT para resolver otimização.

1.4.3 Quadro comparativo

A Tabela 1 mostra um quadro contendo um resumo comparativo entre os principais trabalhos relacionados à temática de particionamento de hardware e software, e de verificação de modelos, seja baseada nas teorias do módulo da satisfabilidade (SMT) ou em satisfabilidade booleana (SAT). O trabalho aqui proposto busca exatamente resolver o particionamento de hardware e software, que nada mais é que um problema de otimização, realizado através da técnica de verificação de modelos baseada nas teorias do módulo da satisfabilidade. O método proposto visa encontrar uma solução ótima global por meio da checagem implícita de um conjunto de possíveis soluções do problema de particionamento.

Trabalhos relacionados	Particionamento de HW/SW			Verificação de modelos				
	Solução exata	Heurística de busca	Híbrido	Checa código C	Checa código C++	Baseado em SMT	Baseado em SAT	Faz particionamento de HW/SW
Mann et al (2007), Wang e Zhang (2010), Sapienza et al (2013)	X							
Arató et al (2003), Wang et al (2006), Luo et al (2010), Jianliang e Manman (2011), Jiang et al (2012), Jigang et al (2010)		X						
Arató et al (2005), Eimuri e Salehi (2010), Huong e Binh (2012), Li et al (2013)			X					

Kroening e Tautschnig (2014)				X			X	
Ramalho et al (2013), Cordeiro (2011)					X	X		
Ganai e Gupta (2006)				X			X	
Trabalho proposto	X			X	X	X		X

Tabela 1 - Comparativo resumido dos principais trabalhos relacionados

FONTE: Conforme indicada.

1.5 Organização da dissertação

Neste capítulo, descreveram-se o contexto, a motivação, os objetivos deste trabalho além de terem sido apresentados os trabalhos relacionados, reunindo algumas referências sobre o tema. Os próximos capítulos deste texto estão organizados da seguinte forma:

- O Capítulo 2 faz uma breve introdução sobre o tema otimização, depois discorre sobre verificação de hardware e software, verificação de modelos, verificação de modelo limitado, verificação de modelos usando as teorias do módulo da satisfabilidade, satisfabilidade, algoritmos de busca em árvore binária, modelagem matemática informal e formal do problema de particionamento de hardware e software;
- O Capítulo 3 descreve a metodologia e os algoritmos implementados para realizar o particionamento de hardware e software, tanto através da técnica proposta no trabalho quanto por outras técnicas para que se possa fazer uma análise comparativa entre as técnicas;
- No Capítulo 4, apresentam-se as configurações utilizadas para executar os testes e os resultados dos experimentos executados, para diferentes tipos de vetores de testes, sejam eles criados tanto pelo autor do trabalho quanto com *benchmarks* disponíveis no mercado;
- Por fim, o Capítulo 5 apresenta as conclusões, destacando a importância da verificação de modelos baseada nas teorias do módulo da satisfabilidade para resolver problemas de particionamento de hardware e software.

2 Fundamentação teórica

Neste capítulo, alguns conceitos são introduzidos, como otimização, verificação de hardware e software, verificação de modelos, verificação de modelo limitado, verificação de modelo baseado nas teorias do módulo da satisfabilidade, satisfabilidade e algoritmos de busca em árvore binária. Além disto, é apresentada a modelagem matemática informal e formal do problema de particionamento de hardware e software.

2.1 Definições Básicas

Esta seção introduz os conceitos básicos que serão usados nas seções seguintes deste capítulo e por todo o trabalho.

Definição I (Árvore binária). É uma estrutura de dados, representando um modo particular de armazenamento e organização de dados para facilitar seu uso posterior, composta de nós ou nodos, onde cada nodo contém: uma referência à esquerda, uma referência à direita e um elemento de dado (valor). Na árvore binária, o nodo mais ao topo da árvore é chamado de raiz, e cada árvore possui apenas uma raiz. Além disso, os elementos associados a cada nodo são habitualmente chamados de filhos desse nodo. Os nodos sem filhos de uma árvore são chamados de folhas. Na árvore binária, cada nodo pode ter no máximo dois filhos. Um nodo pai é aquele com filhos, mas que não tem referências a seus pais. A Figura 1 ilustra uma árvore binária na qual o nodo 6 tem dois filhos, os com identificação 5 e 11, e um nodo pai, identificado como 7. O nodo raiz, o de número 2 no alto da árvore, não tem pais.

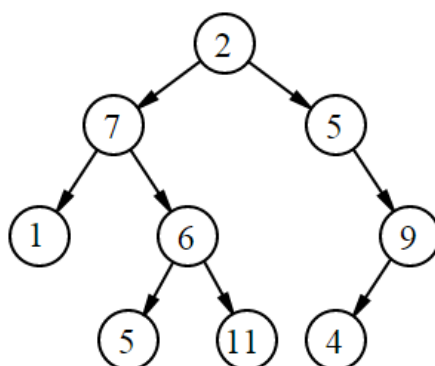


Figura 1 - Uma árvore binária não ordenada

FONTE: Autoria própria (2014).

Árvores possuem uma série de vantagens em relação a outras estruturas de dados, sendo as principais para o foco deste trabalho: árvores refletem relações estruturais entre os dados, árvores são usadas para representação hierárquica e fornecem um eficiente meio de busca. Todas essas características são importantes no trabalho aqui apresentado.

Definição II (Grafos). São estruturas, $G = (V, E)$, contendo um conjunto não vazio de objetos denominados vértices (V) ou nodos e um conjunto de pares não ordenados de V , chamados de arestas (E ou *edges*). Nesta dissertação, as arestas possuem direção (indicada por uma seta na representação gráfica), sendo denominadas de grafo direcionado ou grafo orientado, e vértices e/ou arestas apresentam um peso associado (número). A Figura 2 ilustra um grafo direcionado que será usado várias vezes neste trabalho, contendo 10 nodos e 13 arestas.

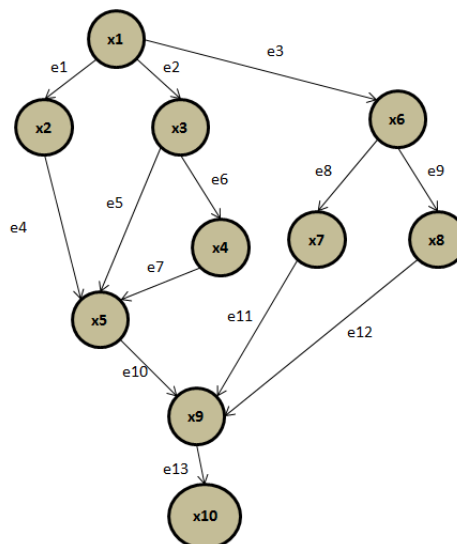


Figura 2 - Grafo direcionado

FONTE: Autoria própria (2014).

Definição III (Sistema de transição de estados ou máquina de estados). Uma máquina abstrata do sistema de transição de estados M pode ser definida como $M = (S, T, S_0)$, onde S é um conjunto de estados, $S_0 \subseteq S$ representa um conjunto de estados iniciais e $T \subseteq S \times S$ é a relação de transição, ou seja, pares de estados especificando como o sistema se move de estado para estado. As transições são identificadas como $Y = (S_i, S_{i+1}) \in T$ entre dois estados S_i e S_{i+1} com uma fórmula lógica que contém as restrições dos valores das variáveis do programa. Um sistema de transição modela o comportamento de diversos tipos de sistemas, como pode ser verificado no exemplo de uma máquina de vender bebidas da Figura 3. Neste sistema, a máquina pode entregar refrigerante ou cerveja. Os estados iniciais são representados por uma seta sem estado anterior. O espaço de estado é formado por $S =$

$\{\text{pagamento}, \text{seleciona}, \text{cerveja}, \text{refrigerante}\}$ enquanto o único estado inicial é $S_0 = \{\text{pagamento}\}$. A transição *insere_moeda* indica a ação do usuário de inserir uma moeda na máquina, e as transições *entrega_cerveja* e *entrega_refrigerante* representam as ações (da máquina) de entregar a bebida de acordo com a escolha do usuário. A transição τ representa o funcionamento interno da máquina, a partir da escolha do usuário.

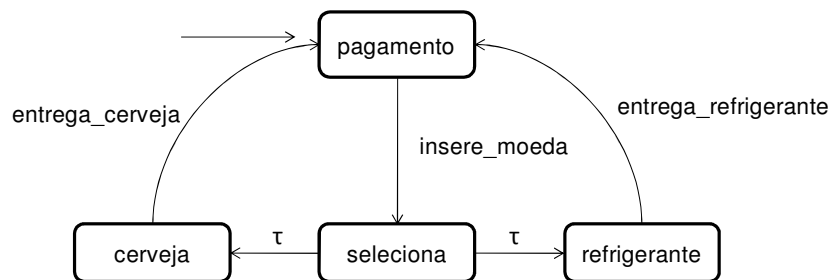


Figura 3 - Sistema de transições de uma máquina de vender bebidas

FONTE: Baier e Katoen (2008).

Em um sistema de transição de estados finitos, podem-se testar todas as possíveis combinações de entradas em um tempo finito. O que não ocorre em um sistema de estados infinitos.

Definição IV (Lógica Proposicional - PL). A sintaxe de PL consiste de símbolos e regras que permitem combinar os símbolos para construir sentenças (ou mais especificamente fórmulas) conforme Cordeiro (2011). De uma forma geral, lógica proposicional ou cálculo sentencial é uma lógica que pode assumir dois valores, partindo-se da suposição que cada sentença só pode ser verdadeira ou falsa. Um valor verdadeiro é um valor indicativo da relação de uma proposição (ou significado da sentença) para verdadeiro. Os elementos básicos de uma PL são as constantes verdadeira e falsa, além das variáveis proposicionais: x_1, x_2, \dots, x_n . Operadores lógicos (tais como conjunção \wedge , negação \neg , disjunção \vee , implicação \implies , equivalência \iff , exclusivo \oplus e expressão condicional *ite*), também conhecidos como operadores Booleanos, fornecem o poder expressivo da PL.

Definição V (Forma normal conjuntiva - CNF). Uma fórmula PL ϕ está na forma normal conjuntiva se ela consiste de uma conjunção de uma ou mais cláusulas, onde cada cláusula é uma disjunção de um ou mais literais. A CNF possui a forma $\bigwedge_i (\bigvee_j l_{ij})$, onde cada l_{ij} é um literal, ou seja, uma variável booleana v ou sua negação \bar{v} . CNF tem a vantagem de ser uma forma muito simples, o que leva a uma fácil implementação de algoritmos. A entrada de um algoritmo para checar a satisfabilidade é usualmente uma fórmula PL na forma normal

conjuntiva. Exemplo de CNF: $(A \vee B \vee \neg C) \wedge (\neg A \vee D) \wedge (B \vee C \vee D)$, que pode também ser representado como um conjunto $\Delta = \{\{A, B, \neg C\}, \{\neg A, D\}, \{B, C, D\}\}$.

Definição VI (Ramificação). Termo usado em busca em árvores (cf. **Definição I**) para resolver problemas computacionais, na qual todas as variáveis de decisão podem assumir valores (sejam binários ou não, dependendo do problema) e que define qual nodo (ou variável) será explorado durante a referida busca naquele momento, ou seja, qual valor será testado para esse nodo.

Definição VII (Retrocesso ou *backtracking*). Termo usado em busca em árvores (cf. **Definição I**) para resolver problemas computacionais, na qual todas as variáveis de decisão podem assumir valores (sejam binários ou não, dependendo do problema) e que define qual nodo (ou variável) será retomada para ramificar para outro valor após já ter sido testada para um valor anteriormente. Envolve ainda o método que efetua o abandono de uma possível solução, mesmo que parcial, quando essa não atende aos requisitos do problema.

Definição VIII (Satisfabilidade booleana ou proposicional - SAT). É o problema de determinar se existe uma interpretação que satisfaz uma dada fórmula booleana. Em outras palavras, a SAT determina se as variáveis de uma dada fórmula booleana podem ser substituídas por valores VERDADEIRO ou FALSO de tal modo que esta fórmula seja avaliada como VERDADEIRA. Se isso acontecer, diz-se que a fórmula foi satisfeita. Caso contrário, a fórmula é insatisfeita ou violada. As teorias do módulo da satisfabilidade (**SMT**) compõem uma extensão da SAT, pois permite enriquecer as fórmulas booleanas com teorias de primeira ordem, tais como as de restrições lineares, vetores, estruturas, aritmética *bit-vector*, funções não interpretadas, dentre outras. O SMT apareceu como uma alternativa para a limitação apresentada pelo SAT na tradução do tamanho das fórmulas booleanas de sistemas complexos atuais e para evitar perda de estrutura na modelagem.

2.2 Otimização

Baseado em Rao (2009), otimização é o ato de obter o melhor resultado sob certas circunstâncias. No projeto, construção e manutenção de qualquer sistema de engenharia, os engenheiros acabam tendo que tomar várias decisões tecnológicas e gerenciais em vários estágios. O objetivo maior de tais decisões pode ser tanto minimizar os esforços requeridos quanto maximizar o benefício desejado. Tanto o esforço requerido quanto o benefício

desejado em qualquer situação prática podem ser expressos como função de certas variáveis de decisão. Otimização pode ser definida como o processo de encontrar as condições que resultam nos valores máximos ou mínimos dessa função.

Não há um método único para resolver todos os problemas de otimização de forma eficiente. Por isso, vários métodos foram desenvolvidos para resolver diferentes problemas de otimização. A técnica mais conhecida é a da programação linear, que é um método de otimização aplicado para a solução de problemas cuja função objetivo e as restrições aparecem como funções lineares de suas variáveis de decisão. As equações de restrições em um problema de programação linear podem ser na forma de igualdades ou desigualdades.

Um caso particular da programação linear é a programação linear inteira (ILP), na qual as variáveis podem assumir apenas valores inteiros. Outro caso particular da programação linear é quando todas as variáveis de decisão de um problema de otimização são permitidas apenas assumirem valor de zero ou um. Este tipo de problema é conhecido como programação zero-um ou ainda programação inteira binária.

Em alguns casos, o tempo necessário para encontrar a solução usando ILP é infactível. Mesmo que se use computadores poderosos, o problema pode levar horas de execução para que a solução ótima seja alcançada. Se o problema de otimização é muito complexo, algumas heurísticas podem ser usadas para resolver o problema. A única desvantagem é que a solução encontrada por meio de heurística pode não ser um mínimo ou máximo global. O que passa a ser um ganho de desempenho torna-se uma perda em correteza da solução.

2.3 Verificação de hardware e software

Os principais métodos de validação e/ou verificação para sistemas complexos, conforme Clarke et al (1999), Clarke et al (2009) e Emerson (2008) são simulação, teste, verificação dedutiva e verificação de modelos.

Simulação e teste envolvem a execução de experimentos antes do sistema estar declarado como pronto. Enquanto se executa a simulação em uma abstração ou modelo do sistema, o teste é executado no produto real. No caso de circuitos eletrônicos, a simulação é executada na etapa do projeto do circuito, enquanto o teste é executado no próprio circuito. Em ambos os casos, estes métodos injetam sinais em determinados pontos no sistema e observam os sinais resultantes em outros pontos. Para o software, a simulação e os testes

geralmente envolvem a geração de determinadas entradas e observação das saídas correspondentes ou respostas do sistema. Estes métodos podem ser uma maneira econômica para encontrar diversos erros. Entretanto, verificar todas as interações possíveis e falhas potenciais de sistemas usando simplesmente as técnicas de simulação e teste é raramente possível ou viável. Conforme Clarke et al (1999), Emerson (2008) e Clarke et al (2004), não há como garantir de forma rigorosa a confiabilidade e a segurança desses sistemas apenas com testes ou simulações.

De acordo com Emerson (2008), devido a cobertura incompleta dos testes e das simulações, métodos alternativos foram buscados. O método mais promissor baseia-se no fato de que programas, e mais especificamente sistemas computacionais, podem ser vistos como objetos matemáticos com um funcionamento que é, em princípio, bem determinado. Daí torna-se possível especificá-lo usando lógica matemática que constitui o comportamento desejado correto. Então pode ser tentada uma prova formal ou ainda estabelecer se o programa atende as suas especificações. Este ramo de estudo foi muito explorado nas últimas cinco décadas e é muitas vezes referido como métodos formais. A verificação dedutiva e a verificação de modelos são instâncias da verificação formal, conforme apresentado por Emerson (2008).

O termo verificação dedutiva se refere normalmente ao uso de axiomas e regras de prova para demonstrar a corretude dos sistemas, conforme Clarke et al (1999). No início da pesquisa envolvendo verificação dedutiva, o foco principal estava em garantir a correção de sistemas críticos. Supunha-se que a importância do comportamento correto era tão grande que o projetista ou perito em verificação (geralmente um matemático ou um especialista em lógica) deveria gastar o tempo que fosse necessário para verificar o sistema. Inicialmente, tais provas eram construídas inteiramente à mão. Com o tempo, os pesquisadores descobriram que ferramentas de software poderiam ser desenvolvidas para forçar o uso correto de axiomas e regras de prova. Tais ferramentas poderiam também aplicar uma busca sistemática para sugerir várias maneiras de se progredir no estágio corrente da prova.

O estilo Floyd-Hoare, foi o método de verificação formal que dominou os anos de 1960, conforme Clarke et al (2009), sendo realizada manualmente. E até 1981, as ferramentas que dominaram a verificação formal foram baseadas em prova de teoremas ou pela exploração exaustiva de estados, conforme Clarke (2008), possibilitando a automatização

desta verificação, destacando-se: as redes de Petri, protocolo de verificação Bochmann e protocolo de verificação Holzmann, conforme Clarke (2008).

