ALESSANDRO JUSTINIANO MENDES

VERIFICAÇÃO DE EQUIVALÊNCIA DE CIRCUITOS COMBINACIONAIS DISSIMILARES ATRAVÉS DO REAPROVEITAMENTO DE CLÁUSULAS DE CONFLITO

Belo Horizonte
04 de dezembro de 2008

ALESSANDRO JUSTINIANO MENDES
Orientador: Antônio Otávio Fernandes

VERIFICAÇÃO DE EQUIVALÊNCIA DE CIRCUITOS COMBINACIONAIS DISSIMILARES ATRAVÉS DO REAPROVEITAMENTO DE CLÁUSULAS DE CONFLITO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
04 de dezembro de 2008

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Verificação de Equivalência de Circuitos Combinacionais Dissimilares Através do Reaproveitamento de Cláusulas de Conflito

ALESSANDRO JUSTINIANO MENDES

Dissertação defendida e aprovada pela banca examinadora constituída por:

Docteur Antônio Otávio Fernandes – Orientador
Universidade Federal de Minas Gerais

Ph. D. Claudionor José Nunes Coelho Júnior
Universidade Federal de Minas Gerais

Ph. D. Diógenes Cecílio da Silva Júnior
Universidade Federal de Minas Gerais

Ph. D. José Monteiro da Mata
Universidade Federal de Minas Gerais

Belo Horizonte, 04 de dezembro de 2008

Àquela que me criou, responsável por tudo que sou e tudo que serei, dedico este trabalho.

Agradecimentos

Primeiramente, agradeço a Deus pelo dom da vida e pela inteligência necessária para cumprir mais esta etapa. À minha mãe e meus familiares, obrigado pelo apoio incondicional e pela compreensão pelos momentos em que estive ausente.

Ao professor Antônio Otávio, que acreditou em mim logo que iniciei minha carreira acadêmica e tornou-se meu orientador e amigo. Professores Ângelo, Claudionor e Renato, obrigado pela orientação durante estes quase sete anos de UFMG. Aos demais professores, agradeço o ensinamento e tenham a certeza de que vocês contribuíram de alguma forma para a minha formação.

Aos meus grandes amigos Diego, Heitor, Marcelo e Thiago, agradeço por sempre estarem ao meu lado, auxiliando até mesmo nas discussões técnicas sobre Ciência da Computação. A todos os amigos e colegas que fiz na Universidade, espero podermos nos encontrar pelas estradas da vida.

Pessoal do CRC, da secretaria do DCC e demais funcionários: obrigado por manter toda a estrutura necessária para que nós, alunos, possamos estudar em um dos melhores cursos de graduação e pós-graduação do país. Continuem com esta dedicação que vocês empregam no dia-a-dia.

Aos colegas do LECOM e da Jasper, sempre dispostos a compartilhar o conhecimento, contribuindo para que realizássemos diversos trabalhos em equipe.

E aos companheiros de Bier Leverpilsen, cujo futebol aos sábados mostrou que a Universidade também pode ser um ambiente de lazer.

Resumo

Os circuitos integrados encontram-se cada dia mais presentes em nossas vidas. Dos celulares que falamos aos carros que dirigimos, em quase todos os momentos é possível encontrarmos um dispositivo eletrônico em ação. Isto gera uma crescente demanda por circuitos mais ágeis e compactos, fazendo com que estes se tornem complexos e caros. Uma parte considerável do tempo e dinheiro dedicados ao projeto e desenvolvimento de circuitos é destinado a verificar a presença de erros dos mesmos. A verificação de equivalência entre dois circuitos combinacionais é uma das técnicas mais utilizadas atualmente para verificar se, dadas as mesmas entradas para dois circuitos combinacionais, em qualquer estágio do projeto, eles geram saídas equivalentes. Por ser um tema atual, diversas abordagens têm sido propostas no intuito de aumentar a capacidade de verificar circuitos cada vez maiores em um menor espaço de tempo, entretanto nenhuma obteve notório sucesso quando os circuitos são dissimilares. Este trabalho apresenta e analisa metodologias para o reaproveitamento das cláusulas de conflito entre partições adjacentes durante a verificação de equivalência entre dois circuitos combinacionais dissimilares particionados, utilizando resolvedores SAT.

Abstract

As time goes by, integrated circuits are becoming ever more present in our lives. From the mobile phones we use to the cars we drive, we have almost constant interaction with electronic devices. This proliferation leads to the necessity for more agile and compact circuits, which in turn, makes them more complex and expensive. To produce error-free circuits, a considerable amount of time and money is spent on hardware verification during the design process. Equivalence checking of two combinational circuits is one of the most widely used techniques, which checks whether two combinational circuits (at any design level) that are given the same input data will produce equivalent output data. During the last few years, researchers have attempted to develop techniques to increase the verification of larger circuits and decrease the time spent on this task, but there has been no notable success for dissimilar circuits. This thesis presents and analyzes methodologies that rely on conflict clause reuse between circuit partitions during the equivalence checking of two dissimilar combinational circuits using a SAT solver.

Sumário

1 Introdução **1**
1.1 Objetivos . . . 2
1.2 Motivação . . . 3
1.3 Organização . . . 4

2 Conceitos Básicos **5**
2.1 Classificação de Circuitos . . . 5
2.2 Miter . . . 7
2.3 Satisfabilidade . . . 7
2.4 Resolvedor SAT . . . 8
2.4.1 Análise de Conflito . . . 13

3 Revisão Bibliográfica **18**
3.1 Diagrama de Decisão Binário . . . 18
3.2 Diagrama de Momento Binário . . . 20
3.3 Geração Automática de Padrões de Teste . . . 21
3.4 Aprendizado Recursivo . . . 23
3.5 Resolvedores SAT . . . 24
3.6 Técnicas Integradas . . . 25
3.7 Trabalhos Relacionados . . . 25

4 Metodologia **28**
4.1 Algoritmo . . . 28
4.2 Particionamento dos Circuitos . . . 30
4.3 Criação do Miter . . . 34
4.4 Conversão do Formato ISCAS para CNF . . . 35
4.5 Seleção de Conflitos Compartilhados . . . 38
4.6 Heurísticas para Reaproveitamento de Cláusulas de Conflito . . . 38
4.6.1 Reaproveitamento de Todas as Cláusulas de Conflito . . . 39
4.6.2 Reaproveitamento de Cláusulas de Conflito com Tamanho Específico . 40
4.6.3 Reaproveitamento de Cláusulas de Conflito que Constituem Funções Booleanas . . . 40

4.6.4 Reaproveitamento de Cláusulas de Conflito Unitárias ou que Constituem Funções Booleanas . . . 47
4.7 Concatenação das Cláusulas de Conflito e Cláusulas que Compõem a Partição 49

5 Resultados **50**
5.1 Características dos Experimentos . . . 50
5.2 Resolvedor SAT Utilizado . . . 50
5.3 Benchmark Utilizado . . . 52
5.3.1 Geração de Circuitos Dissimilares . . . 52
5.3.2 Circuitos Gerados para Testes . . . 52
5.4 Resultados Experimentais . . . 56

6 Conclusão **67**

A Algoritmo para busca em largura em um grafo **69**

Referências Bibliográficas **71**

Lista de Figuras

2.1 Exemplo de circuitos combinacional e seqüencial 6
2.2 Miter construído a partir de dois circuitos combinacionais com duas saídas cada. 7
2.3 Construção de um grafo de implicação (parte 1). 14
2.4 Construção de um grafo de implicação (parte 2). 15
2.5 Cortes no grafo de implicação. 16

3.1 BDD ordenado para a função $[(x_1 \wedge x_3 \wedge x_4) \vee x_2]$. 19
3.2 BDD reduzido e ordenado para a função $[(x_1 \wedge x_3 \wedge x_4) \vee x_2]$. 20
3.3 Justificando $i = 1$ e $j = 0$. 22
3.4 Justificando $i = 0$ e $j = 1$. 22
3.5 Circuito na qual a técnica mostrada na seção 3.3 não funciona. 23

4.1 Circuito com 13 entradas e 5 saídas. 31
4.2 Representação do circuito da figura 4.1 como um grafo. 32
4.3 Circuito da figura 4.1 particionado por TFI. 33
4.4 Miter construído a partir de dois circuitos semelhantes ao da figura 4.1. 34
4.5 Apenas as cláusulas de conflito compartilhadas entre partições adjacentes são aproveitadas. 39

5.1 Circuito original (C) e sua cópia (C') com os operandos invertidos 53
5.2 Multiplicação decimal com troca de operandos 53
5.3 Gráfico da utilização da memória para multiplicadores *Carry-Lookahead*. 59
5.4 Gráfico da utilização da memória para multiplicadores *Dadda Tree*. 66
5.5 Gráfico da utilização da memória para multiplicadores *Wallace Tree*. 66

Lista de Tabelas

4.1 Conversão de expressões booleanas para expressões CNF 37

5.1 Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Carry-Lookahead* a serem verificados 54
5.2 Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Dadda Tree* a serem verificados 55
5.3 Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Wallace Tree* a serem verificados. 55
5.4 Tempo de execução, em segundos, para multiplicadores *Carry-Lookahead* de tamanhos variados 60
5.5 Tempo de execução, em segundos, para multiplicadores *Dadda Tree* de tamanhos variados 61
5.6 Tempo de execução, em segundos, para multiplicadores *Wallace Tree* de tamanhos variados 62
5.7 Número de cláusulas de conflito reaproveitadas para multiplicadores *Carry Lookahead* de tamanhos variados 63
5.8 Número de cláusulas de conflito reaproveitadas para multiplicadores *Dadda Tree* de tamanhos variados 64
5.9 Número de cláusulas de conflito reaproveitadas para multiplicadores *Wallace Tree* de tamanhos variados. 65

Lista de Algoritmos

2.1 Algoritmo utilizado em resolvedores SAT modernos baseado em DPLL 10
4.1 Algoritmo para verificação de equivalência de circuitos combinacionais utilizando particionamento e reaproveitamento de cláusulas de conflito 29
4.2 Algoritmo para particionamento de circuito por TFI 31
4.3 Algoritmo para reaproveitamento de cláusulas de conflito com tamanho específico 40
4.4 Algoritmo para agrupamento de cláusulas de conflito 42
A.1 Algoritmo para busca em largura (BFS) 69

Capítulo 1

Introdução

> 'There are two ways to write error-free programs; only the third one works."
>
> Alan J. Perlis

Este trabalho visa contribuir para a solução do problema da verificação de equivalência entre circuitos combinacionais (CEC, do inglês *Combinational Equivalence Checking*) apresentando e analisando o impacto de técnicas que utilizam SAT e reaproveitam as cláusulas de conflito entre as partições adjacentes de um circuito.

Verificar um circuito integrado é o ato de testar o seu correto funcionamento baseado na especificação do projeto. Esta é uma etapa de grande importância para as empresas da área, pois se um circuito for lançado no mercado com algum defeito, isto pode acarretar prejuízos não somente financeiros mas também pode arruinar a imagem da empresa. Portanto, esta atividade influencia diretamente no preço e no tempo de lançamento do produto no mercado. A verificação de equivalência consiste em provar se dois circuitos apresentam o mesmo comportamento, ou seja, dado um conjunto específico de entradas para os dois circuitos, as saídas devem ser idênticas. De posse dessas informações, definimos o problema de CEC como: dados dois circuitos combinacionais com o mesmo número de entradas e saídas, verificar se para os mesmos valores de entrada, os valores das saídas dos dois circuitos são equivalentes.

O problema de CEC pode ser tratado de forma incremental ao se dividir um circuito em várias partes e verificar cada uma separadamente, reaproveitando as informações relevantes de uma partição, geradas durante a sua verificação, na partição seguinte. Desta forma, um problema maior é dividido em problemas menores a fim de reduzir o tempo de execução da solução.

Uma das maneiras de se verificar cada partição é através da utilização de resolvedores de satisfabilidade, ou como são mais conhecidos, resolvedores SAT [Davis e Putnam, 1960]. Tais programas têm como objetivo procurar assinalamentos válidos para as variáveis de uma dada fórmula booleana, tal que ela seja verdadeira ou provar que tal assinalamento não existe, sendo a fórmula falsa. Caso ela seja avaliada como verdadeira, o resolvedor deve retornar SAT. Caso contrário, retorna UNSAT. Como exemplos de resolvedores SAT atuais, podemos

citar o MiniSat [Eén e Sörensson, 2003], o Berkmin [Goldberg e Novikov, 2002] e o zChaff [Moskewicz et al., 2001].

CEC é um problema co-$\mathcal{NP}$-difícil[1] conforme mostrado em [Molitor e Mohnke, 2004], porém existem circuitos com determinadas características e formas de representá-los que os tornam mais tratáveis. Por exemplo, circuitos representados por diagramas de decisão binária reduzidos e ordenados (ROBDDs, do inglês *Reduced Ordered Binary Decision Diagrams*) [Bryant, 1986] ou por diagramas de momento binário reduzidos e ordenados (ROBMDs, do inglês *Reduced Ordered Binary Moment Diagrams*) [Bryant e Chen, 1995] são verificados em tempo constante. Entretanto, algumas classes de circuitos não são eficientemente representáveis. ROBDDs e ROBMDs para grande parte dos multiplicadores aritméticos, por exemplo, possuem tamanho exponencial. Nestes casos, é recomendável o uso de outras técnicas como geração automática de padrões de teste (ATPG, do inglês *Automatic Test Pattern Generation*) [Brand, 1993] ou SAT. Detalhes sobre estas representações bem como suas limitações podem ser encontradas no capítulo 3.

Este trabalho tem como foco os circuitos dissimilares, ou seja, circuitos que possuem pouca ou nenhuma similaridade estrutural entre si. Em contraste, existem os circuitos similares, que são aqueles que apresentam similaridades estruturais entre eles. Como veremos no capítulo 3, algumas metodologias não são eficientes quando utilizadas em circuitos dissimilares, que a cada dia tornam-se mais populares devido às modernas ferramentas de síntese disponíveis no mercado.

A próxima seção detalha os objetivos a serem atingidos por este trabalho. A seção seguinte descreve a motivação para realizar o mesmo e a última seção apresenta como esta dissertação está organizada.

1.1 Objetivos

O presente trabalho tem por objetivo apresentar e analisar técnicas para o reaproveitamento das cláusulas de conflito entre partições durante a verificação de equivalência entre dois circuitos combinacionais dissimilares particionados. Para tal, será utilizado um resolvedor SAT do estado-da-arte capaz de retornar tais cláusulas. A vantagem em se utilizar resolvedores SAT para tratar instâncias de CEC reside no fato da crescente eficiência nos mecanismos dos resolvedores e por tratar tanto circuitos similares quanto dissimilares.

Os resultados serão gerados com base em *benchmarks* de diversos tipos e tamanhos de multiplicadores. Esta classe de circuitos aritméticos foi escolhida pois, mesmo para circuitos com poucos bits, é considerada uma das mais difíceis a serem verificadas.

Pretendemos responder questões como: é possível obter uma melhora no tempo de execução ao reaproveitar as cláusulas de conflito? É possível reduzir a quantidade utilizada de memória? É possível analisar circuitos maiores? O reaproveitamento das cláusulas de conflito

[1] Um problema pertence à classe co-$\mathcal{NP}$-difícil, se e somente se, seu complemento pertence à classe $\mathcal{NP}$-difícil. O complemento de CEC é o problema da satisfabilidade, bem conhecido por seu caráter NP-difícil [Garey e Johnson, 1979].

é eficiente para circuitos dissimilares?

Este trabalho também ajuda a verificar se alguns dos circuitos gerados pelo BenCGen [Andrade et al., 2008] estão corretos. Por ser uma ferramenta recente de geração de *benchmarks*, até então tinha sido utilizado somente pelos próprios autores. Utilizando técnicas para CEC, poderemos verificar a equivalência entre os circuitos gerados.

1.2 Motivação

A cada dia, os circuitos VLSI (do inglês *Very Large Scale Integrated*) estão mais presentes em nossa vida. Dos celulares que falamos aos carros que dirigimos, em quase todos os momentos é possível encontrarmos um dispositivo eletrônico em ação. Isto gera uma crescente demanda por circuitos mais ágeis e compactos, fazendo com que estes se tornem complexos e caros. Além disso há uma constante pressão do mercado por novos lançamentos em cada vez mais curtos espaços de tempo.

Todos este fatores contribuem diretamente para que as empresas empreguem tempo e dinheiro consideráveis na verificação dos circuitos produzidos. Atualmente, estima-se que para cada projetista haja três engenheiros de verificação [Molitor e Mohnke, 2004] e que esta atividade seja responsável por 50% a 80% do tempo total gasto no projeto [Foster et al., 2004]. Um erro encontrado após o circuito ter sido produzido pode sair muito caro para uma empresa, pois é praticamente impossível consertar erros neste estágio. Por este motivo é necessário que os erros sejam detectados o mais cedo possível, durante a fase inicial de projeto.

A Intel, maior fabricante de microprocessadores do mundo, sofreu as graves conseqüências de um *bug* em um de seus produtos. Em 1994, após o lançamento do produto no mercado, foi encontrado um *bug* na unidade de divisão de ponto flutuante do Pentium[2] que gerou processos à Intel e um prejuízo de quase US$ 500 milhões com a substituição das unidades defeituosas [Beizer, 1995].

Particularmente, a verificação de equivalência tem sido bastante utilizada em dois principais cenários:

1. Verificação entre dois estágios diferentes do projeto: O projeto de circuitos integrados encontra-se dividido em diversos estágios de produção. Inicialmente, o circuito é descrito em RTL (do inglês *Register Transfer Level*), geralmente em uma linguagem de descrição de *hardware* como Verilog ou VHDL. Em seguida, o circuito é submetido a uma ferramenta de síntese, gerando um conjunto de portas lógicas. O circuito sintetizado pode passar por uma etapa de otimização e somente então será fisicamente gerado. Para verificar se, em cada estágio, o circuito corresponde à especificação, é feita uma verificação de equivalência entre dois estágios. Isto garante que durante a transição de um estágio para outro não houve a inserção de erros.

[2]Este *bug* ficou mundialmente conhecido como Pentium FDIV bug. FDIV é a abreviação, em inglês, de *Floating-point Division*.

2. Verificação entre circuito original e circuito modificado: As vezes é conveniente para o projetista realizar alterações no circuito, sem alterar as especificações de entrada e saída do mesmo. Para garantir que não houve a inserção acidental de erros, é feita uma verificação de equivalência entre o circuito original e o novo.

Portanto, este trabalho tem o propósito de contribuir com uma das áreas mais relevantes e atuais do projeto e desenvolvimento de circuitos integrados.

1.3 Organização

O presente trabalho encontra-se organizado em 6 capítulos, distribuídos da seguinte forma: o primeiro capítulo introduz o tema desta dissertação. O capítulo 2 apresenta os conceitos básicos relacionados ao problema de CEC. O capítulo 3 apresenta uma ampla revisão bibliográfica. O capítulo 4 fornece as soluções apresentadas e analisadas neste trabalho para verificar a equivalência entre circuitos combinacionais dissimilares. O capítulo 5 dedica-se aos resultados experimentais obtidos através da implementação das técnicas do capítulo anterior e por fim, o capítulo 6 resume as contribuições, relaciona os trabalhos futuros e conclui esta dissertação.

Capítulo 2

Conceitos Básicos

> "I do not fear computers. I fear the lack of them."
>
> Isaac Asimov

Este capítulo apresenta os conceitos básicos relacionados à área de verificação de equivalência entre circuitos combinacionais. O leitor mais experiente deve se sentir confortável para ir direto para a próxima seção.

Formalmente, a verificação de circuitos pode ser definida da seguinte forma, segundo [Molitor e Mohnke, 2004]:

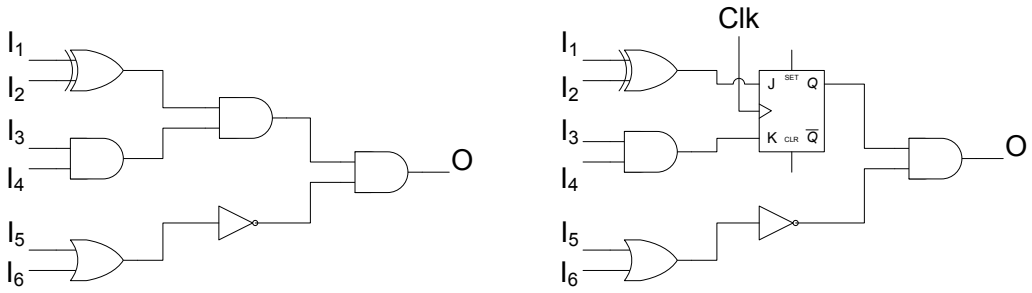
Definição 2.1 *Dadas duas representações d_f e d_g de duas funções booleanas f,g : $\{0,1\}^n \rightarrow \{0,1\}^m$, decidir se as funções booleanas f e g são equivalentes, ou seja, $f(\alpha) = g(\alpha)$ se mantém para todo $\alpha \in \{0,1\}^n$.*

Este capítulo encontra-se dividido em 4 seções. A primeira seção apresenta as características dos circuitos que podem ser usados para sua classificação. A segunda define o conceito de miter. A seção seguinte introduz o problema da satisfabilidade e a última seção explica o funcionamento de um resolvedor SAT.

2.1 Classificação de Circuitos

Os circuitos podem ser divididos em dois grandes grupos:

- **Combinacionais:** Os circuitos combinacionais são aqueles cujas saídas dependem unicamente da entrada atual, não possuindo qualquer tipo de elemento de memória. A figura 2.1a mostra um exemplo de circuito combinacional.
- **Seqüenciais:** Os circuitos seqüenciais caracterizam-se pela presença de memória. Suas saídas dependem da entrada atual e do histórico das entradas anteriores, armazenado como um estado interno. A figura 2.1b mostra um exemplo de circuito seqüencial. O flip-flop JK é o elemento de memória do circuito.



(a) Circuito combinacional (b) Circuito seqüencial

Figura 2.1: Exemplo de circuitos combinacional e seqüencial

Os circuitos a serem verificados devem pertencer ao mesmo grupo. Como o foco deste trabalho é o problema de CEC, somente circuitos combinacionais são de interesse.

Podemos ainda classificar a dupla de circuitos que compõem a instância da CEC em duas categorias, de acordo com o grau de similaridade entre eles:

- **Similares:** Os circuitos similares são aqueles que possuem um considerável grau de similaridade estrutural entre si. Esta característica é explorada pela maioria das técnicas de CEC, com destaque para os resultados excepcionais obtidos com o uso de BDDs. Estes tipos de circuitos eram mais comuns na época em que as ferramentas de síntese eram mais simples, pois o circuito tinha sua estrutura pouco alterada entre os níveis de RTL e portas lógicas.

- **Dissimilares:** Os circuitos dissimilares possuem um baixo ou nenhum grau de similaridade estrutural entre si. São ainda pouco explorados na literatura, pois sua verificação é complexa, sendo intratável mesmo para circuitos de poucos bits. Segundo [Goldberg e Novikov, 2003], as ferramentas de síntese modernas tendem a gerar circuitos dissimilares devido aos avanços ocorridos nos últimos anos na área de verificação formal.

Cada circuito pode ainda ser classificado de acordo com sua estrutura interna [Stanion, 1999], a saber:

- **Dependência Estrutural:** Os circuitos estruturalmente dependentes apresentam os bits das saídas primárias altamente dependentes do circuito utilizado para gerar os bits das saídas primárias anteriores. Circuitos aritméticos freqüentemente apresentam esta característica.

- **Interseção Estrutural:** Os circuitos com interseção estrutural são aqueles cujos bits das saídas primárias dependem parcialmente do circuito utilizado para gerar os bits das saídas primárias anteriores. Em outras palavras, há dependência estrutural somente de parte do circuito que gera os bits das saídas primárias anteriores.

Há também os circuitos sem interseção estrutural, cujos bits das saídas primárias são totalmente independentes do circuito utilizado para gerar os bits das saídas primárias anteriores. Decodificadores e multiplexadores apresentam esta característica.

2.2 Miter

Algumas técnicas de CEC utilizam um miter para provar a equivalência entre dois circuitos. Conforme descrito em [Brand, 1993] e [Goldberg e Novikov, 2003], dados dois circuitos C_1 e C_2, um miter é um circuito gerado em três etapas:

1. Identificação das entradas primárias correspondentes dos dois circuitos.
2. Identificação das saídas primárias correspondentes dos dois circuitos, inserindo uma porta XOR na saída de cada par correspondente.
3. Inserção de uma porta OR que terá como entrada as saídas das portas XOR adicionadas.

