MÁRCIA CAROLINA MARRA DE OLIVEIRA

UM NÚCLEO INTELIGENTE PARA PROCESSAMENTO DISTRIBUÍDO DE RESOLVEDORES SAT EM VERIFICAÇÃO POR EQUIVALÊNCIA

Belo Horizonte

27 de junho de 2006

MÁRCIA CAROLINA MARRA DE OLIVEIRA

UM NÚCLEO INTELIGENTE PARA PROCESSAMENTO DISTRIBUÍDO DE RESOLVEDORES SAT EM VERIFICAÇÃO POR EQUIVALÊNCIA

Dissertação apresentada ao Curso 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

27 de junho de 2006

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Um núcleo inteligente para processamento distribuído de resolvedores SAT em Verificação por Equivalência

MÁRCIA CAROLINA MARRA DE OLIVEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Claudionor José Nunes Coelho Jr. - Orientador
Departamento de Ciência da Computação – ICEx – UFMG

Prof. Ricardo dos Santos Ferreira
Departamento de Informática – DPI – UFV

Prof. Antônio Otávio Fernandes
Departamento de Ciência da Computação – ICEx – UFMG

Prof. Sérgio Vale Aguiar Campos
Departamento de Ciência da Computação – ICEx – UFMG

Belo Horizonte, 27 de junho de 2006.

Resumo

Verificação por Equivalência é um dos componentes chave da metodologia de verificação formal atual para sistemas digitais. Ela é técnica de Verificação Formal mais utilizada atualmente pela indústria para verificação de igualdade entre duas descrições de um circuito. Diversas abordagens baseadas BDDs e SAT obtiveram um considerável sucesso nesta área. No entanto a crescente distância entre a capacidade dos resolvedores atuais e a complexidade das instâncias a serem verificadas motivam a exploração de novas alternativas, em busca de soluções melhores. Esta dissertação apresenta um núcleo inteligente para processamento distribuído de resolvedores SAT em Verificação por Equivalência. Especificamente, o núcleo proposto explora o processamento paralelo de resolvedores SAT e propõe uma nova técnica para a identificação de similaridades estruturais entre os circuitos a serem verificados. Ao final, são apresentados resultados que comprovam a eficiência da metodologia proposta.

Abstract

Equivalence Checking is one the of the key components in formal verification for digital systems. It is also one of the most widely used approaches in industry for functional equivalence verification of different designs. A number of recently proposed BDD and SAT based approaches have met with considerable success in this area. However, the growing gap between the current solvers capabilities and the increasing complexity in digital designs lead to exploring alternative, better solutions. This work proposes an intelligent kernel for distributed processing of SAT solvers in Equivalence Checking. Specifically, the proposed kernel exploits a new technique for identifying structural similarities between the circuits. Finally, results of verification using the proposed methodology are presented.

Este trabalho é dedicado à Prosolina Alves Marra.

Agradecimentos

À madre, pelo amor incondicional... e por sempre me incentivar a seguir adiante, mesmo quando tive dúvidas se estava no caminho certo. Esta vitória também é sua!

Ao Du, pelo carinho e incentivo durante toda esta caminhada. Perto ou longe, você estará sempre dentro do meu coração!

Ao Claudionor, por acreditar em mim e me oferecer tantas oportunidades de crescimento.

Ao Otávio, pela amizade e constante suporte no decorrer desta jornada. É muito importante saber que sempre posso contar com você!

Ao Valdeci, por compartilhar comigo sua grande sabedoria e experiência de vida. São estes momentos que realmente agregam em nossa vida!

Ao André, por acreditar em meu potencial e estar sempre disposto a me orientar com sua valiosa visão de mundo.

Às minhas queridas amigas Vanessa, Marianna, Fernanda e Carol, por compreenderem e me apoairem nos momentos em que tive que me ausentar de seus convívios.

À Polly e Mayara (uma anjinha que está quase chegando...) pelos momentos de terna alegria e genuína amizade.

Aos meus amigos e companheiros de almoço Vinícius (Makish), Breno Vitorino e Thiago Radicchi: por tornarem os momentos de dúvidas e agruras da vida acadêmica muito mais leves e divertidos! Finalmente chegamos lá! Somos mestres!

Ao Fabrício Vivas, pelas ótimas discussões sobre relacionamentos humanos e liderança! Ainda seremos os líderes que idealizamos!

Aos amigos LECOMnianos de todos os tempos, de dinossauros a bebês. Cada um de vocês teve uma contribuição importante neste trabalho.

À toda a secretaria do DCC, pelo carinho e total profissionalismo! Obrigada pela torcida, meninas!

À todos aqueles que de alguma forma contribuíram para a viabilização deste trabalho.

Sumário

Lista de Figuras **x**

1 Introdução **1**
1.1 Escopo do trabalho 3
1.2 Contribuições 4
1.3 Organização 5

2 Revisão Bibliográfica **6**
2.1 Equivalence Checking 6
2.1.1 Mecânica de Equivalence Checking 8
2.1.2 Métodos formais de Equivalence Checking 10

3 Núcleo inteligente para Equivalence Checking **23**
3.1 Metodologia proposta 23
3.2 Trabalhos relacionados 27
3.3 Núcleo inteligente desenvolvido 29
3.3.1 API 29
3.3.2 Módulo Gerenciamento de Problemas 30
3.3.3 Resolvedores SAT 31
3.3.4 Módulo de Identificação de similaridades 32

4 Explorando as similaridades estruturais **33**
4.1 Metodologia de identificação de similaridades estruturais a partir de cláusulas de conflito 36
4.1.1 Geração de cláusulas de conflito em resolvedores SAT 36
4.1.2 Metodologia para identificação de similaridades estruturais . . . 39

5 Resultados Experimentais **47**
5.1 Implementação 47
5.1.1 Resolvedor ZChaff 47
5.1.2 MPI (*Message Passing Interface*) 48
5.1.3 Linux 49
5.2 Resultados experimentais 49
5.2.1 Aprendizado recursivo 49
5.2.2 Verificação de um multiplicador CLA com um multiplicador Booth 68
5.2.3 Processamento Distribuído 69

5.3 Performance da solução adotada 70

6 Conclusões e Trabalhos Futuros 72

Lista de Figuras

1.1 Metodologia proposta para Equivalence Checking 3

2.1 Fluxo típico de projeto de circuitos integrados 7
2.2 Procedimento típico do processo de Equivalence Checking 8
2.3 *Miter*: modelo de equivalência computacional formado pelo produto de máquinas de estado 9
2.4 Taxonomia das técnicas em Equivalence Checking segundo Marques-Silva 11
2.5 Construção do BDD para $(x \otimes y \otimes z)$ 12
2.6 Um exemplo simples: este circuito é um XOR? 13
2.7 Justificativa $i = 0$ e $j = 1$ 15
2.8 Exemplo de verificação por aprendizagem recursiva 18
2.9 Exemplo de circuito e fórmula CNF 19

3.1 Gráfico de correlação 24
3.2 Metodologia proposta para Equivalence Checking 25

4.1 Circuitos contendo equivalências internas 34
4.2 Utilização de similaridades estruturais internas entre os circuitos 35
4.3 Um grafo de implicação típico do processo de resolução SAT 37
4.4 Cortes no grafo de implicações para geração de cláusulas de conflito . . 39
4.5 Particionamento de cláusulas de conflito 46

5.1 Processo de multiplicação binária 50
5.2 Somador de 16 bits implementando esquema CLA em dois níveis 51
5.3 Exemplo de codificação de Booth 53
5.4 Tempos de resposta para a verificação de duas descrições de multiplicadores idênticos 56
5.5 Similaridades encontradas na verificação de duas descrições de multiplicadores idênticos 57
5.6 Utilização de memória pelo ZChaff para a verificação de duas descrições de multiplicadores idênticos 58
5.7 Número de cláusulas de conflito geradas para a verificação de duas descrições de multiplicadores idênticos 59
5.8 Tempos de resposta para a verificação de duas descrições de multiplicadores com inserção de erros aleatórios 60

5.9 Similaridades encontradas na verificação de duas descrições de multiplicadores com inserção de erros aleatórios 61
5.10 Utilização de memória pelo ZChaff na verificação de duas descrições de multiplicadores com inserção de erros aleatórios 62
5.11 Cláusulas de conflito geradas na verificação de duas descrições de multiplicadores com inserção de erros aleatórios 63
5.12 Tempos de resposta para a verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits 64
5.13 Similaridades encontradas na verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits 65
5.14 Utilização de memória pelo ZChaff para a verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits 66
5.15 Cláusulas de conflito geradas na verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits 67
5.16 Tempos de resposta para a verificação de duas descrições de multiplicadores - CLA e Booth 68
5.17 Custo da identificação de similaridades dentro da metodologia de verificação 71

Capítulo 1

Introdução

Este trabalho propõe, analisa e valida um núcleo inteligente para processamento distribuído de resolvedores SAT (Satisfabilidade Proposicional) em Equivalence Checking. É também proposta uma nova metodologia para identificação de similaridades estruturais [1, 2, 3] entre os circuitos a serem verificados a partir da utilização de cláusulas de conflito geradas durante o processo de resolução SAT, com o objetivo de melhorar o desempenho da metodologia Equivalence Checking.

Equivalence Checking é uma técnica bastante utilizada atualmente pela indústria, para a verificação de circuitos integrados. Ela é um método de grande importância entre os métodos formais, tanto do ponto de vista teórico quanto prático e aparece em um vasto número de aplicações CAD (do inglês, *Computer-Aided Design*) relacionadas a verificação de projetos de circuitos integrados. Abaixo são apresentados alguns dos inúmeros cenários em que Equivalence Checking possui uma grande relevância:

- Com o aumento exponencial dos circuitos integrados e da complexidade das ferramentas de síntese que manipulam estes circuitos, é impossível garantir que os circuitos gerados por estas ferramentas estejam livres de erros. Em vez de verificar formalmente as ferramentas de síntese, a abordagem mais utilizada atualmente é verificar, formalmente, se os circuitos gerados por estas ferramentas correspondem funcionalmente às entradas originais. Em geral, as entradas originais correspondem à especificação do circuito fornecida para a ferramenta. Esta verificação é uma instância do problema de Equivalence Checking.

- Quando um circuito é manualmente modificado com o objetivo de conter requisitos especiais que não podem ser gerados a partir de ferramentas CAD, o projetista precisa de garantir que nenhum erro funcional foi introduzido no circuito. Esta tarefa é realizada verificando se o projeto original e o modificado são funcional-

mente idênticos, o que também é uma instância do problema de Equivalence Checking.

- Equivalence Checking aparece ainda como um sub-problema de outros problemas de verificação de nível de abstração superior. Um exemplo é a verificação de circuitos aritméticos, realizada através da garantia de satisfação de uma dada equação de recorrência ou na verificação de equivalência entre duas máquinas de estado, onde realizar uma busca transversal nos estados gerados é muito caro em termos computacionais.

Apesar de Equivalence Checking ser teoricamente considerada um problema coNP-difícil [4], na prática as instâncias do problema são geralmente tratáveis. Para o caso de Equivalence Checking de circuitos combinacionais, as metodologias atuais garantem que os dois circuitos a serem verificados por equivalência possuem um grau mínimo de similaridade estrutural e funcional [5]. Nos últimos anos, diversas abordagens foram propostas explorando esta propriedade. Apesar destas técnicas terem tido um avanço significativo no estado-da-arte de Equivalence Checking, a complexidade inerente ao problema e aumento exponencial do tamanho e complexidade dos circuitos digitais continuam motivando pesquisas nesta área.

A maioria das implementações eficientes que existem atualmente para Equivalence Checking utilizam uma combinação de vários motores, utilizando BDD (do inglês, *Binary Decision Diagrams* [6] como principal técnica de resolução. Apesar de algumas abordagens utilizarem SAT [7] ou motores baseados em SAT (métodos ATPG (do inglês, *Automatic Test Program Generator*) [1] e aprendizagem recursiva [3]), estes métodos não se tornaram muito populares. Conseqüentemente, a utilização de SAT em Equivalence Checking tem sido largamente subordinada a BDDs. Um exemplo deste tipo de subordinação é utilização de SAT para eliminar falsos negativos ou para escolher pares candidatos para deduzir relacionamentos intermediários [8] em BDD.

Existem diversas razões pelas quais este trabalho é focado na utilização de SAT para o núcleo desenvolvido. Em primeiro lugar, houve avanços significativos nos algoritmos SAT. Em segundo lugar, apesar de haver diversas crenças de que BDDs são relativamente mais eficientes para Equivalence Checking, não existem publicações que apresentem comparações quantitativas ou análises detalhadas das ineficiências dos algoritmos SAT. Em terceiro lugar, os algoritmos SAT possuem algumas características inerentes que fazem com que eles se tornem potencialmente mais flexíveis e robustos para aplicações em Equivalence Checking [9].

1.1 Escopo do trabalho

A pesquisa, objeto da presente dissertação de mestrado, forma parte de um trabalho maior, desenvolvido em conjunto com outro aluno de mestrado. O trabalho desenvolvido em conjunto propõe uma nova metodologia para Equivalence Checking de circuitos integrados escritos em linguagens de verificação de hardware (tais como Verilog, VHDL [10]) através da utilização de resolvedores SAT. A metodologia proposta analisa as características dos circuitos a serem verificados e a partir desta análise, particiona os circuitos tendo por base critérios de largura e ciclos. Em seguida, similaridades estruturais entre os circuitos são identificadas e utilizadas como realimentação para partições posteriores do circuito.

A Figura 1.1 apresenta a metodologia proposta.

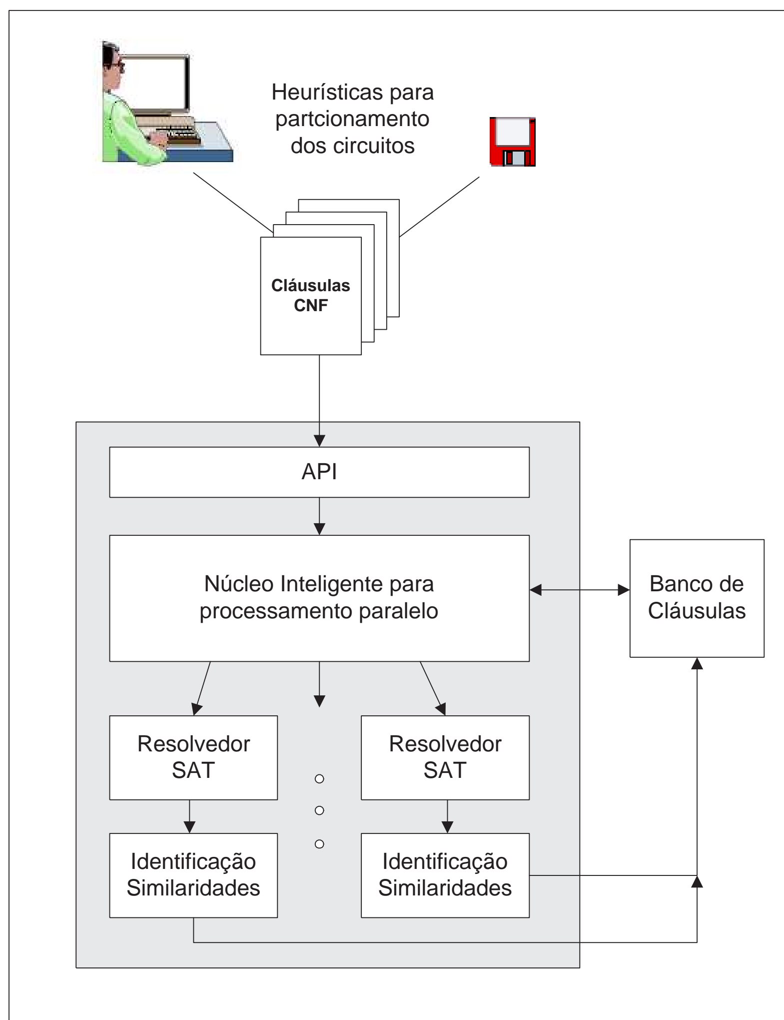


Figura 1.1: Metodologia proposta para Equivalence Checking

Inicialmente os circuitos a serem verificados são submetidos a uma análise estrutural que permite decidir o critério de particionamento a ser adotado e mapear as entradas e saídas primárias de cada um deles. Este processo pode ser realizado manualmente ou automaticamente, através de um aplicativo apropriado. Após terem sido particionados e traduzidos em cláusulas na forma CNF (do inglês, *Conjunctive Normal Form*), os circuitos a serem verificados são submetidos ao núcleo inteligente através de uma API (do inglês, *Application Program Interface*). Dentro do núcleo inteligente, as cláusulas são distribuídas entre os nós previamente selecionados e cada nó executa uma instância do resolvedor SAT. A partir das cláusulas de conflito geradas durante o processo de resolução SAT, são identificadas similaridades estruturais entre os circuitos. As similaridades estruturais alimentam o banco de cláusulas que é utilizado como base de conhecimento para a próxima partição do circuito. Este processo é repetido iterativamente até que todo o circuito tenha sido verificado ou até que tenha sido encontrada a contra-prova de que os circuitos verificados não são equivalentes.