A importância da verificação dedutiva é amplamente reconhecida, entretanto é um processo demorado que pode ser efetuado somente por peritos treinados no raciocínio lógico e que tenham considerável experiência. Estudos de caso, conforme Clarke et al (2009), confirmaram que o método funciona bem para programas pequenos, muito embora mesmo programas pequenos poderem requerer uma prova longa. Entretanto, a verificação manual não conseguiu ser bem ampliado para programas grandes, pois as provas eram muito difíceis de ser construídas.

Uma vantagem da verificação dedutiva é que esta pode ser usada para avaliar sobre sistemas de estados infinitos (cf. **Definição III**), conforme Clarke et al (1999). E esta tarefa pode ser automatizada dentro de certos limites. E ainda, mesmo que a propriedade a ser verificada seja verdadeira, nenhum limite pode ser colocado na quantidade de tempo ou de memória necessária a se encontrar uma solução.

Segundo Clarke (2008), a verificação de modelos (ou do inglês, *model checking*) não surgiu de um vácuo histórico. Havia um importante problema que precisava ser resolvido, chamado verificação de programas concorrentes. Erros de concorrência são particularmente difíceis de encontrar por meio de testes, pois são difíceis de reproduzir.

A dificuldade de se tentar construir provas de programas, usada na verificação dedutiva, levou à busca de um caminho alternativo. E o caminho foi inspirado em lógica temporal (TL ou *temporal logic*), que é basicamente uma extensão da lógica proposicional tradicional (cf. **Definição IV**), um formalismo para descrever mudança ao longo do tempo, de acordo com Demri e Gastin (2012) e Baier e Katoen (2008). Se um programa pode ser escrito em TL, pode ser realizado por um sistema de estados finitos (cf. **Definição III**). Segundo Clarke et al (2009) e Baier e Katoen (2008) isto levou a ideia do verificador de modelos – para verificar se um grafo de estados finitos é um modelo de uma especificação TL. Ainda de acordo com Clarke et al (2009) e Baier e Katoen (2008), outra lógica muito usada é a de árvore computacional (CTL ou *computation tree logic*).

Sendo assim, no início dos anos de 1980, os pesquisadores estadunidenses Edmund Clarke e Allen Emerson propuseram a técnica *model checking*, um método para verificação automática e algorítmica de sistemas concorrentes de estados finitos (cf. **Definição III**); e de

forma independente, os pesquisadores franceses Jean Pierre Queille e Joseph Sifakis propuseram essencialmente o mesmo método. De acordo com Clarke et al (2009), no *model checking*, TL é usada para especificar o comportamento de sistemas concorrentes. Um procedimento eficiente e flexível de busca é usado para encontrar padrões temporais corretos no grafo de estados finitos. A orientação do método é fornecer um método de verificação prático.

Segundo Clarke (2008), a verificação de modelos apresenta uma série de vantagens quando comparada com outras técnicas de verificação formal, tais como: não há a necessidade de se realizar provas, a verificação é automática, é mais rápido, pode apresentar um contraexemplo para auxiliar no diagnóstico do problema detectado, não apresenta problema com a especificação apenas parcial do sistema, lógica temporal pode ser usada para descrever sistemas concorrentes.

2.4 Verificação de modelos

De acordo com Baier e Katoen (2008), verificação de modelos refere-se à técnica que dado um modelo de espaço finito de um sistema e uma propriedade formal, sistematicamente verifica se esta propriedade satisfaz (em algum determinado estado) ao modelo. A técnica é implementada por algoritmos que podem realizar verificação exaustiva automaticamente e, com isso, têm atraído muito o interesse na indústria, de acordo com Clarke (2008).

As propriedades típicas que podem ser examinadas utilizando um verificador de modelo podem ser de natureza qualitativa, tais como: O resultado gerado está de acordo com o esperado? O sistema pode chegar a uma situação de *deadlock*, ou seja, quando dois programas concorrentes estão esperando um pelo outro e, portanto, parando o sistema todo? Existem ainda as propriedades de tempo que podem ser verificadas, como por exemplo se pode um *deadlock* ocorrer em uma dada hora após o sistema ser reiniciado. O verificador de modelo requer ainda uma declaração precisa e inequívoca das propriedades a serem examinadas. Quando feito por meio de um modelo de sistema preciso, este passo leva a descoberta de inconsistências e ambiguidades na documentação informal.

O processo de verificação de modelos engloba três fases bem distintas: modelagem, execução e análise. Na modelagem, o sistema em análise é modelado através de uma linguagem de descrição e também se realiza a formalização da propriedade a ser checada

usando uma linguagem de especificação de propriedade. Modelos de sistema descrevem o comportamento de sistemas de uma forma precisa e não ambígua. Eles são normalmente expressos usando máquina de estado finito (cf. **Definição III**). Para se chegar nas máquinas de estado finito, utilizam-se linguagens de programação usuais, como C, Java e VHDL, dentre outras e, através de extensões, esses programas são convertidos automaticamente em máquinas de estado conforme Baier e Katoen (2008). Deve-se destacar que a especificação de uma propriedade determina “o que” o sistema deve fazer e “o que” não deve fazer, enquanto uma descrição de modelo trata do “como” o sistema se comporta.

Na fase de execução, o verificador de modelo avalia a validade da propriedade no modelo de sistema. E na fase de análise, se a propriedade for satisfeita, ou seja, todos os estados relevantes do sistema foram verificados e não houve violação, verifica-se a próxima propriedade (se houver). Se um estado que viola a propriedade é encontrado, o verificador de modelo fornece o contraexemplo que indica como o modelo poderia chegar nesse estado indesejado. O contraexemplo descreve um caminho de execução que leva do estado inicial do sistema até o estado que viola a propriedade investigada. Com a ajuda de um simulador, o usuário pode repetir o cenário de violação, obtendo desse jeito informação para depuração, adaptando o modelo (ou a propriedade) adequadamente, conforme descrito na Figura 4.

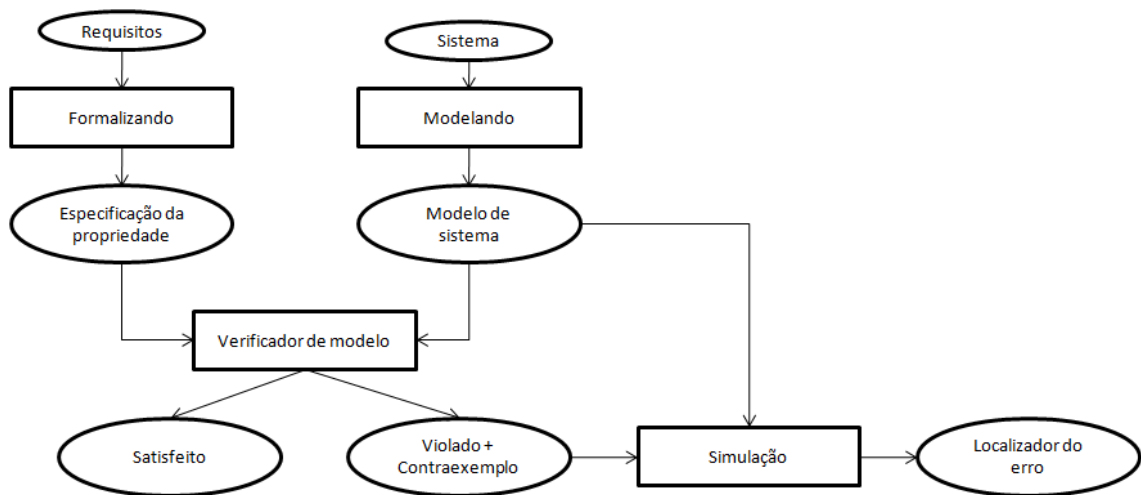


Figura 4 - Esquema típico de um verificador de modelo

FONTE: Baier e Katoen (2008).

Os verificadores de modelo têm sido aplicados com sucesso em vários sistemas de tecnologia da informação e da comunicação. Para exemplificar, também de acordo com Baier e Katoen (2008), *deadlocks* já foram detectados em sistemas de reservas de companhias aéreas, modernos protocolos de comércio eletrônico foram verificados, e muitos estudos de

padrões IEEE para comunicação de aparelhos domésticos foram aperfeiçoados no que concerne suas especificações.

Entretanto, a popularidade da verificação de modelos tem sido freada pelo problema da explosão de estados, na qual o número de estados em um sistema cresce exponencialmente de acordo com o número de componentes do sistema. Muita pesquisa tem sido dedicada para minimizar este problema.

2.5 BMC – Verificação de modelo limitado

Dentre as técnicas mais populares de verificação de modelo, há uma que combina a verificação de modelos com a solução de satisfabilidade (cf. **Definição VIII**). Esta técnica, conhecida como verificação de modelo limitado (*bounded model checking* ou BMC em inglês), conforme proposta originalmente por Biere et al (1999), representa as transições de um sistema como fórmulas lógicas booleanas e, ao invés de explorar todos os estados exaustivamente, desdobra as relações de transição até um certo limite e busca simbolicamente por violações de propriedades dentro desse limite. O BMC transforma essa busca em um problema de satisfabilidade e o resolve por meio de um solucionador comercial de SAT. Para certos tipos de problemas, a técnica BMC oferece melhorias de desempenho grandes em relação às técnicas anteriores, como mostrado por Biere (2009). BMC baseado em satisfabilidade booleana tem sido introduzido como uma técnica complementar para aliviar o problema da explosão de estados, como mostrado por Biere (2009) e Cordeiro et al (2012).

A ideia do BMC é verificar a negação de uma dada propriedade a uma certa profundidade: dado um sistema de transição de estados M , uma propriedade ϕ e um limite k , BMC desdobra o sistema k vezes e o transforma em uma condição de verificação (VC) tal que ϕ é satisfazível se, e apenas se, ϕ tem um contraexemplo de profundidade k ou menor. Verificadores SAT comerciais podem ser usados para checar se ϕ é satisfazível. Na implementação de BMC em software, o limite k restringe o número de iterações de laços e chamadas recursivas de um programa. Este processo está demonstrado na Figura 5.

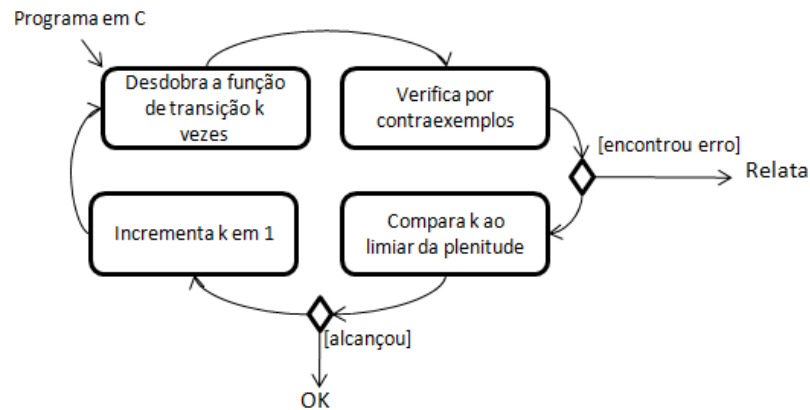


Figura 5. Visão geral do BMC

FONTE: D'Silva et al (2008).

A Figura 6 ilustra um exemplo de aplicação da técnica BMC. Na Figura 6.a tem-se o programa original em linguagem C. A Figura 6.b apresenta os estados gerados a partir do código, onde cada estado representa uma nova atribuição da uma variável. Os estados s_0 , s_1 e s_2 apresentam a inicialização das variáveis do programa. N e M são inicializados com valores não determinísticos sem sinal e i com o valor de zero. Em seguida, o laço *for* gera $2M$ estados, pois são gerados dois estados para cada iteração, um para a atribuição do vetor a e um para o incremento do contador i . No total são gerados $2M+3$ estados, sendo os três primeiros para inicialização das variáveis e o resto é gerado em função do laço *for*. A Figura 6.c ilustra o exemplo de uma propriedade que pode ser verificada com a técnica. Neste, dado um valor de estado s_k , o valor de k pode ser maior que 0 e menor que o limite do vetor, que possui tamanho N . A técnica tenta provar que um programa tem defeito negando a propriedade $\varphi(s_k)$ durante a verificação. Se $s_0 \wedge \dots \wedge s_k \wedge \neg \varphi(s_k)$ é satisfazível, então o programa possui um defeito. Para o exemplo, ocorre um estouro no limite de vetor se $M > N$.

```

01 int main() {
02     unsigned int N, a [N], M;
03     for (unsigned int i=0; i<M; ++i)
04         a[i] = 1;
05     return 0;
06 }
  
```

(a) Exemplo de programa em Linguagem C

$$\begin{aligned}
s_0: & N = \text{nondet_uint} \\
s_1: & M = \text{nondet_uint} \\
s_2: & i_0 = 0 \\
s_3: & a[i_0] = 1 \\
s_4: & i_1 = i_0 + 1 \\
s_5: & a[i_1] = 1 \\
s_6: & i_2 = i_1 + 1 \\
& \dots \\
s_{2M+1}: & a\left[\frac{i_{M-3}}{2}\right] = 1 \\
s_{2M+2}: & \frac{i_{M-2}}{2} = \frac{i_{M-3}}{2} + 1
\end{aligned}$$

(b) Estados gerados a partir do programa da Figura 6.a

$$\phi(s_k): i_k > 0 \wedge i_k \leq N$$

(c) Uma propriedade gerada a partir do programa da Figura 6.a

Figura 6 - Exemplo de aplicação da técnica BMC

FONTE: Gadelha (2013).

Uma ferramenta bem conhecida para implementar BMC em programa ANSI-C/C++ usando solucionador SAT é o CBMC, conforme definido por Kroening e Tautschnig (2014). Ele pode processar código C/C++ usando a ferramenta *goto-cc* que compila o código C/C++ em um programa GOTO equivalente (ou seja, em grafos de controle de fluxo ou CFG em inglês) usando um estilo compatível com o GCC. Os programas GOTO podem ser processados por uma máquina de execução simbólica. CBMC gera as condições de verificação (VCs) usando duas funções recursivas que processam as restrições (suposições e designações de variáveis) e propriedades (condições de segurança e assertivas definidas pelo usuário). Condições de segurança podem ser violações de limites de vetores e matrizes, estouro aritmético e o acesso a informações contidas em endereços de ponteiros nulos.

De uma forma ampla, devido ao uso do BMC original como *back-end*, é possível considerar que tanto o CBMC quanto o ESBMC, que será visto em detalhes a seguir, são verificadores de modelo limitado (BMCs).

2.6 ESBMC - Verificador de modelo eficiente baseado em teorias do módulo da satisfabilidade

Para poder lidar com o aumento da complexidade dos softwares atuais, solucionadores baseados nas teorias do módulo da satisfabilidade (SMT) podem ser usados como *back-end* para resolver os VCs gerados, como mostrado por Clarke et al (2004), Ganai e Gupta (2006), Armando et al (2009) e Cordeiro (2011).

De acordo com Cordeiro et al (2012), verificadores de modelo baseados em SMT podem ser usados para verificar a corretude de software ANSI-C embarcado. Cordeiro (2011) e Cordeiro e Fischer (2011) mostram que o ESBMC pode ser usado para verificar software multitarefa. De acordo com Ramalho et al (2013), o ESBMC pode também ser usado para verificar modelos de software C++ baseados em solucionador SMT.

Entretanto, o ESBMC ainda não foi usado como ferramenta de otimização, mas isto pode ser feito através da característica de checar a negação de uma dada propriedade a uma certa profundidade. A ideia básica é tirar vantagem da característica de verificação de programas do ESBMC e usá-la para verificar todas as possíveis soluções factíveis de uma otimização, e encontrar a melhor solução.

Existem duas diretivas de código C/C++ que podem ser usadas para “ensinar” um verificador de modelo a resolver um problema de otimização: ASSUME e ASSERT. Ambas são declarações de verificação. A declaração ASSERT é usada para forçar uma propriedade como um verificador. Ela pode reportar uma ação baseado no sucesso ou falha da afirmação contida no ASSERT. A declaração ASSERT é uma das diretivas-chaves da verificação, porque ele faz uma declaração sobre o que a propriedade desejada deveria ser.

Para resolver o problema de otimização de um particionamento de hardware/software, a diretiva ASSUME será a responsável por garantir o atendimento das restrições (custo de software neste trabalho), e ASSERT controlará a condição de parada (custo mínimo de hardware neste trabalho). O ESBMC testa um código baseado em diretivas e parâmetros, então é possível testar a função objetivo (ou seja, o problema de otimização) e controlar as restrições. Portanto, com um código C/C++ é possível guiar o ESBMC para resolver um problema de otimização.

A Figura 7 mostra a visão geral com todas as etapas do método de verificação eficiente baseado nas teorias do módulo da satisfabilidade (ESBMC).

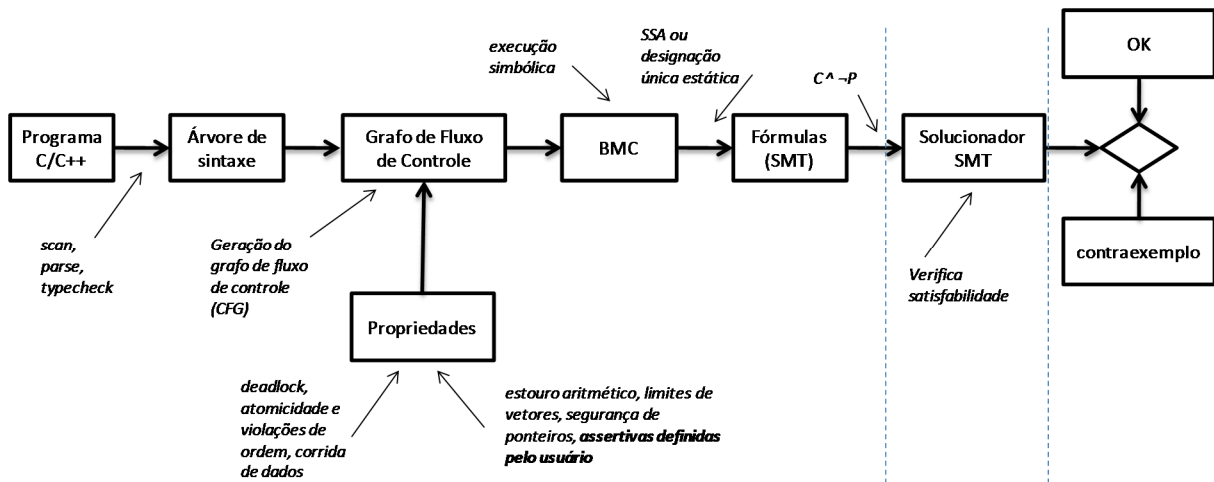


Figura 7 - Visão geral do ESBMC

FONTE: Autoria própria (2014).