A saída da porta OR é denominada saída do miter. O valor desta saída será 1 se no mínimo uma de suas entradas possuir o valor 1. Isto significa que em pelo menos uma das portas XOR adicionadas, as entradas possuíam valores distintos, o que acontece se e somente se, para o mesmo assinalamento das entradas correspondentes dos dois circuitos, uma saída primária de um circuito e a correspondente saída do outro circuito tiverem valores diferentes. Portanto, o problema de CEC entre dois circuitos equivale a testar a satisfabilidade do miter dos dois circuitos. Caso a saída possua o valor lógico 1, os circuitos não são equivalentes. Caso possua o valor lógico 0, os circuitos são equivalentes.

A figura 2.2 mostra um miter construído a partir de dois circuitos combinacionais. Ambos possuem o mesmo número de entradas (seis) e saídas (duas). As entradas correspondentes foram identificadas com o mesmo número, cada dupla de saídas correspondentes foram conectadas a uma porta XOR e as saídas das duas portas adicionadas foram conectadas a uma porta OR, sendo esta a saída do miter.

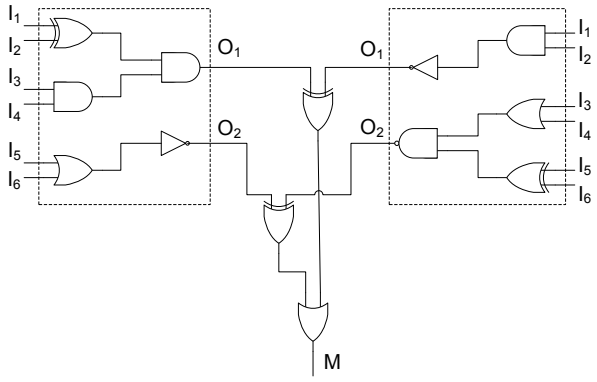


Figura 2.2: Miter construído a partir de dois circuitos combinacionais com duas saídas cada.

2.3 Satisfabilidade

O problema da satisfabilidade (SAT) de circuitos foi o primeiro a ser provado $\mathcal{NP}$-Completo [Cook, 1971], sendo base para o Teorema de Cook, assim enunciado:

Teorema 2.2 *Satisfabilidade está em $\mathcal{P}$ se e somente se $\mathcal{P} = \mathcal{NP}$.*

Este teorema apresenta um dos maiores problemas em aberto da Ciência da Computação, estudado por milhares de pesquisadores do mundo todo: $\mathcal{P} = \mathcal{NP}$?

Formalmente, o problema de SAT pode ser definido da seguinte forma:

Definição 2.3 *Dado uma expressão booleana na forma normal conjuntiva, o problema de SAT consiste em verificar se existe uma atribuição de valores para as suas variáveis que torne a expressão verdadeira (satisfazível). Caso tal atribuição não exista, o problema não é satisfazível.*

O conceito de forma normal conjuntiva pode ser assim enunciado:

Definição 2.4 *Uma expressão booleana encontra-se na forma normal conjuntiva (CNF, do inglês Conjunctive Normal Form), se esta for uma conjunção (AND) de cláusulas, onde cada cláusula representa uma disjunção (OR) de literais e cada literal é uma variável, complementada (NOT) ou não.*

Diz-se também que uma fórmula em CNF é um produto de somas. É importante ressaltar que uma cláusula não pode conter variáveis repetidas e as cláusulas que contém somente um literal recebem o nome de cláusulas unitárias. A expressão abaixo apresenta um exemplo de uma fórmula representada em CNF:

$$(A \vee B) \wedge (\overline{A} \vee B \vee C) \wedge (\overline{A} \vee E) \wedge (\overline{B} \vee \overline{D})$$

Observe que qualquer circuito representado em CNF pode ser construído com portas NOT, AND e OR.

2.4 Resolvedor SAT

Definidos o problema de SAT e o conceito de CNF, um resolvedor SAT é um *software* capaz de resolver instâncias de SAT. Como SAT é um problema $\mathcal{NP}$-difícil [Cormen et al., 2001], instâncias muito grandes ainda são intratáveis. Apesar disso, há de se destacar os grandes avanços obtidos na área nos últimos anos, tendo [Silva e Sakallah, 1996] como ponto de partida.

Para explicarmos o funcionamento dos resolvedores SAT modernos, utilizaremos como referência [Molitor e Mohnke, 2004], [Eén e Sörensson, 2003] e [Silva e Sakallah, 1996].

O algoritmo DP (Davis-Putnam) publicado em [Davis e Putnam, 1960] foi o precursor dos algoritmos amplamente utilizados nos resolvedores SAT atuais. Foi criado para verificar a validação de uma fórmula lógica de primeira ordem. Dois anos mais tarde, foi refinado e renomeado para DPLL[1] (Davis-Putnam-Logemann-Loveland) [Davis et al., 1962]. Como exemplo de resolvedores SAT modernos, podemos citar o Berkmin [Goldberg e Novikov, 2002],

[1] Este algoritmo também é conhecido como DLL (Davis-Logemann-Loveland).

o MiniSat [Eén e Sörensson, 2003] e o zChaff [Moskewicz et al., 2001]. Todos estes resolvedores baseiam-se no algoritmo DPLL, com retrocesso não-cronológico por análise de conflito e aprendizado de cláusula [Silva e Sakallah, 1996], e propagação de restrição booleana (BCP, do inglês *Boolean Constraint Propagation*) utilizando literais vigiados[2][Moskewicz et al., 2001].

Em termos gerais, resolvedores SAT dirigidos por conflito, como os citados anteriormente, possuem uma estrutura de dados capaz de representar cláusulas e assinalamentos, utilizam um mecanismo de inferência conhecido como propagação unitária e realizam uma busca a fim de de derivar informação.

Propagação unitária é um mecanismo em que, tão logo uma cláusula torne-se unitária devido ao assinalamento corrente, o literal restante é assinalado como verdadeiro e como conseqüência, a variável é assinalada como falsa ou verdadeira, dependendo dela estar complementada ou não, respectivamente. Este processo é executado continuamente até que nenhuma informação possa mais ser propagada. O procedimento seguinte é a parte mais complexa de um resolvedor SAT. Através de uma heurística, variáveis são selecionadas e tem valores assinalados[3] até que haja um conflito, que ocorre quando todos os literais de uma cláusula foram assinalados como falsos. Neste momento, uma cláusula de conflito é construída e adicionada ao banco de cláusulas. Os assinalamentos efetuados são então anulados por uma técnica de retrocesso até que a cláusula de conflito torne-se unitária e possa ser propagada para a continuação do procedimento.

O algoritmo 2.1 apresenta o algoritmo utilizado pelos resolvedores SAT atuais, baseado em DPLL. Todos os resolvedores anteriormente citados utilizam tal algoritmo, porém a diferença está na implementação das funções **Propaga**, **Decide**, **Analisa** e **Retrocede**. É justamente a heurística implementada nestas funções que faz com que um resolvedor SAT seja melhor que o outro em determinadas instâncias de problemas.

O algoritmo inicia selecionando uma variável que ainda não tenha sido assinalada e assume um valor para ela. Esta variável é denominada variável de decisão e o valor assumido pode ser verdadeiro ou falso. As conseqüências deste assinalamento serão propagadas, pela função **Propaga**, possivelmente resultando em novos assinalamentos de variáveis. Todas as variáveis assinaladas como conseqüência da variável de decisão pertencem ao mesmo nível de decisão. O primeiro nível de decisão é chamado de nível-raiz e ocorre antes da primeira *assumption*. A partir de então, para cada *assumption* realizada tem-se um novo nível de decisão. Todos os assinalamentos são armazenados em uma pilha, denominada trilha (do inglês *trail*), na ordem em que foram realizadas e agrupadas por nível de decisão. Esta organização será útil durante o processo de retrocesso.

Segundo [Moskewicz et al., 2001], aproximadamente 90% do tempo total gasto pelo resolvedor SAT é gasto na etapa de propagação. Os resolvedores baseados em DPLL apresentam duas regras de BCP: regra da cláusula de um literal[4] (do inglês *one-literal clause rule*)

[2]Na literatura, esta técnica é conhecida como *watched literals*.

[3]Diz-se que o resolvedor SAT realizou uma *assumption*. Embora a palavra *assumption* possa ser traduzida como pressuposição, essa tradução não é utilizada na área.

[4]Também conhecida como regra da cláusula unitária.

Algoritmo 2.1 Algoritmo utilizado em resolvedores SAT modernos baseado em DPLL

Entrada: Uma fórmula booleana em CNF.
Saída: SAT caso a fórmula seja satisfazível, UNSAT caso contrário.

```
1: while (VERDADEIRO) do
2:   Propaga() {Propaga as cláusulas unitárias}
3:   if (não há conflito) then
4:     if (todas as variáveis foram assinaladas) then
5:       return SAT
6:     else
7:       Decide() {Seleciona uma nova variável e faz o assinalamento}
8:     end if
9:   else
10:    Analisa() {Analisa o conflito e adiciona uma cláusula de conflito}
11:    if (conflito encontrado no nível-raiz) then
12:      return UNSAT
13:    else
14:      Retrocede() {Desfaz os assinalamentos}
15:    end if
16:  end if
17: end while
```

e regra afirmativo-negativo[5] (do inglês *affirmative-negative rule*). A primeira regra pode ser assim definida:

Definição 2.5 *Seja ϕ uma fórmula booleana em CNF que contém alguma cláusula que consiste de somente um literal α. Então, ϕ pode ser modificada removendo-se todas as cláusulas que contenham α e apagando todas as ocorrências de $\overline{\alpha}$ das cláusulas restantes.*

Em outras palavras, se ϕ contém uma cláusula unitária com um literal α, então α deve ser verdadeiro. Isto implica que, caso α seja uma variável θ, esta variável deve ser assinalada para verdadeiro. Caso seja $\overline{\theta}$, ela recebe o valor falso. Este valor será propagado para as demais cláusulas de ϕ. Aquelas que contém α podem ser eliminadas pois ela será verdadeira independente dos demais literais. Aquela que contém $\overline{\alpha}$ podem ter este literal removido, pois seu valor independe desta variável. Esta regra traz consigo o seguinte lema:

Lema 2.6 *Seja ϕ_1 uma fórmula booleana em CNF e ϕ_2 a fórmula resultante após a regra da cláusula de um literal ser aplicada a ϕ_1. Então, ϕ_2 é satisfazível se e somente se ϕ_1 é satisfazível.*

Maiores detalhes sobre a regra da cláusula de um literal podem ser encontrados em [Davis e Putnam, 1960].

A regra afirmativo-negativo pode ser assim definida:

[5] Também conhecida como regra do literal puro ou regra da variável fixa monótona.

Definição 2.7 *Seja ϕ uma fórmula booleana em CNF que contém uma variável θ que aparece somente como um literal positivo ou negativo. Então, todas as cláusulas que contém θ podem ser removidas.*

Em outras palavras, se todos os literais α de ϕ que contém a variável θ são exclusivamente θ ou $\overline{\theta}$, as cláusulas que contém α podem ser removidas pois α pode ser assinalado como verdadeiro. Esta regra resulta no seguinte lema:

Lema 2.8 *Seja ϕ_1 uma fórmula booleana em CNF e ϕ_2 a fórmula resultante após a regra afirmativo-negativo ser aplicada a ϕ_1. Então, ϕ_2 é satisfazível se e somente se ϕ_1 é satisfazível.*

Para maiores detalhes sobre a regra afirmativo-negativo, consulte [Davis e Putnam, 1960].

[Silva e Sakallah, 1996] propõe que cada cláusula tenha um contador com o número de variáveis assinaladas. Uma cláusula unitária é detectada quando o contador atinge uma unidade a menos do que o número total de literais da cláusula. Porém, estes contadores precisam ser ajustados sempre que há um retrocesso e verificados a cada nível de decisão. Para evitar tais limitações, [Moskewicz et al., 2001] propõe que, para cada cláusula, dois literais não assinalados sejam escolhidos para serem vigiados. Cada literal possui uma lista contendo as cláusulas que os estão vigiando. Quando um literal vigiado torna-se verdadeiro, as cláusulas são verificadas para saber se há informação para ser propagada ou para selecionar um novo literal não assinalado para ser vigiado. A vantagem desta técnica é que não há necessidade de ajustar as listas quando há um retrocesso. Além disso, não é necessário verificar todas as cláusulas quando há um assinalamento, somente as que possuem literais vigiados assinalados naquele nível de decisão.

A etapa de propagação finaliza quando há um conflito ou quando não for mais possível propagar informação. Caso haja um conflito, a função **Analisa** é invocada para produzir uma cláusula de conflito. Se o conflito ocorreu no nível-raiz, o algoritmo finaliza retornando UNSAT. Caso contrário, a função **Retrocede** é acionada para desfazer os assinalamentos até que a cláusula de conflito seja unitária. A trilha será utilizada para para desfazer as decisões de cada nível, até que um dos literais da cláusula aprendida não seja mais assinalado (no momento em que o conflito ocorre, todos os literais são falsos). Se os assinalamentos do conflito permanecem inalterados por alguns níveis de decisão, é vantajoso escolher o nível mais baixo. Esta técnica, introduzida em [Silva e Sakallah, 1996], é conhecida como retrocesso não-cronológico (do inglês *non-chronological backtracking*) e apresentou-se como uma alternativa bem mais eficiente do que retrocesso cronológico até então utilizando, onde o retrocesso era feito entre níveis adjacentes. Devido a sua relevância para o presente trabalho, detalhes sobre a análise do conflito e produção das cláusulas de conflito será explicado separadamente na subseção 2.4.1.

Se a etapa de propagação for finalizada devida à impossibilidade de se propagar informações, caso todas as variáveis tiverem sido assinaladas, o algoritmo finaliza retornando SAT. Caso ainda haja variáveis a serem assinaladas, a função **Decide** é acionada para selecionar uma variável a ser assinalada. Esta função tem grande importância no desempenho do resolve-

dor e sua eficiência pode variar de acordo com o domínio do problema [Silva, 1999]. Na prática, três heurísticas se destacam: máxima ocorrência em cláusulas de tamanhos mínimos (MOM, do inglês *Maximum Occurrences on clauses of Minimum sizes*), regras de Jeroslow-Wang [Jeroslow e Wang, 1990] e Soma Decrescente Independente do Estado da Variável (VSIDS, do inglês *Variable State Independent Decaying Sum*) [Moskewicz et al., 2001].

As duas primeiras heurísticas dão preferência a variáveis ou literais que ocorrem em cláusulas menores, para que estas cláusulas possam rapidamente ser transformadas em cláusulas unitárias para serem propagadas. Na heurística MOM, apenas o número de ocorrência das variáveis nas cláusulas menores é levado em consideração. Nas regras de Jeroslow-Wang, uma função JW de peso é associada a cada literal. Considerando ϕ uma fórmula booleana em CNF que contém cláusulas ψ constituídas de literais α, a função JW é dada por:

$$\sum_{\psi \in \phi, \alpha \in \psi} 2^{-|\psi|}.$$

A heurística VSIDS, utilizada no zChaff, tem obtido bons resultados para problemas de circuitos. Cada literal possui um contador, iniciado em 0. Sempre que uma cláusula é adicionada ao banco de cláusulas, os contadores associados com os literais da cláusula são incrementados. Periodicamente, todos os contadores são divididos por uma constante. A variável cujo literal possui o contador de maior número é selecionada. O MiniSat propõe e utiliza uma versão do VSIDS com uma pequena modificação: um contador é associado a cada variável em detrimento de cada literal. Esta foi uma das modificações que permitiu que o MiniSat atingisse melhores resultados que o zChaff para as mesmas instâncias de problemas.

Para exemplificar a execução de um resolvedor SAT, considere a seguinte fórmula em CNF:

$$\begin{aligned}\phi \;=\; & (\overline{\theta_8} \vee \theta_{11}) \wedge (\overline{\theta_9} \vee \theta_{12}) \wedge (\overline{\theta_1} \vee \theta_2 \vee \theta_8) \wedge (\theta_1 \vee \theta_9) \wedge (\overline{\theta_1} \vee \theta_3 \vee \theta_8 \vee \theta_9) \wedge \\ & (\overline{\theta_2} \vee \overline{\theta_3} \vee \theta_4) \wedge (\overline{\theta_4} \vee \theta_5 \vee \overline{\theta_6}) \wedge (\overline{\theta_4} \vee \theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6}) \wedge (\theta_1 \vee \theta_7 \vee \overline{\theta_{10}})\end{aligned}$$

Suponhamos que o resolvedor tenha selecionado o literal $\overline{\theta_8}$ para assinalar. Aplicando BCP em $\phi \wedge \overline{\theta_8}$, temos:

$$\begin{aligned}\phi \;=\; & (\overline{\theta_9} \vee \theta_{12}) \wedge (\overline{\theta_1} \vee \theta_2) \wedge (\theta_1 \vee \theta_9) \wedge (\overline{\theta_1} \vee \theta_3 \vee \theta_9) \wedge (\overline{\theta_2} \vee \overline{\theta_3} \vee \theta_4) \wedge \\ & (\overline{\theta_4} \vee \theta_5 \vee \overline{\theta_6}) \wedge (\overline{\theta_4} \vee \theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6}) \wedge (\theta_1 \vee \theta_7 \vee \overline{\theta_{10}})\end{aligned}$$

Como nenhuma cláusula unitária foi formada, o algoritmo seleciona um novo literal: $\overline{\theta_9}$. Aplicando BCP em $\phi \wedge \overline{\theta_9}$, obtemos:

$$\begin{aligned}\phi \;=\; & (\overline{\theta_1} \vee \theta_2) \wedge (\theta_1) \wedge (\overline{\theta_1} \vee \theta_3) \wedge (\overline{\theta_2} \vee \overline{\theta_3} \vee \theta_4) \wedge (\overline{\theta_4} \vee \theta_5 \vee \overline{\theta_6}) \wedge \\ & (\overline{\theta_4} \vee \theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6}) \wedge (\theta_1 \vee \theta_7 \vee \overline{\theta_{10}})\end{aligned}$$

A propagação gerou uma cláusula unitária contendo o literal θ_1. Portanto, este literal é assinalado para verdadeiro e seu valor propagando, resultando em:

$$\phi = (\theta_2) \wedge (\theta_3) \wedge (\overline{\theta_2} \vee \overline{\theta_3} \vee \theta_4) \wedge (\overline{\theta_4} \vee \theta_5 \vee \overline{\theta_6}) \wedge (\overline{\theta_4} \vee \theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6})$$

Desta vez, a propagação gerou duas cláusulas unitárias: θ_2, θ_3. Ao serem propagadas, a fórmula gerada resume-se a:

$$\phi = (\theta_4) \wedge (\overline{\theta_4} \vee \theta_5 \vee \overline{\theta_6}) \wedge (\overline{\theta_4} \vee \theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6})$$

Mais uma cláusula unitária foi gerada, desta vez com o literal θ_4. Com isso, temos a seguinte fórmula:

$$\phi = (\theta_5 \vee \overline{\theta_6}) \wedge (\theta_6) \wedge (\overline{\theta_5} \vee \overline{\theta_6})$$

Devido a θ_6, uma nova propagação é necessária:

$$\phi = (\theta_5) \wedge (\overline{\theta_5})$$

Enfim, um conflito foi identificado. A subseção seguinte mostrará como este conflito deve ser analisado e como gerar uma cláusula de conflito.

2.4.1 Análise de Conflito

Sempre que há um conflito, este deve ser analisado e alguma atitude deve ser tomada para garantir que ele não se repita. Uma simples heurística consiste em atribuir um valor diferente à última variável assinalada antes da ocorrência do conflito. Para isto, basta retroceder ao nível anterior ao assinalamento e assinalar para a mesma variável o valor complementado. Outra proposta é retroceder até uma variável de decisão e complementar o valor assinalado para ela, conforme mostrado a seguir. O processo inicia com a seleção do conjunto de assinalamentos de variáveis que causou o conflito. Estes assinalamentos devem ter sido originados por uma *assumption* ou pelo resultado de algum propagação. Os assinalamentos que ocorreram devido a propagação são analisados para se obter os assinalamentos responsáveis pela sua propagação, e assim sucessivamente até que uma condição de término seja atingida, resultando em um conjunto de assinalamento de variáveis que implicaram o conflito. Uma cláusula proibindo tal assinalamento é adicionada ao conjunto de cláusulas. Esta cláusula é denominada cláusula de conflito.

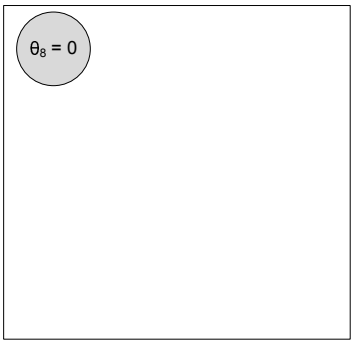
As cláusulas de conflito servem para direcionar a etapa de retrocesso e evitar que o mesmo conflito ocorra novamente. Entretanto, a propagação torna-se mais lenta devido ao crescente aumento do conjunto de cláusulas. Para resolver este problema, alguns resolvedores implementam heurísticas para reduzir o número de cláusulas, mantendo somente as consideradas úteis. No MiniSat por exemplo, uma heurística semelhante ao VSIDS é aplicada a cláusulas. Cada cláusula possui um contador relacionado que é incrementado toda vez que a cláusula é utilizada durante a análise de um conflito. As cláusulas menos utilizada são periodicamente

removidas.

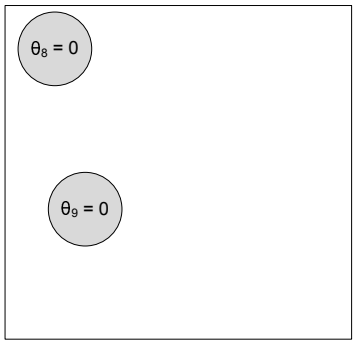
A análise de conflito pode ser bastante eficiente se representada através de uma estrutura denominada grafo de implicação. Este grafo é direcionado e definido da seguinte forma:

- Cada vértice do grafo corresponde a um assinalamento.
- Os antecessores de um vértice do grafo são os assinalamentos anteriores que resultaram na implicação deste vértice.
- Vértices que não tem antecessores correspondem a variáveis de decisão.
- Vértices de conflito são adicionados ao grafo para indicar a ocorrência dos mesmos. Os vértices adjacentes a um vértice de conflito correspondem aos assinalamentos que causaram o conflito.

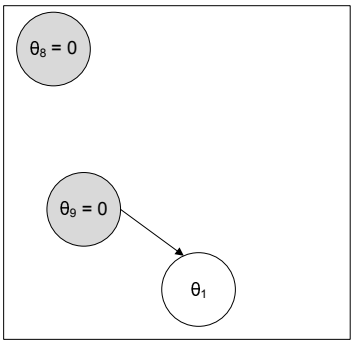
As figuras 2.3 e 2.4 mostram passo-a-passo da construção do grafo de implicação para a fórmula ϕ do exemplo anterior. Os vértices cinza correspondem às variáveis de decisão, os brancos correspondem aos assinalamentos propagados e o preto corresponde a um conflito.



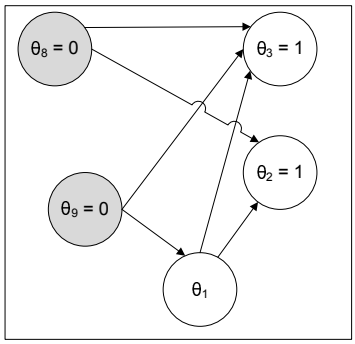
(a) Assinalamento da variável de decisão θ_8.



(b) Assinalamento da variável de decisão θ_9.

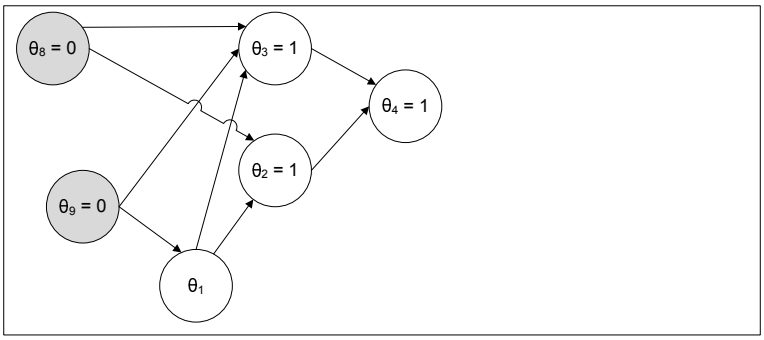


(c) Implicação do assinalamento de θ_9.