Na Figura 1.1, a parte em cinza refere-se ao trabalho na presente dissertação. Foi proposto e validado um núcleo inteligente para processamento paralelo de resolvedores SAT para Equivalence Checking que utiliza um módulo de identificação de similaridades estruturais baseado em cláusulas de conflito.

1.2 Contribuições

As principais contribuições deste trabalho podem ser agrupadas em duas áreas distintas:

Núcleo inteligente: É proposto, analisado e validado um núcleo inteligente para Equivalence Checking em ambiente de arquitetura paralela. Este núcleo utiliza estrutura de dados e comunicação otimizadas entre os nós participantes e define uma API que permite ao projetista controlar de forma inteligente as instâncias do problema que estão sendo resolvidas em cada um dos nós.

Metodologia para identificação de similaridades: É proposta uma nova metodologia para identificação de similaridades estruturais entre circuitos a partir de cláusulas de conflito geradas durante o processo de resolução SAT. A metodologia desenvolvida apresenta ótimos resultados na verificação de circuitos que possuem inclusão estrutural.

1.3 Organização

Esta dissertação está organizada em 5 capítulos. O capítulo 1 introduz o tema da dissertação. O capítulo 2 fornece uma breve revisão bibliográfica sobre a metodologia Equivalence Checking e as principais técnicas utilizadas atualmente nesta área. O capítulo 3 apresenta o núcleo inteligente proposto para Equivalence Checking enquanto o capítulo 4 é dedicado à metodologia para identificação de similaridades estruturais entre os circuitos a serem verificados. O capítulo 5 é dedicado à apresentação dos resultados experimentais e o capítulo 6 ressalta as conclusões e as possíveis extensões do trabalho atual.

Capítulo 2

Revisão Bibliográfica

Este capítulo fornece uma breve revisão bibliográfica sobre a metodologia Equivalence Checking e as principais técnicas utilizadas atualmente nesta área.

2.1 Equivalence Checking

O processo de transformar a representação de um modelo de alto nível de um projeto em uma implementação física envolve muitos passos. Em cada passo são realizadas diversas operações nas representações, tornando-se necessária realizar Equivalence Checking entre elas. A tarefa de provar a igualdade entre as várias representações do projeto tem sido historicamente um desafio.

A Figura 2.1 apresenta o papel da Equivalence Checking dentro de um fluxo típico de projeto de circuito integrado. Neste fluxo, o projeto é inicialmente especificado em nível RTL (do inglês, *Register Transfer Level*), utilizando uma linguagem de descrição de hardware, tal como Verilog ou VHDL [10] ou descrito em nível de micro arquitetura. Esta especificação de alto nível é então sintetizada para o nível de listas de portas lógicas. Em seguida, passos de otimização de seqüência são aplicados ao projeto, com o objetivo de otimizar área, tempo, testabilidade ou dissipação de potência. A implementação otimizada é então mapeada utilizando bibliotecas através de tecnologias de mapeamento.

Durante este processo, é necessário verificar a equivalência de duas representações, descritas em um mesmo nível ou níveis diferentes de abstração. Por exemplo, é bastante crítico e importante verificar a equivalência entre a implementação otimizada e a especificação no nível RTL, para garantir que nenhum erro foi introduzido durante o processo, especialmente quando estiver envolvido esforço humano nesta etapa. De forma semelhante, é necessário verificar a equivalência entre a implementação em nível

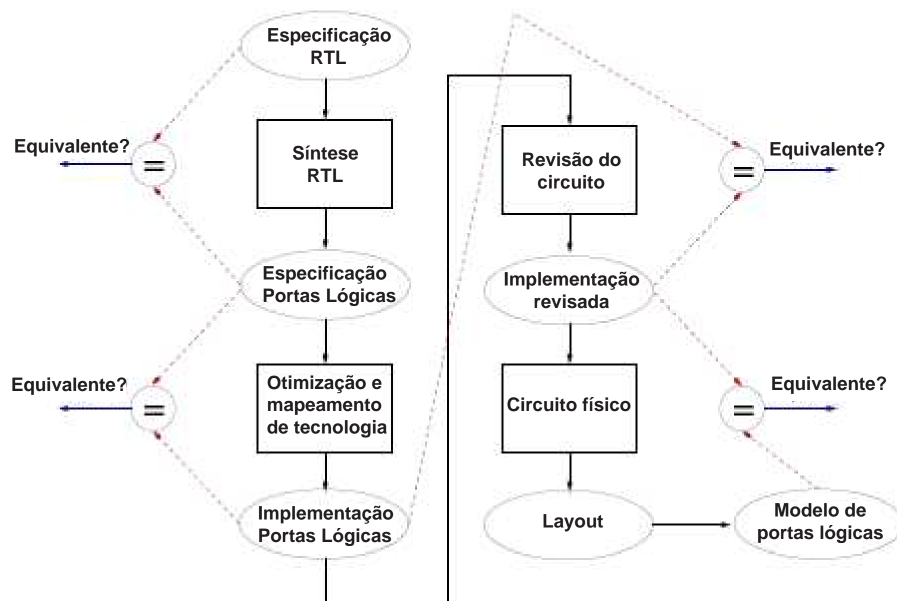


Figura 2.1: Fluxo típico de projeto de circuitos integrados

de portas lógicas e o modelo de portas lógicas extraídos do layout, para garantir que nenhum erro foi introduzido durante o processo de projeto físico.

A abordagem tradicional [11] para superar o desafio de Equivalence Checking no fluxo do projeto é abandonar o modelo RTL, utilizando apenas uma representação - o modelo de portas lógicas. Conseqüentemente, o modelo de portas lógicas passa a ser o modelo de referência do projeto (*gold model*) e é utilizado para realizar a verificação funcional, análise de tempos e outras formas de verificação física. No entanto, utilizando esta abordagem, não é possível fazer a distinção entre a verificação física e funcional, impedindo o fluxo de verificação.

Uma abordagem alternativa [11], também bastante utilizada, é manter a representação em RTL como o modelo de referência durante a verificação funcional. Desta forma, os projetistas estabelecem a igualdade através da execução de testes de simulação regressiva em ambos os modelos (níveis RTL e porta lógica) e comparam a saída dos resultados. Esta alternativa também possui diversos desafios, já que a utilização de simulação para prova de equivalência entre circuitos gera resultados incompletos.

Além disto, ambas as abordagens apresentadas comprometem seriamente os objetivos do cronograma do projeto, já que a simulação é um gargalo no fluxo do projeto. Estes problemas motivaram a busca pela melhoria de métodos de Equivalence Checking.

Atualmente, as técnicas de argumentação matemática oferecem uma grande melhoria com relação ao uso de simulações e vetores de teste para estabelecer provas de igualdade entre vários projetos. Esta forma de condução de raciocínio, chamado de Verificação Formal, garante sistematicamente que a implementação do projeto (por exemplo, o modelo revisado) satisfaz a sua especificação (por exemplo, o modelo de referência). O que, tradicionalmente, levaria semanas e dias para ser apenas parcialmente verificado através da abordagem de simulação, pode ser verificado completamente, em termos de horas e minutos, utilizando Equivalence Checking, uma das metodologias de Verificação Formal.

Equivalence Checking está revolucionando o processo atual de verificação de projetos. Este trabalho é focado em Equivalence Checking de circuitos combinacionais. Apesar da Equivalence Checking ser conhecida como pertencente à classe de complexidade coNP-difícil, várias técnicas de verificação combinadas com práticas efetivas de projeto podem garantir o seu sucesso.

2.1.1 Mecânica de Equivalence Checking

A Figura 2.2 apresenta o procedimento típico de um processo de Equivalence Checking.

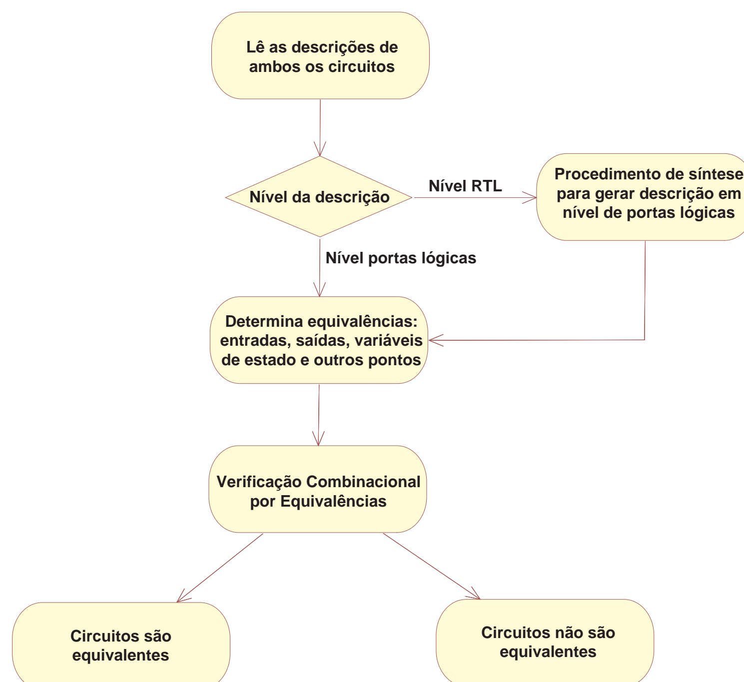


Figura 2.2: Procedimento típico do processo de Equivalence Checking

Inicialmente os circuitos são lidos a partir de uma descrição e, caso algum deles esteja em um nível de abstração diferente do nível portas lógicas, é realizada uma conversão a partir das ferramentas de síntese utilizadas atualmente. Em seguida, são identificadas as equivalências iniciais necessárias para realizar a verificação: as entradas e saídas, as variáveis de estado (tais como *latches*) e outros pontos possivelmente relevantes dos dois circuitos são mapeados. Este mapeamento pode ser realizado tanto manualmente, pela interação do projetista, quanto automaticamente, utilizando nomes e funções. O próximo passo é submeter o circuito a uma das técnicas utilizadas para realizar Equivalence Checking. A tarefa de Equivalence Checking é apontar se os circuitos verificados são ou não equivalentes.

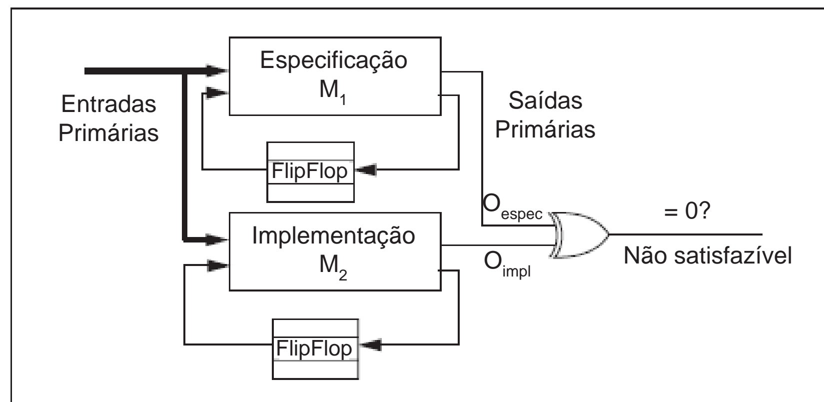


Figura 2.3: *Miter*: modelo de equivalência computacional formado pelo produto de máquinas de estado

Na Figura 2.3 pode ser visto um modelo de equivalência computacional típico. Tradicionalmente, a verificação da equivalência funcional de dois circuitos é realizada através da construção de suas representações canônicas, como por exemplo tabelas-verdade e BDDs. Os dois circuitos são equivalentes se, e somente se, as suas representações canônicas forem isomórficas. Para verificar a equivalência de dois circuitos, os circuitos são geralmente vistos como máquinas de estados finitos (FSM). Assumindo que duas máquinas possuem um estado de reinicialização conhecido, elas são equivalentes se, e somente se, os estados de reinicialização forem equivalentes.

Uma máquina de estados produto de dois circuitos a serem verificados pode ser formada conectando pares de entradas e saídas dos circuitos a serem verificados. Este modelo é conhecido como *miter* [1].

A Figura 2.3 apresenta o modelo. Ele é construído criando uma máquina de produto entre duas máquinas de estado finitos (FSMs): M_1 e M_2. Este produto é formado através de 3 passos:

- identificação dos pares de entrada e saída entre as duas FSMs;
- conexão de cada par de entradas primárias juntas;
- conexão de cada par de saídas primárias à porta lógica XOR.

Para provar a equivalência entre estas duas máquinas é necessário apenas verificar se a saída da porta lógica XOR (formada pelas saídas da máquina de produto) é sempre 1 para qualquer seqüência de entrada.

Considere um problema de provar a equivalência combinacional. Dado dois circuitos combinacionais, $F_{espec}(X)$ e $F_{impl}(X)$, onde X representa um vetor definido das variáveis de entrada $(x_1, x_2, ..., x_n)$, é possível estabelecer a equivalência entre estes dois circuitos provando que a equação 2.1 é satisfazível para todos os valores de X:

$$F_{espec}(X) \oplus F_{impl}(X) \tag{2.1}$$

Várias abordagens foram desenvolvidas para resolver este problema. A seção 2.1.2 apresenta algumas das abordagens utilizadas atualmente.

2.1.2 Métodos formais de Equivalence Checking

Equivalence Checking pode ser categorizada como combinacional ou seqüencial. Na Equivalence Checking Combinacional, os dois circuitos são acíclicos, estão descritos no nível de portas lógicas e a tarefa é determinar se eles computam a mesma função booleana. É interessante notar que a equivalência combinacional pode ser utilizada para provar a equivalência de dois circuitos seqüenciais, desde que estes circuitos utilizem a mesma codificação de seus estados.

Na Equivalence Checking Seqüencial, os dois circuitos a serem verificados podem utilizar codificações de estados totalmente diferentes. A tarefa neste caso é determinar se os dois circuitos irão apresentar diferença em seus valores de saída quando forem iniciados utilizando seqüências de entrada arbitrárias.

A Figura 2.4 apresenta a taxonomia de métodos formais de Equivalence Checking segundo Marques-Silva [7]. Nesta classificação, as abordagens de verificação são dividas em duas grandes categorias: (1) abordagem simbólica e (2) abordagem estrutural. As *abordagens simbólicas* utilizam técnicas simbólicas utilizando, em sua maioria, BDDs como forma de representação. Uma forma alternativa de abordagem simbólica é a utilização de linguagens de programação lógica, tais como *Prolog*. As *abordagens estruturais* são abordagens que exploram as similaridades estruturais entre os dois circuitos a serem verificados. As abordagens estruturais podem ainda ser divididas em 3

grupos, de acordo com a técnica em que se baseiam: (a) baseada em geração automática de padrões de teste (ATPG), (b) baseada em aprendizado recursivo e (c) baseado no problema da Satisfabilidade Proposicional (SAT).