Para o usuário da técnica, as etapas entre a definição do programa em C ou C++, a informação das propriedades a serem verificadas e a resposta do ESBMC ocorrem de forma oculta e direta, mas aqui a técnica será explicada em detalhes.

Tudo se inicia com um programa em linguagem C ou C++ que descreve o problema a ser otimizado. Em seguida, o ESBMC realiza um *scan* (ou *lexer*) no programa, transformando o código em uma lista de fichas (ou *tokens*) para poder identificar e separar os comandos, operadores lógicos, números, parênteses e outros. Em seguida, um *parse* transforma a lista de fichas em uma árvore de sintaxe. Há ainda a verificação de tipos, outro passo comum em compiladores para evitar a chamada de funções com o número errado de parâmetros, converter tipos de dados, evitar divisão por zero, evitar o uso de variáveis e expressões sem que tenham sido declaradas previamente, evitar índices de vetores fora de limites e evitar que funções não retornem valores.

Para se chegar ao grafo de fluxo de controle (representado pelo programa *goto*), o usuário deve informar as propriedades que serão verificadas. Especificamente no problema de particionamento de hardware/software, toda a verificação de limites de vetores e matrizes, divisão por zero e verificação de ponteiros podem ser suprimidos do processo, ficando o foco na assertiva que irá controlar a solução do problema e causar a apresentação do contraexemplo, que é o valor mínimo do hardware.

O resultado desta etapa é um código que substitui os laços por *gotos*. Isto pode ser verificado na Figura 8.b.

Em seguida ocorre uma execução simbólica do programa *goto*, gerando-se o que se conhece por SSA, ou designação única estática ou ainda *static single assignment form* em inglês. Isto pode ser verificado na Figura 8.c.

```

01  ...
02  for (i=0; i < num; i++){
03      x[i] = nondet_bool();
04  }
05  aux = 0;
06  ...

```

(a) Pedaco de código em Linguagem C

```

01  ...
02  i=0;
03  if !(i < num) then GOTO 9 //equivalente da linha 2 da Figura 8.a
04  ASSERT i > 0 // limite inferior do vetor x
05  ASSERT i < num //limite superior do vetor x
06  x[i] = NONDET(_Bool); //equivalente da linha 3 da Figura 8.a
07  i = i + 1;
08  GOTO 3
09  aux = 0; //equivalente da linha 5 da Figura 8.a
10  ...

```

(b) Programa *goto* equivalente ao código da Figura 8.a

```

01  ...
02  i1 == 0
03  x1 == x0 WITH [0 := nondet_symbol(symex::nondet1)]
04  i2 == 1
05  x2 == x1 WITH [1 := nondet_symbol(symex::nondet3)]

```

```

06 | i3 == 2
07 | x3 == x2 WITH [2 := nondet_symbol(symex::nondet5)]
08 | i4 == 3
09 | x4 == x3 WITH [3 := nondet_symbol(symex::nondet7)]
10 | ...
11 | i(num+1) == num //para exemplificar que ele vai até num+1
12 | i(num+2) == 0
13 | ...

```

(c) SSA equivalente ao código da Figura 8.a

Figura 8 - Etapas de conversão de um código em Linguagem C no ESBMC

FONTE: A autoria própria (2014).

Em seguida, fórmulas que correspondem às condições de verificação (VCs), as restrições C (*constraints*) e as propriedades P , são geradas a partir do SSA, usando as teorias do módulo da satisfabilidade. Finalmente, a fórmula $C \wedge \neg P$ é passada para um solucionador de SMT para verificar a sua satisfabilidade. A Figura 9 ilustra as restrições e as propriedades geradas a partir do código C da Figura 8.

$$C := \left[\begin{array}{l} g1 := (i_1 = 0) \\ \wedge \\ x1 := \text{store}(x_0, \text{nondet_symbol}, 0) \\ g2 := (i_2 = 1) \\ \wedge \\ x2 := \text{store}(x_1, \text{nondet_symbol}, 1) \\ g3 := (i_3 = 2) \\ \wedge \\ x3 := \text{store}(x_2, \text{nondet_symbol}, 2) \end{array} \right]$$

$$P := \left[\begin{array}{l} x_0 \geq 0 \wedge x_0 \leq 1 \\ \wedge \\ i_0 \geq 0 \wedge i_0 \leq \text{num} \end{array} \right]$$

Figura 9 - Restrições e propriedades a serem satisfeitas, geradas à partir do código da Figura 8

FONTE: A autoria própria (2014).

Neste ponto, se a propriedade que está sendo verificada for violada, é que o ESBMC, através do solucionador SMT, apresentará a condição de violação (que no caso do particionamento é a solução do hardware mínimo); caso contrário, o programa termina sem encontrar nenhuma violação (o que corresponde a não encontrar uma solução para o problema).

2.7 CBMC x ESBMC

Dentre as principais diferenças entre o CBMC e ESBMC, pode-se destacar:

- O CBMC usa um solucionador de lógica proposicional SAT (cf. **Definição VIII**) enquanto o ESBMC usa um solucionador SMT, de teorias do módulo da satisfabilidade. Portanto a codificação é diferente em ambos;
- O SMT surgiu como alternativa ao SAT, visando contornar o problema do aumento do tamanho das fórmulas proposicionais (em virtude do aumento da complexidade dos sistemas atuais) e a perda de estrutura durante a modelagem dos sistemas avaliados;
- O ESBMC traduz as expressões de um programa em fórmulas livres de quantificadores e aplica um conjunto de técnicas de otimização/simplificação para evitar sobrecarga no solucionador. Isto traz um ganho de desempenho em relação ao CBMC baseado em SAT;
- Várias modificações foram realizadas no modelo CBMC para ser usado no ESBMC: como a geração de condições de verificação de programas de forma prévia para possibilitar a detecção de problemas antes de se chegar ao solucionador em si (por exemplo, vazamento de memória), simplificação dos desdobramentos das fórmulas, dentre outros;
- Um interpretador do contraexemplo (caso um problema não seja satisfeito) foi criado para facilitar a apresentação do problema detectado para o usuário.

2.8 Solucionadores SAT

Lógica proposicional (cf. **Definição IV**) tem sido reconhecida, através dos séculos, como fundamental nas inferências e interpretações das mais variadas áreas, indo desde a filosofia até a matemática. A evolução causou sua formalização em álgebra booleana e a história mostrou que vários problemas combinatoriais podem ser expressos como problemas de satisfabilidade proposicional ou SAT.

O interesse em satisfabilidade (cf. **Definição VIII**) tem aumentado com anos por uma série de razões, conforme Franco e Martin (2009), sendo ajudado pelo fato de muitos problemas estarem sendo solucionados atualmente de forma mais rápida por solucionadores SAT modernos do que por outros meios. Mas a razão principal deste crescimento está no fato de que a satisfabilidade está na interseção de várias teorias e ciências, como lógica, teoria de

grafos, ciência da computação, engenharia da computação e pesquisa operacional. Muitos problemas, de qualquer desses campos, possuem múltiplas traduções para satisfabilidade, e existem muitas ferramentas matemáticas disponíveis hoje para que um solucionador SAT auxilie na sua solução de uma forma rápida.

Atualmente, conforme dados do SAT Competition 2014, 44 solucionadores SAT se inscreveram na competição, que se realizou em julho na Áustria, conforme Belov et al (2014). O que mostra a disponibilidade de um grande número de ferramentas SAT, muitas delas de código aberto.

SAT é ainda interpretado de uma forma bem ampla, incluindo, além da satisfabilidade proposicional aqui citada, o domínio das fórmulas booleanas quantificadas (QBF ou *quantified Boolean formulae*), problemas de satisfação de restrições (CSP ou *constraints satisfaction problems*) e teorias do módulo da satisfabilidade (SMT), conforme descrito em Sebastiani e Tacchella (2009).

2.9 Solucionadores SMT

Uma das últimas etapas da ferramenta ESBMC envolve a passagem de fórmulas em lógica de primeira ordem para um solucionador SMT.

Conforme lista de participantes do evento que ocorre anualmente e no qual se realiza uma competição de solucionadores SMT, o SMT-COMP¹ ou *Satisfiability Modulo Theories Competition* que foi realizado, na sua edição mais recente, em julho de 2014 em Viena na Áustria, mais de 20 solucionadores SMT podem ser encontrados atualmente no mercado, sejam eles pagos ou de código aberto, conforme Cok et al (2014).

Cada solucionador possui suas especificidades e métodos para resolver um problema SMT, mas atualmente a técnica mais usada, de acordo com Darwiche e Pipatsrisawat (2009), Moura e Bjorner (2008) e Sebastiani e Tacchella (2009), é uma variante do algoritmo DPLL (Davis-Putnam-Logemann-Loveland), mais conhecida como CDCL (*Conflict Driven Clause Learning*). O CDCL é um método classificado como completo, conforme Gomes et al (2007), no qual se realiza ramificação (cf. **Definição VI**) e *backtracking* (cf. **Definição VII**) exaustivos e é garantido que retorne se uma fórmula booleana é satisfeita ou não. Existem ainda métodos classificados como incompletos que normalmente são baseados na busca por

¹ <http://www.smtcomp.org/>

meio de processos estocásticos, apresentando melhor desempenho, mas sem a garantia de se demonstrar a satisfabilidade/não satisfabilidade de uma fórmula booleana.

O estado da arte de solucionadores SMT depende essencialmente de solucionadores SAT, ou seja, de alguma forma cada solucionador emprega sua técnica para que a codificação dos seus problemas seja SAT compatível. O solucionador Z3 da Microsoft ou ainda o Boolector da FMV, usados neste trabalho, são solucionadores que usam o CDCL e são classificados como completos, conforme Moura e Bjorner (2008) e Gomes et al (2007). Devido a importância que o CDCL tem no tempo de solução de um problema ao ESBMC, pelo fato de ser o responsável por realizar a busca pela solução do problema, serve de motivação para que se detalhe o funcionamento do algoritmo CDCL neste capítulo.

2.10 Algoritmo DPLL

O algoritmo Davis-Putnam-Logemann-Loveland ou DPLL é um processo de busca sistemática para encontrar a resposta da satisfabilidade de uma dada fórmula booleana ou para provar que ela não é satisfeita. Os pesquisadores Martin Davis e Hilary Putnam criaram em 1960 a ideia básica por trás do algoritmo. Entretanto, foi apenas em 1962 que Martin Davis em conjunto com George Logemann e Donald Loveland apresentaram a forma eficiente que a tornou largamente utilizada até hoje.

A Figura 10 ilustra o pseudocódigo do funcionamento original da técnica, cujos parâmetros de entrada são a CNF (cf. **Definição V**) de um problema (Δ) e a profundidade (d) da árvore de busca (ou seja, o número de variáveis booleanas do problema). O algoritmo retorna um conjunto de literais (L) ou INSATISFEITO. Variáveis são denominadas P_1, P_2, \dots, P_n . Uma variável situada em um nível abaixo da variável que está sendo avaliada, ou seja, a próxima variável a ser ramificada, é definida como P_{d+1} . O condicionamento da CNF em uma variável, no processo de ramificação, para o nível abaixo do atual é indicado por $\Delta|d+1$ ou $\neg\Delta|d+1$, dependendo do valor a variável está sendo testada (verdadeiro ou falso).

Algoritmo DPLL (CNF Δ , profundidade d)	
01	Se $\Delta = \{\}$ então retorne $\{\}$ //se chegou em um nó cujos valores das variáveis tornam o caminho um modelo para Δ
02	Senão se $\{\} \in \Delta$ então retorne INSATISFEITO

03	Senão se $L = \text{DPLL}(\Delta P_{d+1}, d+1) \neq \text{INSATISFEITO}$ então retorne $L \cup \{P_{d+1}\}$
04	Senão se $L = \text{DPLL}(\Delta \neg P_{d+1}, d+1) \neq \text{INSATISFEITO}$ então retorne $L \cup \{\neg P_{d+1}\}$
05	Senão retorne INSATISFEITO

Figura 10 - Pseudocódigo do Algoritmo DPLL

FONTE: Darwiche e Pipatsrisawat (2009).

Dentre as características do método DPLL original, pode-se destacar, conforme Darwiche e Pipatsrisawat (2009):

- O algoritmo realiza uma busca na árvore binária abrindo os nodos na ordem das variáveis de decisão, ramificando uma de cada vez (por exemplo, primeiro x_1 , depois x_2 e assim por diante);
- O algoritmo faz uma ramificação em profundidade ou DFS (*depth first search*), começando pela raiz ou nodo inicial e indo ramificando o máximo que puder cada ramo antes de realizar o retorno ou *backtracking*;
- O *backtracking* ocorre sempre para a última ramificação realizada (do nodo x_i ele retorna sempre para o nodo x_{i-1});
- E o algoritmo não necessariamente chega às folhas da ramificação, caso a combinação atual da ramificação do nodo já não satisfaça os requisitos impostos para a busca na árvore binária.

A Figura 11 ilustra uma árvore binária para três variáveis A, B e C. Na figura, o literal t significa ramificar o valor da variável de decisão para verdadeiro (*true*) e f significa ramificar para valor falso (*false*). O literal w_i indica a folha de um possível caminho da árvore (possível solução ou não do problema).

A ramificação do nodo A para o valor *true*, por exemplo, é registrado na linha 03. Já a ramificação para o valor de *false* está indicada na linha 04.

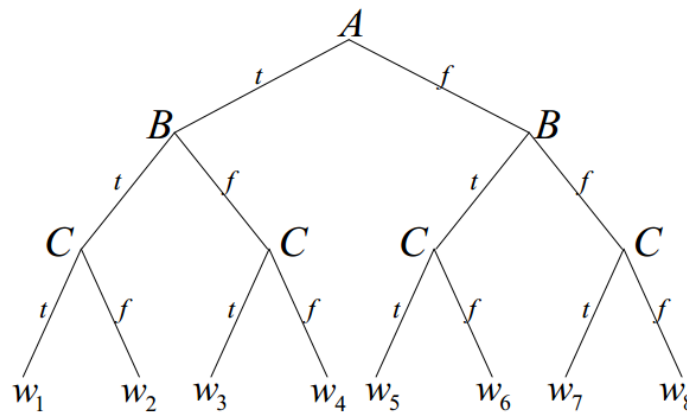


Figura 11 - Árvore de busca resultante da enumeração de todos os possíveis estados das variáveis A, B e C

FONTE: Darwiche e Pipatsrisawat (2009).

Considerando-se a seguinte CNF: $\Delta = \{\{\neg A, B\}, \{\neg B, C\}\}$, no nodo de busca assinalado como F na Figura 12, ao se aplicar o algoritmo DPLL (Figura 10) com o condicionamento de Δ nos literais A, $\neg B$ (que representam os valores das variáveis ramificadas no ponto F), chega-se a:

$$\Delta|A, \neg B = \{\{falso, falso\}, \{verdadeiro, C\}\} = \{\{\}\}$$

Ou seja, o algoritmo concluirá automaticamente que nenhum dos caminhos w_3 e w_4 são modelos para Δ , pois não importa o valor que a variável C irá assumir que o resultado da lógica proposicional será sempre *falso*. Isto está indicado na linha 2 do pseudocódigo da Figura 10.

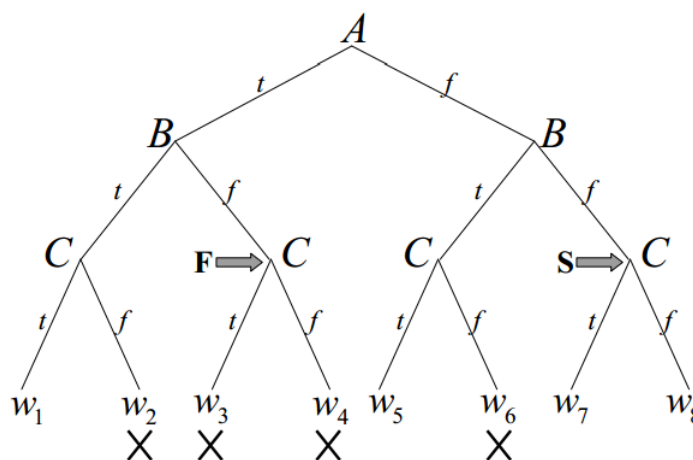


Figura 12 - Árvore de busca resultante da enumeração de todos os estados das variáveis A, B e C, explicitando designações que não são modelos para a CNF do exemplo.

FONTE: Darwiche e Pipatsrisawat (2009).

Já se for observado o nodo assinalado como S na Figura 12, este representa os caminhos w_7 e w_8 os quais são modelos de Δ . Isto pode ser detectado pelo condicionamento de Δ nos literais $\neg A$, $\neg B$ (que representam os valores das variáveis neste ponto S, ou seja, A e B como *false*), conforme aplicação da linha 1 do pseudocódigo da Figura 10:

$$\Delta | \neg A, \neg B = \{\{verdadero, falso\}, \{verdadero, C\}\} = \{\}$$

Portanto, todas as cláusulas ficam subjulgadas imediatamente ao se ajustar os valores de A e B para *false*, independente do valor que a variável C possa assumir, já que o resultado final sempre vai ser verdadeiro e atender a CNF. Com isso conclui-se que os caminhos w_7 e w_8 são modelos de Δ , sem a necessidade de inspecioná-los individualmente.