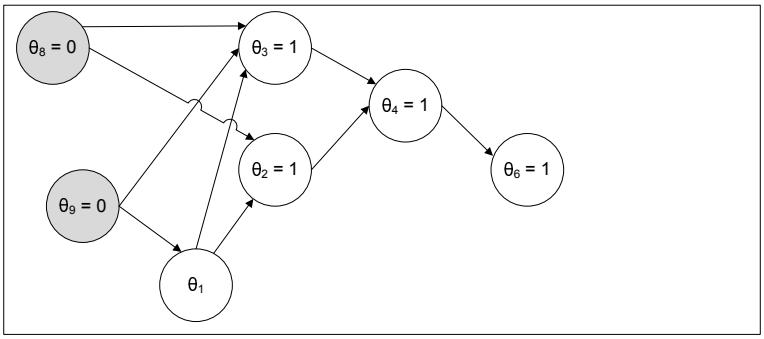


(d) Implicação do assinalamento de θ_1, θ_8 e θ_9.

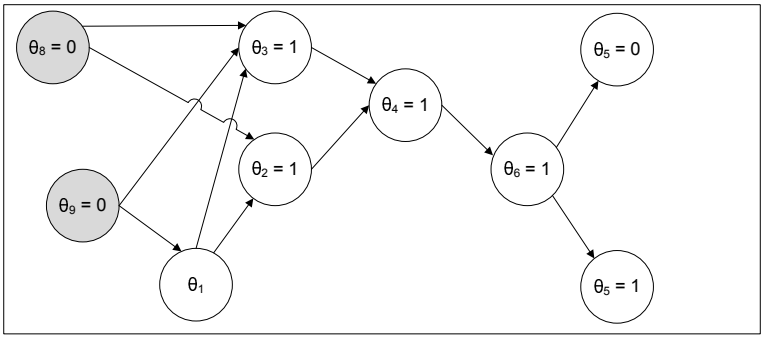
Figura 2.3: Construção de um grafo de implicação (parte 1).



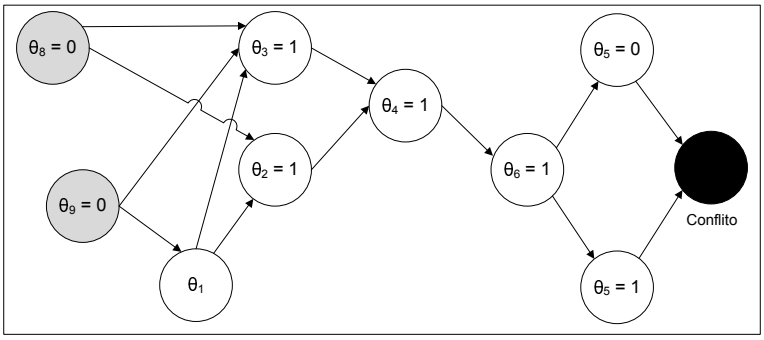
(a) Implicação do assinalamento de θ_2 e θ_3.



(b) Implicação do assinalamento de θ_4.



(c) Implicação do assinalamento de θ_6.



(d) Conflito detectado na variável θ_5.

Figura 2.4: Construção de um grafo de implicação (parte 2).

Cada figura equivale a um assinalamento e propagação mostrados no exemplo anterior. Por exemplo, a figura 2.3c mostra que a propagação do assinalamento de θ_9 gerou uma cláusula unitária que resultou na atribuição do valor[6] 1 para a variável θ_1.

A figura 2.4d mostra o grafo de implicação gerado no exato momento em que o conflito na variável θ_5 foi detectado. Para encontrar os assinalamentos responsáveis pelo conflito, basta inverter as arestas do grafo e, a partir do vértice de conflito, caminhar no grafo até que os vértices de decisão sejam encontrados. No exemplo dado, os assinalamentos $\theta_8 = 0$ e $\theta_9 = 0$ foram os responsáveis pelo conflito. Sendo assim, a seguinte cláusula foi responsável pelo conflito:

$$\lambda = \overline{\theta_8} \wedge \overline{\theta_9}$$

Uma cláusula de conflito β é gerada a partir da negação de λ. Portanto:

$$\begin{aligned} \beta &= \overline{\lambda} \\ \beta &= \overline{\overline{\theta_8} \wedge \overline{\theta_9}} \\ \beta &= \theta_8 \vee \theta_9 \end{aligned}$$

Ou seja, o assinalamento mostrado no exemplo gerou um conflito que implicou a cláusula de conflito $(\theta_8 \vee \theta_9)$.

Alguns resolvedores implementam um algoritmo que explora o fato de que se efetuados cortes no grafo de implicação de forma que os vértices de decisão sejam mantidos de um lado (lado da razão) e o vértice de conflito do outro lado (lado do conflito), é fácil identificar uma cláusula de conflito [Zhang et al., 2001]. Todos os vértices do lado da razão que tem pelo menos uma aresta direcionada para o lado do conflito, integram a razão do conflito.

A figura 2.5 apresenta o grafo de implicação da figura 2.4d com três cortes. Cada um destes cortes implica uma cláusula de conflito: $(\theta_8 \vee \theta_9)$, $(\overline{\theta_2} \vee \overline{\theta_3})$ e $(\overline{\theta_4})$.

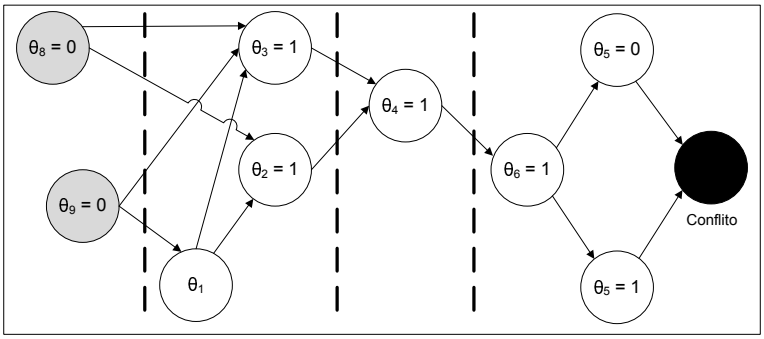


Figura 2.5: Cortes no grafo de implicação.

Esta técnica traz o importante conceito de cláusulas fortes. A cláusula $(\overline{\theta_4})$ é dita mais forte que as demais por ela possuir um número menor de literais. É mais fácil um assinalamento

[6]No contexto deste trabalho, o valor lógico 1 significa verdadeiro e 0 significa falso.

ser feito para ela do que para as cláusulas com mais variáveis. Sendo assim, [Silva e Sakallah, 1996] propõe uma técnica denominada ponto de implicação único (do inglês *unique implication point*), assim definida:

Definição 2.9 *Um vértice v_1 do grafo de implicação domina um vértice v_2, se e somente se, qualquer caminho que liga o último vértice de decisão até o vértice v_2 passe pelo vértice v_1. Um vértice que domina o vértice de conflito é denominado ponto de implicação único.*

Portanto, um ponto de implicação único representa um assinalamento que pode propagar seqüências de assinalamentos que produzirão o conflito.

É importante ressaltar que instâncias diferentes de problemas podem ter um desempenho diferente dependendo do algoritmo utilizado na análise de conflito. Maiores informações podem ser encontradas em [Silva e Sakallah, 1996], [Zhang et al., 2001] e [Silva e Silva, 1999].

Capítulo 3

Revisão Bibliográfica

> "Computer Science is no more about computers than astronomy is about telescopes."
>
> Edsger W. Dijkstra

O objetivo deste capítulo é apresentar uma revisão bibliográfica sobre as principais metodologias utilizadas para resolver instâncias do problema de verificação de circuitos combinacionais: Diagrama de Decisão Binário (BDD), Diagrama de Momento Binário (BMD), Geração Automática de Padrão de Testes (ATPG), Aprendizado Recursivo (RL), resolvedores SAT e técnicas que integram duas ou mais metodologias, como por exemplo, o uso de BDD em conjunto com um resolvedor SAT.

Este capítulo encontra-se dividido em 7 seções, cada uma dedicada a uma técnica diferente: BDD, BMD, ATPG, RL, resolvedores SAT e técnicas integradas. Por fim, serão listados os trabalhos relacionados.

3.1 Diagrama de Decisão Binário

Nas duas últimas décadas, o uso de Diagrama de Decisão Binário (BDD, do inglês *Binary Decision Diagram*) foi decisivo para o avanço das técnicas de CEC. Um BDD é um grafo acíclico direcionado utilizado para representar funções booleanas. Foi introduzido por [Lee, 1959] mas tornou-se popular somente após os avanços introduzidos por [Akers, 1978], [Moret, 1982] e [Bryant, 1986].

Na prática, os problemas de CEC utilizam BDDs Reduzidos e Ordenados (ROBDDs, do inglês *Reduced Ordered BDDs*), sendo que diversos autores adotam BDDs como sinônimo para ROBDDs [Clarke et al., 1999] e assim manteremos neste trabalho.

Uma das principais características dos BDDs é sua dependência quanto ao ordenamento de suas variáveis. Dependendo desta ordenação, o número de vértices pode variar de linear a exponencial, e como veremos mais adiante, o tamanho do BDD gerado tem sido uma limitação para resolver, por exemplo, problemas de CEC para circuitos dissimilares. Daí a necessidade

em se manter os BDDs reduzidos e ordenados.

Para se obter um BDD de uma função booleana, o primeiro passo consiste na construção da árvore de decisão da função, sendo que, em cada caminho da raiz até a folha, cada variável aparece somente uma vez. As variáveis também devem sempre obedecer a mesma ordem nos caminhos. Os vértices terminais representam o valor para um caminho e cada caminho representa um assinalamento para as variáveis da função.

A figura 3.1 apresenta um BDD ordenado para a função $[(x_1 \wedge x_3 \wedge x_4) \vee x_2]$. Os círculos representam os vértices internos, correspondentes às variáveis x_1, x_2, x_3 e x_4. Os quadrados correspondem aos vértices terminais e representam os valores booleanos 0 e 1. As arestas encontram-se direcionadas de cima para baixo e as arestas finas representam assinalamento 0 e as mais grossas representam o assinalamento 1.

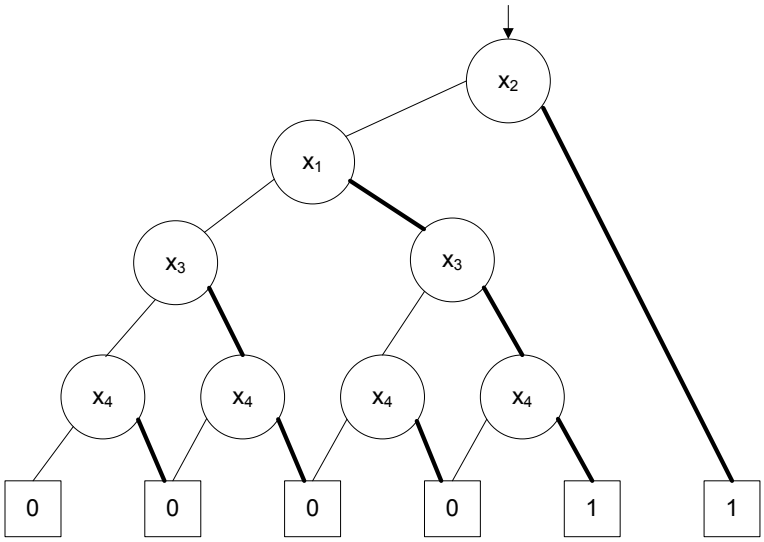


Figura 3.1: BDD ordenado para a função $[(x_1 \wedge x_3 \wedge x_4) \vee x_2]$.

Para que um BDD torne-se reduzido e ordenado, basta aplicar três regras para a redução:

1. Unir todos os vértices terminais que possuem o mesmo rótulo.
2. Unir todos os vértices internos que possuem os mesmos rótulos e sucessores.
3. Remover os vértices redundantes.

A figura 3.2 mostra o mesmo BDD da figura 3.1 porém reduzido.

Com isto, temos uma estrutura canônica[1], o que vai permitir a aplicação de BDDs para resolução de diversos problemas da área de circuitos integrados, inclusive instâncias de CEC.

Para verificar a equivalência entre dois circuitos, basta construir os BDDs referentes aos mesmos. Como os BDDs construídos serão estruturas canônicas, basta comparar os dois, o que pode ser feito em tempo constante [Clarke et al., 1999]. Caso eles sejam idênticos, os circuitos são equivalentes.

[1]ROBDDs possuem a propriedade da canonicidade: dada uma ordem fixa das variáveis, existe um único BDD para o conjunto.

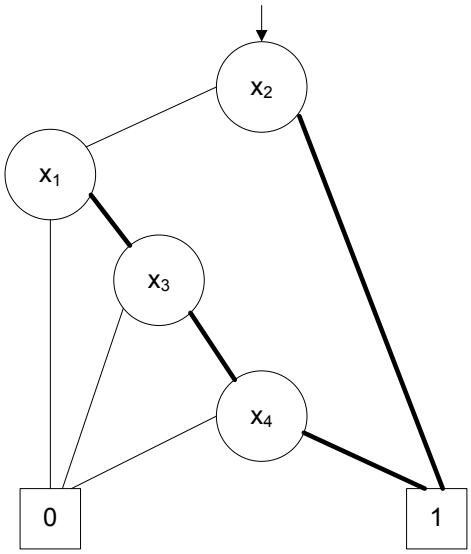


Figura 3.2: BDD reduzido e ordenado para a função $[(x_1 \wedge x_3 \wedge x_4) \vee x_2]$.

Os primeiros trabalhos que sugeriam o uso de BDDs em instâncias de CEC apareceram ao final da década de 80, com [Fujita et al., 1988] e [Malik et al., 1988], considerando a representação de circuitos em nível de bit. Três anos depois, [Bryant, 1991] provou que o BDD para um multiplicador de números inteiros cresce exponencialmente com relação ao seu número de bits.

Em 1994, [C. A. J. van Eijk e G. L. J. M. Janssen, 1994] apresentou uma técnica que explorava as similaridades entre os circuitos a serem verificados. Esta idéia introduziu uma melhora significativa no tempo de verificação, sendo utilizada, com variações, até os dias atuais. Conforme o esperado, esta técnica não apresentava melhora na verificação de circuitos dissimilares. Os trabalhos mais recentes de BDD aplicado a CEC focaram basicamente, mas não exclusivamente, em heurísticas para cortes do circuito ([Kuehlmann e Krohm, 1997] e [Xu et al., 2003]) ou na integração entre BDDs e outras técnicas (como por exemplo, resolvedores SAT), conforme veremos na seção 3.6).

As desvantagens do uso de BDDs são evidentes quando aplicados a circuitos dissimilares, não equivalentes ou até mesmo aritméticos, em especial a multiplicadores[2]. Além disso, verifica circuitos somente no nível de portas lógicas, o que limita o seu uso durante a etapa de projeto, quando um circuito precisa ser verificado entre diferentes níveis.

3.2 Diagrama de Momento Binário

A fim de superar as limitações do uso de BDDs para circuitos aritméticos, [Bryant e Chen, 1995] propôs uma alternativa, denominada Diagrama de Momento Binário (BMD, do inglês *Binary Moment Diagram*). Com esta técnica, é possível representar circuitos (em especial, multiplicadores) ocupando um espaço bem menor para armazenamento da estrutura de da-

[2]Os multiplicadores enquadram-se em uma classe de circuitos particularmente difícil de ser verificada, devido à característica da explosão de estados [van der Schoot e Ural, 1997]

dos, com relação ao BDD. O ganho de espaço se deve à representação do circuito em nível de palavra. Dessa forma, os autores puderam verificar multiplicadores de 256 bits em alguns minutos. Assim como para os BDDs, utilizaremos o termo BMD como sinônimo para ROBMD.

Dentre as propostas de melhorias para a representação em BMDs, destacam-se o trabalho de [Hamaguchi et al., 1995], que propôs um método bastante eficiente para a construção de BMDs e a proposta de [Chen e Chen, 2001] para uma nova estrutura, denominada *PHDDs (do inglês *Multiplicative Power Hybrid Decision Diagrams*, para verificação de multiplicadores dissimilares.

Os BMDs também possuem suas desvantagens. Talvez a maior delas seja o uso de uma descrição do circuito em nível de palavras, pois esta não costuma ser utilizado durante o projeto de circuitos. Este tipo de representação também prejudica a sua comparação com outras metodologias de CEC, que geralmente realizam a comparação em outros níveis de abstração[3]. BMDs não são eficientes o suficiente para representar divisão de inteiros, pois crescem exponencialmente, conforme mostrado em [Ozguner et al., 2001]. Outra desvantagem dos BMDs é que a maioria dos trabalhos disponíveis na literatura provam que um circuito livre de erros pode ser provado, de forma eficiente, ser livre de erros ([Bryant e Chen, 1995] e [Hamaguchi et al., 1995]), porém não provam a eficiência dos BMDs para verificar que um circuito com erros realmente possui errors [Wefel e Molitor, 2000].

3.3 Geração Automática de Padrões de Teste

A Geração Automática de Padrões de Teste (ATPG, do inglês *Automatic Test Pattern Generation*) é uma metodologia similar a que utiliza resolvedores SAT. A principal diferença entre elas reside no fato da baseada em SAT utilizar a representação do circuito em CNF, enquanto ATPG utiliza a representação como redes booleanas. [Kunz et al., 2002] apresenta uma comparação detalhada entre SAT e ATPG.

A verificação de equivalência por ATPG baseia-se na geração de padrões de testes que causam falhas no circuito verificado, se comparado com um circuito correto. Para analisar as falhas, existe um modelo denominado *stuck-at-fault*, que representa uma falha lógica consistindo de um sinal que mantém-se sempre em um mesmo valor. Caso este valor seja 0, o modelo é denominado *stuck-at-0*, caso seja 1, *stuck-at-1*.

Para verificar se um sinal possui uma falha do tipo *stuck-at*, um padrão que ative a falha deve ser encontrado. Este processo é denominado justificação ou ativação. Com isso, este mesmo padrão deve ser aplicado aos circuitos com e sem falha, para que o valor seja propagado até uma saída primária. Um miter deve então ser construído e caso sua saída seja 1, os circuitos não são equivalentes. Isto equivale a testar a saída por uma falha do tipo *stuck-at-0*, o que satisfará o miter.

A figura 3.3 mostra um exemplo de aplicação do método. Para justificar $k = 1$, uma das entradas do miter deve ser 0 e a outra deve ser 1. O processo inicia com a decisão de quais

[3] O mais comum é a verificação no nível de portas lógicas.

valores assinalar para as entradas i e j do miter. Façamos $i = 1$ e $j = 0$. Por implicação, $j = 0$ nos faz deduzir que $h = 1$, $e = 1$, $g = 1$, $f = 0$ e $a = 0$. Para justificar $f = 0$, o sinal b ou c ou os dois devem valer 0. Para qualquer um desses valores, $d = 0$, o que, juntamente com $a = 0$ implica que $i = 0$. Entretanto, isto conflita com a idéia proposta de que $i = 1$. Portanto, vamos retroceder e tentar justificar $i = 0$ e $j = 1$.

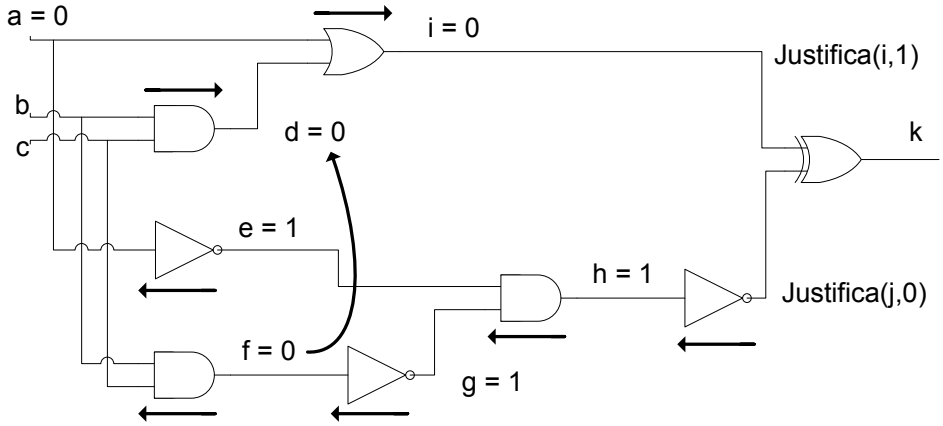


Figura 3.3: Justificando $i = 1$ e $j = 0$.

A figura 3.4 mostra o processo. O fato de $i = 0$ implica $a = 0$, $d = 0$ e $e = 1$. Para justificar $d = 0$, no mínimo b ou c devem valer 0. Isto implica $f = 0$ que, juntamente com $e = 1$ justifica $h = 1$. Portanto, $j = 0$, conflitando novamente com nossa suposição inicial $i = 0$ e $j = 1$. Com isso, temos a prova que os dois circuitos são funcionalmente equivalentes.

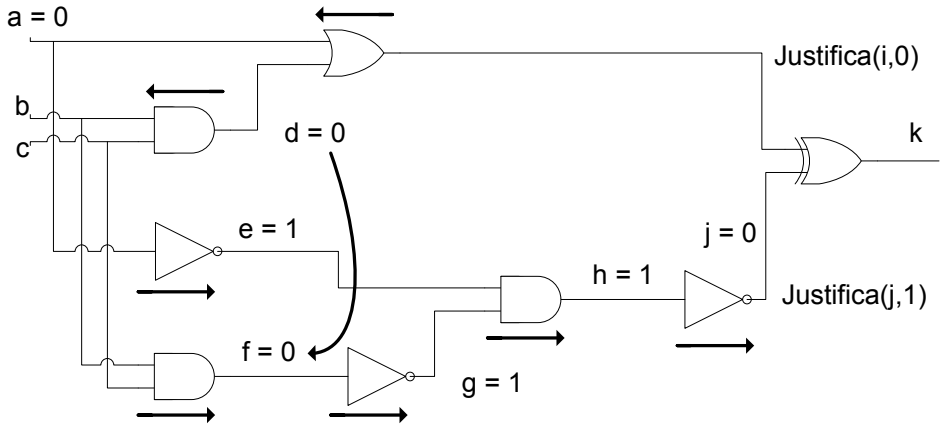


Figura 3.4: Justificando $i = 0$ e $j = 1$.

A técnica mostrada identifica somente implicações locais. Como a eficiência do processo de justificação depende diretamente das regras de implicações, é interessante combinar implicações locais com implicações globais, que analisam áreas maiores do circuito [Schulz e Auth, 1989]. Uma técnica de implicação global denominada aprendizado estático, apresentada por [Schulz et al., 1988], foi bastante utilizada até o surgimento de uma nova técnica denominada aprendizado recursivo. Esta técnica será apresentada na seção 3.4.

O primeiro algoritmo geral de ATPG foi apresentado por [Roth, 1966]. Baseado neste, vários outros algoritmos foram propostos. Dentre estes, podemos citar [Fujiwara e Shimono, 1983], [Kirkland e Mercer, 1987], [Bhattacharya e Hayes, 1990] e [Brand, 1993], sendo que este último obteve resultados significativos para o uso de ATPG em CEC. Esta proposta foi uma das primeiras a levar em consideração as similaridades internas dos circuitos durante a verificação de equivalência. Entretanto, com o crescente aumento da lógica presente nos circuitos integrados, ATPG mostrou-se uma técnica pouco escalável.

3.4 Aprendizado Recursivo

O Aprendizado Recursivo (RL, do inglês *Recursive Learning*), técnica inicialmente proposta em [Kunz, 1993] e aprimorada em [Kunz e Pradhan, 1994], surgiu como uma alternativa mais eficiente que o aprendizado estático [Schulz et al., 1988] utilizado em ATPG. No circuito da figura 3.5, quando $h = 0$, não é possível efetuar implicações locais e nem globais utilizando aprendizado estático, pois tal implicação não é possível devido aos assinalamentos para e e x_5 serem independentes entre si.