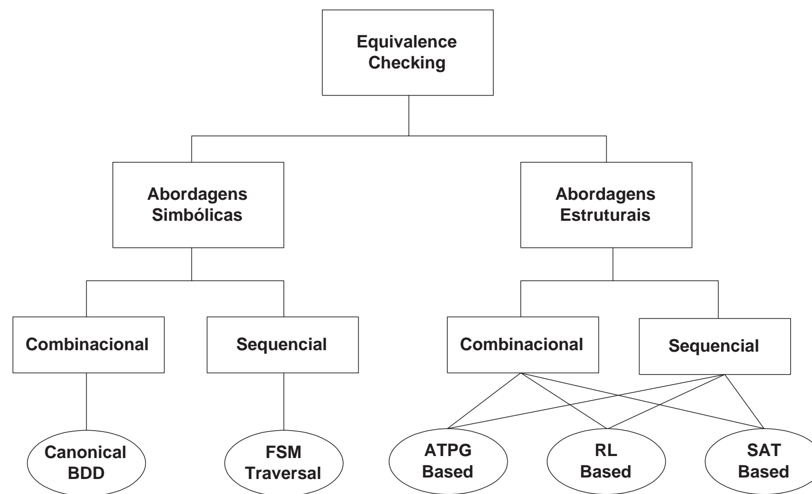


Figura 2.4: Taxonomia das técnicas em Equivalence Checking segundo Marques-Silva

As abordagens estruturais, apesar de dependerem de similaridades estruturais entre os circuitos a serem verificados, têm demonstrado empiricamente uma maior capacidade para gerenciar circuitos grandes, após otimização lógica intensa. Por outro lado, as abordagens simbólicas baseadas em BDD possuem independência da estrutura dos circuitos a serem verificados e portanto, são mais adequadas para verificação de circuitos que sejam diferentemente codificados ou que tenham estruturas completamente diferentes.

Nas próximas subseções serão descritas as técnicas mais importantes em Equivalence Checking. Um foco maior será dado às técnicas baseadas em SAT para realizar verificação por equivalência, já que esta técnica foi escolhida para ser utilizada no presente trabalho.

Diagramas de Decisão Binários (BDD)

Diagramas de Decisão Binário, ou BDDs, são grafos acíclicos diretos, utilizados para representarem funções booleanas. Os BDDs foram introduzidos em sua forma atual por Bryant [6], apesar das idéias gerais já serem conhecidas há mais tempo (como por exemplo os programas de derivação na literatura teórica da ciência da computação).

Conceitualmente, um BDD pode ser construído para uma função booleana como descrito abaixo. Inicialmente, construa a árvore de decisão para a função desejada, obedecendo as restrições de que no caminho da raiz até a folha, cada variável pode

aparecer apenas uma vez e que nos caminhos as variáveis apareçam sempre na mesma ordem. Cada caminho da árvore construída deve representar um assinalamento para as variáveis da função. Os vértices finais, ou folhas, representam o valor para um determinado caminho. Um exemplo de árvore inicial pode ser visto na Figura 2.5 (a). Em seguida aplique o máximo possível das 2 regras de redução a seguir: (1) una todos os nós duplicados (nós com os mesmos rótulos e filhos) e (2) se ambos os ponteiros para filhos de um nó apontam para um mesmo nó, retire o nó já que ele é redundante. Estas operações de redução possuem complexidade de tempo linear com relação ao tamanho do grafo. A aplicação das duas regras pode ser vista na Figura 2.5 (b). O grafo acíclico direto resultante é o BDD para a função, como mostrado na Figura 2.5 (c). Na prática, BDDs são gerados e manipulados em sua forma reduzida, sem terem nunca sido construídos como árvores de decisão.

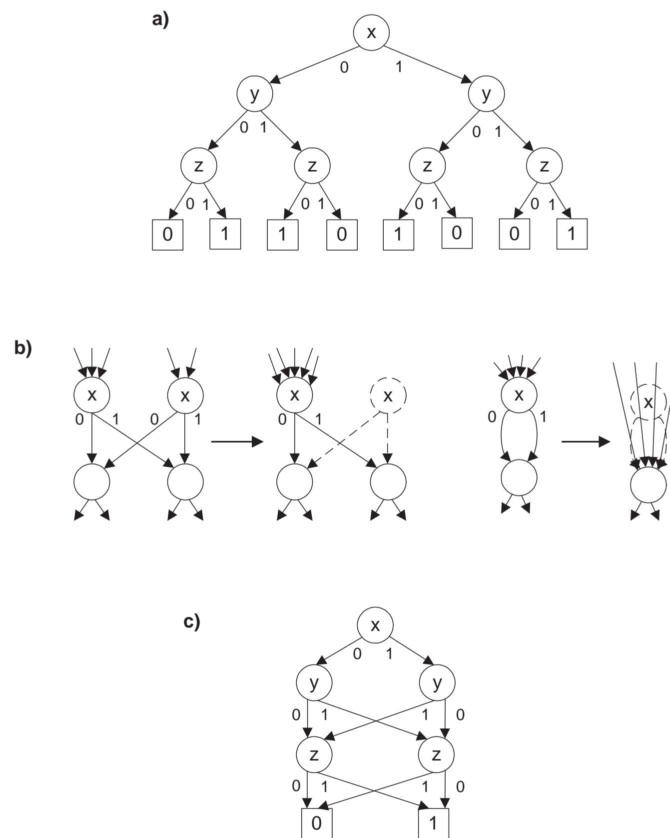


Figura 2.5: Construção do BDD para $(x \otimes y \otimes z)$

Em uma implementação típica, todos os BDDs utilizados são reduzidos ao máximo possível para maximizar os compartilhamentos de nós, de forma que a função seja representada por um ponteiro para o seu nó raiz. Por exemplo, na Figura 2.5, a função $(x \otimes y \otimes z)$ é representada por um ponteiro para o seu nó do topo enquanto a função

$(y \vee z)$ é representada por apenas um ponteiro para o nó de rótulo y mais à esquerda, ao invés de várias cópias dos nós.

Os BDDs são muito sensíveis ao ordenamento de suas variáveis. Em geral, a escolha da ordem das variáveis pode fazer a diferença entre tamanho linear e exponencial dos BDDs em relação ao número de variáveis.

Como BDDs são representações canônicas, eles são muito utilizados para verificar a equivalência entre dois circuitos. Para realizar Equivalence Checking entre dois circuitos, é construído BDDs para as saídas em termos das entradas primárias. Dado que os BDDs são representações canônicas, os 2 circuitos implementam a mesma função se, e somente se, eles possuem o mesmo BDD.

Por exemplo, considere a verificação do circuito apresentado na Figura 2.6 implementando um OR exclusivo, ou seja, um XOR. Inicialmente, rotule as entradas primárias para os BDDs com as variáveis y e z. Em seguida, construa um BDD para cada porta lógica de saída como uma função de suas entradas - rotulando a porta lógica OR com o BDD para $(y \vee z)$, a porta lógica NAND com o BDD para $\neg(y \wedge z)$ e a porta lógica AND com $((y \vee z) \wedge \neg(y \wedge z))$. Para o circuito da especificação, construa o BDD para $(y \otimes x)$. Uma vez que as duas expressões implementam a mesma função booleana, elas possuem o mesmo BDD, o que verifica que o circuito da Figura 2.6 é uma OR exclusivo.

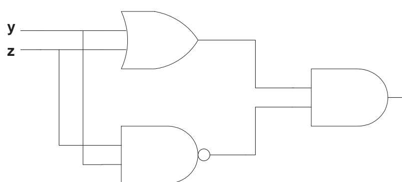


Figura 2.6: Um exemplo simples: este circuito é um XOR?

Na prática, esta abordagem é limitada pelo tamanho dos BDDs gerados, que é extremamente sensível a função a ser verificada e a ordem de variáveis utilizada. Para alguns exemplos patológicos, como os multiplicadores, até mesmo 16 bits geram circuitos muito grandes para serem resolvidos. Tipicamente, circuitos com até poucas centenas de entradas primárias podem, na maioria das vezes, ser verificados. Para circuitos maiores, são necessários métodos mais sofisticados, tais como métodos que utilizam OBDDs (do inglês, *Ordered Binary Decision Diagram*) [12].

Geração automática de padrões de teste (ATPG)

Equivalence Checking baseada em ATPG (geração automática de padrões de teste) é bastante próxima às técnicas de Equivalence Checking baseadas em SAT. A principal

diferença entre os resultados SAT e ATPG é o fato de que os algoritmos SAT operam sobre fórmulas CNF enquanto que os algoritmos ATPG operam normalmente sobre redes booleanas. Enquanto as pesquisas em torno de SAT foram iniciadas no contexto de prova automática por teorema, as pesquisas em ATPG foram inicialmente guiadas para a área de testes de circuitos. Equivalence Checking baseada em SAT será discutida na Seção 2.1.2.

Seja N uma rede booleana e seja $\phi(N) \in \beta_{n,m}$ a função booleana representada por N. Entende-se por rede booleana um grafo dirigido onde os nós são portas lógicas e as arestas são conexões entre as portas lógicas. A presença de uma falha de fabricação ρ transforma N em um novo circuito N_ρ. Assuma que N_ρ seja também uma rede booleana e que seu comportamento seja descrito pela função $\phi(N_\rho) \in \beta_{n,m}$. O problema de testar falhas ρ é encontrar um vetor de entrada $\alpha \in \{0,1\}^n$ tal que

$$\phi(N)(\alpha) \neq \phi(N_\rho(\alpha)) \tag{2.2}$$

O vetor α é chamado de vetor de teste ou vetor distinto para a falha ρ. Se N é uma função booleana com uma única saída para uma implementação de hardware, então o vetor α é um vetor de teste para alguma falha ρ se, e somente se

$$\phi(N)(\alpha) \otimes \phi(N_\rho)(\alpha) = 1 \tag{2.3}$$

As falhas físicas são parcialmente modeladas por falhas lógicas. Esta abordagem reduz a complexidade de verificação uma vez que falhas físicas diferentes podem ser modeladas pelas mesmas falhas lógicas. Além disto, o problema de análise de falhas torna-se um problema mais lógico do que físico.

Em muitas tecnologias, uma pequena diferença entre os sinais terra ou energia e uma linha de sinal ou uma linha de sinal aberta unidirecionalmente pode fazer com que um sinal fique fixo em um nível de voltagem. Este comportamento é modelado pelo modelo *stuck-at-fault*. Um *stuck-at-fault* é uma falha lógica que consiste de um sinal sendo mantido em um valor lógico fixo $b \in \{0,1\}$. Este valor é conhecido como s-a-b.

Para testar um sinal e para s-a-b, é necessário encontrar um vetor que ative a falha, ou seja, assinalar o valor b' ao sinal e no circuito com falhas. Esta fase é usualmente conhecida como processo de justificativa ou processo de ativação. Desta forma, os diferente valores são aplicados na localização da falha, tanto no circuito com falhas quanto no circuito correto. Esta diferença deve ser propagada até as saídas primárias.

A Figura 2.7 apresenta um exemplo de uma justificativa utilizada em ATPG. As redes booleanas F e G são funcionalmente equivalentes se, e somente se, não existir

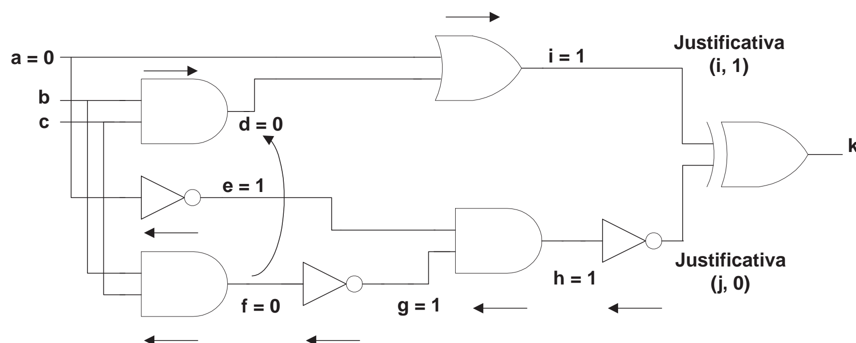


Figura 2.7: Justificativa $i = 0$ e $j = 1$

nenhum assinalamento para as entradas primárias do circuito apresentado tal que a linha de saída seja avaliada para 1. Para testar a saída para s-a-0, é necessário aplicar um vetor de testes que ative a falha, ou seja, que satisfaça o circuito *miter*. Desta forma, Equivalence Checking pode ser vista como um caso de testes para *stuck-at-fault*. Apenas a fase de ativação deve ser transferida para as outras etapas; a fase de propagação pode ser descartada.

Considere o *miter* da Figura 2.7. Para justificar o valor 1 na saída k, existem duas alternativas: assinalar a linha de sinal i ou a linha de sinal j para 1. Desta forma, a busca pela solução envolve um processo de decisão. Inicialmente, vamos tentar justificar $i = 1$ e $j = 0$. Por implicação atrás, o assinalamento do sinal $j = 0$ induz ao assinalamento $h = 1$. Iterativamente, o passo anterior implica nos assinalamentos $e = 1$, $g = 1$, $a = 0$ e $f = 0$. Para justificar $f = 0$, é necessário que a linha de sinal b ou c seja assinalado para 0. Ambas as alternativas induzem o sinal de linha d diretamente para o valor 0. Assumindo que as linhas de sinais a e d estão assinalados para 0, o sinal de linha i também deve ser 0. No entanto, isto produz um conflito, já que queremos justificar $i = 1$. Desta forma, é necessário realizar um backtrack para tentar justificar $i = 0$ e $j = 1$. Como este processo de justificativa também resulta em um conflito, não existe nenhum assinalamento para as entradas primárias do miter tal que ative s-a-0 na saída k. Desta forma, a saída k não pode ser testada para s-a-0 e os dois circuitos são funcionalmente equivalentes.

O problema ATPG foi provado ser um problema NP-Completo [13], ou seja, não existe nenhum algoritmo que resolva o problema em tempo polinomial. No entanto, diversos algoritmos já foram propostos para resolver o problema. A maioria deles são bastante eficientes e exploram heurísticas que:

- Tentam encontrar todos os assinalamentos de sinais necessários para um teste o mais cedo possível;

- Tentam realizar a busca o mínimo possível acima do espaço de decisão.

O primeiro algoritmo completo proposto para padrões de teste foi apresentado por Ruth em 1966 e é conhecido como *Algoritmo-D* [14]. Um algoritmo de teste é completo se ele é capaz de propagar uma falha até uma saída observável caso o circuito apresente falhas.

A partir deste, diversos outros algoritmos extremamente eficientes foram propostos, tais como PODEM [15], FAN [16], TOPS [17], Socrates [18], entre outros.

Aprendizado Recursivo (RL)

O aprendizado recursivo permite realizar implicações precisas [19] para um conjunto de valores assinalados em um subconjunto de linhas no circuito lógico. Realizar implicações precisas significa identificar todos os valores de sinal que são unicamente determinados como conseqüência de uma dada situação de assinalamento de valores, ou seja, o procedimento de implicação precisa determinar todos os valores de sinal necessários para a consistência de um dado assinalamento das variáveis. Para descrever o procedimento, é necessário a definição dos termos portas justificadas e portas não justificadas [19]. É assumido o alfabeto lógico ternário (0, 1, X) onde X é o valor *don't care*. Um sinal é chamado especificado se ele é assinalado para os valores lógicos 0 ou 1; ele é não especificado se ele tem o valor X.

Definição 1 Dado uma porta lógica g em uma rede combinacional que tem pelo menos um sinal de entrada e saída especificado e o valores assinalados a g são logicamente consistentes: a porta g é chamada não justificada se existe uma ou mais combinações de valores de assinalamento nos sinais de entrada ou saída não especificados de g, que leva a um conflito em g. Se este valor não existir, então g é dita justificada.

Considere uma porta lógica AND de 2 entradas cuja saída é 0. Se uma das entradas é assinalada para 1 e a outra não é especificada, então a porta lógica é não justificada porque haveria um conflito caso o valor 1 fosse assinalado para a entrada não especificada. No entanto, se a entrada especificada fosse 0, a porta lógica seria justificada pois nenhum conflito iria ocorrer com o assinalamento da outra entrada. O termo *linha não justificada* é normalmente utilizado no caso especial de uma porta não justificada, onde o sinal especificado está na saída da porta lógica. Portas lógicas não justificadas descrevem as localizações no circuito onde os valores devem ser injetados durante o processo de aprendizado recursivo para determinar as suas conseqüências lógicas. A

próxima definição determina quais valores de sinal devem ser injetados nas portas não justificadas.