Ao se aplicar o algoritmo DPLL nos demais caminhos da Figura 12, w_1 e w_5 são modelos para o CNF proposto, e w_2 e w_6 não. Ainda com relação à mesma Figura, todos os caminhos que não são modelos para o CNF proposto estão marcados com X.

2.11 Algoritmo CDCL

Com o passar do tempo, o algoritmo DPLL passou por vários refinamentos, tornando-se um dos mais eficientes algoritmos da atualidade, conforme Gomes et al (2007), Moura e Bjorner (2008), Darwiche e Pipatsrisawat (2009) e Sebastiani e Tacchella (2009). Esses refinamentos serviram ainda para corrigir limitações do algoritmo DPLL original.

Como mudanças principais do DPLL original para o CDCL, pode se destacar:

- A ordem de ramificação (cf. **Definição VI**) dos nodos não é mais sequencial. O algoritmo utiliza uma heurística para decidir a sequência de ramificação das variáveis booleanas do problema. Várias heurísticas foram desenvolvidas nos últimos anos, tais como o método Tau, a regra de Jeroslow-Wang, o estimador Franco e o estimador Johnson, conforme Kullmann (2009);
- Inclusão do recurso de propagação unitária ou resolução unitária ou BCP (*boolean constraint propagation* ou propagação de restrições binárias): que possibilita deduzir, sem chegar aos níveis mais próximos das folhas da árvore de busca (cf. **Definição I**), se um nodo leva a um sucesso ou a uma violação. Em resumo, se uma cláusula for binária, isto é, se contém somente um literal, esta pode ser satisfeita somente atribuindo o valor necessário para fazer este literal se tornar verdadeiro. Na prática,

isto leva à atribuição do valor verdade a outras cláusulas que também possuem este literal, diminuindo assim o espaço de busca e ganhando-se desempenho;

- O *backtracking* (cf. **Definição VII**) deixa de ser cronológico. No DPLL original, após ramificar, o retorno sempre ocorre para o nodo imediatamente anterior ao que foi recentemente ramificado. No CDCL, entretanto, o algoritmo volta para um nodo cujo nível de decisão tem importância maior para a solução do problema, não necessariamente ao anterior ao que foi ramificado mais recentemente. Os solucionadores SAT e SMT modernos usam heurísticas variantes da VSIDS (*variable state independent decaying sum*), conforme Marques-Silva et al (2009), que são caracterizadas pelo fato do próximo literal escolhido ser o que aparece com mais frequência na cláusula resultante das ramificações, o que significa que há um contador para cada literal/variável. O algoritmo combina ainda a noção de “*conflict driven learning*”, no qual guarda a razão de um conflito, conforme proposto por Silva e Sakallah (1996). O resultado de uma análise de conflito pode ser salvo como uma inferência na forma de uma cláusula. Se, mais tarde, durante a busca em árvore, um subconjunto de variáveis coincide com uma inferência salva, o mecanismo de *backtracking* pode ser aplicado para evitar repetir uma busca que previamente resultou em conflito, este é o conceito por trás do aprendizado de cláusulas, conforme Franco e Martin (2009);
- A ordem de ramificação, que no algoritmo DPLL original começa com o teste das variáveis primeiro para o valor *verdadeiro* para depois testar para *falso*, deixa de seguir esta lógica. O algoritmo CDCL decide, no nodo de ramificação, de acordo com uma heurística, pra que valor cada variável será testado primeiro, e no *backtracking* o valor complementar ao primeiro será escolhido para nova ramificação.

A Figura 13 ilustra o pseudocódigo típico do algoritmo que ficou conhecido como DPLL+ ou CDCL (*conflict driven clause learning*) que essencialmente, conforme Marques-Silva et al (2009), segue a organização do algoritmo DPLL. Com relação ao DPLL, as maiores diferenças estão na chamada da função ANÁLISE_DE_CONFLITO cada vez que um conflito é detectado e a chamada da função BACKTRACK quando o *backtracking* é realizado.

	Algoritmo CDCL (CNF Δ , designação de variáveis v)
01	se (PROPAGAÇÃO_UNITÁRIA (Δ , v) == CONFLITO) então

```

02   retorne INSATISFEITO
03    $d\ell \leftarrow 0$  //variável nível da decisão recebe nível de ramificação atual
04   enquanto (! TODAS_AS_VARIÁVEIS_FORAM_DESIGNADAS ( $\Delta$ ,  $v$ )) faça
05       ( $x$ ,  $v$ ) = ESCOLHE_VARIÁVEL_DE_RAMIFICAÇÃO ( $\Delta$ ,  $v$ ) //decide que variável ramificar
06        $d\ell \leftarrow d\ell + 1$  //desce mais um nível na árvore de busca
07        $v \leftarrow v \cup \{(x, v)\}$  //atribui valor da variável conforme heurística (verdadeiro ou falso)
08       se (PROPAGAÇÃO_UNITÁRIA ( $\Delta$ ,  $v$ ) == CONFLITO) então
09            $\beta$  = ANÁLISE_DE_CONFLITO ( $\Delta$ ,  $v$ )
10           se ( $\beta < 0$ ) então //chegou no nodo raiz
11               retorne INSATISFEITO
12           senão
13               BACKTRACK ( $\Delta$ ,  $v$ ,  $\beta$ )
14                $d\ell \leftarrow \beta$ 
15   retorne SATISFEITO

```

Figura 13 - Pseudocódigo do Algoritmo CDCL

FONTE: Marques-Silva et al (2009).

Em adição à função principal CDCL, as seguintes funções auxiliares são utilizadas:

- PROPAGAÇÃO_UNITÁRIA: consiste da aplicação iterativa da regra da propagação unitária. Se uma violação é identificada, então uma indicação de conflito é devolvida;
- ESCOLHE_VARIÁVEL_DE_RAMIFICAÇÃO: consiste da função que escolhe a variável que será ramificada e para qual valor ela será designada (verdadeiro ou falso);
- ANÁLISE_DE_CONFLITO: analisa um conflito e aprende novas cláusulas à partir dos conflitos encontrados;
- BACKTRACK: retorna a ramificação para o nível computado pela função ANÁLISE_DE_CONFLITO;
- TODAS_AS_VARIÁVEIS_FORAM_DESIGNADAS: testa se todas as variáveis já foram ramificadas. Em caso afirmativo, o algoritmo termina indicando se a fórmula CNF (Δ) foi satisfeita.

A Figura 14.a ilustra uma ramificação típica ao se aplicar o algoritmo DPLL e a Figura 14.b ilustra o mesmo problema ao se aplicar o algoritmo CDCL. Fica claro que a ordem e os valores de ramificação das variáveis mudam, bem como o *backtracking* (em azul) quando uma violação ocorre: o espaço de busca diminui e tem-se um ganho de desempenho.

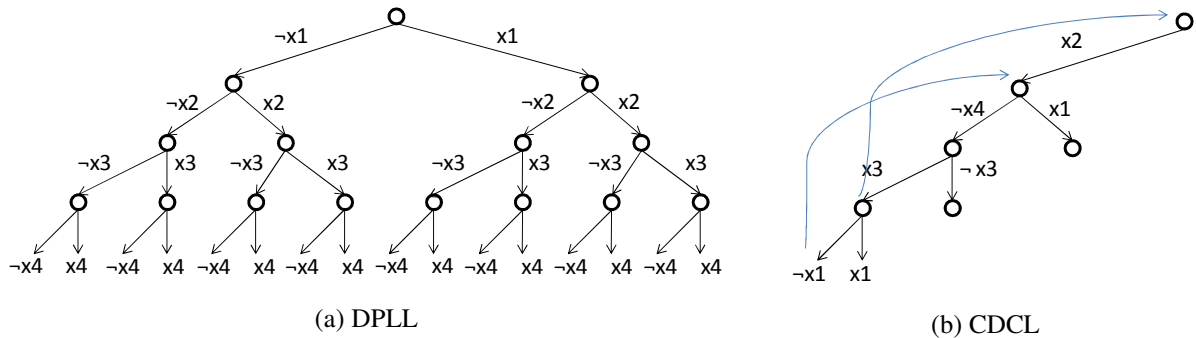


Figura 14 - Comparativo de ramificação: DPLL e CDCL

Fonte: Autoria Própria (2014).

2.12 Modelagem matemática informal do coprojeto de primeira geração

O modelo adotado pode ser descrito através de cinco características, conforme definido em Arató et al (2003), Arató et al (2005) e Mann et al (2007):

- Há apenas um contexto de software, ou seja, há apenas um processador de uso geral e apenas um contexto de hardware (FPG, por exemplo). Em outras palavras, os componentes do sistema só podem ser mapeados para um dos dois contextos;
- A implementação em software de um componente tem um custo de software associado, que é o tempo de execução deste componente se executado em software;
- Similarmente, a implementação em hardware de um componente tem um custo, que pode ser a área, dissipação de calor, ou consumo de energia;
- Baseado na premissa que o hardware é significativamente mais rápido que software, o tempo de execução de um componente em hardware é considerado zero;
- Finalmente, se dois componentes são mapeados para o mesmo contexto, então não há *overhead* de comunicação (ou custo de comunicação) entre eles. E se eles são mapeados para contextos diferentes, então há um *overhead* de comunicação.

A consequência dessas considerações é que o escalonamento não é tratado para o coprojeto de primeira geração. Os componentes de hardware não precisam de escalonamento,

porque o tempo de execução é assumido como sendo zero. Esta configuração, de acordo com Teich (2012), descreve um tipo de coprojeto de primeira geração onde o foco está no biparticionamento.

2.13 Modelagem matemática formal do coprojeto de primeira geração

As entradas do problema são as seguintes:

- Um grafo simples direcionado (cf. **Definição II**) $G = (V, E)$, chamado de grafo de tarefas do sistema ou grafo acíclico orientado (DAG, do inglês *directed acyclic graph*);
- Os vértices $V = \{v_1, v_2, \dots, v_n\}$ representam os nodos que são os componentes do sistema a ser particionado;
- As arestas (E) representam as comunicações entre os componentes;
- Adicionalmente, cada nodo v_i tem um custo de hardware $h(v_i)$, se for implementado em hardware e um custo de software $s(v_i)$, se for implementado em software;
- Finalmente, $c(v_i, v_j)$ representam os custos de comunicação entre v_i e v_j se forem implementados em contextos diferentes (hardware ou software).

Baseado em Arató et al (2003), P é chamada uma partição de hardware/software se é uma bipartição de V : $P = (V_H, V_S)$, onde $V_H \cup V_S = V$ e $V_H \cap V_S = \emptyset$. As arestas cruzadas são $E_P = \{(v_i, v_j) : v_i \in V_S, v_j \in V_H \text{ ou } v_i \in V_H, v_j \in V_S\}$. O custo de hardware de P é dado pela Equação (1), e o custo de software de P , ou seja, a soma dos custos de software dos nodos com o custo de comunicação, é dado pela Equação (2):

$$H_P = \sum_{v_i \in V_H} h_i \quad (1)$$

$$S_P = \sum_{v_i \in V_S} s_i + \sum_{(v_i, v_j) \in E_P} c(v_i, v_j) \quad (2)$$

Ainda baseado em Arató et al (2003), três diferentes problemas de otimização e decisão podem ser definidos. Neste trabalho, o foco estará no mais comum encontrada em sistemas de tempo real, no qual o custo de software inicial (S_0) é dado. Busca-se encontrar

uma partição P de HW-SW tal que $S_P \leq S_0$ e o H seja mínimo, ou seja, busca-se minimizar o custo de hardware de um sistema embarcado buscando-se uma partição que deve atender a restrição de encontrar um custo de software menor que um inicialmente dado. Como se trata de um sistema em tempo real, e o tempo de software está associado ao tempo de execução, o problema está bem caracterizado nesse modelo. Com relação à complexidade deste tipo de problema, Arató et al (2003) também demonstraram que é *NP-Hard*.

2.14 Coprojetos de segunda e terceira gerações

No coprojetos de segunda geração, a arquitetura é de um SOC (*system on chip* ou *system on a chip*) com múltiplos núcleos consistindo de diferentes tipos de máquinas de processamento (PE, *processing engines* em inglês) conforme definido em Luo et al (2010). Uma PE pode ser um processador de propósito geral, um ASIC, ou ainda um FPGA. Portanto trata-se de um SOC com multiprocessadores que compartilham a mesma memória, como mostrado na Figura 15.

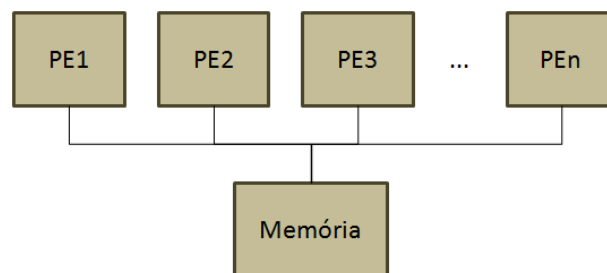


Figura 15 - Arquitetura SOC com múltiplos núcleos

FONTE: Luo et al (2010).

Há ainda uma terceira geração de coprojetos de hardware/software, na qual a tecnologia se torna heterogênea, com a inclusão de periféricos, por exemplo, conforme definido em Teich (2012). Neste trabalho a pesquisa sobre o uso do ESBMC para realizar particionamento de HW/SW vai até a primeira geração de coprojetos.

2.15 Resumo

Neste capítulo, foram introduzidos conceitos básicos para entendimentos do resto deste documento, tais como árvores binárias, grafos, máquinas de estado, bem como definido e explanado o que seria um processo de otimização, que compreende a essência de um problema de particionamento de HW/SW. Para se chegar aos verificadores de modelo

baseados nas teorias do módulo da satisfabilidade, foi feita uma conceituação cronológica começando pela verificação de hardware e de software, passando-se pela verificação de modelos, verificação de modelos de contexto limitado, até se chegar no ESBMC, o qual foi detalhado quanto às suas etapas. O verificador estado da arte CBMC foi ainda descrito e comparado com o ESBMC, havendo ainda seções específicas para se descrever como os solucionadores SAT e SMT funcionam, pois estes são os itens mais importantes no que concerne o desempenho das técnicas. O capítulo apresenta ainda o detalhamento dos mecanismos pelos quais um solucionador encontra a resposta do problema de particionamento de HW/SW e finaliza com a apresentação da modelagem matemática formal do coprojeto de primeira geração. Como resultado, o conteúdo apresentado nesse capítulo fornece todo o embasamento necessário para compreensão do trabalho desenvolvido, que será descrito nas próximas seções.

3 Particionamento de hardware/software para sistemas embarcados

Neste capítulo, aplica-se o modelo formal de particionamento apresentado na seção 2.12 do Capítulo anterior a um exemplo, são apresentadas as técnicas usadas neste trabalho para realizar o particionamento de hardware/software para sistemas embarcados, descrevendo-se como cada técnica resolve o problema de particionamento, as ferramentas usadas e os respectivos algoritmos.

3.1 Aplicando o modelo formal a um exemplo

Para resolver o problema de particionamento de hardware/software, é necessário modelar matematicamente o sistema e então estabelecer o real problema a ser resolvido. A ideia aqui é exemplificar o processo de particionamento de hardware/software de um sistema embarcado de 10 nodos e 13 arestas, como representado na Figura 16. O mesmo princípio valerá para 25, 50, 100 ou mais nodos.

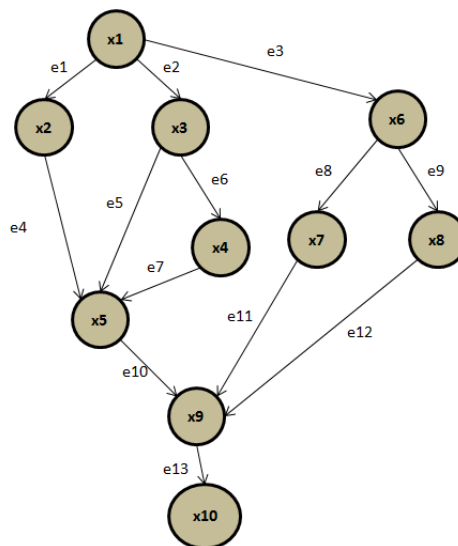


Figura 16 - Grafo direcionado do sistema a ser particionado

FONTE: Autoria própria (2014).

Os dados fornecidos (escolhidos pelo autor) para cada nodo são os seguintes:

- Custo de hardware de cada nodo $h = [5 \ 10 \ 3 \ 4 \ 8 \ 5 \ 10 \ 3 \ 4 \ 8]$;
- Custo do software de cada nodo $s = [10 \ 15 \ 7 \ 4 \ 9 \ 10 \ 15 \ 7 \ 4 \ 9]$;

- E o custo de comunicação de cada aresta se implementado em contextos diferentes $c = [5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5]$.

Para criar o algoritmo, é necessário definir uma matriz chamada $E \in \{-1, 0, 1\}^{13 \times 10}$, que é a matriz de incidência transposta do grafo G , ou seja,

$$E[i, j] := \begin{cases} -1 & \text{se a aresta } i \text{ começa no nodo } j \\ 1 & \text{se a aresta } i \text{ termina no nodo } j \\ 0 & \text{se a aresta } i \text{ não é incidente ao nodo } j \end{cases}$$

Seja $x \in \{0, 1\}^{10}$ um vetor binário indicando a partição, ou seja,

$$x[i] := \begin{cases} 1 & \text{se o nodo } i \text{ for particionado para Hardware} \\ 0 & \text{se o nodo } i \text{ for particionado em software} \end{cases}$$

Os componentes de um vetor, formado pelo módulo da multiplicação da matriz E pelo vetor de partição x , ou seja, $|Ex|$, indicam as arestas que cruzam o limite entre os contextos de hardware e software.