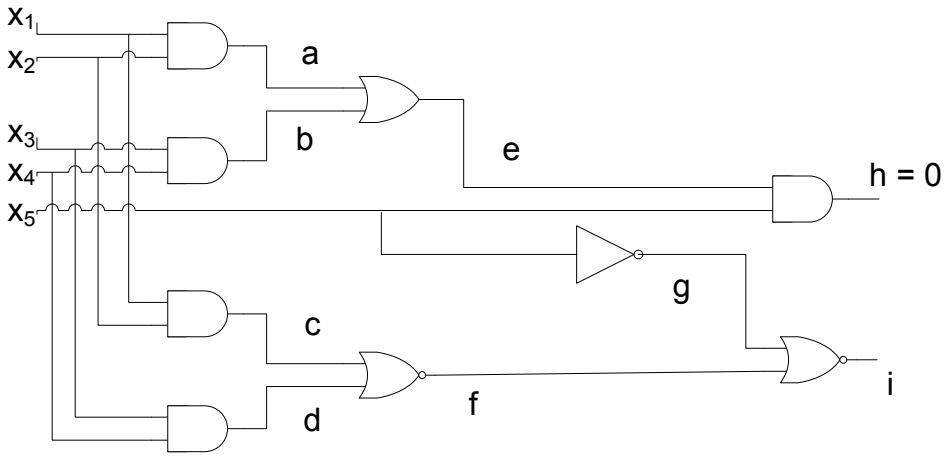


Figura 3.5: Circuito na qual a técnica mostrada na seção 3.3 não funciona.

Entretanto, é possível efetuar implicações a partir de $h = 0$ utilizando outros métodos. Observe que pelo menos e ou x_5 deve ser assinalado para 0. No primeiro caso, $g = 1$ e $i = 0$ podem ser diretamente implicados. No segundo caso, $a = 0$ e $b = 0$ são implicados, mas para justificar estes assinalamentos, uma nova decisão deve ser tomada. Em ambos os casos, $x_1 = 0$ e $x_2 = 0$, $c = 0$. Portanto, $a = 0$ implica $c = 0$. Da mesma forma, $b = 0$ implica $d = 0$. Sendo assim, $f = 1$ e $i = 0$ são implicados, revelando que para justificar $h = 0$, i deve ser assinalado com o valor 0.

RL destaca-se por ter sido uma das primeiras técnicas a explorar as vantagens proporcionadas ao se considerar as similaridades estruturais dos circuitos a serem verificados, conforme [Kunz, 1993] e [Silva e Silva, 1999].

A desvantagem do uso de RL é o crescimento exponencial da complexidade de tempo com relação ao nível de recursão. Entretanto, caso os circuitos sejam similares, o nível de recursão

geralmente é pequeno o suficiente para a técnica ser utilizada com eficiência.

3.5 Resolvedores SAT

O uso de resolvedores SAT baseado em DPLL para tratar instâncias de CEC é a metodologia mais recente das apresentadas até o momento. Este aparente atraso com relação às demais técnicas deve-se ao fato de que até alguns anos atrás, os resolvedores SAT não eram eficientes o suficiente para tratar problemas de CEC (e outros inúmeros problemas). Em 1992, [Larrabee, 1992] foi um dos primeiros trabalhos a sugerir o uso de resolvedores SAT em problemas de ATPG com circuitos combinacionais.

O resolvedor GRASP [Silva e Sakallah, 1996] é considerado por muitos como o ponto de partida para o recente aumento na eficiência dos resolvedores. Introduzindo idéias como retrocesso não-cronológico e aprendizado de cláusula, este resolvedor foi capaz de resolver, em poucos minutos, alguns problemas até antes considerados intratáveis. A partir de então, diversas outras publicações surgiram com propostas que aumentam a eficiência dos resolvedores SAT. Para maiores informações sobre algumas destas propostas e sobre o funcionamento dos resolvedores SAT baseados em DPLL, consulte a seção 2.4 ou as referências [Silva e Silva, 1999], [Moskewicz et al., 2001], [Zhang et al., 2001], [Goldberg e Novikov, 2002] e [Eén e Sörensson, 2003].

Os primeiros trabalhos que utilizam SAT para resolver problemas de CEC, envolviam o uso de outras técnicas. [Gupta e Ashar, 1998] propôs o uso de SAT e BDD. Em [Silva e Silva, 1999], um dos criados do GRASP apresentou uma técnica que integrava SAT com RL. Entretanto, o método proposto não conseguiu alcançar os resultados existentes para algoritmos baseados em BDD, BMD e ATPG. Dois anos mais tarde, [Goldberg et al., 2001] conseguiu, pela primeira vez, obter resultados semelhantes aos melhores obtidos por outras técnicas, utilizando somente SAT. Isto contribuiu diretamente para que esta técnica se firmasse como uma alternativa para CEC.

Os trabalhos mais recentes que utilizam exclusivamente SAT não envolvem necessariamente modificações nos resolvedores, mas sim técnicas para pré-processamento do circuito que envolvem diferentes formas de particionar um circuito ou análise estrutural do mesmo. Por exemplo, [Silva, 2000], [Subbarayan e Pradhan, 2004] e [Bacchus e Winter, 2004] realizam um pré-processamento nas cláusulas em CNF imediatamente antes de submetê-las ao resolvedor. Estas técnicas são mais gerais e no caso de CEC, técnicas que executam um pré-processamento antes da conversão do circuito para CNF geralmente obtém um melhor resultado. Por exemplo, [Arora e Hsiao, 2003], [Arora e Hsiao, 2004] e [Andrade, 2008] obtiveram resultados significativos ao derivarem implicações a partir do grafo de implicações do miter a ser verificado, convertê-las para CNF e adicioná-las ao problema a ser submetido ao resolvedor SAT.

Para que instâncias de CEC possam ser submetidas a um resolvedor SAT, basta construir um miter conectando os dois circuitos a serem verificados, conforme descrito na seção 2.2, e converter o circuito para sua descrição em CNF, pois este é o formato requerido como entrada

para a grande maioria dos resolvedores SAT. Ao submeter-se o circuito para o resolvedor, deve-se assumir que saída do miter será 1. Caso isto seja verdade, o resolvedor retorna SAT e os circuitos não são equivalentes. Caso contrário, o resolvedor retorna UNSAT e os circuitos são equivalentes.

A utilização de resolvedores SAT em CEC é vantajosa pois ataca as deficiências das outras técnicas. Tanto circuitos similares quanto dissimilares podem ser analisados de forma eficiente, bem como circuitos em diferentes níveis de projeto, desde que convertidos para CNF. Além disso, é uma área recente onde avanços são publicados freqüentemente. Como desvantagem, podemos citar a perda das informações estruturais do circuito durante a conversão para CNF.

3.6 Técnicas Integradas

As técnicas integradas são aquelas que envolvem o uso de duas ou mais técnicas para resolver problemas de CEC. Esta idéia é baseada no fato de que dependendo do problema a ser tratado, uma técnica se sobressai a outra. A dificuldade está em classificar os problemas de acordo com as técnicas, ou seja, saber exatamente qual metodologia será mais eficiente com determinada entrada.

Esta dificuldade apareceu já na primeira proposta de integração de técnicas: a ferramenta proposta por [Mukherjee et al., 1997] não funcionava bem para todos todas as classes de circuitos. Um ano depois, [Burch e Singhal, 1998] propôs que se a escolha da técnica não fosse adequada, a execução poderia ser interrompida e uma nova técnica selecionada, sem descartar as informações obtidas durante a análise anterior.

Para evitar uma solução mais complexa composta por várias técnicas, alguns trabalhos propõem o uso de duas técnicas somente. Embora não tenha conseguido superar os melhores resultados até então existentes, [Silva e Silva, 1999] propôs o uso de SAT em conjunto com RL. Mas o mais comum são técnicas envolvendo a integração de BDD e SAT: [Gupta e Ashar, 1998], [Reda et al., 2000], [Reda e Salem, 2001] e [Damiano e Kukula, 2003]. Excelentes resultados tem sido produzidos por essas técnicas, superando os obtidos quando cada técnica é utilizada individualmente. Isto é esperado devido à própria natureza das metodologias que integram mais de uma proposta.

Apesar da superioridade das técnicas integradas, é importante ressaltar que elas dependem diretamente da eficiência cada vez maior das técnicas individuais. Por isso, as pesquisas na área caminham em paralelo, afinal, um aumento de desempenho em uma determinada metodologia pode significar também uma melhora proporcional no desempenho de uma técnica que empregue aquela metodologia em conjunto com outras.

3.7 Trabalhos Relacionados

A utilização de resolvedores SAT baseados em DPLL para resolver problemas de verificação de equivalência entre circuitos combinacionais tem se consagrado como uma proposta eficiente graças aos recentes avanços da área [Prasad et al., 2005] desde a proposta inicial por

[Davis e Putnam, 1960] e [Davis et al., 1962], e por não possuir as mesmas limitações que outras técnicas baseadas em BDD, BMD, RL e ATPG.

Os trabalhos de [Gupta e Ashar, 1998] e [Silva e Silva, 1999] foram uns dos primeiros a sugerir o uso de SAT para resolver problemas de CEC e, embora não obtiveram resultados que superassem os atingidos por BDDs e utilizassem SAT em conjunto com outras técnicas, deram os primeiros indícios de que SAT poderia ser tão bom quanto estes. Isto foi comprovado alguns anos mais tarde, quando [Goldberg et al., 2001] propôs uma heurística de cortes para os circuitos sob verificação a fim de que estes fossem submetidos a um resolvedor SAT.

A idéia de particionar os circuitos para facilitar a análise pelos resolvedores foi adotada por diversos pesquisadores. [Lu et al., 2003a] e [Lu et al., 2003b] apresentam uma técnica denominada Aprendizado Explícito (do inglês, *Explicit Learning*), onde partições são criadas a partir do relacionamento entre os sinais de forma que cada partição seguinte incorpore a anterior. Com isso, cláusulas derivadas das partições menores são propagadas para as maiores, diminuindo o tempo de execução. A técnica apresentada no presente trabalho assemelha-se em alguns aspectos ao Aprendizado Explícito, porém o particionamento efetuado não necessita incluir toda a partição anterior na seguinte, e a equivalência é provada para cada partição separadamente, reaproveitando somente as cláusulas de conflito que interessam às partições adjacentes.

Mais recentemente, [Disch e Scholl, 2007] adotou a mesma técnica de particionamento utilizada no presente trabalho, mas ao invés de selecionar quais cláusulas de conflito seriam reaproveitadas entre as partições, ele propõe o reaproveitamento de todas as cláusulas de conflito. Outra diferença com relação ao presente trabalho é que a ordem de análise das partições é baseada em uma heurística e as cláusulas armazenadas são removidas de acordo com determinada estratégia, para evitar que o número excessivo de cláusulas influencie negativamente no tempo de execução do resolvedor SAT.

Outra abordagem recente, [de Oliveira, 2006] propõe a análise das partições de forma distribuída caso os circuitos sejam dissimilares e caso contrário, a exploração de similaridades entre os circuitos submetidos ao resolvedor reaproveitando somente as cláusulas de conflito que implicam em funções booleanas (mais especificamente, equivalência).

Assim como [de Oliveira, 2006], boa parte das técnicas de CEC exploram as similaridades estruturais entre os dois circuitos a serem verificados, como por exemplo [Burch e Singhal, 1998], [C. A. J. van Eijk e G. L. J. M. Janssen, 1994] e [Kuehlmann e Krohm, 1997]. Com isso, há técnicas muito mais eficientes para verificação de circuitos similares do que dissimilares. Visando ocupar este espaço da área de verificação, [Stanion, 1999] propôs uma técnica que levava em consideração a dependência estrutural de cada circuito. Porém, o uso de BDDs limitava a escalabilidade da técnica. Quatro anos depois, [Goldberg e Novikov, 2003] introduziu o conceito de Especificação Comum (CS, do inglês *Common Specification*), baseado na similaridade estrutural entre circuitos. Mesmo para circuitos com baixa similaridade a idéia é que se a CS dos circuitos possui um grande bloco, maior a similaridade entre os circuitos e um algoritmo que conheça a CS será capaz de resolver o problema em tempo linear, com relação ao número de blocos. A proposta mostrou-se eficiente, entretanto, obter a CS entre

dois circuitos é um processo não trivial.

A proposta de [Andrade et al., 2007], base para o presente trabalho, foi expandir a técnica de reaproveitamento de cláusulas de conflito apresentada em [de Oliveira, 2006] e aplicar a circuitos dissimilares com dependência estrutural. Além de selecionar as cláusulas que implicam em equivalência, decidiu-se por selecionar também as cláusulas que implicam em equivalência complementada. Segundo os autores, a técnica proposta proporciona um aumento na performance da análise tanto de circuitos similares quanto dissimilares.

Capítulo 4

Metodologia

> "Computers are good at following instructions, but not at reading your mind."
>
> Donald Knuth

O objetivo deste capítulo é apresentar a metodologia proposta para resolver o problema da CEC em circuitos dissimilares. A técnica apresentada baseia-se no particionamento do circuito e na verificação seqüencial de cada partição utilizando um resolvedor SAT com o reaproveitamento das cláusulas de conflito específicas de partições anteriores, a fim de evitar buscas redundantes no espaço de solução, conforme proposto por [de Oliveira, 2006] e [Andrade et al., 2007].

A próxima seção apresenta o algoritmo base utilizado neste trabalho e as seções seguintes explicam e analisam o funcionamento e implementação do mesmo: será apresentada a técnica utilizada para o particionamento do circuito, a construção do miter, a conversão dos circuitos de entrada do padrão ISCAS para o DIMACS CNF, a seleção das cláusulas de conflito pertencentes a partições adjacentes, a apresentação de quatro heurísticas para o reaproveitamento das cláusulas de conflito geradas na partição anterior e a concatenação das fórmulas a serem submetidas ao resolvedor SAT.

4.1 Algoritmo

O algoritmo 4.1 trata o problema da verificação de equivalência de circuitos combinacionais, segundo [Andrade et al., 2007], realizando particionamentos e reaproveitando as cláusulas de conflito das partições anteriores.

O algoritmo requer como entrada dois circuitos, no formato ISCAS ([Brglez e Fujiwara, 1985] e [F. Brglez e Kozminski, 1989]), C_1 e C_2, equivalentes no número de entradas e de saídas primárias, ou seja, os circuitos deverão possuir o mesmo número de bits de entrada e o mesmo número de bits de saída. Ele retorna verdadeiro caso estes circuitos sejam equivalentes ou falso caso contrário.

Algoritmo 4.1 Algoritmo para verificação de equivalência de circuitos combinacionais utilizando particionamento e reaproveitamento de cláusulas de conflito

Entrada: Dois circuitos C_1 e C_2 com entradas e saídas equivalentes.
Saída: Verdadeiro caso os circuitos sejam equivalentes, falso caso contrário.

1: **Partição** $P_1, P_2 \leftarrow \emptyset$
2: **Circuito** $M_{atual}, M_{anterior} \leftarrow \emptyset$
3: **FórmulaCNF** $\phi, \phi_{conflitos} \leftarrow \emptyset$
4: **for** ($i \leftarrow 0$ to **NúmeroDeBitsDaSaída**(C_1)) **do**
5: $M_{anterior} \leftarrow M_{atual}$
6: $P_1 \leftarrow$ **ParticionaPorTFI**(C_1, i)
7: $P_2 \leftarrow$ **ParticionaPorTFI**(C_2, i)
8: $M_{atual} \leftarrow$ **CriaMiter**(P_1, P_2)
9: $\phi \leftarrow$ **ConverteParaCNF**(M_{atual})
10: **if** ($i \neq 0$) **then**
11: $\phi_{conflitos} \leftarrow$ **ExtraiConflitosCompartilhados**($\phi_{conflitos}, M_{atual} \cap M_{anterior}$)
12: $\phi_{conflitos} \leftarrow$ **HeurísticaDeReaproveitamento**($\phi_{conflitos}$)
13: $\phi \leftarrow$ **ConcatenaFórmulas**($\phi, \phi_{conflitos}$)
14: **end if**
15: **if** (**ResolvedorSAT**($\phi, \phi_{conflitos}$) = SAT) **then**
16: **return** FALSO
17: **end if**
18: **end for**
19: **return** VERDADEIRO

A partir de cada bit de saída, começando pelo menos significativo, uma partição é criada em cada circuito (P_1 e P_2 em C_1 e C_2, respectivamente) pela função **ParticionaPorTFI**. O algoritmo escolhido para o particionamento é o que utiliza o TFI[1], que será discutido na seção 4.2. A seguir, o miter M_{atual} é gerado pela função **CriaMiter**, que recebe como entrada as partições equivalentes de cada circuito. Informações sobre a construção do miter podem ser encontradas na seção 4.3. A função **ConverteParaCNF** converte o circuito do miter do formato ISCAS para o DIMACS CNF, pois este é o formato de entrada amplamente utilizado pelos resolvedores SAT. A seção 4.4 discute esta transformação.

Os três próximos passos nunca são executados na primeira iteração pois não há conflitos gerados anteriormente. A função **ExtraiConflitosCompartilhados** extrai das cláusulas de conflito geradas na iteração anterior, apenas aquelas que pertencem à interseção do miter atual com o miter anterior, armazenando-as na variável $\phi_{conflitos}$. Consulte a seção 4.5 para maiores informações. As cláusulas de conflito são então filtradas pela função **HeurísticaDeReaproveitamento** que definirá quais cláusulas realmente serão aproveitadas na iteração atual, dependendo da heurística de reaproveitamento utilizada. A seção 4.6 apresenta quatro propostas de heurísticas: reaproveitamento de todas as cláusulas de conflito (subseção 4.6.1), reaproveitamento de cláusulas de conflito com tamanho específico (subseção 4.6.2), reaproveitamento de cláusulas de conflito que constituem funções booleanas (subseção 4.6.3) e reaproveitamento de cláusulas de conflito unitárias ou que constituem funções booleanas

[1] Do inglês *transitive fanin* ou fanin transitivo.

(subseção 4.6.4). As cláusulas de conflito são então concatenadas com a fórmula que representa o miter atualmente analisado, através da função **ConcatenaFórmulas**, conforme mostrado na seção 4.7.

Finalmente o circuito é submetido ao resolvedor SAT, através da função **ResolvedorSAT** que recebe como parâmetro o circuito em CNF e a variável $\phi_{conflitos}$ para armazenar as cláusulas de conflito que serão repassadas para a próxima iteração. Caso o resolvedor retorne SAT, significa que as partições não são equivalentes, portanto os circuitos não são equivalentes e o algoritmo é finalizado retornando *FALSO*. Caso retorne UNSAT, a próxima partição será analisada. Caso todas as partições tenham retornado UNSAT, o algoritmo finaliza retornando *VERDADEIRO*, o que significa que os circuitos são equivalentes. Informações sobre o resolvedor SAT utilizado e justificativa para sua escolha encontram-se descritas no próximo capítulo, na seção 5.2.

A complexidade de tempo do algoritmo 4.1 pode ser calculada pela análise de cada função que o compõe. O algoritmo terá então a mesma complexidade da função que dominar assintoticamente[2] as demais. É possível observar que o algoritmo possui complexidade de ordem exponencial ($O(2^n)$) para o pior caso, sendo n o número de variáveis do problema. Conforme descrito na seção 2.4, esta é a complexidade para o pior caso de um resolvedor SAT de uso geral e conforme poderá ser visto nas seções seguintes, a função **ResolvedorSAT** possui o maior grau de complexidade se comparada com as demais.

4.2 Particionamento dos Circuitos

O particionamento de um problema em partes menores é um método amplamente utilizado na resolução de problemas maiores e mais complexos, até mesmo alguns aparentemente intratáveis. Uma das técnicas mais utilizadas para particionamento é a de divisão e conquista. Resumidamente, ela consiste em particionar determinado problema em sub-problemas, encontrando a solução para cada parte e então combinando as soluções em uma única para o problema inicial. Caso os sub-problemas continuem complexos, particionamentos recursivos podem ser aplicados até que esses possam ser resolvidos.

O particionamento de circuitos tem sido bastante utilizado e é essencial pois os mesmos têm se tornado maiores (em número de elementos lógicos) e mais complexos a cada dia. A tendência, de acordo com a Lei de Moore apresentada em [Moore, 1965], é que o número de transistores de um circuito integrado cresça exponencialmente, duplicando a cada dezoito meses aproximadamente.

Conforme já mencionado, este trabalho utiliza o particionamento por TFI, que funciona da seguinte maneira: a partir de uma das saídas primárias do circuito, percorre-se toda a lógica que influencia o valor daquela saída, até que as entradas primárias sejam alcançadas. Como o objetivo é particionar todo o circuito, isto deve ser feito para as n saídas do mesmo.

[2]Segundo [Knuth, 1968], uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes positivas c e m tais que, para todo $n \geq m$, temos $g(n) \leq cf(n)$. Isto equivale a dizer que $g(n)$ é $O(f(n))$.

Por exemplo, o circuito apresentado na figura 4.1 que possui 13 bits de entrada e 5 bits de saída, será particionado em 5 circuitos compostos por uma saída cada, alguma lógica interna e 3 a 5 entradas, variando de acordo com a partição, conforme mostrado na figura 4.3, a ser discutida ao final desta seção.

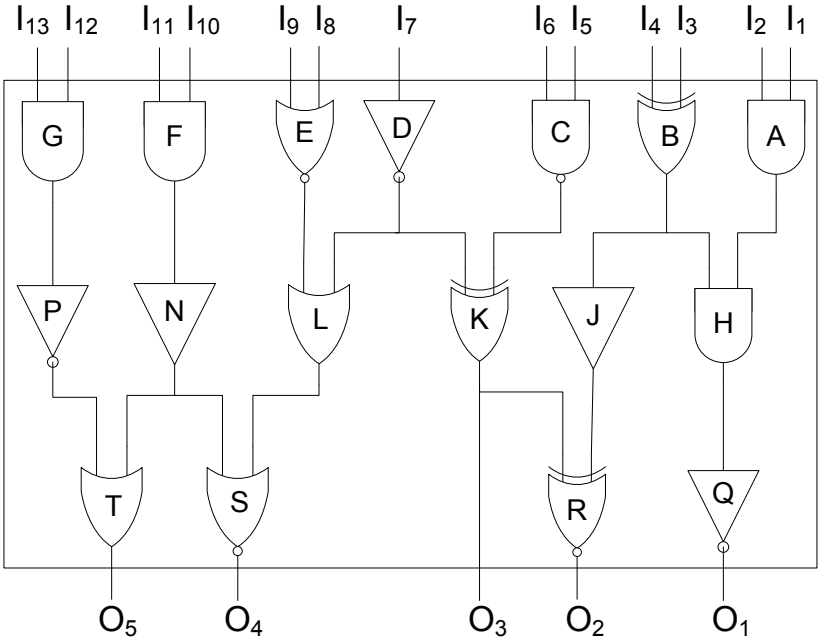


Figura 4.1: Circuito com 13 entradas e 5 saídas.

O algoritmo 4.2 expande a função **ParticionaPorTFI** do algoritmo 4.1. Ele requer como entrada o circuito inicial, no formato ISCAS, e a porta de saída pela qual o particionamento será iniciado. O passo inicial é a representação do circuito como um grafo acíclico direcionado[3]. Como trata-se de um circuito combinacional, este pode ser representado como um grafo, geralmente esparso[4], cujos vértices representam as entradas, as saídas e os elementos lógicos e as arestas representam as ligações entre eles, direcionadas na direção da propagação do sinal.

Algoritmo 4.2 Algoritmo para particionamento de circuito por TFI

Entrada: Um circuito C no formato ISCAS e a saída i do circuito.
Saída: Uma partição do circuito.

1: **Partição** $P \leftarrow \emptyset$
2: **Grafo** $G_1, G_2 \leftarrow \emptyset$
3: $G_1 \leftarrow$ **CircuitoParaGrafo**(C)
4: $G_1 \leftarrow$ **Transposto**(G_1)
5: $G_2 \leftarrow$ **BuscaVértices**(G_1, i)
6: $P \leftarrow$ **GrafoParaCircuito**(G_2)
7: **return** P

[3]Um grafo acíclico direcionado é aquele que não possui ciclos e cujo conjunto de arestas possui uma relação binária com o conjunto de vértices. Conhecido na literatura como DAG, do inglês *direct acyclic graph*.

[4]Um grafo esparso é aquele cujo número de arestas é bem menor que o número de vértices ao quadrado.

A figura 4.2 apresenta o circuito da figura 4.1 representado como um grafo acíclico direcionado.