Definição 2 Sejam $f_1, f_2, ..., f_n$ sinais de entrada ou saída não especificados da porta lógica não justificada g, e sejam $V_1, V_2, ..., V_n$ os valores lógicos que especificam cada um deles. O conjunto de assinalamentos de sinal, $J = f_1 = V_1, ..., f_n = V_n$ é chamado justificativa de g se a combinação de valores em J justifica g.

Em cada porta lógica não justificada é necessário examinar as conseqüências lógicas para um *conjunto completo de justificativas.*

As implicações *diretas* têm um papel importante neste processo. As implicações diretas referem-se à avaliação do conjunto de valores de assinalamento em cada porta lógica que possui um evento e a propagação deste assinalamento de valores de acordo com a conectividade do circuito. Eventos são variáveis cujos valores foram alterados durante o processo de implicação para um dado conjunto de valores de assinalamento S, incluindo sinais de saída de portas não justificadas.

As implicações indiretas são realizadas a partir de aprendizado. Aprendizado são justificativas para portas lógicas não justificadas, injetadas temporariamente para examinar a sua conseqüência lógica.

Realizando o processo de encontrar implicações diretas e indiretas recursivamente é possível determinar todos os assinalamentos necessários. Desta forma, dados dois circuitos a serem verificados, é inserida uma porta lógica XOR tendo como entradas as saídas de cada um dos circuitos. A partir do assinalamento da saída do XOR para 0 (caso em que os circuitos são equivalentes), é aplicada a técnica de aprendizagem recursiva para verificar se este assinalamento é possível.

Esta abordagem possui complexidade exponencial de tempo no pior caso. Em circuitos de maior porte de aplicação industrial, pode ser necessária a aplicação de técnicas de otimização[20].

Um exemplo da aplicação de aprendizagem recursiva em Equivalence Checking pode ser visto na Figura 2.8.

Na Figura 2.8 são mostrados dois circuitos a serem verificados, nas caixas cinzas. Estes circuitos fazem parte de um circuito maior, *miter*, que foi criado através da introdução de uma porta lógica XOR que é alimentada pela saídas dos circuitos a serem verificados. Para realizar Equivalence Checking entre estes dois circuitos, é necessário verificar se a saída do XOR, com o nome z é igual a um. Em caso positivo, os circuitos não são equivalentes.

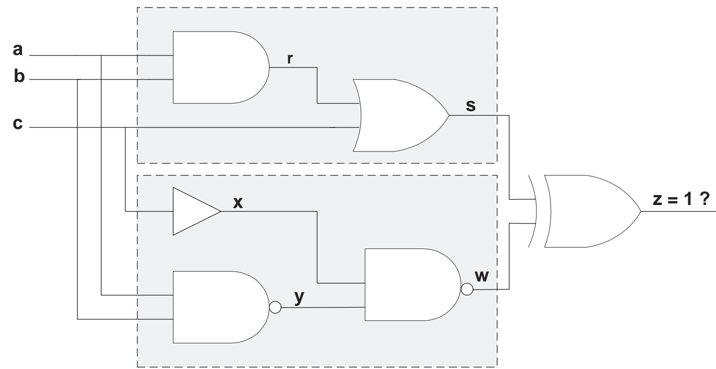


Figura 2.8: Exemplo de verificação por aprendizagem recursiva

Para que a variável z assuma o valor 1, existem duas possibilidades de assinalamentos: $\{s = 0, w = 1\}$ ou $\{s = 1, w = 0\}$. É necessário analisar as duas opções:

1. **Análise da justificativa { s = 0 , w = 1 }:**
   $s = 0$ implica em $a = 0$ e $r = 0$
   $a = 0$ implica em $x = 0$ e justifica $w = 1$
   As justificativas $r = 0, a = 0$ e $b = 0$ não alteram a consistência.

Desta forma, a justificativa { s = 0 , w = 1 } é consistente e os circuitos não são funcionalmente equivalentes.

2. **Análise da justificativa { s = 1 , w = 0 }:**
   $w = 0$ implica em $x = 1$ e $y = 1$
   $a = 1$ implica em $x = 1$ e justifica $s = 1$
   As justificativas $y = 1, a = 0$ e $b = 0$ não alteram a consistência.

Desta forma, a justificativa { s = 1 , w = 0 } é consistente e os circuitos não são funcionalmente equivalentes.

Como foram analisadas as duas alternativas e elas são consistentes entre si, os dois circuitos analisados não são equivalentes.

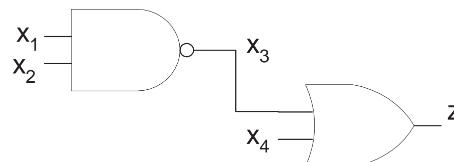
SAT (Satisfabilidade Proposicional)

Satisfabilidade Proposicional (SAT) é o problema de decidir se, para uma determinada fórmula booleana, é possível encontrar um conjunto de assinalamentos para as variáveis tal que a fórmula seja avaliada como *Verdadeira* ou provar que

tal assinalamento não existe.

O problema SAT é um dos problemas NP-Completo mais estudados atualmente devido a sua grande importância tanto em pesquisas teóricas quanto em aplicações práticas. Originalmente motivado pelo trabalho de T. Larrabee, em geração de testes automáticos [21], os modelos e técnicas em SAT vem sendo aplicados em testes de falha por atraso, Equivalence Checking combinacional, cálculo de atraso em circuitos, síntese lógica e geração de vetores funcionais, dentre outras aplicações. Além disto, SAT também desempenha um papel central na resolução de instâncias de BCP (binate covering problems) [22, 23, 24, 11] em particular em instâncias onde as restrições são difíceis de satisfazer, como por exemplo, no cálculo de padrões de teste de tamanho mínimo [24]. SAT também desempenha um papel chave em outros domínios, incluindo por exemplo Inteligência Artificial [25, 26]e Pesquisa Operacional [27]. Os recentes avanços nos algoritmos SAT trouxeram melhorias consideráveis que vêm sendo validadas em diferentes áreas de aplicação [25, 28, 26].

Com relação as aplicações de SAT em CAD, na maioria dos casos, a formulação do problema original inicia-se com a descrição do circuito, a partir de uma dada propriedade do circuito que necessita ser validada. No caso de Equivalence Checking, a propriedade a ser validada é a saída do XOR entre os dois circuitos. A formulação do circuito resultante é então mapeada em uma instância SAT, utilizando fórmulas na forma conjuntiva normal (CNF).



$$(x_1 + x_3) \cdot (x_2 + x_3) \cdot (\overline{x_1} + \overline{x_2} + \overline{x_3}) \cdot$$
$$(\overline{x_3} + z) \cdot (\overline{x_4} + z) \cdot (x_3 + x_4 + \overline{z})$$

a) Assinalamentos consistentes

$$(x_1 + x_3) \cdot (x_2 + x_3) \cdot (\overline{x_1} + \overline{x_2} + \overline{x_3}) \cdot$$
$$(\overline{x_3} + z) \cdot (\overline{x_4} + z) \cdot (x_3 + x_4 + \overline{z}) \cdot (\overline{z})$$

b) Com a propriedade z = 0

Figura 2.9: Exemplo de circuito e fórmula CNF

A fórmula CNF para um circuito combinacional é a conjunção das fórmulas

CNF para cada porta de saída do circuito, onde a fórmula CNF de cada porta denota um assinalamento válido de entrada-saída para a porta. Na Figura 2.9 pode ser visto o exemplo de um circuito associado a uma fórmula CNF e a especificação de uma propriedade. A derivação de fórmulas CNF para portas lógicas simples pode ser encontrada em [21]. Se a fórmula CNF para a porta lógica for vista como um conjunto de cláusulas, então a fórmula CNF j para o circuito é definida como o conjunto união (ou conjunção) das fórmulas CNF para cada porta lógica. Desta forma, dado um circuito combinacional, é possível criar a fórmula CNF para o circuito, bem como provar uma dada propriedade do circuito.

Os algoritmos SAT operam sobre fórmulas CNF e conseqüentemente podem ser aplicados para a resolução de instâncias de SAT associadas a circuitos combinacionais. Exemplos de instâncias SAT associadas a circuitos combinacionais incluem fórmulas CNF para geração de padrões de teste [21] e expressões de cálculo de atrasos em circuitos [29].

A organização geral de um algoritmo genérico de SAT pode ser visto na Figura 2.1. Este algoritmo SAT genérico captura a organização de vários dos algoritmos mais competitivos existentes atualmente [25, 28, 26].

O algoritmo conduz uma busca, através do espaço de assinalamentos possíveis, para as variáveis da instância do problema. Em cada estágio da busca, um assinalamento de variável é selecionado com a função `Decide()`. Um nível de decisão d é associado com cada seleção de uma assinalamento. Todos os assinalamentos implicados a partir da seleção são identificados com a função `Deduz()`, que corresponde à derivação direta de implicações [25, 28]. Todas as vezes que uma cláusula torna-se insatisfeita, a função `Deduz()` retorna uma indicação de conflito que é analisada pela função `Diagnostico()`. O diagnóstico de um dado conflito retorna o nível de decisão de *backtracking*, ou seja, o nível de decisão para o qual o processo deve recuar. A função `Apague()` apaga todos os assinalamentos implicados resultantes de cada seleção de assinalamento. Organizações diferentes de algoritmos SAT podem ser modeladas por este algoritmo genérico. Atualmente, a maioria dos algoritmos SAT eficientes são caracterizados por várias das propriedades chave a seguir:

1. A análise do conflito pode ser usada para implementar estratégias de busca para *backtracking* não cronológico. Portanto, seleções de assinalamento que sejam consideradas irrelevantes podem ser descartadas durante a busca [25, 28, 26]

```
 1 // Argumentos de entrada:   nivel de decisao atual d
   // Argumentos de saida:     nivel de de decisão de backtrack b
   // Valor de retorno:        SATISFAZIVEL ou NAO_SATISFAZIVEL
   //
   SAT (d, &b ) {
 6      If (Decide (d) != DECISAO)
            return SATISFAZIVEL;
        while (TRUE) {
            if (Deduz (d) != DECISAO) {
                if (SAT (d + 1, b)  ==  SATISFAZIVEL)
11                  return SATISFAZIVEL;
                else if (  != d || d == 0) {
                    Apague (d);
                    return NAO_SATISFAZIVEL;
                }
16          }
            if ( Diagnostico (d, b) == CONFLITO) {
                return NAO_SATISFAZIVEL;
            }
        }
21 }
```

Listagem 2.1: Algoritmo genérico para SAT com busca e backtracking

2. A análise de conflito pode também ser usada para identificação e armazenagem de novas implicações de funções booleanas associadas com a fórmula CNF. A armazenagem de cláusulas possui um papel importante nos algoritmos SAT mais recentes porém, na maioria dos casos, as cláusulas grandes que são armazenadas são eventualmente descartadas [25, 28]

3. Outras técnicas vêm sendo desenvolvidas. Aprendizagem baseada em relevância [25] aumentam a vida de cláusulas grandes armazenadas que serão eventualmente removidas. Assinalamentos necessários induzidos por Conflito[28] denotam assinalamentos para variáveis que são necessários para prevenir a ocorrência de um dado conflito durante a busca.

A utilização de modelos CNF e algoritmos SAT tem importantes vantagens:

1. Existência de algoritmos SAT extensivamente validados que podem ser utilizados, eliminando a necessidade de algoritmos dedicados.

2. Novas melhorias e novos algoritmos SAT podem ser facilmente aplicados a cada aplicação.

Por outro lado, a utilização de fórmulas CNF e algoritmos SAT associados também são caracterizados por algumas desvantagens:

1. Como observado em [30], as informações estruturais do circuito, muitas vezes de grande importância são perdidas.

2. Em muitos problemas EDA (*Electronic Design Automation*), um grande número de instâncias SAT têm que ser resolvidas para cada circuito. Portanto, o mapeamento de uma dada descrição do problema em SAT, pode representar uma grande porcentagem do tempo de execução total.

3. Técnicas poderosas de raciocínio baseadas no circuito, tal como aprendizado recursivo [31, 32], não são facilmente aplicadas.

Capítulo 3

Núcleo inteligente para Equivalence Checking

Este capítulo é dedicado a metodologia proposta no presente trabalho e ao núcleo inteligente desenvolvido para processamento distribuído de resolvedores SAT. Na Seção 3.1 é apresentada em detalhes a metodologia de verificação proposta e a Seção 3.2 é dedicada aos trabalhos relacionados à metodologia proposta. A Seção 3.3 apresenta o núcleo inteligente desenvolvido durante esta dissertação de mestrado.

3.1 Metodologia proposta

A metodologia proposta no presente trabalho permite a verificação de circuitos integrados descritos em linguagens de verificação de hardware utilizando resolvedores SAT.

Na metodologia desenvolvida, o circuito a ser verificado é preprocessado de forma a classificar o grau de intersecção entre os cones de lógica particionados. Esta etapa de preprocessamento permite decidir o tratamento que será dado aos circuitos:

- Quando há um grande compartilhamento entre os cones de lógica, é utilizado um aprendizado booleano através de iterações de instâncias SAT.
- No caso de não haver compartilhamento entre os cones de lógica, é utilizado processamento distribuído.

Na Figura 3.1, é apresentado o gráfico mostrando a relação entre o grau de compartilhamento e o grau de intersecção dos circuitos a serem verificados.

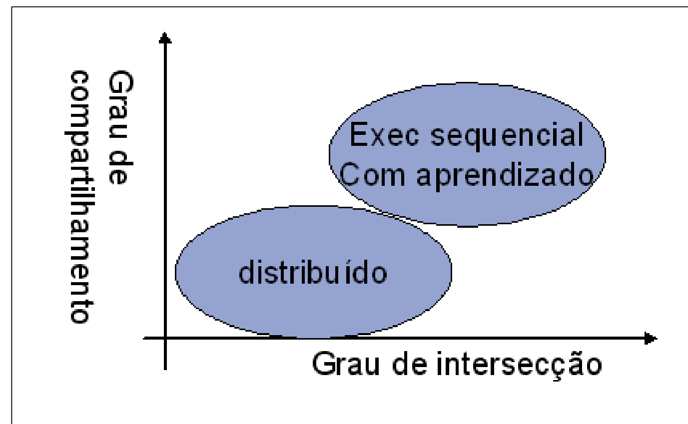


Figura 3.1: Gráfico de correlação

Quando há utilização de aprendizado booleano, as características dos circuitos são analisados e a partir desta análise são realizados partições no circuito (baseado em critérios de largura e ciclos). Cada partição é submetida a um nó resolvedor SAT, que identifica similaridades estruturais entre os circuitos. As similaridades estruturais encontradas são utilizadas para realimentar as partições do circuito, restringindo o espaço de soluções a ser buscado. Este processo é repetido iterativamente até que todo o circuito tenha sido verificado ou até que seja encontrada um prova de que os circuitos não são equivalentes.

Na Figura 3.2, é apresentado um fluxo típico de verificação na metodologia proposta.

Inicialmente os circuitos a serem verificados são submetidos a uma análise estrutural que permite decidir qual critério de particionamento deve ser adotado. Os critérios de particionamento utilizados na abordagem proposta são baseados em *ciclo* e *largura*. O critério baseado em ciclo particiona o circuito em ciclos de *clock*, separando lógicas ativadas em diferentes ciclos. O critério baseado em largura particiona o circuito de forma hierárquica, separando-o por conjunto de bits. Para que seja possível realizar particionamento baseado no critério de largura, o problema a ser verificado deve apresentar a característica de inclusão estrutural. Um exemplo deste tipo de circuito é um multiplicador CLA (do inglês, *Carry Look-Ahead*). Neste tipo de multiplicador, o *carry* possui uma lógica própria de computação sendo que o *carry* do bit i é calculado com base no *carry* do bit $i-1$. Esta característica permite que o circuito seja particionado em largura, separando a lógica por bits de saída, onde cada bit irá depender apenas da lógica relacionada ao bit anterior.