O problema de otimização usado em todas as técnicas demonstradas neste trabalho pode então ser formulado como:

$$\text{Minimizar } hx \tag{3a}$$

Sujeito às restrições:

$$s(1 - x) + c|Ex| \leq S_0 \tag{3b}$$

$$x \in \{0, 1\}^{10} \tag{3c}$$

A formulação acima pode ser transformada em uma ILP equivalente pela introdução das variáveis auxiliares $y \in \{0, 1\}^{13}$ para eliminar o módulo da Equação 3b.

$$\text{Minimizar } hx \tag{4a}$$

Sujeito às restrições:

$$s(1 - x) + cy \leq S_0 \tag{4b}$$

$$Ex \leq y \tag{4c}$$

$$-Ex \leq y \tag{4d}$$

As Equações (3) e (4) são equivalentes. Entretanto, quando se utiliza ferramentas comerciais para resolver o problema de otimização, a forma apresentada pelas Equações (4b), (4c) e (4d) não são adequadas. Essas ferramentas comerciais usam a seguinte representação, com apenas uma restrição de desigualdade:

$$\min_x f^T x \text{ tal que } A \cdot x \leq b \quad (5)$$

Portanto, a ideia básica é transformar as Equações (4b), (4c) e (4d) em uma única matriz. Isto é feito decompondo as três equações (que são inicialmente três matrizes) em termos das variáveis de decisão x e y , e depois as colocando de volta na forma de uma única matriz e isolando qualquer variável para o lado esquerdo da equação. Os exemplos seguintes mostram o processo de transformar uma formulação algébrica da Equação (4b) na matriz que representa o exemplo em questão.

$$[10 \ 15 \ 7 \ 4 \ 9 \ 10 \ 15 \ 7 \ 4 \ 9] \begin{bmatrix} 1 - x_1 \\ 1 - x_2 \\ 1 - x_3 \\ 1 - x_4 \\ 1 - x_5 \\ 1 - x_6 \\ 1 - x_7 \\ 1 - x_8 \\ 1 - x_9 \\ 1 - x_{10} \end{bmatrix} + [5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5] \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \end{bmatrix} \leq S_0 \quad (6)$$

Ou ainda, em notação algébrica:

$$-10x_1 - 15x_2 - 7x_3 - 4x_4 - 9x_5 - 10x_6 - 15x_7 - 7x_8 - 4x_9 - 9x_{10} + 5y_1 + 5y_2 + 5y_3 + 5y_4 + 5y_5 + 5y_6 + 5y_7 + 5y_8 + 5y_9 + 5y_{10} + 5y_{11} + 5y_{12} + 5y_{13} \leq S_0 - 90 \quad (7)$$

E de volta na representação de matriz, com todas as variáveis do lado esquerdo:

$$[10 \ 15 \ 7 \ 4 \ 9 \ 10 \ 15 \ 7 \ 4 \ 9 \ 10 \ 15 \ 7 \ 4 \ 9 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5] \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_{10} \\ y_1 \\ \vdots \\ y_{13} \end{bmatrix} \leq S_0 - 90 \quad (8)$$

Fazendo o mesmo para as Equações (4c) e (4d), e colocando-as, todas juntas, o resultado final é mostrado na Equação (9), independente do número de variáveis. Os índices mostram a dimensão de cada componente da matriz resultante.

$$\begin{bmatrix} -S_{1 \times V} & C_{1 \times E} \\ E_{E \times V} & -I_E \\ -E_{E \times V} & -I_E \end{bmatrix}_{(1+2*E)x(V+E)} \cdot \begin{bmatrix} x_{V \times 1} \\ y_{E \times 1} \end{bmatrix}_{(V+E)x1} \leq \begin{bmatrix} S_0 - 9 * V_{1x1} \\ 0_{E \times 1} \\ 0_{E \times 1} \end{bmatrix}_{(1+2*E)x1} \quad (9)$$

A forma final descrita pela Equação (9) pode ser usada por qualquer uma das técnicas descritas neste trabalho. Entretanto, devido ao fato do ESBMC poder manusear as Equações (3) e (9) da mesma maneira, pois utiliza o ambiente C/C++ e tem mais liberdade de codificação, o trabalho de pesquisa aqui descrito realizará testes com dois algoritmos de ESBMC: um para os mesmos parâmetros adotados por ferramentas comerciais, ou seja, Equação (9); e outra versão sem a inclusão das variáveis auxiliares, ou seja, Equação (3b).

3.2 Algoritmo baseado em ILP: programação inteira binária

Para a formulação ILP do problema de particionamento de hardware/software foi usada a ferramenta matemática MATLAB na versão R2011a e com o toolbox de Otimização, da fabricante MathWorks conforme Matlab (2014). MATLAB é uma linguagem de alto nível, dinamicamente tipificada, conhecida por ser uma ferramenta matemática de estado da arte, de acordo com Tranquillo (2011), e por ser amplamente usada pela comunidade de engenheiros eletricitas e de computação, de acordo com Hong e Cai (2010).

As Equações (4a) e (9) foram usadas, bem como a função *bintprog* do MATLAB, que resolve problemas de programação inteira binária. Esta função usa uma programação linear (LP) baseada em um algoritmo de *branch-and-bound* que busca uma solução ótima através de uma série de problemas de relaxação LP, nas quais os requisitos das variáveis de serem inteiras binárias são substituídas pela limitação $0 \leq x \leq 1$. De acordo com Daskalaki et al (2004), o algoritmo pode ser descrito como um processo de 3 etapas:

- Primeiro ele busca por uma solução inteira factível (que atende os requisitos);
- Então ele atualiza o ponto de melhor inteiro binário factível encontrado até então à medida que a busca em árvore cresce;
- E finalmente, verifica que nenhuma solução factível melhor é possível por meio da solução do problema de programação linear.

O algoritmo do MATLAB cria uma busca em árvore (cf. **Definição I**), ramificando. No passo de ramificação (cf. **Definição VI**), o algoritmo escolhe uma variável x_j cujo valor atual não é um número inteiro e adiciona a restrição $x_j = 0$ para formar um ramo e a limitação

$x_j = 1$ para formar o outro ramo. Este processo pode ser representado por uma árvore binária na qual os nodos representam as restrições adicionadas. A Figura 17 ilustra a árvore binária completa para um problema com 5 variáveis. Deve-se notar que, em geral, a ordem das variáveis que são escolhidas para ramificar não segue necessariamente a ordem usual sequencial.

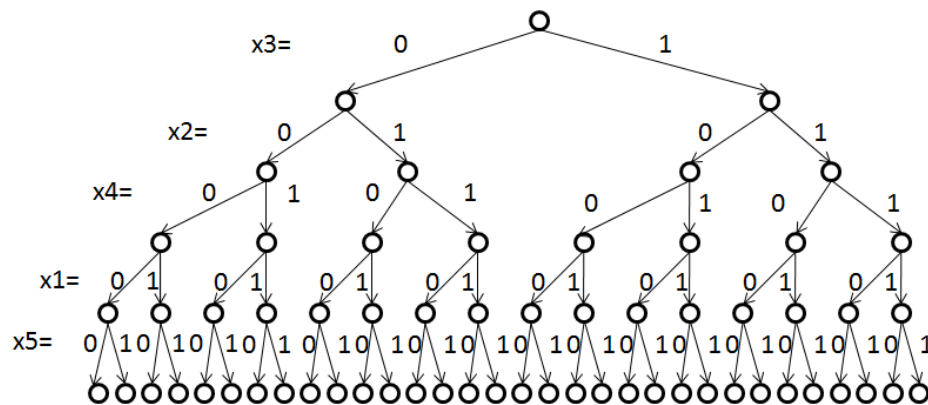


Figura 17 - Árvore binária para o exemplo de particionamento com 5 variáveis

FONTE: Autoria própria (2014).

Em cada nodo, o algoritmo do MATLAB soluciona um problema de relaxação de programação linear (LP) usando as restrições do nodo e decidindo se ramifica ou se move para outro nodo, dependendo do resultado. Três possibilidades são possíveis:

- Se a solução do problema de relaxação LP no nodo atual é infactível (não atende a restrição) ou se seu valor ótimo é maior do que o melhor até então encontrado, o algoritmo remove o nodo da árvore e não busca novos ramos abaixo deste nodo. O algoritmo do MATLAB então se move para um novo nodo de acordo com o parâmetro que pode ser ajustado na opção *NodeSearchStrategy*;
- Se o algoritmo do MATLAB encontra uma solução inteira factível com valor da função objetivo menor que o melhor até então encontrado, o algoritmo atualiza a melhor solução encontrada e move para o próximo nodo;
- Se o resultado do problema de relaxação LP é ótimo, mas não com solução inteira e o valor da função objetivo do problema é menor que a melhor solução até então encontrada, o algoritmo do MATLAB ramifica de acordo com o método especificado na opção *BranchStrategy*.

A solução do problema de relaxação LP fornece o limite inferior do problema de programação inteira binária. Se a solução encontrada é um vetor inteiro binário, então passa a

ser o limite superior do problema de programação inteira binária. Quando a árvore de busca cresce com o acréscimo de nodos, o algoritmo do MATLAB atualiza os limites inferior e superior. Os limites dos valores da função objetivo servem como limiar para cortar ramos desnecessários.

O algoritmo MATLAB para a função *bintprog* pode potencialmente buscar todos os 2^n vetores inteiros binários, onde n é o número de variáveis do problema.

A Figura 18 apresenta o pseudocódigo do problema de particionamento do exemplo-modelo de 10 nodos e 13 arestas da seção 3.1.E para ser resolvido por ILP no MATLAB. Todas as etapas do método descrito aqui para explicar como o MATLAB resolve um problema ILP são realizadas na linha 11 da referida Figura, com a chamada da função *bintprog*.

01	Declarar o número de nodos e arestas
02	Declarar custo de hardware de cada nodo como um vetor (h)
03	Declarar o custo de software de cada nodo como um vetor (s)
04	Declarar o custo de comunicação de cada aresta como um vetor (c)
05	Declarar o custo do software inicial (S_0)
06	Declarar a matriz de incidência transposta do grafo $G(E)$
07	Criar a matriz identidade I (dimensão = número de arestas)
08	Tendo como referência a (Eq. 9), criar a matriz $A = \begin{bmatrix} -S & C \\ E & -I \\ -E & -I \end{bmatrix}$
09	Criar a matriz $b = \begin{bmatrix} S_0 - 9 * nodos_{1x1} \\ 0_{arestas \times 1} \\ 0_{arestas \times 1} \end{bmatrix}$
10	Iniciar cronômetro
11	Resolver ILP usando a função <i>bintprog</i> , passando h , A e b como parâmetros
12	Parar cronômetro
13	Apresentar a solução

Figura 18 - Pseudocódigo de um programa no MATLAB para resolver ILP

Fonte: Autoria própria (2014).

3.3 Algoritmo genético

Da mesma forma que com o ILP, foi usada a ferramenta matemática MATLAB, entretanto foi adotada a versão R2014a com o toolbox de otimização conforme Matlab (2014). Os motivos para a escolha desta ferramenta são os mesmos citados na seção anterior e a mudança de versão deve-se ao fato da versão R2011a apresentar limitações quando as variáveis de busca são números reais inteiros.

Embora o algoritmo baseado em ILP possa resolver qualquer tipo de otimização do particionamento de hardware/software, o tempo de processamento para encontrar a solução torna-se uma séria limitação quando a complexidade do problema aumenta. Nos anos mais recentes, foram desenvolvidos métodos de otimização diferentes das técnicas de programação matemática tradicionais. Estes métodos foram chamados de métodos modernos ou não tradicionais de otimização, de acordo com Rao (2009), e são usualmente mais rápidos que a ILP. GAs emergem destes métodos e, filosoficamente falando, são baseados na teoria de Darwin da sobrevivência do mais apto.

Alguns autores, como Belegundu e Chandrupatla (2011), nomeiam GA não como uma ferramenta de otimização, mas como uma ferramenta de busca heurística, pelo fato de que o algoritmo não garante que se alcance a melhor solução global do problema.

De acordo com Rao (2009), GAs são baseados nos princípios de genética natural e seleção natural. Os elementos básicos da genética natural, tais como reprodução, cruzamento e mutação, são usados no procedimento de busca genética. De acordo com Haupt e Haupt (2004), os GAs diferem dos métodos tradicionais em quatro grandes aspectos:

- GAs trabalham com a codificação de um grupo de parâmetros e não com os parâmetros propriamente ditos;
- GAs trabalham com uma população e não com um ponto único;
- GAs usam informação de custos e recompensa e não derivadas ou outros recursos auxiliares;
- E, finalmente, GAs usam regras de transição probabilísticas e são não determinísticos.

A implementação de GAs segue passos bem definidos, conforme demonstrado por Mitchell (1999). Inicialmente, há uma codificação: uma codificação inicial de indivíduos (zeros e uns) deve ser definida. O problema de particionamento é muito feliz neste ponto,

porque a possível solução é exatamente representada por um vetor binário. Este é o ponto de partida para os demais passos do GA.

Depois, uma população inicial deve ser gerada, ou seja, indivíduos gerados aleatoriamente serão os candidatos para resolver o problema de particionamento. Baseado na avaliação experimental de Arató et al (2003), o tamanho dessa população deve ser definido como 300 indivíduos para produzir melhores resultados (o valor padrão do MATLAB é 20).

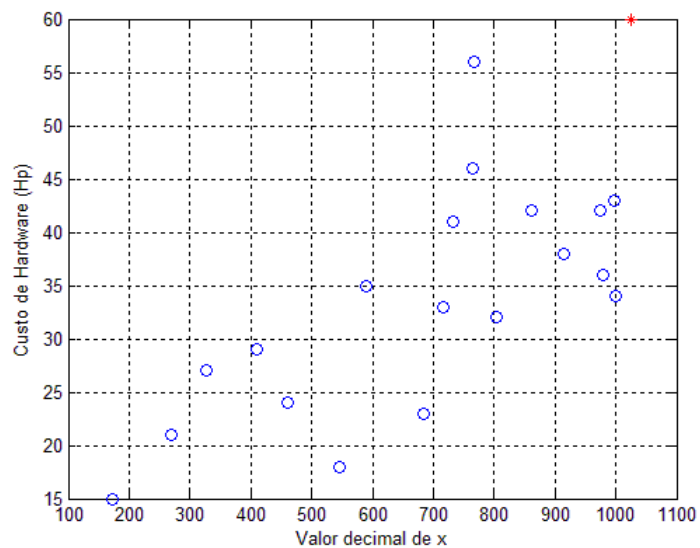
O terceiro passo é a criação de uma nova população, ou geração, realizada da seguinte forma, de acordo com Rao (2009): cada membro da população atual recebe uma pontuação através do cálculo do seu valor de *fitness* (que nesse caso é o custo de hardware). Através de uma técnica probabilística, denominada de *roulette-wheel* ou roleta em português, há a escolha de indivíduos para o conjunto de acasalamento que será usado para os operadores seguintes, na qual os indivíduos com melhor valor de *fitness* da população atual têm probabilidade maior de serem “escolhidos” para a próxima população, de forma a garantir o princípio da sobrevivência do mais apto. A partir daí, os membros escolhidos, denominados de elite, serão os pais usados para produzir nova população. Os filhos serão produzidos por meio de mudanças aleatórias dos bits de um único pai (mutação) ou pela combinação dos valores dos vetores de um par de pais (cruzamento). Dessa forma, o GA substitui a população atual por uma nova geração.

O passo final define como o algoritmo será parado. O MATLAB usa uma série de critérios de parada, todos configurados como parâmetros no algoritmo: quando o número máximo de gerações (iterações) é alcançado, quando um limite de tempo de execução é alcançado, quando o valor da função *fitness* para o melhor ponto na população atual é menor ou igual ao limite de *fitness*, quando a mudança relativa no valor da função de *fitness* entre uma geração e outra alcançar a tolerância da função ou quando não há melhoria na função objetivo durante um intervalo de tempo pré-estabelecido.

O algoritmo MATLAB começa criando a população inicial aleatória de 20 indivíduos, conforme exemplo ilustrado na Figura 19.a para o caso de teste de 10 nós e 13 arestas apresentado na seção 3.1 deste capítulo. Lembrando que, para este exemplo, a solução ótima considerando a restrição $S_0 = 0$ é dada por $x = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$, ou seja, o valor decimal de x é $1.024 (2^{10})$ e o custo mínimo de hardware resultante é de 60. Na Figura 19.b, há a representação gráfica destes 20 indivíduos, com destaque para o valor ótimo a ser alcançado representado por um asterisco na cor vermelha.

Valor binário	Valor decimal	Custo de hardware
0110011111	998	43
1011001101	717	33
1100010011	803	32
0101100110	410	29
0111001111	974	42
0110001010	326	27
1111111101	767	56
0011010101	684	23
0111000010	270	21
0010101111	980	36
0011001110	460	24
0111001001	590	35
1011101011	861	42
1000010001	545	18
0010101111	980	36
1011101101	733	41
1011111101	765	46
0001011111	1000	34
1100100111	915	38
0011010100	172	15

(a) População inicial de 20 indivíduos do algoritmo GA



(b) Representação gráfica da população e do custo de hardware

Figura 19 - População inicial de um problema a ser resolvido com GA

FONTE: Autoria própria (2014).

No passo seguinte, o GA usa a população atual para criar os filhos que comporão a próxima geração. O algoritmo do MATLAB seleciona os indivíduos da atual população, conforme método descrito anteriormente nesta seção, para serem os pais e formarem o conjunto de acasalamento, que contribuirão com seus genes – no caso do particionamento, com os valores das variáveis de decisão ou nodos - para seus filhos. Três tipos de filhos são criados para a próxima geração:

- Filhos Elite serão aqueles indivíduos selecionados probabilisticamente em função do seu valor *fitness* da atual população que sobreviverão para a próxima geração/população;
- Filhos Cruzamento serão aqueles criados pela combinação de vetores de seus pais;
- Filhos Mutação serão aqueles criados pela mudança randômica, ou mutação, de um pai escolhido da população atual.