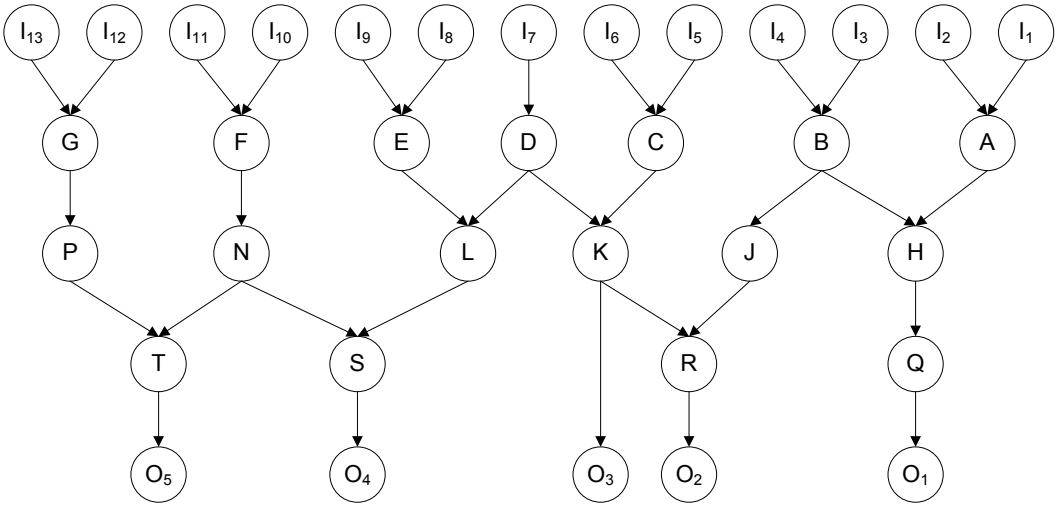
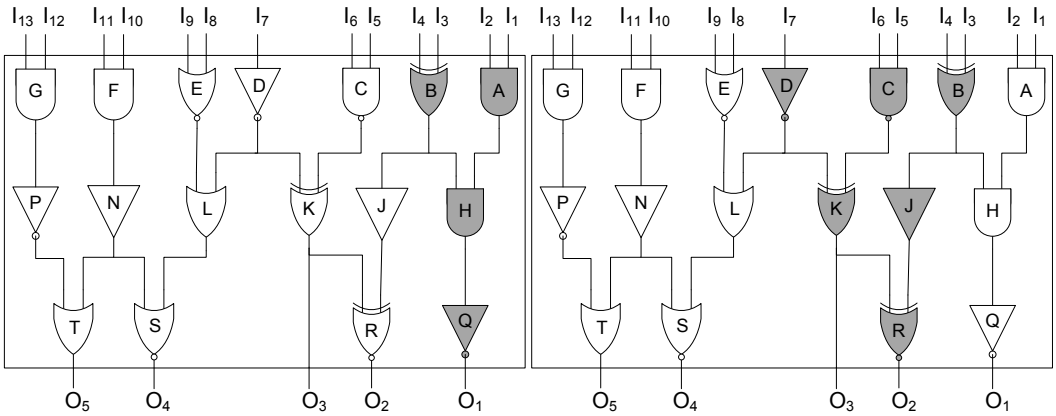


Figura 4.2: Representação do circuito da figura 4.1 como um grafo.

A representação do circuito em forma de grafo é feita pelo procedimento **CircuitoParaGrafo**, que recebe como entrada o circuito C no formato ISCAS e retorna o grafo G_1. Em seguida, o procedimento **Transposto** inverte a direção de todas as arestas de G_1 para que a função **BuscaVértices** retorne um grafo G_2, sendo $G_2 \subset G_1$. G_2 deve conter todos os vértices de G_1 alcançáveis a partir do vértice i. Em um grafo, isto pode ser feito utilizando-se um algoritmo de busca, como por exemplo, busca em largura ou busca em profundidade. Para este trabalho, o algoritmo escolhido foi o de busca em largura. Maiores informações sobre este algoritmo encontram-se disponíveis no apêndice A. Por fim, G_2 é convertido de grafo para um circuito no formato ISCAS, sendo retornado pelo procedimento.

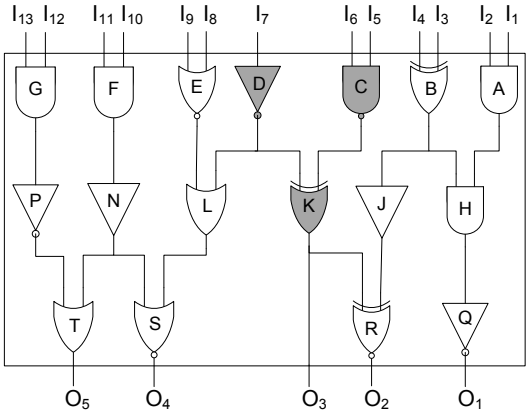
A figura 4.3 apresenta o circuito da figura 4.1 particionado por TFI. O circuito foi dividido em 5 partições, uma para cada porta de saída. Em cada quadro, as portas lógicas coloridas representam uma partição, juntamente com as saídas e entradas ligadas a elas. Por exemplo, a figura 4.3a mostra a partição relativa à porta O_1. As portas lógicas A, B, H e Q integram a partição juntamente com as quatro entradas I_1, I_2, I_3 e I_4. Já a figura 4.3d mostra a partição composta pela saída O_1, entradas I_7, I_8, I_9, I_10 e I_11 e lógicas D, E, F, L, N e S.

A complexidade da função **ParticionaPorTFI** é a mesma da função **BuscaVértices** que por sua vez é a mesma do algoritmo para busca em largura. Considerando o grafo com um número A de arestas e V de vértices, a complexidade, para o pior caso, é $O(|V| + |A|)$ conforme demonstrado em [Cormen et al., 2001]. Como cada vértice do grafo representa uma variável, temos que $V = n$. Sendo o grafo um DAG esparso, o número de arestas é bem menor que 2^n. Com isso, $O(2^n)$ domina assintoticamente $O(n + |A|)$.

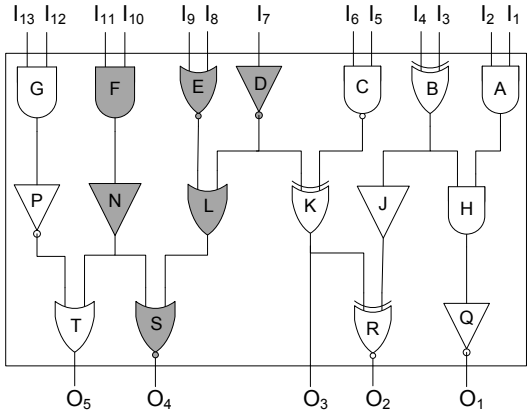


(a) Partição a partir de O_1

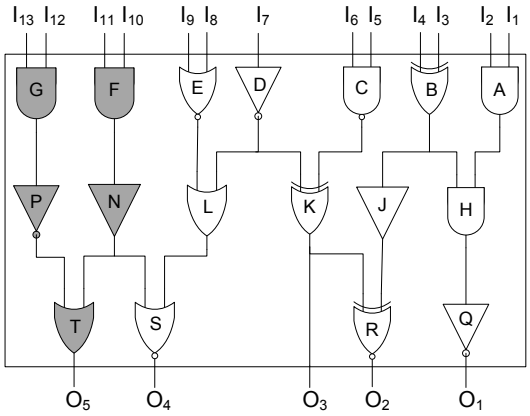
(b) Partição a partir de O_2



(c) Partição a partir de O_3



(d) Partição a partir de O_4



(e) Partição a partir de O_5

Figura 4.3: Circuito da figura 4.1 particionado por TFI.

4.3 Criação do Miter

A subseção 2.2 introduziu o conceito de miter. A técnica descrita neste trabalho necessita de um miter para provar a equivalência entre os dois circuitos verificados.

A figura 4.4 apresenta um miter construído a partir das saídas de dois circuitos, C_1 e C_2, semelhantes ao da figura 4.1.

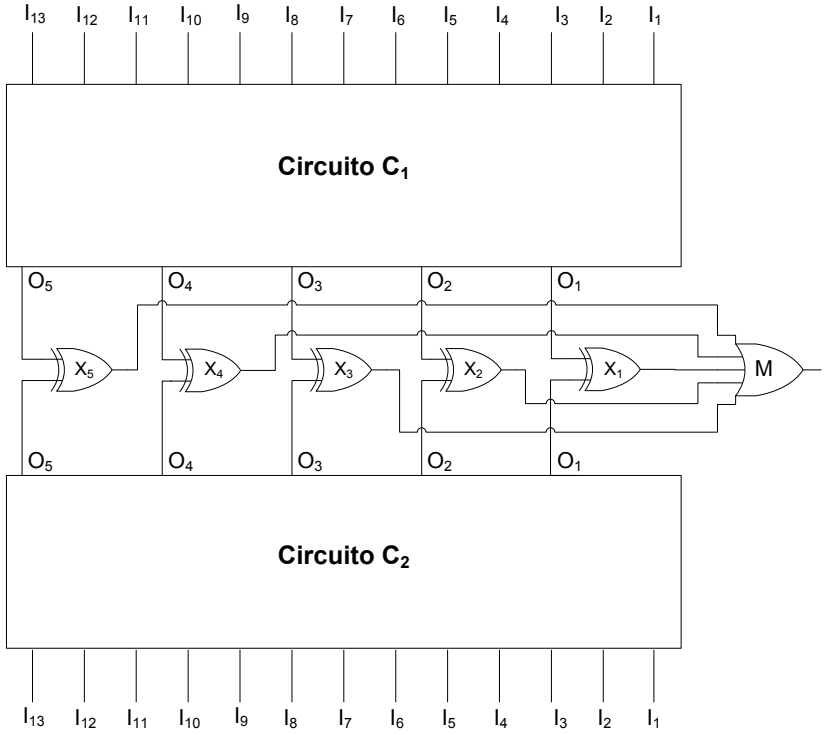


Figura 4.4: Miter construído a partir de dois circuitos semelhantes ao da figura 4.1.

As entradas correspondentes dos dois circuitos foram identificadas e nomeadas de forma equivalente, de I_1 a I_{13}. Já as saídas equivalentes foram conectadas como entrada para uma porta XOR. Sendo assim, foram adicionadas 5 portas XOR (X_1 a X_5), uma para cada par de saídas (O_1 a O_5, respectivamente). As saídas das portas XOR foram então ligadas a uma porta OR (M).

Para o funcionamento do algoritmo, basta que a função **CriaMiter** retorne um circuito composto pelas duas partições passadas como argumento, P_1 e P_2, cujas entradas equivalentes estejam identificadas e as duas saídas sejam entradas de uma porta XOR. Como teremos adicionado somente uma porta, não será necessário acrescentar uma porta OR à saída do miter.

A complexidade da função **CriaMiter** é constante ($O(1)$) com relação ao tamanho do número de variáveis do problema, pois esta apenas identifica as portas de entrada equivalentes e adiciona uma porta XOR nas saídas das duas partições a serem verificadas.

4.4 Conversão do Formato ISCAS para CNF

A função **ConverteParaCNF** recebe como entrada o miter atual (M_{atual}) para convertê-lo do formato ISCAS para DIMACS CNF. Conforme descrito na seção 4.1, é necessária esta conversão pois o padrão CNF é o formato de entrada amplamente utilizado pelos resolvedores SAT atuais e o padrão ISCAS é o principal utilizado por *benchmarks* da área de circuitos, como por exemplo os *benchmarks* ISCAS 85 [Brglez e Fujiwara, 1985], ISCAS 89 [F. Brglez e Kozminski, 1989] e os gerados através da ferramenta BenCGen, proposta por [Andrade et al., 2008] e ideais para para o problema de CEC. Estes últimos inclusive serão utilizados para a análise de resultados deste trabalho.

O formato ISCAS é convenientemente utilizado para representar circuitos aritméticos, possuindo as seguintes portas: AND, NAND, OR, NOR, BUFF, NOT, XOR, XNOR além de aceitar a especificação das entradas (INPUT) e saídas (OUTPUT). Cada variável deve representar somente um bit e não pode começar com um '_' ou conter os seguinte caracteres: ',', ')', '(', '='. Uma variável também não pode aparecer mais de uma vez do lado esquerdo de uma igualdade. Comentários devem começar com um '#' e com exceção de BUFF e NOT que aceitam somente uma variável como entrada, as outras portas aceitam como entrada duas ou mais variáveis. Um exemplo de arquivo no formato ISCAS pode ser visto a seguir:

```
INPUT(in1)
INPUT(in2)
INPUT(in3)
# Isto é um exemplo de comentário
d = AND(in2, in3)
e = NOR(in1, in3)
f = AND(in1, in2)
g = XOR(in1, in2, in)
h = OR(f, e)
out1 = NOT(g)
out2 = OR(h, d)
OUTPUT(out1)
OUTPUT(out2)
```

As três primeiras linhas definem *in1*, *in2* e *in3* como as entradas do circuito. A seguir, tem-se uma linha com comentário seguida por sete linhas representando as conexões das entradas do circuito até as suas saídas. Por exemplo, a expressão:

```
d = AND(in2, in3)
```

representa uma porta AND com duas entradas (*in2* e *in3*) e uma saída *d*. Por fim, as duas últimas linhas definem *out1* e *out2* como as saídas do circuito.

O formato DIMACS CNF é um padrão bastante difundido e utilizado para representar fórmulas booleanas em CNF. O início do arquivo pode conter linhas com comentários, iniciadas

pela letra *c*. A seguir, uma linha iniciada com *p cnf* define o número de variáveis e o número de cláusulas. Na seqüência, cada linha representa uma cláusula onde um número positivo representa um literal positivo e um número negativo representa um literal negativo. Estas linhas devem ser finalizadas com um 0. Um exemplo de arquivo no formato DIMACS CNF pode ser visto a seguir:

```
c Isto é um exemplo de comentário
p cnf 6 4
1 -3 0
2 3 -1 0
2 4 0
-4 -5 6 0
```

A primeira linha representa um comentário. A segunda define que a fórmula possui 6 variáveis e 4 cláusulas. É um erro um arquivo CNF conter um número de variáveis ou cláusulas diferentes do especificado. As linhas restantes representam efetivamente a fórmula booleana.

Suponha $1 = A$, $2 = B$, $3 = C$, $4 = D$, $5 = E$ e $6 = F$. Como cada linha é uma cláusula, temos a seguinte fórmula representada no exemplo:

$$(A \vee \overline{C}) \wedge (B \vee C \vee \overline{A}) \wedge (B \vee D) \wedge (\overline{D} \vee \overline{E} \vee F)$$

Apesar de bastante utilizada, uma desvantagem na representação de um circuito em CNF consiste no fato deste conter menos informação estrutural sobre o circuito do que o formato ISCAS. Por exemplo, não é possível afirmar com exatidão quais são as entradas e saídas de um circuito em CNF.

Para converter o circuito do formato ISCAS para o DIMACS CNF, será utilizada a técnica conhecida como transformação de Tseitin, descrita em [G. S. Tseitin, 1968] e [Larrabee, 1992]. Tal técnica consiste em transformar as igualdades em implicações lógicas, introduzindo uma nova variável para a saída de cada porta lógica. O exemplo a seguir mostra a transformação da expressão booleana de uma porta lógica AND em sua expressão CNF.

Suponha que I_1 e I_2 sejam as entradas e O a saída de uma porta AND. Portanto:

$$O = I_1 \cdot I_2 \tag{4.1}$$

Como uma igualdade $X = Y$ é logicamente equivalente a $(X \rightarrow Y) \cdot (Y \rightarrow X)$, a equação 4.1 equivale a

$$(O \rightarrow (I_1 \cdot I_2)) \cdot ((I_1 \cdot I_2) \rightarrow O).$$

Considerando que $X \rightarrow Y$ é logicamente equivalente a $\overline{X} + Y$, as implicações são convertidas em disjunções obtendo a fórmula

$$(\overline{O} + (I_1 \cdot I_2)) \cdot (\overline{(I_1 \cdot I_2)} + O)$$

que organizada na forma de produto de somas, torna-se

$$(\overline{O} + I_1) \cdot (\overline{O} + I_2) \cdot (\overline{I_1} + \overline{I_2} + O)$$

que é a expressão em CNF para uma porta AND com duas entradas.

Deduções similares podem ser aplicadas para portas AND com três ou mais entradas. Por exemplo, supondo que I_1, I_2 e I_3 sejam as entradas, a expressão em CNF para

$$O = I_1 \cdot I_2 \cdot I_3$$

equivale a

$$(\overline{O} + I_1) \cdot (\overline{O} + I_2) \cdot (\overline{O} + I_3) \cdot (\overline{I_1} + \overline{I_2} + \overline{I_3} + O).$$

As deduções não se restringem à porta AND, podendo ser aplicadas a todas as outras portas lógicas do formato ISCAS: NAND, OR, NOR, BUFF, NOT, XOR e XNOR. A tabela 4.1 mostra a conversão da expressão booleana destas portas lógicas para uma expressão CNF equivalente.

Porta Lógica	**Expressão booleana**	**Expressão em CNF**
AND	$O = I_1 \cdot I_2$	$(\overline{O} + I_1) \cdot (\overline{O} + I_2) \cdot (O + \overline{I_1} + \overline{I_2})$
NAND	$O = \overline{I_1 \cdot I_2}$	$(O + I_1) \cdot (O + I_2) \cdot (\overline{O} + \overline{I_1} + \overline{I_2})$
OR	$O = I_1 + I_2$	$(O + \overline{I_1}) \cdot (O + \overline{I_2}) \cdot (\overline{O} + I_1 + I_2)$
NOR	$O = \overline{I_1 + I_2}$	$(\overline{O} + \overline{I_1}) \cdot (\overline{O} + \overline{I_2}) \cdot (O + I_1 + I_2)$
BUFF	$O = I$	$(O + \overline{I}) \cdot (\overline{O} + I)$
NOT	$O = \overline{I}$	$(\overline{O} + \overline{I}) \cdot (O + I)$
XOR	$O = I_1 \oplus I_2$	$(O + \overline{I_1} + I_2) \cdot (O + I_1 + \overline{I_2}) \cdot$ $(\overline{O} + I_1 + I_2) \cdot (\overline{O} + \overline{I_1} + \overline{I_2})$
XNOR	$O = \overline{I_1 \oplus I_2}$	$(\overline{O} + \overline{I_1} + I_2) \cdot (\overline{O} + I_1 + \overline{I_2}) \cdot$ $(O + \overline{I_1} + \overline{I_2}) \cdot (O + I_1 + I_2)$

Tabela 4.1: Conversão de expressões booleanas para expressões CNF

À primeira vista, a transformação de Tseitin pode parecer desvantajosa pela inserção de novas variáveis. Entretanto, esta transformação permite que a conversão seja realizada em tempo linear com relação ao tamanho da fórmula, pois, conforme mostrado em [Groote e Zantema, 2002], sem a inserção de novas variáveis, a conversão dos formatos é necessariamente exponencial. Portanto, a complexidade da função **ConverteParaCNF** é ($O(n)$).

4.5 Seleção de Conflitos Compartilhados

Para evitar que variáveis de uma partição anterior que não façam parte da partição atual sejam adicionadas ao problema, a função **ExtraiConflitosCompartilhados** extrai das cláusulas de conflito, $\phi_{conflitos}$, geradas a partir da análise da partição anterior, apenas aquelas que pertencem à interseção dos dois miters. Isto é o mesmo que selecionar somente as cláusulas compostas por variáveis presentes nos miters adjacentes ($M_{atual} \cap M_{anterior}$). Desta forma, o número de cláusulas de conflito será reduzido porém suficiente para cobrir toda a área compartilhada pelas dois miters.

A figura 4.5 mostra dois circuitos particionados. As áreas coloridas indicam que duas ou mais partições compartilham parte de sua lógica, ou seja, há variáveis compartilhadas por duas ou mais partições, considerando-se a fórmula CNF das mesmas. Por exemplo, as partições P_1 e P_2 dos dois circuitos compartilham áreas denominadas S_{12} e as partições P_4 e P_5 compartilham as áreas coloridas S_{45}.

A função **ExtraiConflitosCompartilhados** possui complexidade $O(n)$ para o pior caso, que ocorre quando todas as n variáveis das duas partições são compartilhadas.

4.6 Heurísticas para Reaproveitamento de Cláusulas de Conflito

A função **HeurísticaDeReaproveitamento** é responsável pela seleção de quais cláusulas de conflito $\phi_{conflitos}$ geradas na análise da partição anterior serão incorporadas às cláusulas que representam a partição atual. Ao propor este reaproveitamento das cláusulas de conflito, o objetivo é restringir o espaço de solução, ou seja, diminuir a quantidade de assinalamentos possíveis para a fórmula CNF, reduzindo assim o tempo necessário para verificação da partição (e consequentemente, do problema completo) pelo resolvedor SAT. Desta forma, a função **HeurísticaDeReaproveitamento** juntamente com as funções **ExtraiConflitos** e **ConcatenaFórmulas** realizam um pré-processamento da entrada a ser enviada para o resolvedor SAT, na expectativa de torná-la menos complexa. Esta tentativa é válida pois grande parte do tempo de execução do algoritmo é devido à função **ResolvedorSAT** e não há adição de variáveis ao problema, somente de cláusulas.

Quatro heurísticas são analisadas neste trabalho: reaproveitamento de todas as cláusulas de conflito, reaproveitamento de cláusulas de conflito com tamanho específico, reaproveitamento de cláusulas de conflito que constituem funções booleanas e reaproveitamento de cláusu-

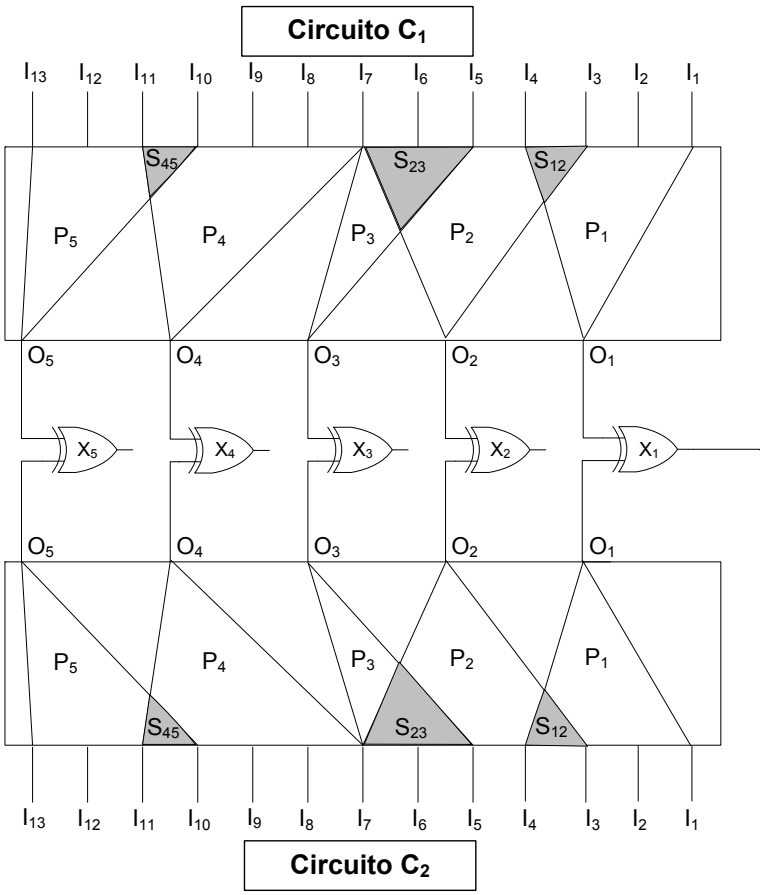


Figura 4.5: Apenas as cláusulas de conflito compartilhadas entre partições adjacentes são aproveitadas.

las de conflito unitárias ou que constituem funções booleanas. As subseções seguintes apresentam e analisam as técnicas.

4.6.1 Reaproveitamento de Todas as Cláusulas de Conflito

Proposto em [Andrade et al., 2007], o reaproveitamento de todas as cláusulas de conflito da partição anterior baseia-se na idéia de que quanto maior o número de cláusulas de conflito, mais restrito será o espaço de solução. Isto ocorre pois o número de assinalamentos possíveis tende a diminuir, desde que não haja a adição de novas variáveis, o que é garantido pela função **ExtraiConflitosCompartilhados** (vide seção 4.5), executada previamente.

Em termos de implementação, não é necessário fazer nada pois a saída da função **ExtraiConflitosCompartilhados** é o conjunto das cláusulas de conflito da partição anterior cujas variáveis pertencem também à partição atual. Portanto, este conjunto será a entrada $\phi_{conflitos}$ da função **ConcatenaFórmulas**.

4.6.2 Reaproveitamento de Cláusulas de Conflito com Tamanho Específico

Proposto em [Andrade et al., 2007], o reaproveitamento das cláusulas de conflito de tamanho específico baseia-se na premissa de que nem todas as cláusulas de conflito são úteis, especialmente aquelas que possuem um número maior de literais [Arora e Hsiao, 2003]. Portanto, somente cláusulas com um número menor ou igual ao especificado serão utilizadas.