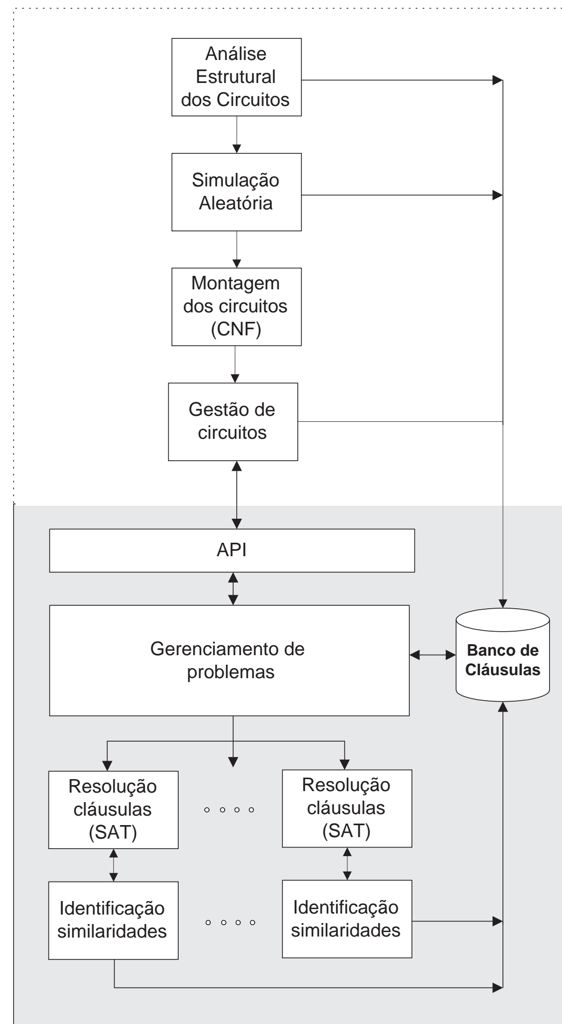


Figura 3.2: Metodologia proposta para Equivalence Checking

Durante a fase de análise estrutural, caso sejam identificadas linhas de sinais idênticas do circuito, estas linhas são codificadas em cláusulas de restrição (representando a similaridade encontrada) e são inseridas no banco de cláusulas. O banco de cláusulas é responsável por armazenar todas as similaridades estruturais detectadas durante o processo de verificação. Estas cláusulas podem ser utilizadas na construção de partições posteriores do problema, restringindo o espaço de busca do resolvedor e conseqüentemente a complexidade do problema que está sendo verificado.

O segundo passo da metodologia é aplicar simulação paralela aleatória aos circuitos, permitindo a identificação de pares sinais candidatos à similaridade estrutural entre as descrições [33, 34]. Estes pares podem dar origem a mais algumas cláusulas de restrição, que são inseridas no banco de cláusulas.

O terceiro passo da metodologia é realizar a montagem dos circuitos em cláusulas na forma CNF (do inglês, *Conjunctive Normal Form*). Os sinais de entradas das duas descrições são mapeados, garantindo que os sinais de entrada correspondentes de cada descrição sejam representados pelas mesmas variáveis. Em seguida, as descrições são convertidas para cláusulas na forma CNF [35]. Nesta etapa de conversão são inseridas as portas lógicas necessárias para a formação do *miter* a ser verificado, como apresentado na Seção 2.1.1.

Uma vez particionados e montados os circuitos a serem verificados, as cláusulas são distribuídas entre os nós previamente selecionados. O núcleo proposto possui uma API que permite ao desenvolvedor ou aplicativo de camada superior visualizar e selecionar os nós ativos na rede para resolução do problema. Cada nó ativo na rede executa uma instância do resolvedor SAT.

O resolvedor SAT realiza o processamento das cláusulas de entrada. Durante o processo de resolução SAT, são geradas cláusulas de conflito que direcionam a busca do resolvedor para espaços de solução ainda não investigados. Estas cláusulas de conflito são utilizadas posteriormente pelo módulo de identificação de similaridades estruturais para encontrar sinais internos equivalentes dos circuitos.

As similaridades estruturais encontradas são convertidas para o formato apropriado e inseridas no banco de cláusulas, que é utilizado como base de conhecimento para a próxima partição do circuito.

O processo central da metodologia de verificação proposta neste trabalho é composta por três tarefas principais:

- Submissão de partições dos circuitos aos resolvedores SAT;
- Identificação de similaridades estruturais entre os circuitos a partir de cláusulas de conflito geradas pelo resolvedor SAT;
- Realimentação das partições posteriores do circuito com as similaridades estruturais identificadas.

Este processo é repetido iterativamente até que todo o circuito tenha sido verificado ou até que tenha sido encontrada uma contra-prova de que os circuitos verificados não são equivalentes.

A parte colorida em cinza na Figura 3.2 representa o núcleo inteligente implementado no presente trabalho. A Seção 3.3 é dedicada a sua apresentação.

3.2 Trabalhos relacionados

Os métodos de verificação baseados em resolvedores SAT tem surgido como uma solução promissora para o problema de verificação. Melhorias significativas na tecnologia SAT na última década levaram ao desenvolvimento de resolvedores SAT poderosos [36, 37, 28, 26]. Os métodos de verificação baseados nestes resolvedores estão se mostrando bastante eficientes em termos de capacidade e efetividade, como é possível de constatar em diversos estudos de caso industriais e acadêmicos [38, 39, 40].

Várias das técnicas propostas para verificação baseadas em SAT utilizam características inerentes ao SAT (tais como cláusulas geradas e processos de implicação internos) para tentar reduzir a complexidade do problema a ser resolvido. O objetivo é tentar reduzir o espaço de soluções a ser verificado, de forma a tornar as técnicas de verificação mais eficientes e poderosas.

As técnicas baseadas em abstração são um exemplo de técnicas que utilizam SAT para diminuir a complexidade dos projetos a serem verificados. Abstração é a tarefa de remover informações do sistema que não são relevantes para a solução do problema e é bastante utilizada em Verificação de Modelos (*Model Checking*) de sistemas de grande escala. Nesta seção serão discutidas duas abordagens de verificação que utilizam abstração: abstração baseada em contra-exemplos e abstração baseada em provas.

A abordagem de abstração baseadas em contra-exemplos, também conhecida como CBA (do inglês, *Counterexample-based abstraction*) têm o objetivo de manipular projetos muito grandes. Nesta abordagem, são encontrados contra-exemplos para pequenos modelos abstratos de projetos concretos e eles são utilizados para refinar os projetos concretos iterativamente até que um resultado seja obtido pela Verificação de Modelos ou que os recursos disponíveis tenham terminado [41]. Uma das primeiras tentativas de utilização de resolvedores SAT em refinamentos por abstração baseada em contra-exemplos foi descrita por Clarke em [42]. Na abordagem descrita, o resolvedor SAT é utilizado para verificar se o caminho encontrado por um contra-exemplo durante a verificação de modelos é real ou não. Isto é realizado verificando a satisfabilidade do contra-exemplo no projeto concreto. Se o caminho não for real, são utilizadas técnicas de aprendizado de máquinas e ILP (do inglês, *Integer Linear Programming*) para realizar o refinamento. Em um esforço subseqüente [43], foram utilizadas técnicas baseadas em SAT para realizar o refinamento. Foram

propostas heurísticas que utilizam as pontuações de decisões do processo de resolução SAT para escolher *latches* candidatos para refinamento. Neste caso, a tarefa é identificar um conjunto de *latches* capaz de excluir o contra-exemplo não real.

A abordagem de abstração baseadas em provas, também conhecida como PBA (do inglês, *Proof-based Abstraction*), foi inicialmente proposta por [44]. Esta abordagem explora uma das características dos resolvedores SAT: no caso de não-satisfabilidade do problema, o resolvedor pode gerar uma prova da não satisfabilidade. Em Verificação de Modelos com Fronteiras (*Bounded Model Checking*, isto corresponde a prova de que não existe um contra-exemplo para a propriedade de k passos ou menos. Mesmo que em geral isto não implique em nada para a propriedade, é possível utilizar esta prova para identificar as restrições que são importantes para a propriedade (pelo menos nos k primeiros passos). A partir desta identificação, é gerada uma abstração guiada, que pode ser usada para verificar completamente a propriedade utilizando os métodos padrões de verificação de modelos. A diferença entre a abstração baseada em provas e a abstração baseada em contra-exemplos é que na técnica baseada em contra-exemplos, os contra-exemplos gerados pelo verificador de modelos são ignorados. No caso da técnica baseada em provas, a abstração é baseada provas fornecidas pelo resolvedor SAT e não em refutar contra-exemplos. Esta abordagem possui a vantagem de rejeitar todos os contra-exemplos até um determinado tamanho, ao invés de utilizar apenas um contra-exemplo produzido pelo verificador de modelos.

Ainda em verificação de modelos com fronteiras, em [45] é proposta uma metodologia baseada em resolvedores SAT e processamento distribuído. Na abordagem proposta, os modelos são particionados por heurísticas definidas e submetidos a resolvedores SAT de forma distribuída. Caso seja encontrada uma solução possível em uma das partições, as cláusulas geradas são replicadas para as outras partições do problema de forma a reduzir parte do espaço de busca por soluções. A restrição desta abordagem é que existe a necessidade de conhecimento prévio da estrutura do problema, já que as cláusulas geradas pelo resolvedor SAT são replicadas apenas para partições que possuam intersecção com a partição onde as cláusulas foram geradas.

3.3 Núcleo inteligente desenvolvido

O núcleo inteligente desenvolvido durante o presente trabalho é responsável por realizar a distribuição e gerenciamento dos problemas processados de forma distribuída e por realizar a interface com o desenvolvedor ou aplicativo de camada superior.

O núcleo inteligente é um *framework* flexível para Equivalence Checking, que permite que diferentes heurísticas de particionamento dos problemas sejam realizadas, além da possibilidade de utilização de diferentes resolvedores SAT e metodologias para identificação de similaridades. Ele possui interfaces de comunicação bem definidas que possibilitam a conexão com outros módulos de resolvedores e identificação de similaridades.

O núcleo foi construído a partir de estruturas de dados e comunicação otimizadas que reduzem o tempo de processamento relacionado a comunicação dos dados, aumentando o desempenho da tarefa de verificação.

As principais características do núcleo inteligente são :

- API bem definida de interação com o usuário ou aplicativo de camada superior;
- Seleção de resolvedor SAT e módulo de identificação de similaridades a ser utilizado durante a verificação;
- Ajuste de limites de memória e tempo a serem utilizados pelo resolvedor SAT;
- Seleção do conjunto de máquinas a serem utilizadas na resolução do problema
- Balanceamento de carga entre os nós utilizados para resolução do problema;
- Operações de verificação de estado e interrupção de problemas submetidos.

O núcleo inteligente é composto por 5 módulos: API, Módulo de Gerenciamento de problemas, Resolvedor SAT, Módulo de identificação de similaridades e banco de cláusulas. Nas próximas subseções será apresentada uma descrição de cada módulo e suas principais funcionalidades.

3.3.1 API

Durante o presente trabalho foi definida uma Interface de Programação de Aplicativos (API) para o núcleo inteligente que permite ao desenvolvedor ou aplicativo de uma camada superior interagir com o núcleo.

A API definida é composta por um conjunto de primitivas suportadas pelo núcleo inteligente que permite o total controle e gerenciamento dos problemas submetidos ao núcleo.

A submissão de problemas é realizada através de uma estrutura de dados bem definida que permite ao desenvolvedor ou aplicativo de camada superior controlar os parâmetros de resolução de cada problema. Os dados definidos para entrada são:

- Resolvedor SAT a ser utilizado;
- Limite de memória que poderá ser utilizada pelo resolvedor;
- Limite de tempo de CPU que poderá ser utilizado pelo resolvedor;
- Módulo de similaridades a ser utilizado na resolução;
- Número de variáveis que compõe o problema;
- Número de cláusulas que compõe o problema;
- Cláusulas na forma CNF que constituem o problema.

3.3.2 Módulo Gerenciamento de Problemas

O módulo Gerenciamento de Problemas é a parte central do núcleo inteligente desenvolvido. Ele é responsável por suportar as operações exportadas através da API além de realizar interface com os resolvedores SAT disponíveis para resolução do problema.

Inicialmente os problemas são recebidos e o módulo Gerenciamento de Problemas verifica se existe alguma máquina estabelecida para resolver o problema. Caso não exista, é realizado o balanceamento de carga entre as máquinas disponíveis e o problema é submetido ao nó com menor carga.

Em seguida é verificado se o resolvedor SAT e módulo de identificação de similaridades estão presentes na máquina escolhida. Em caso positivo o módulo instancia a API com o resolvedor SAT, passando o problema a ser resolvido, limites eventualmente ajustados e o módulo de identificação de similaridades a ser utilizado.

O resolvedor SAT resolve o problema recebido e instancia a API com o módulo de similaridades estruturais, passando como argumento as cláusulas de conflito geradas durante o processo de solução. No momento em que todas as

similaridades estruturais estiverem identificadas, o nó retorna o resultado ao módulo Gerenciamento de Problemas. Este módulo, através da API definida, retorna o resultado para o desenvolvedor ou aplicativo de camada superior.

O módulo Gerenciamento de Problemas utiliza o paradigma computacional "Mestre e Escravo", onde este módulo é o mestre e as máquinas que rodam instâncias dos resolvedores SAT e do módulo de identificação de similaridades são escravos.

3.3.3 Resolvedores SAT

O objetivo do resolvedor SAT é encontrar um assinalamento verdadeiro de variáveis para um conjunto de cláusulas proposicionais na forma CNF, como foi apresentado na Seção 2.1.2. Especificamente no caso de Equivalence Checking, dentro da metodologia proposta, encontrar um assinalamento válido significa encontrar a equivalência de dois circuitos.

O núcleo inteligente definido suporta a integração com diversos resolvedores SAT que utilizam como entradas cláusulas na forma CNF, agrupadas em arquivos no formato DIMACS [46].

Existem um grande número de resolvedores SAT disponíveis ao público atualmente. A Tabela 3.1 apresenta alguns dos resolvedores mais eficientes conhecidos atualmente. Estes resolvedores participaram da Competição SAT 2005 [47], que é um concurso que vem sendo realizado desde 2002 e acontece junto com o Simpósio Internacional de Teoria e Aplicação de Provas de Satisfabilidade.

O propósito desta competição é identificar novos *benchmarks* desafiadores e promover novos resolvedores SAT assim como compará-los com resolvedores considerados no estado-da-arte. A competição é toda realizada utilizando o sistema SAT-Ex [48].

Os benchmarks aos quais os resolvedores são submetidos na Competição SAT podem ser agrupados em três grandes grupos:

Problemas industriais criados a partir de aplicações industriais reais.

Problemas aleatórios criados a partir de algum critério mais geral, tal como um número fixo de variáveis e cláusulas.

Problemas preparados manualmente criados cuidadosamente buscando explorar características específicas das fórmulas.

A Tabela 3.1 apresenta os resultados obtidos pelos 10 melhores resolvedores em 2005. Na primeira parte da Tabela, são apresentados o nome do resolvedor, o número de problemas as quais o resolvedor foi submetido e o número de problemas que foram resolvidos de forma correta. Em seguida, dos problemas resolvidos de forma correta, é apresentados a distribuição por categoria de problemas (*benchmarks* gerados de forma aleatória, manualmente e criados a partir de aplicações reais industriais). Maiores detalhes sobre os resultados e referências aos resolvedores podem ser encontrados em [49].

Nome	**Tentativas**	**Resolvidos**	**Aleatórios**	**Manuais**	**Industriais**
SatELiteGTI	1657	768	54	312	402
minisat_static	1657	731	56	299	376
csat	1657	641	27	271	343
vallst.sh	1657	623	17	324	282
zchaff_rand	1657	611	12	276	323
HaifaSat	1657	610	3	251	356
Jerusat1.31_B	1657	596	11	267	318
Jerusat1.31_A	1657	596	7	276	313
HaifaSat2	1657	594	5	249	340
zchaff	1657	584	16	266	302

Tabela 3.1: Resultado dos dez melhores resolvedores SAT na Competição SAT 2005

3.3.4 Módulo de Identificação de similaridades

O módulo de identificação de similaridades é responsável por encontrar similaridades estruturais entre os circuitos que estão sendo verificados. As similaridades encontradas podem ser utilizadas para diminuir o espaço de busca dos resolvedores SAT, diminuindo a complexidade do problema a ser resolvido.

Durante o presente trabalho foi desenvolvido um módulo de identificação de similaridades baseado em cláusulas de conflito geradas pelo resolvedor SAT durante o processamento do problema. O Capítulo 4 é dedicado ao módulo de identificação de similaridades desenvolvido.