O esquema da Figura 20 ilustra os três tipos de filhos, considerando, para o caso apresentado na Figura 19, que o algoritmo vai escolher dois pais de melhor valor *fitness* para produzir parte da nova população. O melhor valor *fitness* apresentado pela população da Figura 19 é o $x = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1]$, com o custo de hardware de 56; e o segundo melhor valor *fitness* é 46, dado pelo vetor $x = [1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1]$.

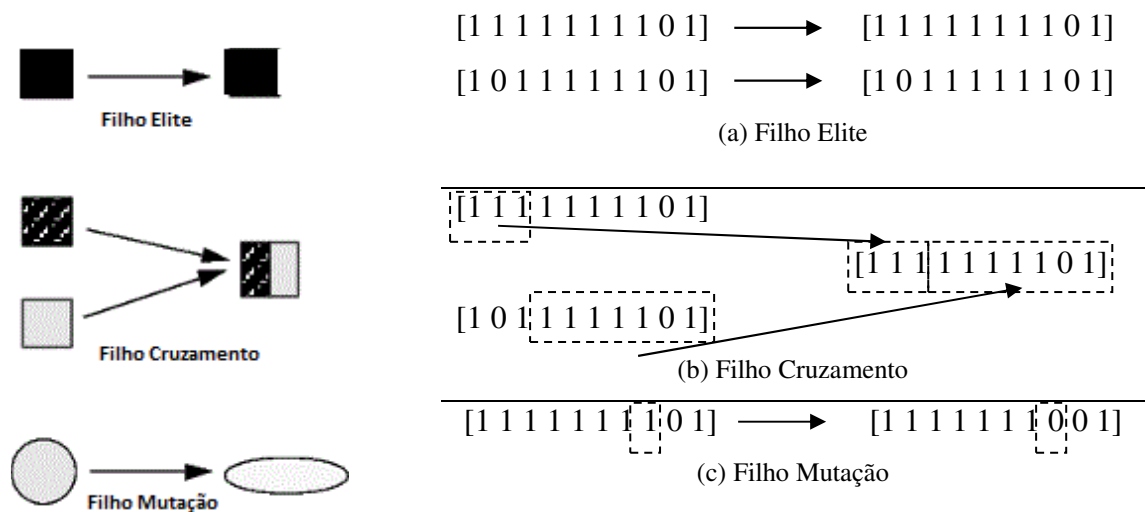


Figura 20 - Os três tipos de filhos da função GA do MATLAB

FONTE: Matlab (2014) e Autoria Própria (2014).

Assim, dois filhos Elite da próxima geração representam uma cópia exata dos pais escolhidos da geração atual, conforme Figura 20.a.

Na geração do Filho Cruzamento, o algoritmo do MATLAB cria filhos pela combinação de pares de pais da população atual. Para isso, através da opção `@crossoversinglepoint` da função `CrossoverFcn` do MATLAB, o algoritmo seleciona aleatoriamente um ponto de cruzamento (7 no exemplo) e pega parte de um pai e mescla com o restante dos genes do outro pai, criando um novo filho, conforme indicado pela Figura 20.b.

Na geração do Filho Mutação, vários métodos podem ser usados, mas considerando-se a utilização do algoritmo de ponto-único do MATLAB, primeiro escolhe-se um pai de bom valor de *fitness*, em seguida um gene aleatório deste pai é escolhido e seu valor é mudado conforme uma probabilidade randômica pré-estabelecida pelo método, conforme mostrado na Figura 20.c, onde se escolheu o 2º gene e este mudou de valor.

Por padronização do método, quando se aplica GA a problemas de restrições lineares, como é o caso do particionamento de HW/SW, os filhos continuam sendo uma solução factível do problema.

A Figura 21 mostra os Filhos da segunda geração para os pais Elite escolhidos na Figura 20, e indica se são Elite, Cruzamento ou Mutação.

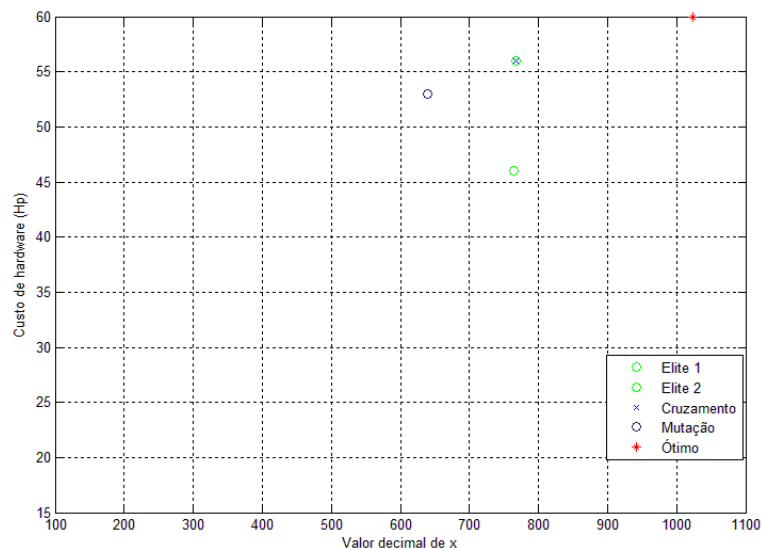
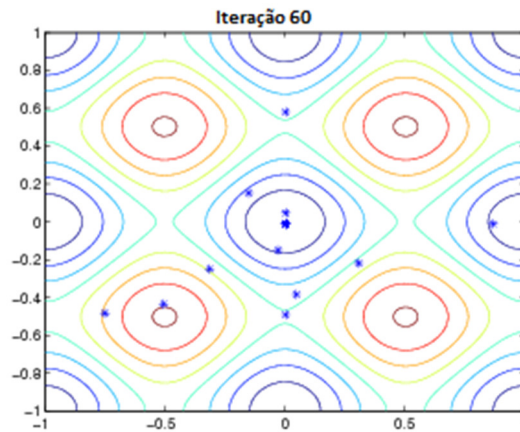


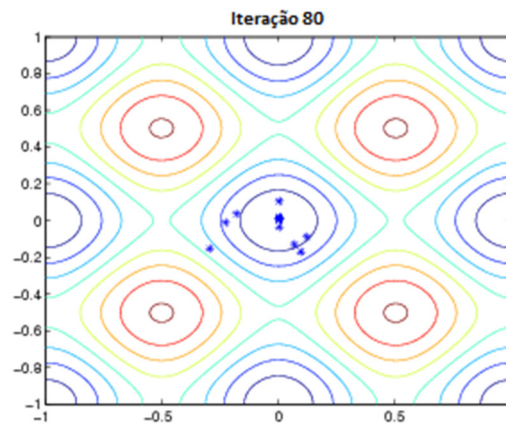
Figura 21 - Filhos resultantes da técnica de GA

FONTE: Autoria própria (2014).

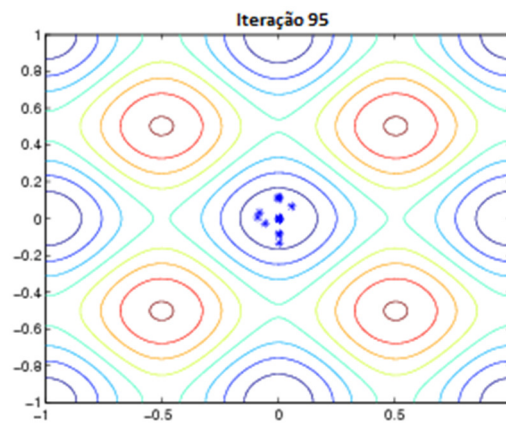
As Figura 22.a, b e c ilustram as populações geradas nas iterações 60, 80 e 95 de um exemplo genérico de uso da técnica de GA. A cada nova geração, os filhos se aproximam e convergem para um dado valor, que representa a solução do problema de otimização.



(a) População gerada na iteração 60 do GA



(b) População gerada na iteração 80 do GA



(c) População gerada na iteração 95 do GA

Figura 22 - Ilustração das gerações da técnica de GA e sua aproximação a um valor ótimo

FONTE: Matlab (2014).

A Figura 23 mostra o pseudocódigo do programa criado no MATLAB para resolver o problema de particionamento HW/SW com GA. Todas as etapas citadas aqui de como a função do MATLAB implementa o GA são realizadas na chamada da função *ga* na linha 16.

01	Declarar o número de nodos e arestas
02	Declarar custo de hardware de cada nodo como um vetor (h)
03	Declarar o custo de software de cada nodo como um vetor (s)
04	Declarar o custo de comunicação de cada aresta como um vetor (c)
05	Declarar o custo do software inicial (S_0)
06	Declarar a matriz de incidência transposta do grafo G (E)
07	Criar a matriz identidade I (dimensão = número de arestas)
08	Tendo como referência a (Eq. 9), criar a matriz $A = \begin{bmatrix} -S & C \\ E & -I \\ -E & -I \end{bmatrix}$
09	Criar a matriz $b = \begin{bmatrix} S_0 - 9 * nodos_{1 \times 1} \\ 0_{arestas \times 1} \\ 0_{arestas \times 1} \end{bmatrix}$
10	Definir o limite inferior da solução como zero para todas as variáveis (lb)
11	Definir o limite superior da solução como um para todas as variáveis (ub)
12	Definir a função <i>fitness</i> como o custo de hardware
13	Declarar que todas as variáveis da solução irão assumir valores inteiros (<i>intcon</i>)
14	Mudar o valor padrão da população e o número de gerações para 300
15	Iniciar cronômetro
16	Resolver GA usando a função <i>ga</i> , passando h , A , b , lb , ub e <i>intcon</i> como parâmetros
17	Parar Cronômetro
18	Apresentar a solução

Figura 23 - Pseudocódigo do programa do MATLAB para o GA

FONTE: Autoria própria (2014).

3.4 Análise do problema de particionamento usando ESBMC

Aqui se deve destacar que embora o ESBMC possa realizar todo tipo de verificação com relação a corretude do código em C que contém o problema de particionamento de HW/SW, iremos suprimir todas as verificações padrão, tais como a verificação de limites de

vetores e matrizes, divisão por zero, permitindo que a técnica encontre uma violação no programa apenas quando resolver o problema de particionamento de hardware/software. Para tal, na linha de comando, devem-se passar parâmetros específicos na hora de executar o ESBMC².

A Figura 24 mostra o uso da técnica ESBMC com as mesmas restrições e condições impostas ao ILP e ao GA. Dois valores devem ser controlados para obter os resultados e se realizar a otimização. Um é o valor inicial do custo de software, como definido na seção 2.13 do Capítulo 2. O outro é a condição de parada que quebra o algoritmo, valor de hardware mínimo, ou seja, a solução do problema de particionamento.

Da mesma forma que os algoritmos ILP e GA, o algoritmo ESBMC começa com a declaração dos custos de hardware, software, comunicação, S_0 , a matriz de incidência transposta e a matriz identidade. Neste ponto, as matrizes A e b são construídas, e o algoritmo começa a ficar diferente dos apresentados previamente neste trabalho.

Pela sua característica de verificador de modelo e de avaliador de corretude, é possível definir formalmente ao ESBMC com que tipo de valores as variáveis serão testadas. Um loop inicia, na linha 12, com a declaração para povoar as variáveis de decisão com valores booleanos não determinísticos. Estes valores mudarão a cada iteração (ou verificação do modelo), tentando gerar uma possível solução que atenda as restrições do problema. Se isto é alcançado, então uma solução factível foi encontrada, pois a diretiva ASSUME garante esse atendimento das restrições ($Ax \leq b$).

No fim do algoritmo, um predicativo controla a condição de parada, ou seja, uma declaração de verdadeiro-falso. Isto é realizado pela declaração ASSERT. Se o predicado é FALSO, então a otimização termina, ou seja, a solução do particionamento de hardware/software foi encontrada. A declaração ASSERT testa a função objetivo, ou seja, o custo do hardware, e parará se o custo de hardware encontrado for menor ou igual que a solução ótima. Entretanto, se o ASSERT retornar uma condição de VERDADEIRO, ou seja, o custo do hardware encontrado for maior que a solução ótima, então o verificador de modelo reinicia e uma nova possível solução é gerada e testada até que ASSERT gere uma condição de FALSO.

² Linha de comando: `esbmc nome_do_arquivo.c --no-bounds-check --no-pointer-check --no-div-by-zero-check -no-slice --boolector`

Deve-se lembrar de que uma condição de FALSO no tempo de execução aborta a execução do ESBMC e este apresenta o contraexemplo que causou a parada, o custo de hardware ótimo.

Neste trabalho, uma versão alternativa do algoritmo do ESBMC foi também testada, conforme descrito na Figura 25, onde as Equações (3) são usadas. No algoritmo ESBMC-2 não se faz necessário adicionar as variáveis auxiliares porque o ESBMC consegue trabalhar com o módulo da matriz de incidência transposta da Equação (3b). Isto causa a redução no número de variáveis a serem solucionadas. De forma similar ao descrito no Algoritmo ESBMC-1, a ideia por trás da declaração ASSUME é de controlar a restrição do custo do software e do ASSERT é de testar a função objetivo.

Sendo assim, a solução do problema, que apresenta o contraexemplo na técnica ESBMC, está indicado na linha 18 da Figura 24 e da Figura 25 (custo do hardware particionado).

```

1  Inicializar variáveis
2  Declarar o número de nodos e arestas
3  main {
4      Declarar custo de hardware de cada nodo como um vetor ( $h$ )
5      Declarar o custo de software de cada nodo como um vetor ( $s$ )
6      Declarar o custo de comunicação de cada aresta como um vetor ( $c$ )
7      Declarar o custo do software inicial ( $S_0$ )
8      Declarar a matriz de incidência transposta do grafo  $G$  ( $E$ )
9      Criar a matriz identidade  $I$  (dimensão = número de arestas)
10     Tendo como referência a (Eq. 9), criar a matriz  $A = \begin{bmatrix} -S & C \\ E & -I \\ -E & -I \end{bmatrix}$ 
11     Criar a matriz  $b = \begin{bmatrix} S_0 - 9 * nodos_{1 \times 1} \\ 0_{arestas \times 1} \\ 0_{arestas \times 1} \end{bmatrix}$ 
12     Definir as variáveis de solução ( $x_i$ ) como booleanas

```

```

13 Realizar verificação {
14     Povoar  $x_i$  com valores não determinísticos
15     Armazenar na matriz “temp” o resultado da matriz  $A_{i \times j}$  multiplicada por  $x_i$ 
16     Fazer o teste ESBMC (comando ASSUME) para o requisito “temp”  $\leq b$  (i.e.  $Ax \leq b$ )
17     Calcular a função objetivo baseada no valor factível de  $x_i$  encontrado pelo ESBMC
18     Verificação controlada pela função objetivo (custo de hardware) com ASSERT
19 }
20 Verificação formal realiza testes sistemáticos com diferentes valores de  $x_i$ 
21 }

```

Figura 24 - Pseudocódigo do ESBMC Algoritmo 1 (ESBMC-1)

FONTE: Autoria própria (2014).

```

1 Inicializar variáveis
2 Declarar o número de nodos e arestas
3 main{
4     Declarar custo de hardware de cada nodo como um vetor ( $h$ )
5     Declarar o custo de software de cada nodo como um vetor ( $s$ )
6     Declarar o custo de comunicação de cada aresta como um vetor ( $c$ )
7     Declarar o custo do software inicial ( $S_0$ )
8     Declarar a matriz de incidência transposta do grafo  $G(E)$ 
9     Criar a matriz identidade  $I$  (dimensão = número de arestas)
10    Tendo como referência a (Eq. 9), criar a matriz  $A = \begin{bmatrix} -S & C \\ E & -I \\ -E & -I \end{bmatrix}$ 
11    Criar a matriz  $b = \begin{bmatrix} S_0 - 9 * nodos_{1 \times 1} \\ 0_{arestas \times 1} \\ 0_{arestas \times 1} \end{bmatrix}$ 
12    Definir as variáveis de solução ( $x_i$ ) como booleanas

```

```

13 Realizar verificação {
14     Povoar  $x_i$  com valores não determinísticos
15     Calcular  $s(I-x)+c*|E*x|$  e armazenar na variável “restrição”
16     Realizar teste ESBMC (comando ASSUME) para o requisito “restrição”  $\leq S_0$ 
17     Calcular a função objetivo baseada no valor factível de  $x_i$  encontrado pelo ESBMC
18     Verificação controlada pela função objetivo (custo de hardware) com ASSERT
19 }
20 Verificação formal realiza testes sistemáticos com diferentes valores de  $x_i$ 
21 }

```

Figura 25. Pseudocódigo do ESBMC Algoritmo 2 (ESBMC-2)

FONTE: A autoria própria (2014).

3.5 Resumo

Neste capítulo foi apresentada a metodologia proposta para solução de um problema de particionamento de hardware e software de sistemas embarcados através de verificação formal baseada em SMT, com o ESBMC. Esta é a contribuição maior do presente trabalho, destacando-se ainda que ferramentas de verificação formal são usualmente utilizadas para verificar corretude de código e não para resolver problema de otimização. Foi ainda descrita e apresentada duas outras formas pelas quais um problema de particionamento pode ser resolvido, por Programação Linear Inteira (ILP) e por Algoritmo Genético (GA), com seus respectivos pseudocódigos. As técnicas ILP e GA são mais difundidas para realizar particionamento de HW/SW por meio de otimização, embora apenas o ILP e o ESBMC possam garantidamente obter a solução exata do problema. O conjunto das três técnicas, ILP, GA e ESBMC, sendo ainda a ESBMC desdobrada em duas devido à sua especificidade para representação matemática do problema, servem de referência para a realização dos testes do Capítulo seguinte, no qual o tempo de resposta e a corretude são avaliados comparativamente entre as três técnicas aqui detalhadas, bem como o impacto da escolha do valor do custo inicial do software para a solução da otimização.