Um possível algoritmo para esta técnica é apresentado no algoritmo 4.3. O algoritmo recebe como entrada o conjunto das cláusulas de conflito geradas pela função **ExtraiConflitosCompartilhados** e um número inteiro positivo x que representa o tamanho máximo das cláusulas de conflito a serem selecionadas. O número de literais de cada cláusula é comparado com x e caso seja maior, a cláusula é removida de $\phi_{conflitos}$. Ao final da execução, $\phi_{conflitos}$ conterá todas as cláusulas com tamanho menor ou igual a x.

Algoritmo 4.3 Algoritmo para reaproveitamento de cláusulas de conflito com tamanho específico

Entrada: Um conjunto $\phi_{conflitos}$ de cláusulas em CNF e um inteiro positivo x.
Saída: Um subconjunto de $\phi_{conflitos}$ sendo cada cláusula menor ou igual a x.

```
1: for (i ← 0 to (NúmeroDeCláusulas(φ_conflitos) − 1)) do
2:    if (NúmeroDeLiterais(φ_conflitos[i]) > x) then
3:       φ_conflitos ← RemoveCláusula(φ_conflitos[i])
4:    end if
5: end for
6: return φ_conflitos
```

Considerando que as funções **NúmeroDeCláusulas**, **NúmeroDeLiterais** e **RemoveCláusula** são constantes, o algoritmo 4.3 possui complexidade de tempo linear com relação ao número de cláusulas, afinal todas as cláusulas de $\phi_{conflitos}$ devem ser verificadas.

4.6.3 Reaproveitamento de Cláusulas de Conflito que Constituem Funções Booleanas

A técnica de reaproveitamento das cláusulas de conflito que constituem funções booleanas consiste em selecionar somente as cláusulas resultantes da análise da partição anterior que juntas formam algum tipo de função booleana, tal como AND, OR, NAND, NOR, equivalência (BUFF), equivalência complementada (NOT), XOR, e XNOR. Esta idéia baseia-se na identificação de similaridades estruturais internas de cada uma das duas partições que compõem o miter. A tabela 4.1 mostra como identificar cada uma das expressões booleanas procuradas em CNF.

Inicialmente apresentada em [de Oliveira, 2006], a técnica identificava similaridades do tipo equivalência entre dois circuitos, ou seja, funcionava para circuitos similares. O algoritmo foi expandido por [Andrade et al., 2007] para também identificar equivalência complementada, e os autores afirmam que esta abordagem funciona para circuitos dissimilares que possuem um

alto grau de similaridade interna.

A principal vantagem desta técnica é o fato de ser desnecessário um conhecimento prévio sobre a estrutura interna do circuito para a seleção das cláusulas, pois ela está baseada nas cláusulas de conflito geradas pelo resolvedor SAT. Isto permite o uso do formato CNF de forma eficiente e sem grandes acréscimos de tempo.

Encontrar um subconjunto de cláusulas de conflito que formam algumas das expressões booleanas procuradas é um problema $\mathcal{NP}$-Completo [Cook, 1971], análogo ao problema da soma de subconjuntos[5]. Este problema consiste em responder à questão: dados um conjunto finito numérico S e um número finito t, existe um subconjunto $S' \subseteq S$ cuja soma dos elementos seja igual a t?

Um possível algoritmo com complexidade de tempo exponencial é verificar se cada combinação possível entre as n cláusulas de conflito forma alguma das expressões procuradas. Claramente este algoritmo não é eficiente pois cada um dos 2^n conjuntos de cláusulas gerados devem ser verificados se formam uma das m funções booleanas. Com isto, a complexidade de tempo do algoritmo é $O(m2^n)$.

Para resolver este problema, uma metodologia mais eficiente foi proposta em [de Oliveira, 2006]. Esta metodologia pode ser dividida em duas etapas:

1. Criação de um número reduzido de grupos contendo as cláusulas com duas ou três variáveis que possuem um certo grau de correlação.

2. Identificação das expressões booleanas dentre os elementos de cada grupo.

Desta forma, o problema torna-se computacionalmente viável.

O algoritmo 4.4 apresenta uma possível implementação para a primeira etapa da metodologia descrita nesta seção. Baseado no algoritmo *Cluster Growth* [Gajski et al., 1992] para particionamento de circuitos, a técnica agrupa as cláusulas de conflito que possuem duas ou três variáveis compartilhadas.

Resumidamente, o algoritmo *Cluster Growth* recebe como entrada um conjunto de objetos não particionados para serem distribuídos em grupos de acordo com uma determinada função. No presente caso, esta função é determinada pelo número de variáveis em comum entre as cláusulas (objetos).

O algoritmo 4.4 recebe como entrada um conjunto de cláusulas e seu passo inicial consiste em descartar aquelas constituídas por mais de três variáveis, através da função **RemoveCláusulasMaiores**. Isto é possível pois quanto maior a cláusula, menor a probabilidade de se encontrar outras cláusulas que juntamente com esta constituirão alguma das expressões booleanas da tabela 4.1 e todas as funções listadas podem ser geradas com cláusulas constituídas por dois ou três literais. Observe que o algoritmo da função **RemoveCláusulasMaiores** é o mesmo do algoritmo 4.3). Na seqüência, as cláusulas unitárias também são removidas pois nenhum das funções booleanas de interesse são constituídas por cláusulas unitárias.

[5] O problema da soma de subconjuntos é conhecido na literatura como *subset sum* [Cormen et al., 2001]

Algoritmo 4.4 Algoritmo para agrupamento de cláusulas de conflito

Entrada: Um conjunto $\phi_{conflitos}$ de cláusulas em CNF.
Saída: Um conjunto G de grupos de cláusulas que possuem duas ou três variáveis em comum.

```
1: φ_conflitos ← RemoveCláusulasMaiores(φ_conflitos, 3)
2: φ_conflitos ← RemoveCláusulasUnitárias(φ_conflitos)
3: for (i ← 0 to (NúmeroDeCláusulas(φ_conflitos) − 1)) do
4:    criaGrupo ← 1
5:    for (j ← 0 to (NúmeroDeGrupos(G) − 1)) do
6:       variáveisIguais ← 0
7:       for (x ← 0 to (NúmeroDeVariáveis(G[j]) − 1)) do
8:          for (y ← 0 to (NúmeroDeVariáveis(φ_conflitos[i]) − 1)) do
9:             if (φ_conflitos[i][y] = G[j][x]) then
10:               variáveisIguais++
11:            end if
12:         end for
13:      end for
14:      if ((variáveisIguais = 2) || (variáveisIguais = 3)) then
15:         InsereNoGrupo(φ_conflitos[i], G[j])
16:      end if
17:      if ((variáveisIguais = 2) && (NúmeroDeVariáveis(φ_conflitos[i]) = 2)
         && (NúmeroDeVariáveis(G[j]) = 2)) || ((variáveisIguais = 3) &&
         (NúmeroDeVariáveis(G[j]) = 3)) then
18:         criaGrupo ← 0
19:      end if
20:   end for
21:   if criaGrupo = 1 then
22:      criaNovoGrupo(φ_conflitos[i])
23:   end if
24: end for
25: G ← RemoveGruposPequenos(G)
26: return G
```

Em seguida, é calculado o número de variáveis compartilhadas entre cada cláusula e cada grupo. Um grupo é constituído pela cláusula inicial, que servirá de base para comparação entre suas variáveis e as de outras cláusulas, e por cláusulas adicionadas durante a execução do algoritmo quando há semelhança entre duas ou três variáveis. Caso a cláusula comparada não possua um grupo onde todas as suas variáveis sejam compartilhadas, um novo grupo é criado para esta cláusula, através da função **criaNovoGrupo**. O mesmo não ocorre quando a cláusula comparada possui duas variáveis semelhantes às de um grupo cuja cláusula inicial possui exatamente duas variáveis ou quando a cláusula comparada possui as mesmas três variáveis de um determinado grupo.

O último passo do algoritmo é a remoção de dois grupos específicos de cláusulas pela função **RemoveGruposPequenos**:

- Grupos constituídos por somente uma cláusula: o número de cláusulas das expressões

booleanas da tabela 4.1 variam entre duas a quatro cláusulas.

- Grupos constituídos por duas cláusulas sendo que ao menos uma delas contém três literais: as cláusulas que possuem três literais pertencem a funções booleanas formadas por, no mínimo, três cláusulas.

Após a execução do algoritmo 4.4, a técnica prossegue para a etapa de identificação das expressões booleanas dentre os elementos de cada grupo. Como os grupos tendem a ser limitados e de tamanho reduzidos, o tempo gasto na resolução do problema diminui consideravelmente em comparação com o algoritmo que verifica todas as combinações possíveis entre todas as cláusulas iniciais.

A seguir, apresentamos um exemplo para melhor entendimento da solução adotada.

Considere o seguinte conjunto de cláusulas de conflito geradas pelo resolvedor SAT aplicado a uma partição anterior e filtrado pela função **ExtraiConflitosCompartilhados**:

```
1 -3 -9
2 1 3
-2 5 8
1 -3 5 -7
1 -5 7
-1 9
-10
3 6 8 10
5 -7 2
6 4
3 -1
-11
8 -5 -2
-6 -4
```

A primeira parte do algoritmo é a identificação e remoção de cláusulas constituídas por mais de três literais:

```
1 -3 -9
2 1 3
-2 5 8
1 -5 7
-1 9
-10
5 -7 2
6 4
3 -1
-11
```

```
8 -5 -2
-6 -4
```

Observe que as cláusulas *1 -3 5 -7* e *3 6 8 10* foram removidas do conjunto. O próximo passo é remover as cláusulas unitárias:

```
1 -3 -9
2 1 3
-2 5 8
-5 1 7
-1 9
5 -7 2
6 4
3 -1
8 -5 -2
-6 -4
```

Foram identificadas duas cláusulas unitárias, *-10* e *-11*, que foram devidamente removidas. Sendo o conjunto atual constituído somente por cláusulas contendo dois ou três literais, tem início o processo de agrupamento. A primeira cláusula disponível é *1 -3 -9*. Como durante o agrupamento somente as variáveis são levadas em consideração, o primeiro grupo, denominado grupo 0, é formado pelas variáveis 1, 3 e 9 e a respectiva cláusula é adicionada ao mesmo:

```
Grupo 0: 1 3 9
         1 -3 -9
```

A segunda cláusula disponível é *2 1 3*, que possui as variáveis 1 e 3 em comum com o grupo 0. Portanto ela será adicionada a este grupo e também a um novo grupo criado com base em suas variáveis:

```
Grupo 0:  1  3  9          Grupo 1:  1  2  3
          1 -3 -9                    2  1  3
          2  1  3
```

A cláusula seguinte, *-2 5 8*, não possui nenhuma variável em comum com o grupo 0 mas possui uma variável em comum com o grupo 1: 2. Entretanto, ela não será adicionada a este grupo pois somente cláusulas contendo duas ou três variáveis compartilhadas são adicionadas. O grupo 2 será então criado para comportar as três variáveis:

```
Grupo 0:  1  3  9     Grupo 1:  1  2  3     Grupo 2:  2  5  8
          1 -3 -9               2  1  3              -2  5  8
          2  1  3
```

Com a cláusula *1 -5 7* ocorre o mesmo que com a anterior: há somente uma variável em comum com os outros grupos. Portanto, ela terá um grupo próprio:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8            1 -5  7
          2  1  3
```

A próxima cláusula, *-1 9*, possui suas duas variáveis em comum com a cláusula base do grupo 0, sendo adicionada a este grupo. Ela também será adicionada a um novo grupo pois, apesar de ter suas duas variáveis compartilhadas com o grupo 0, a cláusula base deste grupo possui três literais, sendo portanto, cláusulas de tamanhos diferentes:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8            1 -5  7
          2  1  3
         -1  9

Grupo 4:  1  9
         -1  9
```

Na seqüência, a cláusula *5 -7 2* é adicionada aos grupos 2, devido ao compartilhamento das variáveis 2 e 5, 3, devido as variáveis 5 e 7, e 5, grupo este criado para comparar as três variáveis da cláusula com as das cláusulas seguintes:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8            1 -5  7
          2  1  3                                5 -7  2            5 -7  2
         -1  9

Grupo 4:  1  9     Grupo 5:  2  5  7
         -1  9               5 -7  2
```

A cláusula *6 4* não possui nenhuma variável em comum com nenhum dos grupos já criados. Portanto basta criar um novo grupo para suas variáveis e adicionar a cláusula ao mesmo:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8            1 -5  7
          2  1  3                                5 -7  2            5 -7  2
         -1  9

Grupo 4:  1  9     Grupo 5:  2  5  7  Grupo 6:  6  4
         -1  9               5 -7  2            6  4
```

Ao executar o algoritmo para a cláusula *3 -1*, ela é adicionada aos grupos 0 e 1 pois estes compartilham duas de suas três variáveis com a cláusula analisada, e o grupo 7 é criado para comportar as variáveis 1 e 3:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8             1 -5  7
          2  1  3             3 -1                5 -7  2             5 -7  2
         -1  9
          3 -1

Grupo 4:  1  9     Grupo 5:  2  5  7  Grupo 6:  6  4     Grupo 7:  1  3
         -1  9               5 -7  2             6  4               3 -1
```

As três variáveis da penúltima cláusula, *8 -5 -2*, são as mesmas que compõem o grupo 2. Portanto ela é adicionada a este grupo, além dos grupos 3 e 5, devido ao compartilhamento de 2 variáveis:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8             1 -5  7
          2  1  3             3 -1                5 -7  2             5 -7  2
         -1  9                                    8 -5 -2             8 -5 -2
          3 -1

Grupo 4:  1  9     Grupo 5:  2  5  7  Grupo 6:  6  4     Grupo 7:  1  3
         -1  9               5 -7  2             6  4               3 -1
                             8 -5 -2
```

Observe que como as variáveis da cláusula *8 -5 -2* são as mesmas do grupo 2, não houve necessidade de criação de um novo grupo.

A última cláusula somente possui variáveis compartilhadas com o grupo 6, mas como esse compartilhamento é total, ou seja, todas as variáveis de comparação do grupo e da cláusula são as mesmas, o número de grupos não é incrementado:

```
Grupo 0:  1  3  9  Grupo 1:  1  2  3  Grupo 2:  2  5  8  Grupo 3:  1  5  7
          1 -3 -9             2  1  3            -2  5  8             1 -5  7
          2  1  3             3 -1                5 -7  2             5 -7  2
         -1  9                                    8 -5 -2             8 -5 -2
          3 -1

Grupo 4:  1  9     Grupo 5:  2  5  7  Grupo 6:  6  4     Grupo 7:  1  3
         -1  9               5 -7  2             6  4               3 -1
                             8 -5 -2            -6 -4
```

Após a divisão de todas as cláusulas em grupos, basta remover os grupos que contém somente uma cláusula ou aqueles formados por duas cláusulas sendo ao menos uma delas constituída por três literais. Logo, os grupos 1, 4 e 7 devem ser removidos:

```
Grupo 0:  1  3  9      Grupo 2:  2  5  8      Grupo 3:  1  5  7
          1 -3 -9                -2  5  8                1 -5  7
          2  1  3                 5 -7  2                5 -7  2
         -1  9                    8 -5 -2                8 -5 -2
          3 -1

Grupo 5:  2  5  7      Grupo 6:  6  4
          5 -7  2                6  4
          8 -5 -2               -6 -4
```

Com o número reduzido de grupos, basta procurar dentro de cada grupo se existem cláusulas que formam expressões booleanas. Observe que nesta etapa os literais devem ser levados em consideração.

Com o auxílio da tabela 4.1, podemos identificar as expressões. No grupo 0, as cláusulas *1 -3 -9*, *-1 9* e *3 -1* formam uma função AND e portanto serão adicionadas à próxima partição. Os grupos 2, 3 e 5 não possuem cláusulas que constituem funções booleanas. Já no grupo 6, uma função NOT aparece devido às cláusulas *6 4* e *-6 -4*. Estas duas cláusulas também integrarão o resultado do exemplo, que pode ser visto a seguir:

```
 1 -3 -9
-1  9
 3 -1
 6  4
-6 -4
```

4.6.4 Reaproveitamento de Cláusulas de Conflito Unitárias ou que Constituem Funções Booleanas

A técnica de reaproveitamento de cláusulas de conflito unitárias ou que constituem funções booleanas é uma extensão da técnica apresentada na subseção 4.6.3. Ela consiste em reaproveitar as mesmas cláusulas da metodologia anterior adicionalmente a todas as cláusulas de conflito unitárias, geradas na resolução da partição prévia que sejam compartilhadas com a partição atual. Estas cláusulas eram anteriormente excluídas, já que nenhuma das funções booleanas de interesse era formada por cláusulas de um literal.

Esta alteração baseia-se no princípio de que cláusulas menores tendem a ser mais úteis durante o processo de resolução de um problema de satisfabilidade [Arora e Hsiao, 2003]. Como as menores cláusulas são justamente as cláusulas unitárias, a idéia consiste em também reaproveitá-las na tentativa de reduzir o espaço de busca para o resolvedor SAT.

Partindo da técnica de reaproveitamento de cláusulas de conflito que constituem funções booleanas, a implementação torna-se simples: a função **RemoveCláusulasUnitárias** do algoritmo 4.4 deve ser alterada para não descartar as cláusulas unitárias, mas somente removê-las das cláusulas a serem agrupadas, salvando-as então em uma estrutura de dados capaz de ser

acessada ao final da técnica a fim de serem concatenadas à solução final. Sendo assim, a função **RemoveCláusulasUnitárias**, que antes recebia somente o parâmetro $\phi_{conflitos}$, deve receber também um novo parâmetro para o armazenamento das cláusulas unitárias: $\phi_{conflitosUni}$. O conteúdo desta nova estrutura será concatenado com as cláusulas que impliquem funções booleanas.

É importante ressaltar que esta alteração na técnica anterior não altera a sua complexidade, pois as operações de salvar cláusulas que anteriormente eram descartadas e concatenação, são constantes com relação ao número de cláusulas ou variáveis do problema.

Retomando o exemplo dado na subseção 4.6.3, logo após a remoção das cláusulas com mais de quatro literais, as seguintes cláusulas são submetidas à nova função **RemoveCláusulasUnitárias**:

```
1 -3 -9
2 1 3
-2 5 8
1 -5 7
-1 9
-10
5 -7 2
6 4
3 -1
-11
8 -5 -2
-6 -4
```

As duas cláusulas unitárias *-10* e *-11* são identificadas e serão removidas deste conjunto mas serão salvas a parte, para acesso futuro. Sendo assim, a função retorna o mesmo conjunto da técnica anterior:

```
1 -3 -9
2 1 3
-2 5 8
1 -5 7
-1 9
5 -7 2
6 4
3 -1
8 -5 -2
-6 -4
```

Portanto, as expressões booleanas identificadas serão as mesmas:

```
1 -3 -9
```

```
-1  9
 3 -1
 6  4
-6 -4
```

Ao conjunto anterior, basta concatenar as duas cláusulas unitárias, sendo este o resultado para o nosso exemplo:

```
 1 -3 -9
-1  9
 3 -1
 6  4
-6 -4
-10
-11
```

4.7 Concatenação das Cláusulas de Conflito e Cláusulas que Compõem a Partição

A função **ConcatenaFórmulas** é a mais simples de todas: somente concatena as cláusulas de conflito $\phi_{conflitos}$ selecionadas pela função **HeurísticaDeReaproveitamento** e as cláusulas ϕ que compõem a partição a ser analisada. Isto equivale a adicionar uma porta AND ao circuito, conectando as duas fórmulas CNF.

Esta simples operação é constante com relação ao tamanho do problema, possuindo portanto complexidade constante $O(1)$.

Capítulo 5

Resultados

> "Program testing can be used to show the presence of bugs, but never to show their absence!"
>
> Edsger W. Dijkstra

O objetivo deste capítulo é apresentar os resultados da execução dos algoritmos apresentados no capítulo 4. Todas as técnicas foram implementadas e executadas com um *benchmark* de circuitos adequado ao problema de CEC. Este capítulo encontra-se dividido em quatro seções. A primeira seção apresenta as informações básicas sobre os experimentos realizados, a segunda justifica a escolha do MiniSat como resolvedor SAT a ser utilizado e a terceira descreve o *benchmark* selecionado para execução dos testes. Por fim, a última seção expõe e analisa os resultados dos experimentos.

5.1 Características dos Experimentos

As técnicas apresentadas no capítulo 4 foram implementadas utilizando as linguagens C++ e Perl. As duas linguagens foram escolhidas devido à eficiência amplamente conhecida da primeira e à facilidade com a qual a segunda lida com strings. Os testes foram executados em uma máquina com processador Intel Pentium 4 de 2,8 GHz e 2 GB de memória RAM, rodando Fedora Core 7 [Fedora, 2008], uma das distribuições mais populares do sistema operacional Linux.

Os resultados foram obtidos utilizando-se o MiniSat como resolvedor SAT (vide seção 5.2) e circuitos gerados pelo gerador de *benchmarks* BenCGen (vide seção 5.3).

5.2 Resolvedor SAT Utilizado

Um dos ítens mais importantes para composição deste trabalho é a escolha do resolvedor SAT a ser utilizado no algoritmo 4.1, pois conforme descrito na seção 4.1, devido à complexidade do problema da satisfabilidade, de nada adianta um pré-processamento do circuito se o resolvedor

SAT não for eficiente. Para tal, foi feita uma pré-seleção com três resolvedores SAT do estado-da-arte bastante conhecidos: Berkmin [Goldberg e Novikov, 2002], Minisat [Eén e Sörensson, 2003] e Zchaff [Moskewicz et al., 2001]. O escolhido foi o Minisat pelos seguintes fatores:

- O código-fonte é aberto e gratuito, permitindo qualquer adaptação necessária pelo usuário.
- É mantido atualizado pelos autores. A última versão, 2.0, foi lançada no ano passado e até o final deste ano sairá a versão 2.1.
- A comunidade é bastante ativa, possuindo uma lista onde os próprios autores esclarecem dúvidas dos usuários.
- Tem sido bastante utilizado pela comunidade acadêmica.
- É destaque freqüente nas mais importantes competições de resolvedores SAT: SAT-Race [SAT'08, 2008], que utiliza apenas *benchmarks* industriais e SAT Competition [Competitions, 2007], que utiliza instâncias industriais, feitas a mão e aleatórias. Na SAT-Race deste ano, o MiniSat ficou em primeiro lugar em duas de três categorias. Na última SAT Competition, ficou entre os três primeiros colocados em duas das três categorias.

De acordo com [Andrade, 2008], o Berkmin é o resolvedor SAT mais adequado para tratar problemas de CEC. No entanto, este resolvedor encontra-se desatualizado (sua última atualização foi em 2003) e possui código-fonte fechado. Como o Berkmin não imprime as cláusulas de conflito por padrão, seria necessário acesso ao seu código para fazer esta alteração e utilizá-lo no presente trabalho. O Zchaff também não tem sido atualizado: com exceção de um pequeno conserto de *bug* em 2007, sua última versão com melhorias foi lançada em 2004. Após este ano, não obteve boa colocação nas competições entre resolvedores SAT.

Para o presente trabalho, algumas adaptações precisaram ser feitas no MiniSat, sendo elas:

- **Extração das cláusulas de conflito:** Por padrão, o MiniSat somente retorna SAT ou UNSAT para a fórmula submetida. Foi necessário alterar o código para armazenar todas as cláusulas de conflito geradas durante a resolução de cada problema.
- **Saída do miter sendo passada como *assumption*:** Para que o reaproveitamento das cláusulas de conflito seja possível em partições subseqüentes, é necessário que a cláusula unitária que representa a saída do miter seja passada como um *assumption* para o resolvedor SAT e não como uma cláusula concatenada à fórmula CNF que representa o circuito. Isto acontece porque adicionar a saída do miter ao circuito é o mesmo que adicionar uma porta AND ligando esta cláusula a todas as demais, forçando o valor da saída a ser sempre 1, o que pode macular as cláusulas de conflito geradas pelo resolvedor, já que estas estarão sob a influência direta desta cláusula. Portanto, o valor da saída do miter deve ser temporariamente igual a 1, para que as cláusulas sejam aprendidas a

partir do circuito original ou caso contrário, que contenham explicitamente o literal que representa a saída do miter.

5.3 Benchmark Utilizado

Segundo [Harlow, 2000], um *benchmark* para projeto de circuitos integrados é uma coleção de circuitos em um formato comum que pode ser utilizado para avaliar algoritmos e ferramentas dado o domínio de um problema. Portanto, este é um importante instrumento para verificação das técnicas descritas no capítulo 4.