Capítulo 4

Explorando as similaridades estruturais

Na maioria dos ambientes de projetos de circuitos, Equivalence Checking é aplicada para estabelecer a equivalência entre duas descrições de um circuito. Tipicamente uma descrição é derivada de outra através de vários passos, que são partes do processo de projeto do circuito. Muitos destes passos possuem um pequeno efeito na estrutura global do circuito. Desta forma, é bastante provável que existam similaridades estruturais entre os circuitos a serem verificados, ou seja, não somente as saídas são funcionalmente equivalentes mas alguns sinais internos também. Por exemplo, quando é verificada a exatidão de um processo de mapeamento de tecnologia, ambos os circuitos têm uma estrutura global muito parecida. Outro exemplo é a verificação de um circuito que teve alterações manuais realizadas pela equipe de projetistas, onde apenas pequenas partes do circuito foram provavelmente alteradas.

As similaridades estruturais possuem um papel de grande importância em Equivalence Checking. Algumas vezes é muito difícil verificar todo o *miter* (circuitos a serem verificados com as saídas alimentando uma porta lógica XOR) por completo, devido ao seu tamanho. Já se constatou que o grau de dificuldade da Equivalence Checking depende, principalmente, do tamanho do *miter* e não do tamanho da lógica que o envolve[1]. Desta forma a identificação de similaridades estruturais pode ser bastante útil para o processo de Equivalence Checking. As técnicas que exploram estas similaridades reduzem a complexidade do problema a ser resolvido, identificando sinais do circuito idênticos. Se dois sinais dos circuitos a serem verificados são idênticos, estes sinais podem ser unidos e apenas uma das lógicas, associada aos sinais, deve ser verificada. Esta

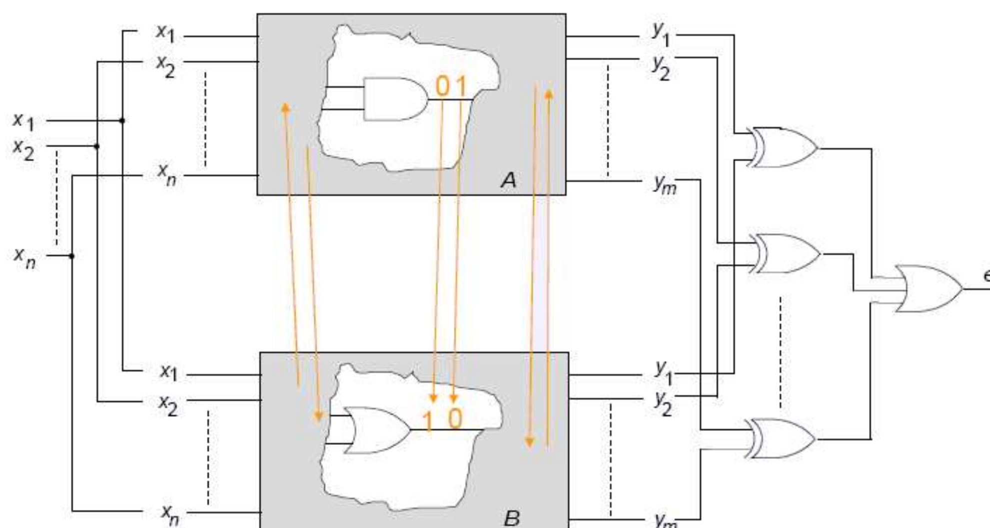


Figura 4.1: Circuitos contendo equivalências internas

redução na quantidade de lógica dos circuitos pode apresentar uma redução significativa no problema de verificação dos circuitos. Na Figura 4.1 pode ser visto um *miter*, composto por dois circuitos a serem verificados, que possuem pontos de equivalência estrutural internos.

Basicamente, o processo de explorar as similaridades estruturais entre os circuitos consiste de dois passos. Iniciando a partir das entradas primárias e movendo em direção às saídas:

1. Identifique equivalências entre os sinais internos do miter;

2. Substitua os nós correspondentes por seus equivalentes identificados.

Por fim, é necessário provar a equivalência dos circuitos provando a satisfabilidade da saída do miter.

Um exemplo de como as similaridades estruturais podem ser utilizadas para reduzir a complexidade dos circuitos a serem verificados é mostrado na Figura 4.2.

Inicialmente são mostrados os dois circuitos a serem verificados com suas saídas alimentando a porta lógica XOR, como pode ser visto na Figura 4.2(a). Ao lado direito é apresentada a primeira similaridade estrutural identificada, entre os sinais d e e. Como os sinais d e e são iguais, eles podem ser ligados e a lógica de maior complexidade associada aos sinais é removida. A Figura 4.2(b) mostra o circuito resultante da união dos dois sinais. Ao lado direito, é mostrado

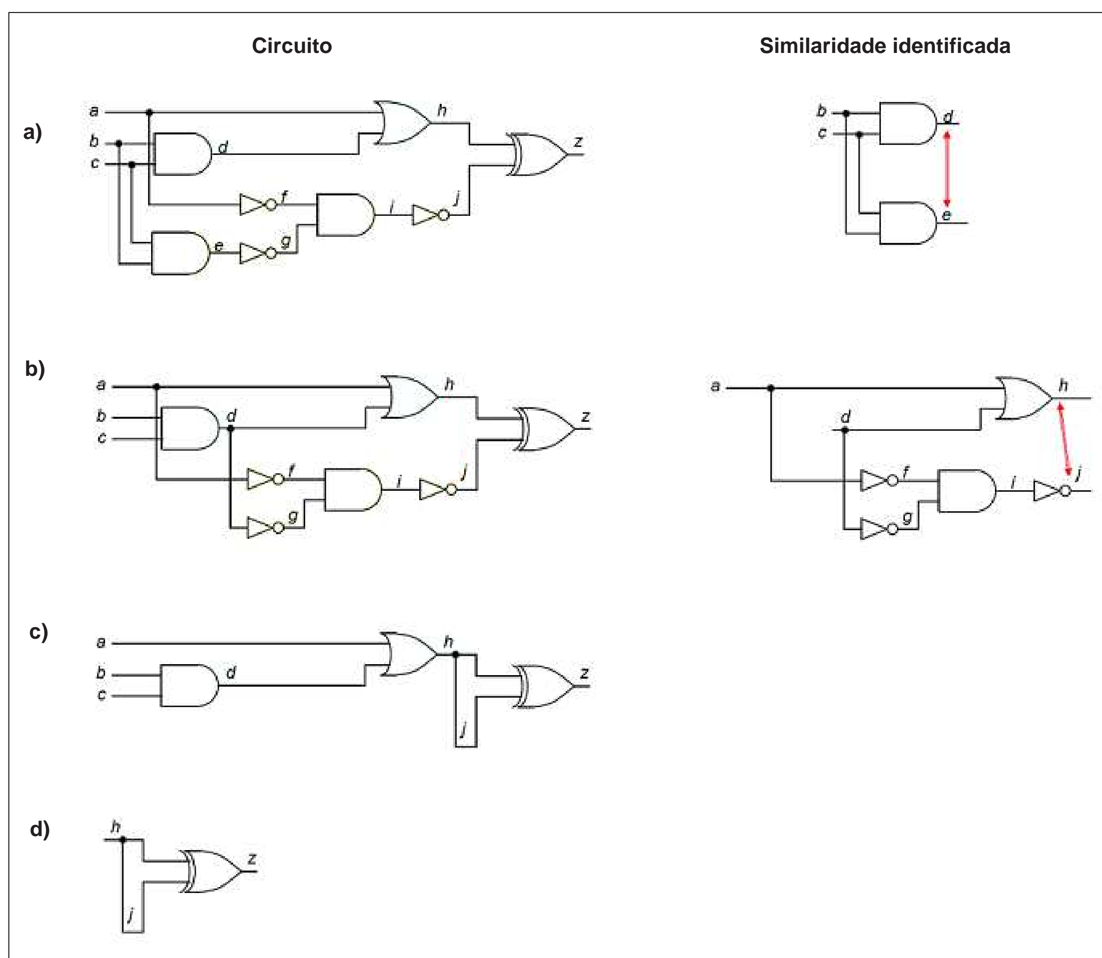


Figura 4.2: Utilização de similaridades estruturais internas entre os circuitos

a segunda similaridade estrutural identificada: os sinais h e j são idênticos. Seguindo o mesmo procedimento, os sinais h e j são ligados e novamente a lógica de maior complexidade associada aos sinais é suprimida. A Figura 4.2(c) apresenta o circuito resultante do último passo da identificação. Por fim, como as duas entradas da porta lógica XOR são alimentadas pelo mesmo sinal, é realizada uma nova simplificação do circuito que conduz ao circuito final após a identificação de todas as similaridades estruturais internas, como mostrado na Figura 4.2(d).

Diversos métodos foram propostos para realizar a identificação de similaridades estruturais entre os circuitos, dentre eles:

- Aplicar padrões randômicos para determinar candidatos para pares de sinais equivalentes e então provar a equivalência através de SAT ou ATPG [1];

- Identificar implicações indiretas e provar variáveis utilizando técnicas de apren-

dizado recursivo [32]

- Utilizar BDDs locais para identificar partições do circuito relevantes[50][51][52].

Neste trabalho é proposta uma nova metodologia para identificação de similaridades estruturais, baseado em cláusulas de conflito geradas durante a verificação dos circuitos pelo resolvedor SAT.

Algumas das vantagens da abordagem proposta são:

- Não é necessário um conhecimento prévio da estrutura do circuito. Baseado nas cláusulas CNF geradas a partir do circuito, o resolvedor gera, durante o seu curso, cláusulas de conflito.

- A metodologia aproveita as cláusulas geradas a partir da resolução para encontrar as similaridades, ou seja, o aumento de tempo introduzido por esta metodologia é menor do que em outras técnicas.

As próximas subseções são dedicadas à apresentação da metodologia.

4.1 Metodologia de identificação de similaridades estruturais a partir de cláusulas de conflito

A metodologia proposta utiliza cláusulas de conflito geradas pelo resolvedor SAT para identificar similaridades estruturais entre os circuitos a serem verificados.

Para permitir o perfeito entendimento da metodologia, é necessário conhecimento sobre os resolvedores SAT e a maneira como estes geram as cláusulas de conflito utilizadas para a identificação das similaridades estruturais. Portanto, a Seção 4.1.1 apresenta uma breve explicação sobre o processo de resolução SAT e a geração de cláusulas de conflito. A Seção 4.1.2 é dedicada à apresentação da metodologia.

4.1.1 Geração de cláusulas de conflito em resolvedores SAT

A organização geral dos algoritmos de resolução SAT foram apresentados no Capítulo 2. Nesta seção será aprofundado um pouco mais sobre a geração das cláusulas de conflito e o seu significado dentro do processo de resolução SAT.

Inicialmente é importante entender o conceito de grafo de implicação, pois é a partir dele que as cláusulas de conflito são geradas.

Durante o processo de resolução SAT, as relações de implicação de assinalamento de variáveis são normalmente expressas como um grafo de implicação. Um grafo de implicação típico é mostrado na Figura4.3.

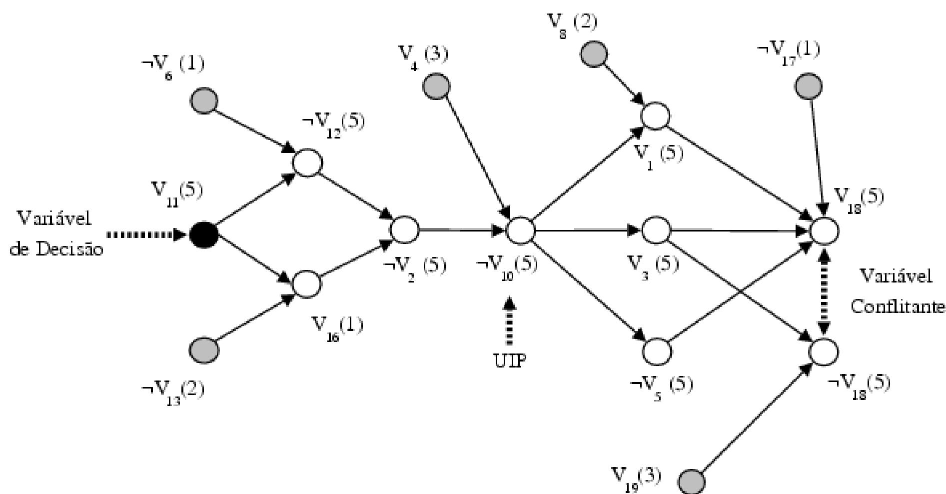


Figura 4.3: Um grafo de implicação típico do processo de resolução SAT

Um grafo de implicação é um grafo acíclico direto onde cada vértice representa um assinalamento de variável. Uma variável é considerada positiva quando o valor 1 foi assinalado para ela. Por outro lado uma variável é considerada negativa quando ela teve o seu valor assinalado para 0. As arestas incidentes em cada vértice do grafo de implicação representam as razões ou justificativas pelas quais a variável foi assinalada para o determinado valor. Os vértices que possuem arestas diretas para outros vértices são chamados de vértices antecedentes. Já os vértices que não possuem arestas incidentes, ou seja, vértices que tiveram os seus valores assinalados diretamente são chamados de vértices de decisão. Cada variável possui um nível de decisão associado a ela, que pode ser visto no grafo entre parênteses, ao lado da variável. A construção deste grafo é parte do processo de resolução SAT, onde assinalamentos são realizados diretamente, resultando em assinalamentos indiretos posteriores. Quando um vértice do grafo (uma variável) aponta para os dois assinalamentos de uma mesma variável (0 e 1), é dito que um conflito ocorreu. A variável que está sendo apontada é chamada de variável de conflito. Na Figura 4.3, pode ser visto uma variável de conflito, com o nome V18. Neste caso, quando ocorre um conflito, é necessário

realizar a análise deste conflito.

Análise de conflito é um procedimento que investiga a razão pela qual o conflito ocorreu e tenta resolvê-la. A análise de conflito indica ao resolvedor SAT que em um determinado espaço de busca não existe uma solução possível e indica um novo espaço de busca para continuar com a procedimento de busca pela solução.

Um dos métodos mais simples para análise de conflito consiste em armazenar, para cada variável de decisão, um rótulo indicando se houve alguma tentativa de assinalamento da variável nas duas fases. Quando um conflito ocorre, o procedimento de análise de conflito procura pela variável de decisão com o maior nível de decisão que não foi marcada anteriormente, marca esta variável e desfaz todos os assinalamentos feitos entre o nível de decisão desta variável e o nível de decisão atual. A partir daí, o procedimento tenta assinalar a outra fase para a variável de decisão. Este tipo de análise de conflito apresenta bons resultados para problemas gerados randomicamente, possivelmente porque este tipo de problemas não tem uma estrutura bem definida e aprender de determinadas partes do espaço de busca não irá em geral ajudar na busca de outras partes do espaço de soluções.

No entanto, para problemas reais, este tipo de análise não apresenta bons resultados. Existem técnicas bem mais avançadas para realizar análise de conflito, que utilizam o grafo de implicações para determinar a razão dos conflitos. Este tipo de abordagem é chamada *conflict directed backjumping*[25] porque ela consegue analisar mais de um nível na pilha de decisão.

Durante a análise de conflitos, o procedimento responsável insere algumas cláusulas no banco de cláusulas. Este processo é chamado de aprendizado de cláusulas. As cláusulas aprendidas são chamadas de cláusulas de conflito e elas guardam informações deduzidas dos conflitos com o objetivo de evitar que o mesmo caminho seja escolhido pelo resolvedor no futuro. Estas cláusulas mostram que certas combinações de assinalamentos de variáveis não são válidas porque elas forçam a variável em conflito a assumir os valores 1 e 0, provocando um conflito durante a resolução.

As cláusulas de conflito são geradas a partir do particionamento do grafo de implicação em duas partes, onde uma das partes contém todas as variáveis de decisão (chamado de *lado da decisão*) e a outra contem as variáveis conflitantes (chamado *lado do conflito*). Todos os vértices do lado das variáveis de decisão que possuem pelo menos uma aresta para um vértice do lado das variáveis de

conflito representam a razão do conflito em questão. Um grafo de implicações biparticionado por ser visto na Figura 4.4.