4 Resultados experimentais

Este capítulo está dividido em quatro partes. A primeira é dedicada à descrição da configuração na qual os experimentos foram realizados, incluindo softwares, versões e ambientes. A segunda apresenta os resultados obtidos quando se realizou os experimentos com vetores de teste de autoria própria, focando em avaliar o impacto da mudança do valor inicial de S_0 em grafos indo de 10 a 100 nodos. Na terceira parte, uma nova rodada de resultados de experimentos é apresentada, mas desta vez S_0 é mantido fixo e um comparativo de desempenho e corretude são realizados com *benchmarks* adotados em trabalhos similares. E na última seção são destacadas as questões que impactam nos resultados obtidos, ou seja, as ameaças à validade do trabalho.

4.1 Configuração experimental

Foi usada a versão 1.24 do ESBMC na versão 64 bits. O solucionador de SMT usado foi o Boolector do Instituto de Modelos Formais e Verificação (FMV) da Áustria, na versão 2.0.1, que é disponibilizado sem custo na internet (<http://fmv.jku.at/boolector/>). E o código foi todo escrito em ANSI-C. Em uma rodada inicial de testes foi avaliado ainda o uso do solucionador Z3 da Microsoft, entretanto os resultados apresentados pelo Boolector foram melhores em termos de desempenho.

O software comercial MATLAB da MathWorks foi a ferramenta comercial usada para resolver a programação ILP (versão R2011a) e o GA (versão R2014a).

Todas as técnicas foram testadas em um desktop executando um sistema operacional Ubuntu 14.04.1 LTS de 64-bits com 24GB de memória RAM e processador i7 da Intel com *clock* de 3,40 GHz.

Em uma rodada inicial de testes, definiram-se os parâmetros ideais para uso no Algoritmo Genético, pois vários ajustes são possíveis, e estes têm impacto significativo no tempo e na precisão da otimização. Para os resultados apresentados neste trabalho, adotaram-se os seguintes parâmetros para o GA: população inicial de 300 indivíduos, número máximo de 300 iterações, taxa de cruzamento de 0,8, fração de migração de 0,2, função *@crossoverscattered* de cruzamento e gradiente de aceitação de uma solução (diferença entre a solução encontrada comparativamente à melhor anteriormente encontrada) de 1×10^{-10} .

Ainda em uma primeira rodada de testes, os tempos de execução em todas as técnicas foram medidos três vezes e uma média foi calculada. Entretanto, os tempos foram muito próximos entre si, demonstrando que não havia necessidade de se executar cada algoritmo mais do que uma vez para se chegar no tempo de solução dos problemas de particionamento de HW/SW. Uma condição de *time out* (TO), apresentada nos resultados deste capítulo, representa um tempo de execução maior que 7.200 segundos sem se obter uma solução, ou seja, duas horas de execução.

4.2 Mudando o valor de S_0

A Tabela 2 mostra os resultados para os casos de teste para 10, 25, 50 e 100 nodos. Para cada número de nodos, cinco diferentes valores do custo inicial do software (S_0) foram selecionados, indo do cenário onde todos os nodos são particionados para hardware a um valor onde todos os nodos são alocados em software. Este valor máximo de S_0 foi obtido através de cálculo manual: para 10 nodos S_0 é 90, para 25 nodos S_0 é 225, para 50 nodos S_0 é 450 e para 100 nodos S_0 é 900. No meio de cada faixa foram escolhidos valores equidistantes.

Nodos (arestas)	S_0	Solução exata		Algoritmo ILP		Algoritmo GA		Algoritmo ESBMC -1		Algoritmo ESBMC -2	
		Custo Hp	Custo Sp	Tempo(s)	Custo Hp	Tempo(s)	Custo Hp	Tempo(s)	Custo Hp	Tempo(s)	Custo Hp
10 (13)	0	60	0	0,83	60	6,47	60	0,20	60	0,15	60
	23	52	14	0,43	52	6,28	56	0,23	52	0,17	52
	45	38	43	0,60	38	5,15	42	0,26	38	0,17	38
	67	30	55	0,46	30	8,75	35	0,23	30	0,17	30
	90	0	90	0,31	0	7,90	17	0,16	0	0,15	0
25 (33)	0	150	0	0,37	150	16,04	150	2,82	150	0,41	150
	56	120	50	0,76	120	14,22	146	3,35	120	1,42	120
	112	87	111	1,43	87	12,92	117	19,45	87	13,05	87
	168	48	166	1,17	48	14,09	89	14,39	48	3,19	48
	225	0	225	0,32	0	14,76	57	0,76	0	0,41	0
50 (64)	0	300	0	0,40	300	29,28	300	13,34	300	1,41	300
	112	236	112	0,70	236	25,39	286	TO	-	5449,41	236
	224	173	223	13,55	173	29,39	254	TO	-	TO	-
	336	92	333	39,55	92	24,78	190	TO	-	TO	-
	450	0	450	0,33	0	26,92	130	2,80	0	1,19	0
100 (132)	0	600	0	0,45	600	87,85	544	81,53	600	7,87	600
	225	462	224	185,16	462	64,23	494	TO	-	TO	-
	450	311	448	24,07	311	54,46	471	TO	-	TO	-
	675	165	674	202,27	165	23,71	421	TO	-	TO	-
	900	0	900	0,66	0	31,20	299	14,26	0	5,25	0

LEGENDA: TO = *timeout*

Tabela 2 - Resultados empíricos de S_0 com 10, 25, 50 e 100 nodos

FONTE: Autoria própria (2014).

Com relação ao caso de 10 nodos e 13 arestas, a avaliação geral mostra que ESBMC-2 foi o mais rápido, seguido bem de perto pelo ESBMC-1 (1,34 vezes mais lento) e pelo ILP (3,25 vezes mais lento que o ESBMC-2). A técnica GA teve o pior desempenho, e ainda com um erro de 8,7% em relação à solução exata. No geral, todas as técnicas resolveram o problema de particionamento em menos de 10 segundos (menos de 1 segundo no caso do ILP e do ESBMC). A influência de S_0 foi sentida na maioria das técnicas: ILP teve uma diferença de 2,68 vezes entre a solução mais rápida e a mais lenta; GA apresentou uma diferença de 1,7 vezes entre os extremos de tempo; e o ESBMC teve uma diferença de desempenho de 1,12 vezes o tempo mais lento em comparação ao mais rápido.

Já nos testes com o grafo de 25 nodos e 33 arestas, ILP teve o melhor desempenho, sendo 4,56 vezes mais rápido que ESBMC-2 e 10 vezes mais rápido que ESBMC-1. GA apresentou o pior desempenho novamente, sendo 17 vezes mais lento que o ILP e apresentando um erro médio de cerca de 35% na solução. No geral, todas as técnicas resolveram o problema de particionamento em menos de 20 segundos. S_0 , desta vez, teve alto impacto na solução do problema de particionamento na maioria das técnicas: na técnica ILP, a diferença entre a solução mais rápida e a mais lenta foi de 4,47 vezes; no GA o tempo manteve-se praticamente constante, enquanto no ESBMC a diferença de desempenho foi de cerca de 31 vezes.

No vetor de teste com 50 nodos e 64 arestas, a técnica ESBMC começou a apresentar suas limitações de desempenho, com várias situações de *timeout*. A técnica ILP novamente obteve os melhores resultados: foi 2,5 vezes mais rápida que GA. GA apresentou um erro médio em relação à solução exata de 43,6%. Comparativamente entre as duas técnicas de ESBMC, o ESBMC-2 apresentou os melhores resultados de tempo e menos situações de *timeout*. GA novamente não teve impacto no que concerne a variação do valor de S_0 , já na ILP a diferença entre o desempenho mais rápida e a mais lenta foi de 120 vezes, na técnica com ESBMC a diferença ficou em 4.571 vezes.

E finalmente, com 100 nodos e 312 arestas, nos 20 testes realizados, o GA foi o que apresentou o melhor desempenho, sendo 1,6 vezes mais rápido que o ILP, mas com um erro médio na solução de 51%. ILP foi o segundo colocado com relação ao desempenho. O ESBMC, para as duas opções de algoritmo, apresentou *timeout* em 3 das 5 avaliações realizadas de cada. Mas no geral, como aconteceram nos demais testes, o ESBMC-2 apresenta um tempo de solução menor que o ESBMC-1. S_0 causou uma diferença no tempo para

solução de cerca de 451 vezes no algoritmo ILP, 3,7 no GA, 5,7 vezes no ESBMC-1 e 1,5 vezes no ESBMC-2, demonstrando seu impacto no desempenho de todas as técnicas.

Em termos de desempenho e corretude, a Tabela 2 mostra que se o problema de particionamento tiver 10 ou menos nodos, então ESBMC-1 e ESBMC-2 são as melhores opções. Iniciando com 25 nodos e maiores, ILP foi a melhor em termos de desempenho e corretude. Especificamente com relação ao ESBMC, se o problema de particionamento tiver até 25 nodos, então o algoritmo ESBMC-2 se mostra como uma opção de técnica viável para resolver o problema de particionamento de HW/SW. E, no geral, o ESBMC-2 sempre apresentou resultados melhores do que o ESBMC-1, dado que a complexidade no ESBMC-1 para resolver o problema de otimização é sempre maior que no ESBMC-2. A técnica com GA apresentou o melhor desempenho a partir de 100 nodos, mas a corretude apresentada não foi boa, apresentando erro no geral que variou de 8,7% a até 51% em relação a solução exata.

O teste serviu para demonstrar que o valor escolhido para S_0 tem um impacto muito grande nas técnicas ILP e ESBMC, principalmente quando a complexidade do problema de particionamento de HW/SW aumenta. Os melhores desempenhos foram resultantes da escolha de S_0 próximo dos extremos de custo, ou seja, particionando-se todo o sistema em software ou todo em hardware.

4.3 Testes de *benchmark*

Para realizar a Parte 2 dos testes, alguns vetores de teste fornecidos por Mann et al (2007) foram usados. A Tabela 3 lista os *benchmarks* desta segunda rodada de testes. Os vértices nos grafos correspondem a instruções de linguagem de alto nível. Custos de software são dimensionais em relação ao tempo e os custos de hardware representam a área ocupada no *chip*. Os primeiros três *benchmarks* vêm do MiBench, conforme apresentado em Guthaus et al (2001). Os *benchmarks* de *clustering* e de lógica *fuzzy* foram criados por Mann et al (2007) e são significativamente grandes. Ainda dos mesmos autores, ainda mais complexos *benchmarks* testaram os limites de aplicabilidade das técnicas descritas neste trabalho.

Nome	Nodos	Arestas	Descrição
CRC32	25	32	Verificação de redundância cíclica de 32-bits. Da categoria de telecomunicação do MiBench, conforme Guthaus et al (2001)
Patricia Insert	21	48	Rotina de inserção de valores. Usado para armazenar tabelas de rotinas. Da categoria de rede do MiBench, conforme Guthaus et al (2001)
Dijkstra	26	69	Calcula o caminho mais curto de um grafo. Da categoria de rede do MiBench, conforme Guthaus et al (2001)
Clustering	150	331	Algoritmo de segmentação de imagens de uma aplicação médica
RC6	329	448	Grafo de criptografia RC6
Fuzzy	261	422	Algoritmo de <i>clustering</i> baseado em lógica <i>fuzzy</i>
Mars	417	600	Código MARS da IBM

Tabela 3 - Descrição dos *benchmarks* usados

FONTE: Autoria própria (2014) e conforme indicado.

A Tabela 4 sumariza o resultado dos testes realizados com os *benchmarks* para os algoritmos ILP, GA, ESBMC-1 e ESBMC-2.

Nome	Nodos	Arestas	So	Solução Exata		Algoritmo ILP		Algoritmo GA		Algoritmo ESBMC-1		Algoritmo ESBMC-2	
				Custo Hp	Custo Sp	Tempo (s)	Custo Hp	Tempo (s)	Erro	Tempo (s)	Custo Hp	Tempo (s)	Custo Hp
CRC32	25	32	20	20	0	1,08	20	7,87	-5,0%	1,29	20	0,38	20
Patricia	21	48	10	47	4	0,41	47	59,33	0,0%	1,796	47	0,49	47
Dijkstra	26	69	20	40	0	0,44	40	11,19	0,0%	6,13	40	0,86	40
Clustering	150	331	50	256	3	4,10	256	379,01	-2,0%	354,50	256	32,65	256
RC6	329	448	600	703	535	452,79	703	1706,20	-14,7%	TO	-	713,71	703
Fuzzy	261	422	4578	13579	2129	TO	-	1691,24	-31,5%	TO	-	TO	-
Mars	417	600	300	887	300	409,37	887	TO	-	TO	-	TO	-

LEGENDA: TO = *timeout*

Tabela 4 - Testes empíricos com os *benchmarks*

FONTE: Autoria própria (2014).

De forma similar ao que aconteceu na primeira parte dos testes, o desempenho demonstrado pela técnica ILP nos *benchmarks* faz com que possa ser declarada como a melhor opção para resolver problemas de particionamento, mesmo considerando que o vetor Fuzzy tenha dado *time out* no ILP. Nenhuma técnica conseguiu resolver 100% dos vetores sem que tenham apresentando, pelo menos, um *time out*. GA foi a única técnica que conseguiu resolver o *benchmark* Fuzzy dentro do tempo limite, embora com um erro de 31,5% em relação à solução exata. Mais uma vez, o ESBMC-2 teve um desempenho melhor do que o ESBMC-1, chegando na solução exata do problema em menos tempo (de 3,4 a 10,5 vezes mais rápido).

Aumentando-se a complexidade do problema, e até se chegar a 150 nodos, a técnica ESBMC, mais especificamente o ESBMC-2, se mostrou uma boa escolha para resolver o problema de particionamento de hardware/software. Isto é porque a solução exata foi encontrada e o tempo de execução foi, na maior parte do tempo, não muito distante do apresentado pelo ILP (de 1,2 a 7,9 vezes maior).

Quando a complexidade dos vetores de teste aumenta, o algoritmo do ESBMC-1 tem a desvantagem de criar um problema ainda mais complexo a ser resolvido, isto porque inclui variáveis auxiliares do tamanho das arestas do grafo. Deve-se lembrar que o número total de variáveis a ser resolvido, no caso do ESBMC-1, é igual ao número de nodos adicionado ao número de arestas, enquanto no ESBMC-2 o algoritmo tem que resolver variáveis do tamanho do número de nodos.

4.4 Considerações sobre os resultados

De uma forma geral, os resultados aqui apresentados são dependentes do tipo de máquina e da configuração usada nos testes empíricos. Eventuais mudanças no código e na versão da ferramenta ESBMC também impactam nos resultados desta técnica, bem como a escolha do solucionador SMT adotado. E uma possível mudança do MATLAB para outra ferramenta de cálculo numérico, no que concernem os algoritmos ILP e GA, pode trazer mudanças nos tempos obtidos nestas técnicas.

4.5 Resumo

Este capítulo apresentou, dividido em duas partes principais, os resultados práticos do uso da técnica de verificação formal baseada nas teorias do módulo da satisfabilidade com outras duas técnicas mais usuais, a programação linear inteira e o algoritmo genético. Na primeira parte, que avaliou a influência da escolha do valor do custo inicial do software no particionamento de HW/SW, ficou claro que há uma influência muito grande deste em todas as técnicas, embora em menor intensidade do GA. No ILP, a diferença entre os tempos, para um mesmo vetor de teste, chegou a até 451 vezes entre o menor e o maior tempo. Já no ESBMC, essa diferença de tempo para se obter a solução chegou a 4.571 vezes. À luz dos resultados, a recomendação que se faz, para tornar a solução mais rápida, é que se escolham valores de S_0 mais próximos dos extremos, ou seja, próximo de um particionamento onde todos os componentes estão em software ou todos estão em hardware. Na segunda parte, que

utilizou *benchmarks*, avaliou-se o limite de uso de cada técnica, estando o ILP em destaque como o melhor desempenho no geral, mas com um bom resultado obtido pelo ESBMC-2, o que serve de comprovação e recomendação para o uso do ESBMC como ferramenta para solucionar problemas de particionamento de hardware e software.

5 Conclusões

Neste trabalho foi demonstrada a viabilidade de se usar uma técnica de verificação formal de software para resolver problemas de particionamento de hardware/software, sem que tenha sido necessário adaptar a ferramenta ESBMC, bastando usar os algoritmos conforme descritos na Figura 24 e na Figura 25. Salvo melhor juízo ou convicção, a proposta aqui apresentada é a primeira na qual se usa uma técnica de verificação baseada em teorias do módulo da satisfabilidade para resolver um problema de particionamento.

Com relação aos testes empíricos do coprojeto de primeira geração, comparando-se as quatro técnicas apresentadas neste trabalho para resolver particionamento de hardware/software, ficou evidente que nenhum é indicado para particionar mais do que 400 nodos ou componentes do sistema a ser otimizado, isto porque o tempo de computação para resolver o problema alcançou algumas horas em um computador desktop padrão. Entretanto, se forem considerados menos que 400 componentes a serem particionados, então é possível usar a técnica de Programação Linear Inteira como a melhor alternativa dentre as técnicas utilizadas. Especificamente com relação à técnica de ESBMC, que é a contribuição maior deste trabalho, esta se mostrou uma ótima alternativa para resolver problemas com até 329 nodos. O Algoritmo Genético apresentou um resultado intermediário dentre as técnicas em relação a desempenho, mas o erro apresentado pela técnica chegou a 51% no pior dos casos.