Dentre os *benchmarks* de circuitos integrados mais populares, destacam-se o ISCAS 85 [Brglez e Fujiwara, 1985], ISCAS 89 [F. Brglez e Kozminski, 1989] e ITC 99 [Davidson, 1999]. Os dois primeiros são considerados ultrapassados, sendo rapidamente verificados por ferramentas atuais. O ITC 99 é limitado por possuir poucos circuitos e todos com tamanho fixo. Recentemente, uma ferramenta chamada BenCGen foi proposta por [Andrade et al., 2008], sendo capaz de gerar 24 tipos diferentes de circuitos com tamanhos variados. De acordo com os autores, é possível gerar mais de 1.000.000 de circuitos combinacionais diferentes. Devido a esta riqueza, optamos por utilizar *benchmarks* gerados pelo BenCGen.

5.3.1 Geração de Circuitos Dissimilares

Para gerar circuitos dissimilares de mesmo tamanho, cada circuito gerado no BenCGen foi duplicado, e os operandos da cópia foram invertidos, conforme mostrado na figura 5.1. O circuito C e sua cópia C' possuem 12 entradas e 12 saídas cada. Cada operando equivale a um conjunto de 6 bits consecutivos das portas de entrada. Portanto, para inverter os operandos de C', basta inverter os 6 bits menos significativos da entrada com os 6 bits mais significativos. Por exemplo, observe que a entrada I_1 do circuito C equivale a entrada I_7 de C', I_2 de C equivale a I_8 de C', e assim por diante, até I_6 de C que equivale a I_{12} de C'. A partir de então, I_7 de C equivale a I_1 de C' até I_{12} de C que equivale a I_6 de C'.

Esta inversão de operandos transforma os circuitos em dissimilares pelo mesmo princípio que ocorre quando há uma inversão dos operandos em uma multiplicação decimal, conforme mostrado em [Stoffel e Kunz, 2004]. A figura 5.2 mostra detalhadamente a multiplicação dos números 357 e 521. Observe que a multiplicação de cada termo que compõe o multiplicador por todo o multiplicando, gera somas intermediárias diferentes quando os operandos são invertidos. Entretanto, o produto permanece o mesmo.

5.3.2 Circuitos Gerados para Testes

Utilizando o BenCGen, circuitos de diferentes tamanhos e com alto grau de similaridades internas foram gerados para três tipos de multiplicadores: *Carry-Lookahead*, *Dadda Tree* e *Wallace Tree*. Os multiplicadores foram escolhidos pois são considerados uma das classes de circuitos combinacionais mais difíceis de serem verificadas, mesmo aqueles de poucos bits.

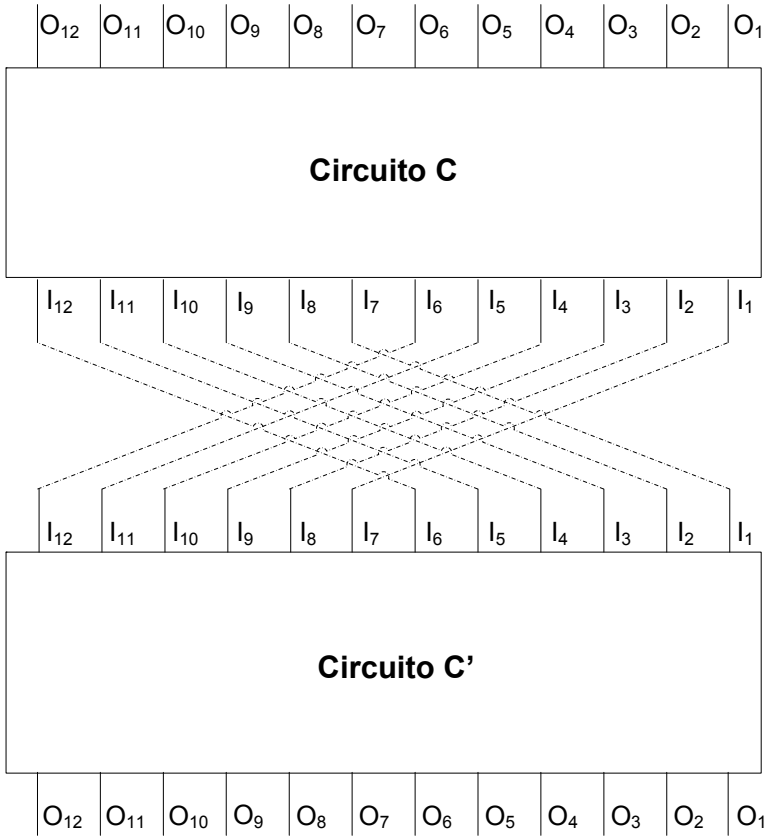


Figura 5.1: Circuito original (C) e sua cópia (C') com os operandos invertidos

```
   357          521
 x 521        x 357
 -----        -----
   357         3647
+  714       + 2605
 -----       -----
  1785        1563
 -----       -----
 185997      185997
```

Figura 5.2: Multiplicação decimal com troca de operandos

Foram gerados circuitos com entradas variando de 6 a 26 bits, ou seja, cada operando deve possuir o mesmo tamanho e este pode variar de 3 a 13 bits.

As tabelas 5.1, 5.2 e 5.3 apresentam as características para cada tipo de circuito gerado pelo BenCGen e utilizado nos testes. Cada uma delas possui três colunas, assim identificadas:

- **Tamanho:** Número de bits do circuito. Por exemplo, 6x6 significa que cada multiplicador possui 12 bits de saída e 12 de entrada, sendo que cada operando é representado por 6 bits. Portanto diz-se que este é um multiplicador de 6 bits.
- **Número de portas lógicas:** Número de portas lógicas da dupla de circuitos originais a serem verificados (excluindo-se o miter).
- **Número de cláusulas:** Número de cláusulas originais da dupla de circuitos a ser verificada (excluindo-se o miter). Este número depende da quantidade e do tipo de portas lógicas, que são convertidas em cláusulas CNF de acordo com a relação mostrada na tabela 4.1.

A tabela 5.1 apresenta as características dos multiplicadores *Carry-Lookahead* gerados pelo BenCGen. Cada elemento corresponde à dupla de circuitos a ser verificada. Por exemplo, os dois multiplicadores 10x10 possuem, juntos, 2056 portas lógicas e 6152 cláusulas CNF.

Tamanho	Número de Portas Lógicas	Número de Cláusulas
3x3	96	286
4x4	202	602
5x5	356	1062
6x6	564	1684
7x7	832	2486
8x8	1166	3486
9x9	1572	4702
10x10	2056	6152
11x11	2624	7854
12x12	3282	9826
13x13	4036	12086

Tabela 5.1: Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Carry-Lookahead* a serem verificados

A tabela 5.2 mostra as características dos multiplicadores *Dadda Tree*, gerados pelo BenCGen, a serem verificados.

Tamanho	Número de Portas Lógicas	Número de Cláusulas
3x3	120	386
4x4	214	690
5x5	340	1098
6x6	498	1610
7x7	688	2226
8x8	910	2946
9x9	1164	3770
10x10	1450	4698
11x11	1768	5730
12x12	2118	6866
13x13	2500	8106

Tabela 5.2: Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Dadda Tree* a serem verificados

Por fim, a tabela 5.3 introduz as características dos multiplicadores do tipo *Wallace Tree* a serem verificados.

Tamanho	Número de Portas Lógicas	Número de Cláusulas
3x3	138	442
4x4	268	864
5x5	440	1424
6x6	722	2350
7x7	990	3228
8x8	1314	4286
9x9	1698	5546
10x10	2092	6838
11x11	2540	8306
12x12	3050	9980
13x13	3592	11758

Tabela 5.3: Número de portas lógicas e cláusulas de cada dupla de multiplicadores *Wallace Tree* a serem verificados.

5.4 Resultados Experimentais

Com o intuito de avaliar o resultado das técnicas descritas no capítulo 4, testes experimentais foram conduzidos e os resultados registrados para análise. Cada dupla de multiplicadores apresentados nas tabelas 5.1, 5.2 e 5.3 da seção 5.3 foi executada por diversas vezes em oito implementações distintas do algoritmo 4.1. Cada versão possui alterações somente no que tange ao reaproveitamento das cláusulas de conflito, sendo algumas somente variações dos parâmetros das técnicas apresentadas na seção 4.6.

As tabelas 5.4, 5.5 e 5.6 apresentam, para cada dupla de circuitos verificados, o tempo de execução, em segundos, das implementações testadas. Para cada execução, foi estabelecido um limite máximo de tempo de 100000 segundos, equivalente a pouco mais de 27 horas.

Cada linha da tabela representa dois circuitos equivalentes de mesmo tamanho e as nove colunas são assim identificadas:

- **Tamanho:** Número de bits do circuito. Por exemplo, 8x8 significa que cada multiplicador possui 16 bits de saída e 16 de entrada, sendo que cada operando é representado por 8 bits. Portanto diz-se que este é um multiplicador de 8 bits.

- **Nenhuma:** Nenhuma cláusula de conflito é reaproveitada entre as partições adjacentes. É uma das abordagens mais tradicionais utilizada na verificação de equivalência com resolvedor SAT. Nesta versão, as funções **ExtraiConflitosCompartilhados**, **HeurísticaDeReaproveitamento** e **ConcatenaFórmulas** do algoritmo 4.1 são ignoradas, fazendo com que cada partição seja passada diretamente para o MiniSat.

- **Todas:** Todas as cláusulas de conflito geradas durante a resolução da partição anterior, são reaproveitadas na análise da partição atual. Os dados são referentes à execução da técnica descrita na subseção 4.6.1.

- **Até** 3 **Literais:** Somente as cláusulas de conflito compostas por até três literais serão reaproveitadas na partição seguinte. Os dados são referentes à execução da metodologia descrita na subseção 4.6.2, porém o algoritmo 4.3 recebe como parâmetro o inteiro positivo 3.

- **Até** 5 **Literais:** Somente as cláusulas de conflito compostas por até cinco literais serão reaproveitadas na partição seguinte. Os dados são referentes à execução da metodologia descrita na subseção 4.6.2, porém o algoritmo 4.3 recebe como parâmetro o inteiro positivo 5.

- **TCFB:** Somente as cláusulas de conflito que constituem as funções booleanas AND, $NAND$, OR, NOR, $BUFF$, NOT, XOR e $XNOR$ serão reaproveitadas. Os dados são relativos à implementação da técnica descrita na subseção 4.6.3.

- **C2FB:** Somente as cláusulas de conflito compostas por dois literais e que representam funções booleanas serão reaproveitadas na partição seguinte. Observe que as únicas funções booleanas totalmente representáveis por cláusulas constituídas de dois literais

são $BUFF$ e NOT, conforme mostrado na tabela 4.1. Esta técnica é uma especialização das descritas nas subseções 4.6.2 (com o algoritmo 4.3 recebendo como parâmetro o inteiro positivo 2) e 4.6.3.

- **TCFB+U:** Além das cláusulas de conflito que constituem as funções booleanas AND, $NAND$, OR, NOR, $BUFF$, NOT, XOR e $XNOR$, também serão reaproveitadas as cláusulas unitárias, conforme descrito na subseção 4.6.4.

- **C2FB+U:** Além das cláusulas de conflito compostas por dois literais e que representam funções booleanas, também serão reaproveitadas as cláusulas unitárias. Esta técnica é uma especialização das descritas nas subseções 4.6.2 (com o algoritmo 4.3 recebendo como parâmetro o inteiro positivo 2) e 4.6.4. Portanto, o conjunto de cláusulas reaproveitáveis será constituído por cláusulas unitárias e expressões que formam $BUFF$, NOT.

Um símbolo do tipo '−' aparece na tabela em uma das seguinte situações:

- O limite de tempo definido foi ultrapassado.
- Um *Segmentation Fault* ocorreu durante a execução do resolvedor SAT.
- A quantidade de memória disponível não foi suficiente.
- Os demais recursos da máquina, como por exemplo capacidade de processamento, não foram suficientes.

Os dois últimos ítens são esperados que ocorram a partir de um certo tamanho de multiplicadores, tendo em vista a natureza da classe $\mathcal{NP}$-Completo na qual o problema está inserido. O nosso objetivo é resolver corretamente problemas cada vez maiores no menor espaço de tempo possível. Sobre a possibilidade de ocorrência de um *Segmentation Fault*, é importante ressaltar que da forma que os algoritmos foram implementados, o resolvedor SAT será o primeiro a retornar tal erro. Por exemplo, houve casos em que foram geradas mais de quatro milhões de cláusulas de conflito em uma partição e todas foram reaproveitadas na partição seguinte, sobrecarregando o resolvedor.

Cada tabela apresenta os resultados para um tipo de multiplicador diferente. A tabela 5.4 refere-se a multiplicadores *Carry-Lookahead*, a tabela 5.5 refere-se a multiplicadores *Dadda Tree* e por fim, a tabela 5.6 apresenta os resultados para multiplicadores *Wallace Tree*. Ao testar três tipos diferentes de multiplicadores, temos dois objetivos:

1. Verificar se os circuitos foram corretamente gerados pelo BenCGen. Todos os trabalhos anteriores que utilizavam o BenCGen eram de autoria dos próprios autores do gerador, portanto este é o primeiro trabalho independente que utiliza os circuitos gerados pelo mesmo.

2. Ao testar somente um tipo de multiplicador, este pode ter algum diferencial em sua estrutura que favoreça ou deprecie o reaproveitamento das cláusulas de conflito. Para

evitar qualquer tipo de vício, optamos por utilizar três dos cinco tipos de multiplicadores gerados pelo BenCGen.

Quanto ao primeiro objetivo, os circuitos mostram-se corretos com relação ao resultado de saída, pois conforme o esperado, os multiplicadores provaram ser equivalentes para todos os testes executados que finalizaram com sucesso. O segundo objetivo também foi alcançado, pois os resultados obtidos mostraram possuir as mesmas características independentemente do tipo de circuito. Observe que isto não significa que os tempos foram semelhantes, mas sim que é possível analisar os dados de todas as tabelas em conjunto e extrair informações consistentes e relevantes.

As tabelas 5.7, 5.8 e 5.9 apresentam o número total de cláusulas de conflito reaproveitadas em cada técnica para multiplicadores do tipo *Carry-Lookahead*, *Dadda Tree* e *Wallace Tree*, respectivamente. A coluna referente à abordagem tradicional terá sempre o número 0, pois esta não reaproveita nenhuma cláusula de conflito.

Analisando as tabelas que contemplam o tempo de execução, para circuitos menores (de até 6 bits), o não reaproveitamento das cláusulas de conflito é mais vantajoso devido à simplicidade da implementação, que requer um menor poder de processamento. Entretanto, esta vantagem refere-se a décimos de segundo, o que, na prática, não faz diferença para o usuário. Tal variação no tempo de execução continua pequena para circuitos de 7, 8 e 9 bits, com exceção da técnica que reaproveita todas as cláusulas de conflito, cujo tempo cresce em um ritmo bem mais acelerado que os demais, inclusive estourando o limite estabelecido antes mesmo que a abordagem tradicional. Este comportamento pode ser explicado com o auxílio das tabelas que mostram o número de cláusulas de conflito reaproveitadas. Observe a tabela 5.7, referente a um multiplicador *Carry-Lookahead*. O número de cláusulas reaproveitadas por esta técnica aumenta consideravelmente, mas não proporcionalmente, a medida que o circuito aumenta. Por exemplo, a verificação de um multiplicador 9x9, reaproveitando todas as cláusulas de conflito, leva em média 20 minutos e reaproveita mais de 200.000 cláusulas. Para as demais técnicas, o tempo de execução é de cerca de 5 minutos (4 vezes menor), e o reaproveitamento é de no máximo 16.141 cláusulas (mais de 12 vezes menor). Neste caso, o grande número de cláusulas influencia negativamente no desempenho do resolvedor SAT, pois estas não são boas o suficiente para solucionar o problema de forma eficiente. É possível argumentar que, caso a quantidade de variáveis seja mantida, quanto maior o número de cláusulas de um problema de SAT, mais fácil será resolvê-lo devido ao menor número de assinalamentos possíveis. Entretanto, é preciso levar em consideração características como o número de literais das cláusulas e como estas são incorporadas pelo resolvedor SAT.

Para os multiplicadores de tamanho 10x10 em diante, as técnicas que reaproveitam as cláusulas de conflito com até dois literais que formam funções booleanas, com e sem cláusulas unitárias, mostraram-se mais eficientes. A razão desta diminuição do tempo deve-se mais ao fato da seleção de cláusulas de tamanho reduzido do que ao fato das cláusulas comporem as funções de equivalência e equivalência negada. Isto corrobora o descrito em [Arora e Hsiao, 2003], que afirma que as cláusulas de conflito menores geralmente são mais úteis.

A afirmação anterior de que o reaproveitamento das cláusulas de conflito que representam

funções booleanas tem pouca influência na melhora do desempenho do algoritmo, é comprovada pelo desempenho referente às execuções com o reaproveitamento de todas as funções booleanas, independente da adição de cláusulas unitárias. Em nenhum teste estas técnicas apresentaram melhor tempo de execução que as demais. O motivo novamente pode estar relacionado com o tamanho das cláusulas de conflito: uma cláusula com três literais pode não ser tão útil quanto se esperava. Outro possível motivo para o fraco desempenho desta implementação reside no fato das cláusulas selecionadas não comporem o conjunto ideal para o resolvedor SAT, o que não necessariamente está relacionado com os tamanhos das mesmas. Exemplificando, a introdução de certas cláusulas podem eliminar alguma cláusula chave para que uma contradição seja imediatamente identificada ou, em outras palavras, o espaço de busca está sendo cortado onde não seria recomendável fazê-lo.

Apesar da visível melhora no tempo de execução obtida com as técnicas que reaproveitam as cláusulas de conflito com até dois literais e que formam funções booleanas, as duas técnicas verificam a equivalência de, no máximo, multiplicadores 12x12 no tempo estipulado, enquanto a abordagem tradicional verifica somente até 11x11. Portanto, existe uma pequena melhora de desempenho quando se reaproveita determinadas cláusulas de conflito.

Com relação à utilização da memória, esta não variou de acordo com a técnica utilizada. Os gráficos ilustrados nas figuras 5.3, 5.4 e 5.5 mostram que, para multiplicadores *Carry-Lookahead*, *Dadda Tree* e *Wallace Tree* respectivamente, o espaço de memória utilizado tanto para a abordagem que não reaproveita cláusulas de conflito quanto para a abordagem com melhor tempo para aquela classe de multiplicadores, é praticamente a mesma.

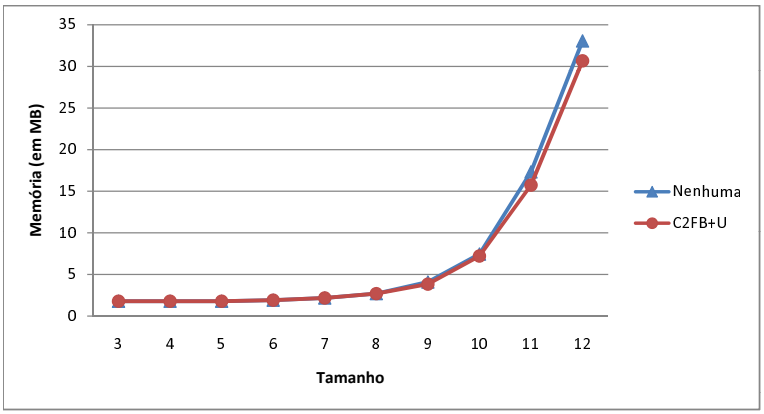


Figura 5.3: Gráfico da utilização da memória para multiplicadores *Carry-Lookahead*.

Tamanho	Nenhuma	Todas	Até 3 Literais	Até 5 Literais	TCFB	C2FB	TCFB+U	C2FB+U
3x3	**0,1**	0,2	0,2	0,2	0,3	0,3	0,3	0,3
4x4	**0,2**	0,3	0,3	0,3	0,4	0,4	0,4	0,4
5x5	**0,3**	0,6	0,5	0,5	0,6	0,6	0,6	0,6
6x6	**1,3**	1,9	1,4	1,4	1,8	1,5	1,9	1,6
7x7	6,1	10,7	**5,6**	6,2	7,9	6,1	7,6	6,0
8x8	47,1	126,9	33,3	**32,2**	50,1	38,8	50,4	39,1
9x9	296,7	1212,1	298,5	309,1	378,0	303,1	333,6	**275,0**
10x10	2415,9	15290,8	2140,1	2628,5	2350,5	**2059,5**	2584,3	2248,6
11x11	20350,1	-	19573,5	14887,9	17984,9	16185,5	16454,9	**14019,5**
12x12	-	-	-	-	-	-	-	**95484,8**
13x13	-	-	-	-	-	-	-	-

Tabela 5.4: Tempo de execução, em segundos, para multiplicadores *Carry-Lookahead* de tamanhos variados

Tamanho	Nenhuma	Todas	Até 3 Literais	Até 5 Literais	TCFB	C2FB	TCFB+U	C2FB+U
3x3	**0,1**	0,2	0,2	0,2	0,2	0,2	0,2	0,2
4x4	**0,2**	0,3	0,3	0,3	0,4	0,4	0,4	0,4
5x5	**0,3**	0,5	0,5	0,5	0,5	0,5	0,5	0,5
6x6	**0,9**	1,4	1,0	1,0	1,3	1,2	1,3	1,2
7x7	4,5	5,3	3,4	**3,3**	4,6	3,8	4,3	3,6
8x8	17,8	52,9	**14,9**	18,1	21,7	15,3	20,7	17,5
9x9	111,2	540,5	86,5	102,7	123,3	**76,1**	115,5	89,7
10x10	694,6	-	667,9	1036,5	820,1	672,3	719,1	**559,7**
11x11	4251,3	-	4580,8	6341,5	3907,2	**2914,2**	3751,4	3340,6
12x12	-	-	26296,4	41040,0	21533,7	**19006,6**	19302,2	19864,7
13x13	-	-	-	-	-	-	-	-

Tabela 5.5: Tempo de execução, em segundos, para multiplicadores *Dadda Tree* de tamanhos variados

Tamanho	Nenhuma	Todas	Até 3 Literais	Até 5 Literais	TCFB	C2FB	TCFB+U	C2FB+U
3x3	**0,1**	0,2	0,2	0,2	0,3	0,3	0,3	0,3
4x4	**0,2**	0,3	0,3	0,3	0,4	0,4	0,4	0,4
5x5	**0,4**	0,6	0,5	0,5	0,6	0,6	0,6	0,6
6x6	**1,0**	1,6	1,1	1,2	1,3	1,3	1,4	1,3
7x7	4,9	11,1	**4,5**	5,1	5,3	4,8	5,3	4,8
8x8	30,8	118,9	25,6	27,9	30,5	29,1	28,9	**24,6**
9x9	153,7	699,0	127,9	176,9	145,6	150,5	152,4	**126,6**
10x10	1494,0	17967,0	1555,2	2843,7	1661,2	1430,2	1449,5	**1328,2**
11x11	9006,9	-	10242,2	19657,1	9481,1	8207,1	8115,1	**7772,2**
12x12	-	-	51187,6	86427,9	38648,1	37300,9	33379,1	**32307,7**
13x13	-	-	-	-	-	-	-	-

Tabela 5.6: Tempo de execução, em segundos, para multiplicadores *Wallace Tree* de tamanhos variados

Tamanho	**Nenhuma**	**Todas**	**Até 3 Literais**	**Até 5 Literais**	**TCFB**	**C2FB**	**TCFB+U**	**C2FB+U**
3x3	0	25	19	24	4	4	6	6
4x4	0	179	58	101	6	6	8	8
5x5	0	1.011	195	390	18	18	22	22
6x6	0	5.008	559	1.178	34	34	38	38
7x7	0	18.022	1.252	3.017	44	44	51	51
8x8	0	69.725	2.916	6.434	74	74	84	84
9x9	0	220.848	7.632	16.141	150	150	158	158
10x10	0	835.207	15.922	34.990	240	240	267	267
11x11	0	-	36.373	67.707	386	386	404	404
12x12	-	-	-	-	-	-	-	624
13x13	-	-	-	-	-	-	-	-

Tabela 5.7: Número de cláusulas de conflito reaproveitadas para multiplicadores *Carry Lookahead* de tamanhos variados

Tamanho	Nenhuma	Todas	Até 3 Literais	Até 5 Literais	TCFB	C2FB	TCFB+U	C2FB+U
3x3	0	26	12	25	0	0	5	5
4x4	0	81	35	60	4	4	9	9
5x5	0	550	116	202	12	12	17	17
6x6	0	2.758	318	640	44	44	59	59
7x7	0	9.211	895	2.025	69	84	86	86
8x8	0	38.368	2.040	5.529	180	170	211	191
9x9	0	142.267	4.795	12.356	395	322	417	370
10x10	0	-	10.726	28.156	653	516	672	544
11x11	0	-	21.688	54.798	1.070	810	1053	844
12x12	-	-	41.800	115.624	1.603	1.222	1.609	1.225
13x13	-	-	-	-	-	-	-	-

Tabela 5.8: Número de cláusulas de conflito reaproveitadas para multiplicadores *Dadda Tree* de tamanhos variados

Tamanho	**Nenhuma**	**Todas**	**Até 3 Literais**	**Até 5 Literais**	**TCFB**	**C2FB**	**TCFB+U**	**C2FB+U**
3x3	0	38	19	32	2	2	8	8
4x4	0	355	52	135	0	0	10	10
5x5	0	1.246	98	347	2	2	13	13
6x6	0	4.354	345	927	18	18	42	42
7x7	0	17.010	855	2.028	66	66	77	77
8x8	0	61.724	1.919	5.258	148	132	172	137
9x9	0	162.520	4.294	12.765	281	262	333	311
10x10	0	817.136	9.900	32.801	463	434	510	462
11x11	0	-	20.342	62.290	817	698	859	697
12x12	0	-	41.385	127.180	1.249	1.048	1.300	1.120
13x13	-	-	-	-	-	-	-	-

Tabela 5.9: Número de cláusulas de conflito reaproveitadas para multiplicadores *Wallace Tree* de tamanhos variados.