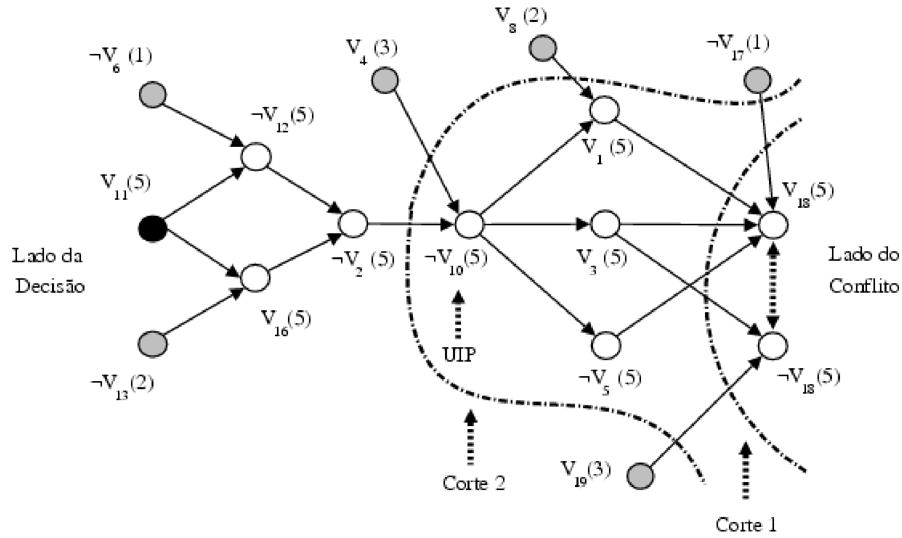


Figura 4.4: Cortes no grafo de implicações para geração de cláusulas de conflito

Para formar o biparticionamento, é necessário definir os cortes. Cortes diferentes correspondem a esquemas de aprendizado diferentes. Na figura 4.4 partir do corte 1, o conjunto de atribuições em 4.1 induz a um conflito.

$$\{V_1 = 1, V_3 = 1, V_5 = 0, V_{17} = 0, V_{19} = 1\} \tag{4.1}$$

A partir desta atribuição, é possível gerar as seguintes cláusulas de conflito:

$$Conflito \ = (V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19}) \tag{4.2}$$

$$\neg Conflito \ = \neg\,(V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19}) \tag{4.3}$$

$$Clausula\ de\ Conflito \ = (\neg V_1 \wedge \neg V_3 \wedge V_5 \wedge V_{17} \wedge \neg V_{19}) \tag{4.4}$$

4.1.2 Metodologia para identificação de similaridades estruturais

A metodologia proposta utiliza as cláusulas de conflito geradas durante o processo de resolução SAT para identificar as similaridades estruturais entre dois

circuitos. Estas similaridades realimentam as partições que serão submetidas posteriormente aos resolvedores SAT, diminuindo a complexidade do problema a ser verificado.

As similaridades estruturais são identificadas a partir de expressões booleanas compostas por até três variáveis. Cada expressão booleana identificada pode corresponder a um ponto no circuito em que existe uma equivalência de sinais.

As expressões booleanas de interesse para a metodologia proposta são apresentadas na Tabela 4.1.

Expressão booleana	**Combinação de cláusulas**
$V_1 \rightarrow V_2$	$(V_1^{'} + V_2)$
$V_1 = V_2$	$(V_1 + V_2^{'})\ (V_1^{'} + V_2)$
$V_1 = V_2^{'}$	$(V_1 + V_2)\ (V_1^{'} + V_2^{'})$
$V_3 = V_1 \wedge V_2$	$(V_3 + V_1^{'} + V_2^{'})\ (V_3 + V_1)\ (V_3 + V_2)$
$V_3 = V_1 \vee V_2$	$(V_3^{'} + V_1 + V_2)\ (V_3 + V_1^{'})\ (V_3 + V_2^{'})$
$V_3 = V_1 \otimes V_2$	$(V_3 + V_1 + V_2^{'})\ (V_3 + V_1^{'} + V_2)\ (V_3^{'} + V_1^{'} + V_2^{'})\ (V_3^{'} + V_1 + V_2)$

Tabela 4.1: Combinação de cláusulas para identificação de similaridades

A partir de agora, o termo expressão booleana será utilizado como sinônimo de similaridade estrutural. Ainda que exista a possibilidade da expressão booleana não corresponder a uma similaridade estrutural de interesse, esta questão somente será avaliada em outro ponto da metodologia.

A tarefa de encontrar todos os subconjuntos de cláusulas que identifiquem expressões booleanas dentre as cláusulas de conflito geradas pelo resolvedor SAT é um problema NP-Completo. Este problema é análogo ao problema da Soma de Subconjunto, amplamente conhecido por ser NP-Completo. Em uma das versões do problema da Soma de Subconjunto, é dado um conjunto de inteiros e um inteiro cujo limite não pode ser ultrapassado. O desafio é encontrar um subconjunto dentre o conjunto de inteiros que satisfaça o limite imposto. No caso do presente trabalho, é necessário encontrar todos os subconjuntos de cláusulas para cada uma das expressões booleanas de interesse (ou seja, para diversos limites). Esta característica torna o problema ainda mais difícil de ser resolvido.

A solução de força-bruta é testar todas as combinações possíveis de agrupamento de cláusulas de conflito e verificar, para cada combinação, se ela satisfaz o padrão necessário para alguma das expressões booleanas de interesse. No entanto, esta abordagem é claramente não eficiente, já que para um conjunto y de cláusulas de conflito geradas com z expressões booleanas de interesse, teríamos

uma complexidade de tempo da ordem de $O(z.y!)$.

Para resolver este problema, foi proposta e desenvolvida uma metodologia que realiza a identificação de similaridades em 2 etapas. Na primeira etapa, as cláusulas de conflito são particionadas em grupos de acordo com uma medida de proximidade determinada e na segunda etapa é realizada uma busca dentro de cada um destes grupos, com o objetivo de realizar a identificação das similaridades. Durante a etapa de particionamento, o domínio de busca pelas expressões booleanas é consideravelmente diminuído, tornando a busca por expressões booleanas um problema tratável.

Desta forma, a metodologia proposta para identificação de similaridades pode ser resumida como a seguir:

1. Particionamento das cláusulas, onde são aplicadas técnicas de particionamento para a separação das cláusulas de conflito em grupos lógicos. Os grupos formados serão utilizados pelo algoritmo para realizar a identificação de expressões booleanas.

2. Identificação das expressões booleanas, onde grupos lógicos particionados de interesse são percorridos de forma a identificar as similaridades.

Nas Seções 4.1.2 e 4.1.2 serão apresentados os detalhes de cada uma destas etapas.

A metodologia proposta possui duas principais vantagens com relação à outras abordagens que exploram similaridades estruturais entre os circuitos:

- A identificação de similaridades é realizada sem a necessidade de conhecimento prévio da estrutura interna dos circuitos a serem verificados. Ela é realizada apenas com base nas cláusulas de conflito geradas a partir do processo de resolução SAT.

- Como a metodologia para identificação de similaridades proposta utiliza cláusulas de conflito que já foram previamente geradas pelo resolvedor SAT, ela possui um pequeno acréscimo de tempo em termos computacionais. A metodologia proposta se mostrou eficiente computacionalmente.

Particionamento das cláusulas de conflito

Como discutido na Seção 4.1.2, é necessário realizar o particionamento das cláusulas de conflito antes do início da identificação de expressões booleanas

de forma a tornar o problema de identificação possível de ser resolvido. O particionamento permite que o problema seja subdivido em sub-problemas menores, que podem ser tratados de forma eficiente.

No contexto deste trabalho, particionamento é definido como a tarefa de juntar objetos em grupos tais que uma dada função objetivo seja otimizada com relação ao conjunto de restrições existentes. No caso do particionamento de cláusulas de conflito, o objetivo é agrupar as cláusulas de forma que os grupos gerados satisfaçam uma função de proximidade definida.

Particionamento é um problema amplamente estudado pela comunidade científica e diversos algoritmos já foram propostos para resolver o problema. Dentre eles, alguns se destacam pelo especial uso no contexto de projeto de circuitos integrados: *cluster growth*, *hierarchical clustering*, *min-cut partitioning* e *simulated annealing*[53]. Cada um destes algoritmos possui propósito específico e entradas e saídas bem definida.

O algoritmo utilizado para o particionamento das cláusulas de conflito é uma adaptação do algoritmo *cluster growth*. Este algoritmo foi escolhido como base para o algoritmo de particionamento utilizado pela metodologia proposta porque ele organiza um conjunto de objetos de acordo com uma função definida pelo usuário. No caso do particionamento de cláusulas para identificação de similaridades estruturais, o objetivo é organizar um conjunto de cláusulas de conflito em grupos de acordo com o número de variáveis compartilhadas entre as cláusulas.

O algoritmo *cluster growth* é iniciado com um conjunto de objetos não particionados e a medida em que é percorrido, ele distribui os objetos em um dado número de agrupamentos de acordo com uma medida de proximidade definida. O algoritmo *cluster growth* consiste de três tarefas principais: seleção de semente, seleção de objetos não agrupados e inserção de objetos em um grupo. O primeiro passo do algoritmo *cluster growth* é selecionar objetos que serão sementes para cada grupo, com objetivo de guiar o processo de agrupamento. Os objetos semente podem ser especificados manualmente, escolhidos randomicamente ou selecionados baseados em atributos do próprio objeto. O segundo passo é determinar a ordem de agrupamento dos objetos que ainda não foram inseridos nos grupos. A ordem é determinada por medidas de proximidade dos objetos. Por fim, o algoritmo insere os objetos que possuem maior valor de proximidade no grupo apropriado. Este processo é repetido até que todos os objetos tenham sido inseridos em um grupo.

O algoritmo de particionamento da metodologia proposta é iniciado com um conjunto de cláusulas de conflito não particionado e seu objetivo é agrupá-las de acordo com uma medida de proximidade baseada no número de variáveis compartilhadas entre as cláusulas. Se uma determinada variável aparece em mais de uma cláusula de conflito então esta variável é compartilhada pelas duas cláusulas em que ela aparece. Esta medida de proximidade é especialmente interessante para a metodologia de identificação de similaridades porque as cláusulas que compõe as expressões booleanas de interesse para o nosso problema são formadas pelas mesmas variáveis, alterando apenas o número de variáveis que as compõem, como pode ser visto na Tabela 4.1. Por exemplo, para a identificação da expressão booleana de interesse $z = x \wedge y$, é necessário o agrupamento das cláusulas apresentadas em 4.5.

$$(z + x^{'} + y^{'})\ (z + x)\ (z + y) \tag{4.5}$$

Todas as três cláusulas que compõem a expressão booleana são formadas pelas mesmas variáveis z, x e y. Por este motivo foi escolhida uma função de proximidade baseada no número de variáveis iguais que as cláusulas possuem entre si. A medida de proximidade das cláusulas pode variar entre zero e três, onde zero significa que as duas cláusulas de conflito que estão sendo comparadas não possuem nenhuma variável em comum e três significa que as duas cláusulas de conflito são formadas pelas mesmas variáveis.

O algoritmo de particionamento é composto por três tarefas: pré-processamento das cláusulas, agrupamento de cláusulas e remoção de grupos pequenos. No primeiro passo do algoritmo é realizado um pré-processamento das cláusulas de conflito descartando todas as cláusulas formadas por mais de três variáveis. Estas cláusulas não devem fazer parte do processo de particionamento, as expressões booleanas de interesse são formadas por cláusulas que possuem no máximo três variáveis. Em seguida as cláusulas de conflito são processadas seqüencialmente. Para cada um dos grupos criados, é computada a função de proximidade entre a cláusula e o grupo. Caso a medida de proximidade seja igual a dois ou três, a cláusula é inserida no grupo. No caso em que a cláusula não possua proximidade com nenhum dos grupos, um novo grupo é criado. Este procedimento é repetido até que todas as cláusulas tenham sido processadas. O último passo do algoritmo é a remoção de grupos que possuam apenas uma cláusula. O algoritmo pode ser visto na Listagem 4.1.

Como um exemplo do funcionamento do algoritmo, considere o conjunto

```
  \\ Entrada: conjunto de clausulas de conflito
  \\ Saida:   conjunto de similaridades identificadas
  \\
4 Particiona (clausulas_conflito) {
      remove_clausulas_mais_3_variaveis();
      for (i = 1 ; i < numero_clausulas; i++) {
          clausula_inserida = 0;
          for (j = 1 ; j < numero_grupos + 1; j++ ) {
9             proximidade = calcula_proximidade(i, j);
              if ( proximidade == 2 || proximidade == 3) {
                  insere_clausula_grupo(i,j);
                  clausula_inserida = 1;
              }
14        }
          if (clausula_inserida == 0) {
              cria_novo_grupo(i);
          }
      }
19    remove_grupos_pequenos();
  }
```

Listagem 4.1: Algoritmo de particionamento adaptado

de cláusulas de conflito mostradas na Figura 4.5. No primeiro passo do algoritmo, as cláusulas compostas por mais de 3 variáveis são retiradas durante a etapa de pré-processamento, como mostrado na Figura 4.5(a). Em seguida, cada cláusula de conflito é processada de forma seqüencial. A Figura 4.5(b) apresenta o processamento das primeiras três cláusulas. Para cada uma delas, foi computada a função de proximidade com os grupos existentes e como o resultado da função não atingiu o padrão mínimo esperado, foram criados novos grupos. O número ao lado de cada cláusula dentro do grupo indica o grau de proximidade da cláusula com o grupo em questão, sendo este definido pelas variáveis em negrito. É interessante notar que, para computar a função de proximidade, apenas o nome da variável é considerado, não importando a sua fase de assinalamento. Na figura, as cláusulas processadas em um determinado passo estão em negrito. O processamento da cláusula $(b+d+e^{'})$ é apresentado na Figura 4.5(c). Inicialmente, é computada a função de proximidade entre a cláusula e cada um dos grupos existentes, sendo que apenas com o grupo 1 a cláusula apresentou um grau de proximidade mínimo (2 variáveis iguais). Desta forma, a cláusula foi inserida ao conjunto de cláusulas do grupo 1 e um novo grupo foi criado para ela (grupo 4). Na Figura 4.5(d) é mostrado o processamento da cláusula $(b^{'}+e^{'}+g^{'})$.

É computado a função de proximidade da cláusula com os grupos existentes e ela é inserida no grupo 2 (com o qual possui grau de proximidade 3) e no grupo 4 (com o qual possui grau de proximidade 2). Como o grau de proximidade com um dos grupos foi máximo, ou seja, as variáveis do grupo e cláusula coincidem, não há necessidade de criação de um novo grupo. Finalmente é possível ver o processamento da última cláusula $(a + e^{'} + g)$ na Figura 4.5(e). A função de proximidade é calculada e a cláusula é inserida nos grupos 1 e 2 com grau de proximidade 2. Como não houve um casamento perfeito, ou seja, a coincidência de todas as variáveis, um novo grupo é criado para a cláusula. O terceiro e último passo do algoritmo é a remoção de grupos pequenos. São removidos todos os agrupamentos que possuem apenas uma cláusula, já que eles não irão compor nenhuma das expressões booleanas de interesse.

A principal adaptação realizada no algoritmo *cluster growth* foi a escolha inicial das sementes dos grupos a serem criados. Nesta abordagem, os grupos são criados dinamicamente, de acordo com o grau de proximidade entre a cláusula que está sendo processada e os grupos já pré-existentes. No algoritmo adaptado, não existe a necessidade de criação de todos os grupos para o início do particionamento.

Realizado o particionamento das cláusulas, o próximo passo é a identificação das expressões booleanas de interesse.

Identificação das expressões booleanas

Após ter sido concluído o particionamento das cláusulas de conflito conforme apresentado na Seção 4.1.2, a tarefa de identificação das expressões booleanas torna-se um problema mais fácil de ser resolvido. Como cada grupo formado pelo particionamento possui apenas cláusulas que compartilham as mesmas variáveis, dada uma expressão booleana de interesse, se esta expressão existir dentre o grupo de cláusulas de conflito, ela será formada por cláusulas que estejam em um mesmo grupo. Esta afirmativa é verdadeira uma vez que todas as expressões booleanas de interesse são formadas por cláusulas que compartilham as mesmas variáveis.