Se considerarmos as soluções de prateleira, isto é, soluções comerciais prontas, como o MATLAB, então ILP e GA são simples de serem usadas e não demandam desenvolvimentos adicionais. Entretanto, estas são soluções pagas, o que pode ser uma desvantagem em alguns casos. Por outro lado, o ESBMC apresenta uma licença do estilo BSD (*Berkeley Software Distribution*) e pode ser baixado e utilizado livre de custos. Portanto, acaba sendo um ponto positivo para os Algoritmos ESBMC-1 e ESBMC-2 apresentados neste trabalho. Mesmo se considerarmos que existem soluções de prateleira com código aberto, como o GNU MathProg por exemplo, que é parte integrante do projeto GNU GLPK, são necessárias várias linhas de código para representar os mesmos algoritmos descritos aqui no MATLAB, ou seja, a complexidade de uso aumenta quando não se usa o MATLAB.

Se o foco for dado nas duas versões do ESBMC aqui apresentadas, os resultados dos testes demonstram que o ESBMC-2 teve um desempenho melhor que o ESBMC-1. Portanto,

quando modelando um problema de particionamento de hardware/software, é possível manter a versão com o módulo da matriz de incidência transposta nas fórmulas aqui apresentadas. Isto implica menos trabalho para preparar a função objetivo e as restrições, além do que causa redução no número de variáveis a serem resolvidas pelo algoritmo. Isto explica o porquê do ESBMC-2 ter apresentado tempos de solução menores que o ESBMC-1.

Quanto à contribuição secundária na avaliação do impacto do valor inicial do custo de software (S_0) usado nas restrições da otimização, o que trabalhos anteriores ainda não tinham tratado, foi demonstrado aqui que S_0 tem um impacto muito grande nos resultados de desempenho, independentemente da técnica escolhida para resolver o problema de particionamento de hardware/software, menos no GA, onde o tempo para se obter uma solução praticamente não muda em virtude da sua forma de resolver a otimização. Mesmo se for analisado um problema com o mesmo número de nodos e arestas, um valor diferente de S_0 pode resultar em um tempo computacional completamente distinto e distante. A conclusão que se chega, com relação ao S_0 é que se for escolhido um valor extremo para ele, ou seja, com todos os componentes em hardware ou em software, então o tempo de solução do problema é relativamente pequeno. Por outro lado, ele aumenta significativamente e pode ser 451 vezes maior na técnica ILP, 25 vezes maior no ESBMC-1 e 4.571 vezes maior no ESBMC-2.

Vale destacar que, além da questão intrínseca da complexidade dos vetores de testes impactando no tempo de resposta das técnicas aqui apresentadas, há ainda o problema do aumento das variáveis a serem solucionadas para o caso do ESBMC-1. Fica claro que a técnica empregada na busca em árvore do ILP é mais eficiente do que a empregada pelo ESBMC. As duas técnicas, ILP e ESBMC, encontram a solução exata e realizam busca em árvore binária para encontrar a solução, mas o aumento da complexidade dos problemas explicita a questão de que os mecanismos e heurísticas usadas pelo ILP são melhores do que as do ESBMC, que na verdade são as do solucionador empregado para resolver o problema.

Uma questão final que merece ser discutida tem a ver com o fato do limite apresentado pelo ESBMC para resolver problemas de particionamento. Ficou demonstrado aqui que, dependendo do valor do custo do software inicial, o ESBMC consegue resolver em um tempo aceitável (menos de 12 minutos) problemas com até 329 nodos e 448 arestas. A pergunta que se faz é se esse tipo de sistema, com 329 nodos, é representativo de um problema real de particionamento de hardware e software. A resposta depende na verdade da granularidade da

modelagem do sistema que se quer resolver, pois alguns pesquisadores propõem modelos de grão-fino, onde cada instrução pode ser mapeada para hardware ou software. E isto pode levar a um problema com dezenas de milhares de nodos ou até mais. Entretanto, existe uma outra corrente de pesquisadores advogando por modelos de grão-grosso, onde as decisões são feitas em componentes maiores, em que até sistemas complexos podem ser compostos de apenas algumas poucas dezenas de nodos para particionar. A princípio, um modelo de grão-fino possibilita uma melhor partição, mas ao custo do aumento exponencial do espaço de busca da solução. Portanto a escolha do modelo é que vai ditar a aplicabilidade do uso de ESBMC como uma ferramenta de particionamento de hardware e software.

5.1 Trabalhos futuros

Um possível futuro trabalho está na aplicação da ferramenta de verificação baseada em teorias do módulo da satisfabilidade para resolver um problema de particionamento para arquiteturas mais complexas, com mais de uma CPU e a mudança de programas com apenas uma tarefa para multiprogramação e multiprocessamento. Isto cobre a avaliação para o coprojeto de segunda e terceira geração.

Especificamente em relação ao GA, esta é uma técnica que tem muitos parâmetros que podem ser modificados e ajustados. Outros trabalhos futuros podem avaliar melhorias de precisão e desempenho, como mudanças na taxa de mutação e cruzamento, o tipo e o tamanho da seleção.

Quanto ao ESBMC, ainda é possível se estudar como diminuir seu tempo de processamento para problemas de particionamento de hardware/software. Isto pode ser realizado avaliando-se os gargalos da sua metodologia de solução interna como, por exemplo, o procedimento de decisão em tempo de execução ou ainda a codificação adotada para o programa. Mudanças no algoritmo do solucionador SMT também podem ser implementadas.

Referências bibliográficas

ARATÓ, P.; JUHÁSZ, S.; MANN, Z.; ORBÁN, A.; PAPP, D. *Hardware/software partitioning in embedded system design*. In: PROCEEDINGS OF THE IEEE INTERNATIONAL SYMPOSIUM OF INTELLIGENT SIGNAL PROCESSING, 2003. p. 192-202.

ARATÓ, P.; MANN, Z.; ORBÁN, A. *Algorithmic aspects of hardware/software partitioning*. In: ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS (TODAES), v. 10, 2005, p. 136–156.

ARMANDO, A.; MANTOVANI, J.; PLATANIA, L. *Bounded model checking of software using SMT solvers instead of SAT solvers*. In: INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER, v. 11, n. 1, 2009, p. 69–83.

BAIER, C.; KATOEN, J-P. *Principles of Model Checking*. London: The MIT Press, 2008.

BELEGUNDU, Ashok; CHANDRUPATLA, Tirupathi. *Optimization Concepts and Applications in Engineering*. Segunda edição. New York: Cambridge University Press, 2011.

BELOV, A.; DIEPOLD, D.; HEULE, M.; JÄRVISALO, M. *Proceedings of SAT COMPETITION 2014. Solver and Benchmark Description*. University of Helsinki. Dinamarca. Disponível em: <https://helda.helsinki.fi/bitstream/handle/10138/135571/sc2014_proceedings.pdf>. Acesso em: 16 novembro 2014.

BHATTACHARYA, A.; KONAR, A.; DAS, S.; GROSAN, C.; ABRAHAM, A. *Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm*. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPLEX, INTELLIGENT AND SOFTWARE INTENSIVE SYSTEMS, 2008, p. 171-176.

BIERE, A. *Bounded model checking*. In: Biere, A.; Heule, M.; van Marren, H.; Walsh, T. (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 457–481.

BIERE, A.; CIMATTI, A.; CLARKE, E.; ZHU, Y. *Symbolic model checking without BDDs*. In: TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF

SYSTEMS (TACAS), Lecture Notes in Computer Science, v. 1579, 1999, p. 193-207.

BRADLEY, S.; HAX, A.; MAGNANTI, T. *Applied Mathematical Programming*. 1st edition. Reading, Massachusetts: Addison-Wesley, 1977.

CLARKE, E. *The birth of model checking*. In: Grumberg, O.; Veith, H. (Org.). 25 YEARS OF MODEL CHECKING. Berlin: Springer-Verlag, 2008, p. 1-26.

CLARKE, E.; EMERSON, E.; SIFAKIS, J. *Model checking: algorithmic verification and debugging*. In: COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY (ACM), v. 52, i. 11, 2009, p. 74-84.

CLARKE, E.; GRUMBERG, O.; PELED, D. *Model Checking*. Cambridge: MIT Press, 1999.

CLARKE, E.; KROENING, D.; LERDA, F. *A Tool for Checking ANSI-C Programs*. In: PROCEEDINGS OF THE TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS (TACAS), LNCS 2988, 2004, p. 168-176.

COK, D.; DEHARBE, D.; WEBER, T. *SMT-COMP 2014 Participants*. SMT-COMP. Gramma Tech, USA. Disponível em: < <http://www.smtcomp.org/>>. Acesso em: 16 novembro 2014.

CORDEIRO, Lucas Carvalho. SMT-based bounded model checking of multi-threaded software in embedded systems. 2011. 220f. Tese (Doutorado em Ciência da Computação) – Faculty of Engineering and Applied Science, University of Southampton, Southampton, Inglaterra.

CORDEIRO, L.; FISCHER, B. *Verifying multi-threaded software using SMT-based context-bounded model checking*. In: PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2011, pp. 331-340.

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. *SMT-based bounded model checking for embedded ANSI-C software*. In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, v. 38, ed. 4, 2012, p. 957-974.

DARWICHE, A.; PIPATSRISAWAT, K. *Complete algorithms*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 99-130.

DASKALAKI, S.; BRIBAS, T.; HOUSOS, E. *An integer programming formulation for a case study in university timetabling*. In: EUROPEAN JOURNAL OF OPERATIONAL RESEARCH, v. 153, 2004, p. 117-135.

D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. *A Survey of Automated Techniques for Formal Software Verification*. In: IEEE TRANSACTIONS ON CAD OF INTEGRATED CIRCUITS AND SYSTEMS, v. 27, issue 7, 2008, p. 1165-1178.

DEMRI, S.; GASTIN, P. *Specification and verification using temporal logics*. In: D'Souza, D.; Shankar, P. (Org.). MODERN APPLICATION OF AUTOMATA THEORY. Volume 2 of IISc Research Monographs Series. Cingapura: World Scientific Publishing, 2012, p. 457-494.

EMERSON, E. *The beginning of model checking: a personal perspective*. In: O. Grumberg, H. Veith (Org.). 25 YEARS OF MODEL CHECKING. Berlim: Springer-Verlag, 2008, p. 27-45.

EIMURI, T.; SALEHI, S. *Using DPSO and B&B Algorithms for Hardware/Software Partitioning in Co-design*. In: PROCEEDINGS OF THE SECOND INTERNATIONAL CONFERENCE ON COMPUTER RESEARCH AND DEVELOPMENT, 2010, p. 416-420.

FRANCO, J; MARTIN, J. *A history of satisfiability*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 3-74.

GADELHA, Mikhail Yasha Ramalho. *Verificação Baseada em Indução Matemática para Programas C++*. Dezembro de 2013. 101 f. Dissertação (Mestrado em Engenharia Elétrica) – Faculdade de Tecnologia, Universidade Federal do Amazonas, Manaus, 2013.

GANAI, M.; GUPTA, A. *Accelerating high-level bounded model checking*. In: PROCEEDINGS OF THE IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD), 2006, p. 794–801.

GOMES, C.; KAUTZ, H.; SABHARWAL, A.; SELMAN, B. *Satisfiability solvers*. In: F. Harmelen, V. Lifschitz, B. Porter (Org.). HANDBOOK OF KNOWLEDGE REPRESENTATION, San Diego: Elsevier Science, 2007, p. 89-134.

GUPTA, R.; DE MICHELI, G. *Hardware/Software Co-Design*. In: PROCEEDINGS OF IEEE, v. 85, n. 3, 1997, p. 349-365.

GUTHAUS, M.; RINGENBERG, J.; ERNST, D.; AUSTIN, T.; MUDGE, T.; BROWN, R. *MiBench: a free, commercially representative embedded benchmark suite*. In: PROCEEDINGS OF THE IEEE 4TH ANNUAL WORKSHOP ON WORKLOAD CHARACTERIZATION, 2001, p. 3-14.

HAUPT, R.; HAUPT, S. *Practical genetic algorithms*. 2nd edition. New York: John Wiley & Sons, 2004.

HONG, L.; CAI, J. *The application guide of mixed programming between MATLAB and other programming languages*. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON COMPUTER AND AUTOMATION ENGINEERING (ICCAE), 2010, p. 185-189.

HUONG, P.; BINH, N. *An approach to design embedded systems by multi-objective optimization*. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON ADVANCED TECHNOLOGIES FOR COMMUNICATIONS, 2012, p. 165-169.

JIANG, Y.; ZHANG, H.; JIAO, X.; SONG, X.; HUNG, W.; GU, M.; SUN, J. *Uncertain Model and Algorithm for Hardware/Software Partitioning*. In: PROCEEDINGS OF THE IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2012, p. 243-248.

JIANLIANG, Y.; P. MANMAN, P. *Hardware/Software partitioning algorithm based on wavelet mutation binary particle swarm optimization*. In: PROCEEDINGS OF THE IEEE 3rd INTERNATIONAL CONFERENCE ON COMMUNICATION SOFTWARE AND NETWORK, 2011, p. 347-350.

JIGANG, W.; SRIKANTHAN, T.; CHEN, G. *Algorithmic Aspects of Hardware/Software Partitioning: 1D Search Algorithms*. In: IEEE TRANSACTIONS ON COMPUTERS, v. 59, issue 4, 2010, p. 532 – 544.

KROENING, D.; TAUTSCHNIG, M. *CBMC – C Bounded Model Checker (Competition Contribution)*. In: TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS (TACAS), Lecture Notes in Computer Science, v. 8413, 2014, p. 389-391.

KULLMANN, O. *Fundamentals of branching heuristics*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009. p. 205-244.

LI, S.; LIU, Y.; HU, S.; HE, X.; ZHANG, Y., ZHANG, P.; YANG, H. *Optimal partition with block-level parallelization in C-to-RTL synthesis for streaming applications*. In: PROCEEDINGS OF THE 18th ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2013, p. 225-230.

LUO, L.; HE, H.; LIAO, C.; DOU, Q. *Hardware/Software partitioning for heterogeneous multicore SOC using particle swarm optimization and immune clone (PSO-IC) algorithm*. In: PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON INFORMATION AND AUTOMATION (ICIA), 2010, p. 490-494.

MANN, Z.; ORBÁN, A.; ARATÓ, P. *Finding optimal hardware/software partitions*. In: FORMAL METHODS IN SYSTEM DESIGN. Springer Science+Business Media, 2007, v. 31, p. 241-263.

MARQUES-SILVA, J.; LYNCE, I.; MALIK, S. *Conflict-driven clause learning SAT solvers*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 131-153.

MATLAB R2014a. Optimization Toolbox: User's Guide. Natick, Massachusetts: The MathWorks, Inc., 2014.

MITCHELL, M. *An introduction to genetic algorithm*. 5th Edition. Cambridge: MIT Press, 1999.

MOURA, L.; BJORNER, N. *Engineering DPLL (T) + Saturation*. In: PROCEEDING OF THE 4th INTERNATIONAL JOINT CONFERENCE ON AUTOMATED REASONING (IJCAR), 2008, p. 475-490.

PRESTWICH, S. *CNF encodings*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 75-97.

RAMALHO, M.; FREITAS, M.; SOUSA, F.; MARQUES, H.; CORDEIRO, L.; FISCHER, B. *SMT-based bounded model checking of C++ programs*. In: INTERNATIONAL

CONFERENCE AND WORKSHOPS ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, IEEE, 2013, p. 147-156.

RAO, S. *Engineering Optimization: Theory and Practice*. 4 ed. Roboken: John Wiley & Sons, 2009.

SAPIENZA, G.; SECELEANU, T.; CRNKNOVIC, I. *Partitioning Decision Process for Embedded Hardware and Software Deployment*. In: IEEE 37th ANNUAL CONFERENCE AND WORKSHOPS ON COMPUTER SOFTWARE AND APPLICATIONS (COMPSACW), 2013, p. 674-680.

SEBASTIANI, R.; TACHELLA, A. *SAT Techniques for Modal and Description Logics*. In: A. Biere, M. Heule, H. van Marren, T. Walsh (Org.). HANDBOOK OF SATISFIABILITY. Amsterdam: IOS Press, 2009, p. 781-824.

SILVA, J.; SAKALLAH, K. *Conflict analysis in search algorithms for satisfiability*. In: PROCEEDINGS EIGHTH IEEE INTERNACIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE, 1996, p. 467-469.

TEICH, J. *Hardware/Software Codesign: The Past, the Present, and Predicting the Future*. In: PROCEEDINGS OF THE IEEE, v. 100, 2012, p. 1411-1430.

TRANQUILLO, Joseph. *Matlab for Engineering and the Life Sciences*. Synthesis Lectures on Engineering. Morgan & Claypool: 2011.

WANG, G.; GONG, W.; KASTNER, R. *Application partitioning on programmable platforms using the ant colony optimization*. In: JOURNAL OF EMBEDDED COMPUTING – EMBEDDED PROCESSORS AND SYSTEMS: ARCHITECTURAL ISSUES AND SOLUTIONS FOR EMERGING APPLICATIONS, v. 2, 2006, p. 119-136.

WANG, H.; ZHANG, H. *Improved HW/SW partitioning algorithm on efficient use of hardware resource*. In: 2nd INTERNATIONAL CONFERENCE ON COMPUTER AND AUTOMATION ENGINEERING (ICCAE), 2010, p. 682-685.

Apêndice A: Publicações

SBESC 2014 (IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais) – aceito e publicado:

TRINDADE, A.; CORDEIRO, L., *Aplicando Verificação de Modelos para o Particionamento de Hardware/Software*. In: ANAIS DO IV SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS (SBESC), 2014.

Journal Design Automation for Embedded Systems - Qualis B1 – submetido em 21/05/2014, aceito em 03/10/2014 com pedido de revisões (*major revision*), revisado em 14/10/2014:

TRINDADE, A.; CORDEIRO, L. *Applying SMT-based Verification to Hardware/Software Partitioning in Embedded Systems*. In: DESIGN AUTOMATION FOR EMBEDDED SYSTEMS (DAEM), em publicação.