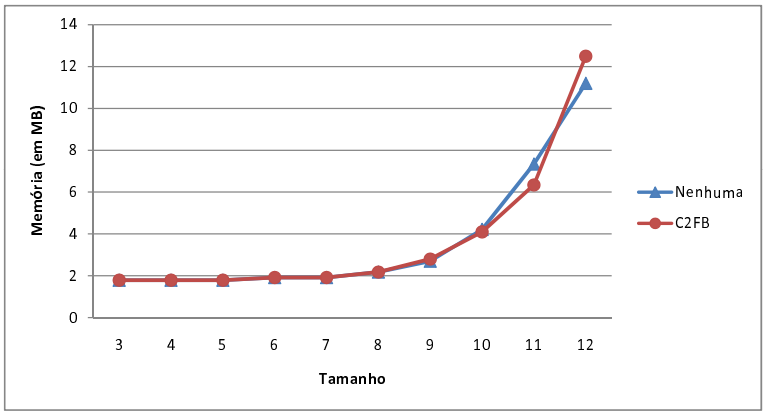


Figura 5.4: Gráfico da utilização da memória para multiplicadores *Dadda Tree*.

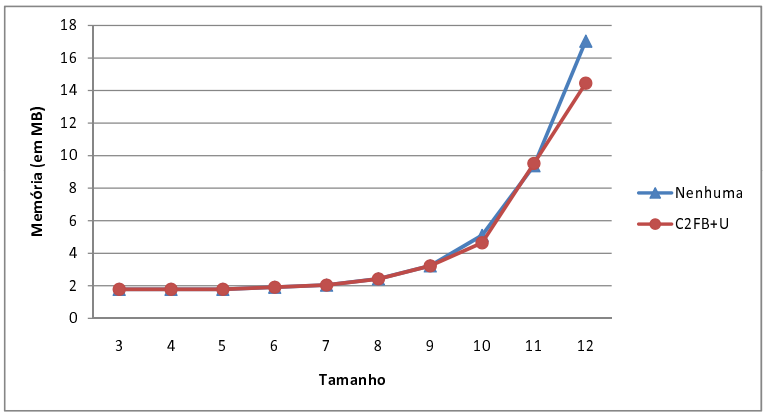


Figura 5.5: Gráfico da utilização da memória para multiplicadores *Wallace Tree*.

Capítulo 6

Conclusão

> "Programming is similar to a game of golf. The point is not getting the ball in the hole but how many strokes it takes."
>
> Harlan Mills

O presente trabalho apresentou e analisou metodologias para o reaproveitamento das cláusulas de conflito entre partições durante a verificação de equivalência entre dois circuitos combinacionais dissimilares particionados. Inicialmente, os circuitos são particionados de forma que cada divisão contenha somente uma porta de saída. Um miter é construído entre os dois circuitos e as partições são submetidas seqüencialmente ao resolvedor SAT. Além de retornar se o problema é satisfazível ou não, o resolvedor deve armazenar todas as cláusulas de conflito geradas durante a resolução da partição atual para o reaproveitamento durante a análise da partição seguinte. A diferença entre as metodologias está na seleção de quais cláusulas de conflito serão reaproveitadas.

Ao reaproveitar as cláusulas de conflito entre partições adjacentes, o objetivo é restringir o espaço de solução, na tentativa de diminuir o trabalho a ser efetuado pelo resolvedor SAT, proporcionando uma melhora no tempo de execução e no consumo de recursos computacionais, permitindo a análise de circuitos cada vez maiores.

As metodologias foram implementadas e verificadas com diferentes tipos de multiplicadores gerados por um recente gerador de *benchmarks* denominado BenCGen. Multiplicadores são particularmente interessantes para CEC pois estão entre as classes de circuitos mais difíceis de serem verificados, mesmo aqueles com poucos bits.

Com base nos resultados experimentais, pode-se afirmar que o reaproveitamento de cláusulas de conflito entre partições nem sempre é uma boa solução. Verificou-se que nem todas as cláusulas são úteis, algumas podem até mesmo piorar o desempenho do resolvedor SAT. Em termos de memória, não há diferença entre as abordagens utilizadas. De todas as técnicas analisadas, as que reaproveitam as cláusulas de conflito com até dois literais que formam funções booleanas, com e sem cláusulas unitárias, mostraram-se mais eficientes em termos de tempo de execução para circuitos maiores. Entretanto, tal eficiência é limitada pois, no

tempo estipulado, enquanto uma abordagem sem o reaproveitamento consegue verificar multiplicadores de até 11 bits, as técnicas citadas verificam somente circuitos com até 12 bits. Este ganho está aquém do esperado, pois há na literatura recente técnicas capazes de verificar multiplicadores com até 32 bits, porém utilizando outras abordagens, como derivação de implicações a partir do miter construído [Andrade, 2008].

Embora não apresente nenhuma nova metodologia que supere os resultados do estado-da-arte de CEC para circuitos dissimilares, este trabalho possui diversas contribuições para a área. Não há ferramentas públicas disponíveis na Internet para particionamento de circuitos no formato ISCAS por TFI e para conversão de circuitos do formato ISCAS para CNF; as duas são bastante úteis não só para problemas de CEC mas para toda a área de projeto de circuitos integrados. As ferramentas geradas neste trabalho estarão disponíveis, juntamente com seu código-fonte, para que o usuário possa alterá-las de acordo com suas necessidades. Este trabalho também auxilia na verificação dos *benchmarks* gerados pelo BenCGen, pois é o primeiro que o utiliza sem ser dos mesmos autores do gerador. Outra importante contribuição é a conclusão de que mesmo os circuitos possuindo similaridades internas, o reaproveitamento das cláusulas de conflito não necessariamente implica em melhora de desempenho na resolução das partições. Observe que as técnicas apresentadas podem ser facilmente adaptadas ou integradas a outras técnicas, além de serem independentes do resolvedor SAT, desde que este retorne as cláusulas de conflito. Além disso, as técnicas apresentadas funcionam de maneira similar, independente dos circuitos serem ou não equivalentes; algumas técnicas tendem a apresentar melhores resultados para somente uma das situações. Os BDDs, por exemplo, são mais eficientes quando os circuitos são equivalentes.

Para trabalhos futuros, recomenda-se analisar o impacto da substituição do MiniSat por outros resolvedores SAT, desde que estes sejam capazes de retornar as cláusulas de conflito. Outra possibilidade é a proposta de novas técnicas para seleção das cláusulas de conflito a serem reaproveitadas, em conjunto com outros métodos de análise das partições como ordenação das mesmas ou verificação distribuída em thread separadas. Por último, sugere-se a análise das técnicas aqui descritas em *benchmarks* de circuitos combinacionais similares. Para tal, sugerimos o uso do BenCGen na geração os circuitos.

Apêndice A

Algoritmo para busca em largura em um grafo

A busca em largura ou BFS (do inglês *breadth-first search*), apresentada no algoritmo A.1, consiste em descobrir todos os vértices a partir de um determinado vértice, percorrendo todos à uma distância x antes de qualquer vértice à uma distância $x+1$. A versão aqui apresentada baseia-se na disponível em [Cormen et al., 2001].

Algoritmo A.1 Algoritmo para busca em largura (BFS)

Entrada: Um grafo G e um vértice de origem $i \in G$.
Saída: Um grafo com todos os vértices de G alcançáveis a partir de i.

```
1: Fila Q ← ∅
2: Vértice v, u
3: Cor cor
4: Grafo F
5: for all u ∈ ListaDeVértices(G) − i do
6:    cor[u] ← BRANCO
7: end for
8: cor[i] ← CINZA
9: Enfileira(Q, i)
10: while Q ≠ ∅ do
11:    u ← Desenfileira(Q)
12:    for all v ← ListaDeAdjacentes(u) do
13:       if cor[v] = BRANCO then
14:          cor[v] ← CINZA
15:          Enfileira(Q, v)
16:       end if
17:    end for
18:    cor[u] ← PRETO
19: end while
20: F ← GrafoDeVérticesPretos(G)
21: return F
```

O algoritmo recebe como entrada o grafo G a ser percorrido e o vértice de origem i. Inicialmente todos os vértices são coloridos de branco, o que significa que ainda não foram visitados. O vértice i é então pintado de cinza, pois este é descoberto com o início do algoritmo, e incluído na fila Q. A função desta fila é armazenar os vértices cinza, pois esta cor significa que o vértice foi descoberto mas sua lista de adjacência não foi totalmente percorrida. A partir de então, enquanto a fila não estiver vazia, um vértice u é desenfileirado e para cada vértice v de sua lista de adjacentes que ainda não tenha sido visitado, este é colorido de cinza e enfileirado. Estando todos os vértices da lista de adjacentes marcados, o vértice u é então colorido de preto. Por fim, o algoritmo retorna um grafo $F \subset G$.

Referências Bibliográficas

Akers, S. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, 27:509 – 518.

Andrade, F. V. (2008). *Contribuições para o problema de verificação de equivalência de circuitos combinacionais*. PhD thesis, Departmento de Ciência da Computação, Universidade Federal de Minas Gerais.

Andrade, F. V.; Oliveira, M. C. M. e Fernandes, A. O. (2007). SAT-based equivalence checking based on circuit partitioning and special approaches for conflict clause reuse. In *Proceedings of IEEE Workshop on Design and Diagnosis of Electronic Circuits and Systems*.

Andrade, F. V.; Silva, L. M. e Fernandes, A. O. (2008). BenCGen: a digital circuit generation tool for benchmarks. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pp. 164–169, New York, NY, USA. ACM.

Arora, R. e Hsiao, M. S. (2003). Enhancing SAT-based equivalence checking with static logic implications. In *Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop*, pp. 63–68, Washington, DC, USA. IEEE Computer Society.

Arora, R. e Hsiao, M. S. (2004). Using global structural relationships of signals to accelerate SAT-based combinational equivalence checking. *Journal of Universal Computer Science*, 10(12):1597–1628.

Bacchus, F. e Winter, J. (2004). Effective preprocessing with hyper-resolution and equality reduction. In *In Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT*, pp. 321–322. Springer.

Beizer, B. (1995). The pentium bug - an industry watershed. Testing Techniques Newsletter, TTN, On-line edition.

Bentley, B.; Baty, K.; Normoyle, K.; Ishii, M. e Yogev, E. (2004). Verification: What works and what doesn't. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 274–274.

Bhattacharya, D. e Hayes, J. P. (1990). *Hierarchical Modeling for VLSI Circuit Testing*. Springer.

Brand, D. (1993). Verification of large synthesized designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 534–537, Los Alamitos, CA, USA. IEEE Computer Society Press.

Brglez, F. e Fujiwara, H. (1985). A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings of the International Symposium on Circuits and Systems*, pp. 663–698.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

Bryant, R. E. (1991). On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213.

Bryant, R. E. e Chen, Y.-A. (1995). Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 535–541, New York, NY, USA. ACM Press.

Burch, J. R. (1991). Using BDDs to verify multipliers. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 408–412, New York, NY, USA. ACM Press.

Burch, J. R. e Singhal, V. (1998). Tight integration of combinational verification methods. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 570–576, New York, NY, USA. ACM Press.

C. A. J. van Eijk e G. L. J. M. Janssen (1994). Exploiting structural similarities in a BDD-based verification method. In *Proceedings of the International Conference on Theorem Provers in Circuit Design*, volume 901, pp. 110–125. Springer-Verlag.

Chen, J.-C. e Chen, Y.-A. (2001). Equivalence checking of integer multipliers. In *Proceedings of the Asia South Pacific Design Automation Conference*, pp. 169–174, New York, NY, USA. ACM Press.

Clarke, E. M.; Grumberg, O. e Peled, D. A. (1999). *Model Checking*. The MIT Press.

Competitions, S. (2007). The international sat competitions web page. Disponível em: `http://www.satcompetition.org/`, Último acesso: 03/11/2008.

Cook, S. A. (1971). The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pp. 151–158.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. e Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, second edição.

Corno, F.; Reorda, M. S. e Squillero, G. (2000). RT-Level ITC 99 benchmarks and first ATPG results. *IEEE Design and Test of Computers*, pp. 44–53.

Damiano, R. e Kukula, J. (2003). Checking satisfiability of a conjunction of BDDs. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 818–823, New York, NY, USA. ACM Press.

Davidson, S. (1999). ITC 99 benchmark homepage. Disponível em: `http://www.cerc.utexas.edu/itc99-benchmarks/bench.html`, Último acesso: 03/11/2008.

Davis, M.; Logemann, G. e Loveland, D. (1962). A machine program for theorem proving. *Communications of the Association for Computer Machinery*, 5(7):394–397.

Davis, M. e Putnam, H. (1960). A computation procedure for quantification theory. *Journal of the Association for Computer Machinery*, 7:201–215.

de Oliveira, M. C. M. (2006). Um núcleo inteligente para processamento distribuído de resolvedores SAT em verificação por equivalência. Master's thesis, Departmento de Ciência da Computação, Universidade Federal de Minas Gerais.

Disch, S. e Scholl, C. (2007). Combinational equivalence checking using incremental SAT solving, output ordering, and resets. In *Proceedings of the conference on Asia South Pacific design automation*, pp. 938–943.

Drechsler, R. (2004). *Advanced Formal Verification*. Kluwer Academic Publishers.

Eén, N. e Sörensson, N. (2003). An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 840–843.

F. Brglez, D. B. e Kozminski, K. (1989). Combinational problems of sequential benchmark circuits. In *Proceedings of the International Symposium on Circuits and Systems*, pp. 1929–1934.

Fedora (2008). Fedora project. Disponível em: `http://fedoraproject.org/`, Último acesso: 02/11/2008.

Foster, H.; Krolnik, A. C. e Lacey, D. J. (2004). *Assertion-Based Design*. Kluwer Academic Publishers.

Fujita, M.; Fujisawa, H. e Kawato, N. (1988). Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 2–5.

Fujiwara, H. e Shimono, T. (1983). On the acceleration of test generation algorithms. *IEEE Trans. Comput.*, 32(12):1137–1144.

G. S. Tseitin (1968). On the complexity of derivations in propositional calculus. In *Structures in Constructive Mathematics and Mathematical Logic Part II*, pp. 115–125. A. O. Slisenko Ed.

Gajski, D. D.; Dutt, N. D.; Wu, A. C.-H. e Lin, S. Y.-L. (1992). *High-Level Synthesis: Introduction to Chip and System Design*. Springer, first edição.

Garey, M. R. e Johnson, D. S. (1979). *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Goldberg, E. (2004). Equivalence checking of dissimilar circuits II. Technical Report CDNL-TR-2004-0830, Cadence.

Goldberg, E. e Novikov, Y. (2002). Berkmin: a fast and robust SAT-solver. In *In Design, Automation, and Test in Europe (DATE 02)*, pp. 142–149.

Goldberg, E. e Novikov, Y. (2003). Equivalence checking of dissimilar circuits. In *Workshop Notes of The International Workshop on Logic Synthesis*.

Goldberg, E.; Prasad, M. e Brayton, R. (2001). Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 114–121, Piscataway, NJ, USA. IEEE Press.

Groote, J. F. e Zantema, H. (2002). Resolution and binary decision diagrams cannot simulate each other polynomially. *Lecture Notes in Computer Science*, 2244:33–38.

Gupta, A. e Ashar, P. (1998). Integrating a boolean satisfiability checker and BDDs for combinational equivalence checking. In *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, p. 222, Washington, DC, USA. IEEE Computer Society.

Hamaguchi, K.; Morita, A. e Yajima, S. (1995). Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 78–82, Washington, DC, USA. IEEE Computer Society.

Harlow, J. E. (2000). Overview of popular benchmark sets. *IEEE Des. Test*, 17(3):15–17.

Hooker, J. N. (1993). Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15.

Jain, J.; Yan, A. N.; Fujita, M. e Sangiovanni-Vincentelli, A. (1997). A survey of techniques for formal verification of combinational circuits. In *Proceedings of the IEEE/ACM International Conference on Computer Design*, p. 445, Washington, DC, USA. IEEE Computer Society.

Jeroslow, R. e Wang, J. (1990). Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, pp. 167–187.

Katz, R. H. e Borriello, G. (2005). *Contemporary Logic Design*. Pearson Prentice Hall, second edição.

Keim, M.; Drechsler, R.; Becker, B.; Martin, M. e Molitor, P. (2003). Polynomial formal verification of multipliers. *Formal Methods in System Design*, 22(1):39–58.

Kirkland, T. e Mercer, M. R. (1987). A topological search algorithm for ATPG. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pp. 502–508, New York, NY, USA. ACM.

Knuth, D. E. (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley, Redwood City, CA, USA.

Kuehlmann, A.; Ganai, M. K. e Paruthi, V. (2001). Circuit-based boolean reasoning. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 232–237, New York, NY, USA. ACM Press.

Kuehlmann, A. e Krohm, F. (1997). Equivalence checking using cuts and heaps. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 263–268.

Kunz, W. (1993). HANNIBAL: an efficient tool for logic verification based on recursive learning. In *Proceedings International Conference on Computer-Aided Design*, pp. 538–543, Los Alamitos, CA, USA. IEEE Computer Society Press.

Kunz, W. e Pradhan, D. K. (1994). Recursive learning: A new implication technique for efficient solutions to CAD-problems: Test, verification and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1143–1158.

Kunz, W.; Silva, J. P. M. e Malik, S. (2002). SAT and ATPG: algorithms for boolean decision problems. pp. 309–341.

Kunz, W. e Stoffel, D. (1997). *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Norwell, MA, USA. Foreword By-Randal E. Bryant.

Larrabee, T. (1992). Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22.

Lee, C. (1959). Representation of switching circuits by binary decision diagrams. *Bell Syst. Tech. Journal*, 38:985–999.

Lu, F.; Wang, L.-C.; Cheng, K.-T. e Huang, R. C.-Y. (2003a). A circuit SAT solver with signal correlation guided learning. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, p. 10892, Washington, DC, USA. IEEE Computer Society.

Lu, F.; Wang, L.-C.; Cheng, K.-T. T.; Moondanos, J. e Hanna, Z. (2003b). A signal correlation guided ATPG solver and its applications for solving difficult industrial cases. In *Proceedings of the conference on Design automation*, pp. 436–441, New York, NY, USA. ACM Press.

Malik, S.; Wang, A. R.; Brayton, R. K. e Sangiovanni-Vincentelli, A. (1988). Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 6–10.

Mishra, P. e Dutt, N. (2004). Funcional validation of programmable architectures. In *Proceedings of the EUROMICRO Systems on Digital System Design*, pp. 12–19.

Molitor, P. e Mohnke, J. (2004). *Equivalence Checking of Digital Circuits - Fundamentals, Principles, Methodos*. Kluwer Academic Publishers.

Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Moret, B. M. E. (1982). Decision trees and diagrams. *ACM Comput. Surv.*, 14(4):593–623.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L. e Malik, S. (2001). Chaff: engineering an efficient SAT solver. In *Proceedings of the conference on Design automation*, pp. 530–535, New York, NY, USA. ACM Press.

Mukherjee, R.; Jain, J.; Takayama, K.; Fujita, M.; Abraham, J. A. e Fussell, D. S. (1997). FLOVER: Flitering oriented combinational verification approach. In *Workshop Notes of International Workshop on Logic Synthesis*.

Ozguner, F.; Marhefka, D.; DeGroat, J.; Wile, B.; Stofer, J. e Hanrahan, L. (2001). Teaching future verification engineers: the forgotten side of logic design. In *DAC '01: Proceedings of the 38th conference on Design automation*, pp. 253–255, New York, NY, USA. ACM.

Prasad, M. R.; Biere, A. e Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7:156–173.

Reda, S. e Salem, A. (2001). Combinational equivalence checking using boolean satisfiability and binary decision diagrams. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pp. 122–126, Piscataway, NJ, USA. IEEE Press.

Reda, S.; Wahba, A. e Salem., A. (2000). M-check: A multiple engine combinational equivalence checker. In *Proceedings of the International Symposium on Circuits and Systems*, pp. 613–616.

Roth, J. P. (1966). Diagnosis of automata failures. *IBM Journal of Research and Development*, 10(4):278–291.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 42–47, Los Alamitos, CA, USA. IEEE Computer Society Press.

SAT'08 (2008). SAT-Race 2008. Disponível em: `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/index.html`, Último acesso: 03/11/2008.

Schubert, T. (2003). High level formal verification of next-generation microprocessors. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 1–6.

Schulz, M. H. e Auth, E. (1989). Improved deterministic test pattern generation with applications to redundancy identification. *IEEE Transactions on Computer-Aided Design*, pp. 811–816.

Schulz, M. H.; Trischler, E. e Sarfert, T. M. (1988). SOCRATES: a highly efficient automatic test pattern generation system. *IEEE Transactions on Computer-Aided Design*, 7(1):126–137.

Silva, J. M. (1999). The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the Portuguese Conference on Artificial Inteligence*, pp. 62–74.

Silva, J. M. e Silva, L. G. (1999). Algorithms for satisfiability in combinational circuits based on backtrack search and recursive learning. In *Workshop Notes of the International Workshop on Logic Synthesis*, pp. 227–241.

Silva, J. P. M. (1995). *Search Algorithms for Satifiability Problems in Combinational Switching Circuits*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan.

Silva, J. P. M. (2000). Algebraic simplification techniques for propositional satisfiability. In *CP '02: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pp. 537–542, London, UK. Springer-Verlag.

Silva, J. P. M. e Sakallah, K. A. (1996). GRASP - a new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 220–227, Washington, DC, USA. IEEE Computer Society.

Stanion, T. (1999). Implicit verification of structurally dissimilar arithmetic circuits. In *Proceedings of the IEEE/ACM International Conference on Computer Design*, pp. 46–50, Washington, DC, USA. IEEE Computer Society.

Stoffel, D. e Kunz, W. (2004). Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(5):586–597.

Subbarayan, S. e Pradhan, D. K. (2004). NiVER: Non increasing variable elimination resolution for preprocessing sat instances. In *In Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT*, pp. 276–291. Springer.

van der Schoot, H. e Ural, H. (1997). A uniform approach to tackle state explosion in verifying progress properties for networks of cfsms.

Velev, M. N. (2004). Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Proceedings of the Conference on Asia South Pacific Design Automation*, pp. 310–315.

Wefel, S. e Molitor, P. (2000). Prove that a faulty multiplier is faulty!? In *GLSVLSI '00: Proceedings of the 10th Great Lakes symposium on VLSI*, pp. 43–46, New York, NY, USA. ACM.

Whittemore, J.; Kim, J. e Sakallah, K. (2001). SATIRE: A new incremental satisfiability engine. In *Proceedings of ACM/IEEE Conference on Design Automation*, pp. 542–545.

Xu, Z.; Yan, X.; Lu, Y. e Ge, H. (2003). Equivalence checking using independent cuts. In *Proceedings of Asian Test Symposium*, p. 482, Los Alamitos, CA, USA. IEEE Computer Society.

Zhang, L.; Madigan, C. F.; Moskewicz, M. H. e Malik, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided design*, pp. 279–285, Piscataway, NJ, USA. IEEE Press.

Zhao, J.; Newquist, J. A. e Patel, J. (1997). Static logic implication with application to fast redundancy identification. In *Proceedings of VLSI Test Symposium*, pp. 288–293.