O algoritmo para identificação de expressões booleanas percorre todos os grupos criados durante a etapa de particionamento tentando encontrar conjuntos de cláusulas que constituam uma das expressões booleanas apresentadas em 4.1. Nesta etapa os assinalamentos das variáveis dentro de cada cláusula devem ser considerados.

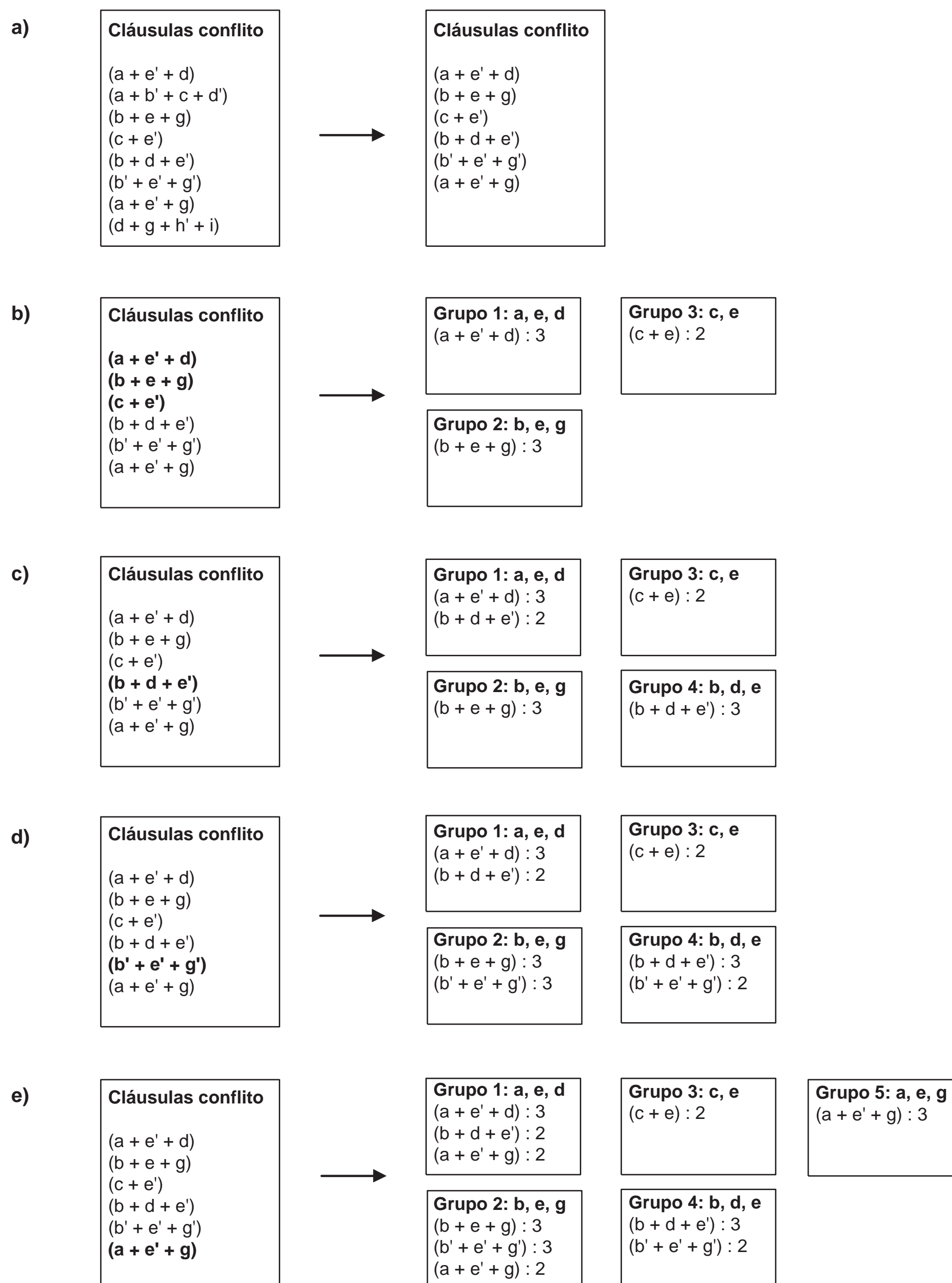


Figura 4.5: Particionamento de cláusulas de conflito

Capítulo 5

Resultados Experimentais

Neste capítulo serão apresentadas as tecnologias utilizadas durante a implementação do presente trabalho e será descrito o processo utilizado para a avaliação da metodologia de verificação proposta. Em seguida, serão discutidos os resultados experimentais.

5.1 Implementação

A implementação realizada no presente trabalho contempla um conjunto de funcionalidades definidas na metodologia de verificação proposta. Foram implementados integralmente o módulo de identificação de similaridades e o núcleo inteligente para processamento distribuído de resolvedores. Com relação a API, foram implementadas apenas as funcionalidades de submissão e interrupção de problemas ao núcleo inteligente.

A implementação realizada utiliza um conjunto de componentes que cooperam para a resolução do problema. Nas próximas seções, eles serão brevemente discutidos.

5.1.1 Resolvedor ZChaff

O resolvedor ZChaff é reconhecidamente um dos melhores resolvedores SAT da atualidade. Isso pode ser comprovado por suas premiações na tradicional *SAT Competition* que acontece todos os anos acompanhando um dos mais importantes eventos da área de resolvedores SAT. Na competição de 2002 ele levou o prêmio de resolvedor mais completo e em 2004 ele levou o prêmio de melhor resolvedor da categoria industrial, que engloba problemas gerados a partir

de casos reais apresentados pela indústria. O ZChaff é mantido pelo Boolean Satisfiability Research Group da Princeton University. Sua terceira versão foi disponibiliza em maio de 2004, o que mostra que é um projeto ativo. Maiores detalhes sobre o ZChaff podem ser encontrados nas referências [37, 54, 55].

5.1.2 MPI (*Message Passing Interface*)

A Interface de Passagem de Mensagens (do inglês, *Message Passing Interface*) é um protocolo de comunicação entre computadores. A MPI surgiu a partir da necessidade sentida pela indústria e centros de pesquisa de definir uma biblioteca padrão para o paradigma de passagem de mensagens. Surgiu então o Fórum MPI, que é um grupo de oitenta pessoas e quarenta organizações representando vendedores de sistemas paralelos, usuários industriais, laboratórios de pesquisa e universidades. Estas pessoas passaram então a desenvolver o padrão MPI buscado os seguintes objetivos:

- Desenvolver uma API se preocupando principalmente com as necessidades dos programadores.
- Permitir comunicação eficiente. Permitir a utilização em ambiente heterogêneos.
- Permitir a utilização com as linguagens Fortran e C.
- Prover uma comunicação confiável.
- Definir uma interface similar com as utilizadas até então.
- Definir uma interface que tenha o mínimo interferência nas plataformas já existentes de hardware e redes de computadores.
- Definir uma interface que suporte a utilização de threads.

As maiores vantagens de MPI sobre outras interfaces é que ela é portável, já que MPI já foi implementada para quase todas as arquiteturas de memória distribuída, e rápida, já que cada implementação é otimizada para o hardware na qual ela executa.

As implementações mais populares dessa interface são a MPICH [56, 57] e a LAM/MPI [58, 59].

A implementação realizada no presente trabalho utiliza LAM/MPI pois ela implementa mais funcionalidades da versão 2 da MPI. Uma exemplo dessas

funcionalidades adicionais é a instanciação de processos por um outro processo. Isto é importante já que permite calcular quantos processos serão necessários diretamente de dentro do programa.

5.1.3 Linux

Todos os componentes citados na seções anteriores possuem suporte para Linux. Esta é a principal razão para a escolha deste sistema operacional como plataforma de experimentos.

Além dessa vantagem o Linux oferece várias ferramentas que facilitam a vida de desenvolvedores, desde compiladores e depuradores a até ambientes de desenvolvimentos integrados (IDE, do inglês, *Integrated Development Environments*). Entre as distribuições Linux usadas, foi escolhida a distribuição Suse. A razão para essa escolha é o excelente sistema de pacotes fornecido por essa distribuição. Este sistema facilitou muito a instalação de todos os aplicativos necessários em um cluster de máquinas.

5.2 Resultados experimentais

Para validar a metodologia proposta nesta dissertação, foram realizados testes de verificação funcional com circuitos de baixo e alto grau de compartilhamento. Nos casos dos circuitos com baixo grau de compartilhamento, foi utilizado o processamento distribuído. Já para os circuitos com alto grau de compartilhamento, foi utilizado o aprendizado recursivo. As próximas duas subseções apresentam os resultados de cada caso.

5.2.1 Aprendizado recursivo

Para validar a metodologia de aprendizado recursivo, foi realizada a verificação funcional de multiplicadores.

Os multiplicadores estão inseridos em uma classe de problemas muito difíceis de serem resolvidos, já que eles que possuem a característica de explosão de estados [60], uma das maiores limitações no processo de verificação atualmente.

O princípio de operação dos multiplicadores paralelos normalmente utilizados em circuitos integrados é bastante simples. Ele tem origem no algoritmo clássico para o produto de dois números binários.

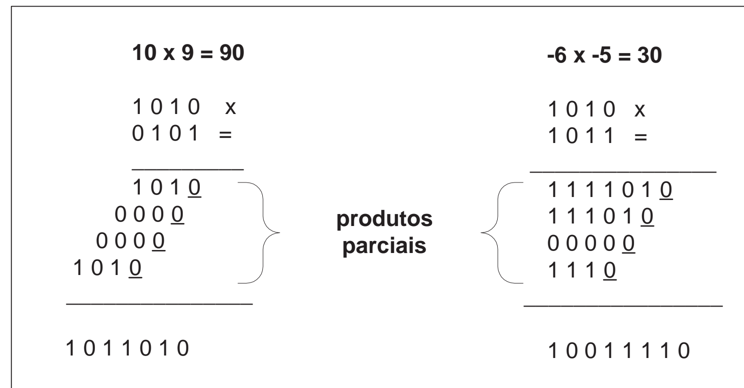


Figura 5.1: Processo de multiplicação binária

A Figura 5.1 mostra o processo realizado por dois multiplicadores: o mais a esquerda possui os operandos sem sinal enquanto que o da direita realiza a multiplicação de dois operandos com sinal.

Em geral, o resultado da multiplicação de duas seqüências de caracteres de n bits é uma seqüência de $2n-1$ bits, resultado da soma dos n produtos parciais, como pode ser visto na Figura 5.1. O produto parcial i, onde $0 \leq i \leq n-1$ é computado a partir do produto do i^{esimo} bit do multiplicador pelo multiplicando, sendo que este bit é deslocado i posições para a esquerda. O processo de multiplicação binária pode ser resumido em duas etapas principais: geração dos produtos parciais e a soma destes produtos. A aceleração do processo de multiplicação é baseado em duas técnicas:

- Redução do número de produtos parciais;
- Aceleração da soma dos produtos parciais

A soma dos produtos parciais é normalmente realizada utilizando somadores completos, também conhecidos por *full-adder*, conectados a uma estrutura de geração, soma e propagação de *carry*.

Duas técnicas utilizadas para redução dos produtos parciais são testadas na presente dissertação: a codificação de Booth [61] e a propagação CLA. As duas formas de redução serão discutidas nas próximas seções.

Carry Look Ahead (CLA)

O esquema mais freqüentemente utilizado para acelerar a propagação do *carry* é chamado de esquema CLA [62](do inglês, Carry Look Ahead). A idéia principal

deste esquema é gerar todos os *carries* de entrada em paralelo, com o objetivo de evitar o tempo de espera de propagação do *carry* desde o somador em que ele está sendo gerado. A Figura 5.2 ilustra a idéia básica do CLA.

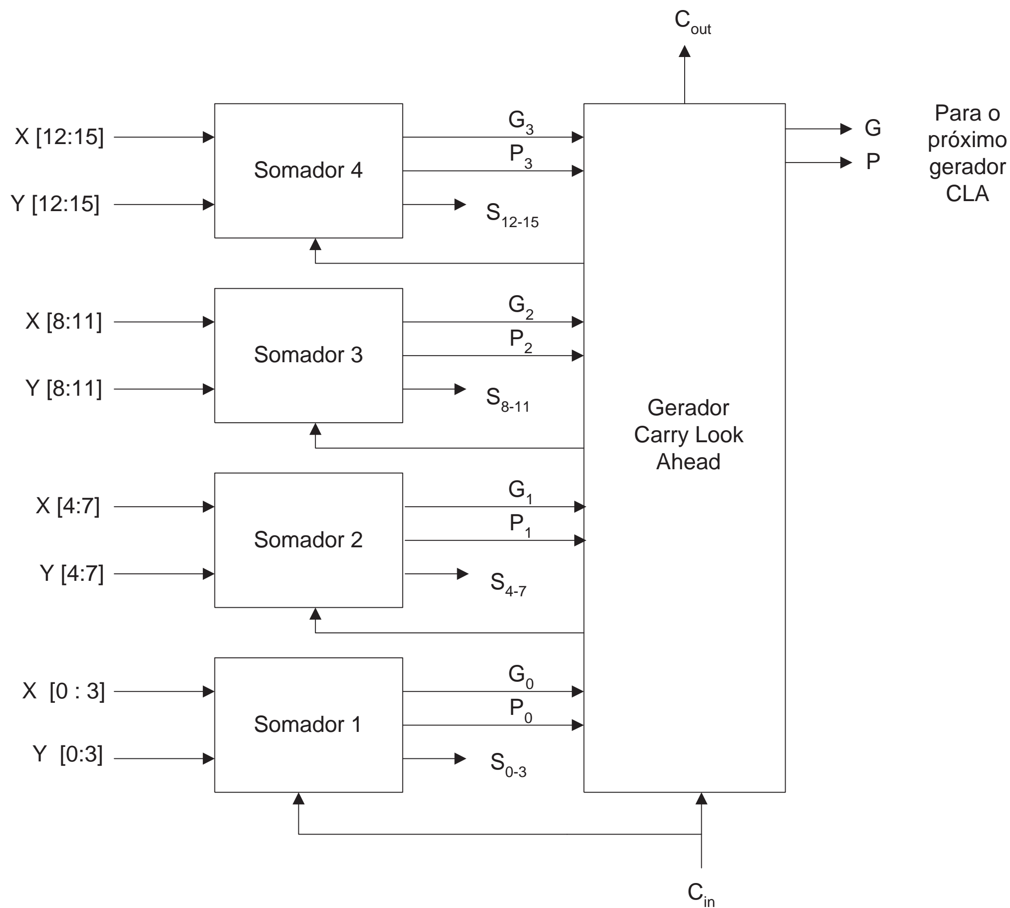


Figura 5.2: Somador de 16 bits implementando esquema CLA em dois níveis

Todo o tratamento do *carry* é realizado no módulo Gerador de CLA. G_i representam os sinais de *carry* gerados internamente. Estes sinais não dependem de sinais externos de entrada $carry_{in}$. Eles dependem apenas das entradas originais. P_i representa a soma onde o $carry_{out}$ é 0 porém os sinais de $carry_{in}$ irão determinar a propagação do *carry*. G_i e P_i são calculados em paralelo, antes que um $carry_{in}$ seja introduzido.

Codificação de Booth

A idéia principal da codificação de Booth é realizar adições e subtrações do multiplicando baseado no número de bits do multiplicador.

O algoritmo verifica em cada iteração dois bits adjacentes do multiplicador de forma a decidir a operação que será realizada (adição ou subtração). Os bits do multiplicador são verificados da direita para a esquerda, ou seja, dos bits menos significantes para os mais significantes. Por definição, o bit 0 a direita é considerado o bit menos significativo.

Supondo que o multiplicando tenha m bits e o multiplicador tenha n bits, o resultado será armazenado no registro de $m + n$ bits e será inicializado com 0. Desta forma, repetidas operações e shifts são aplicados nos resultados parciais, sendo que o resultado é armazenado no acumulador.

Os passos para o algoritmo de codificação de Booth pode ser visto abaixo:

- Inicialize o registrador de resultados com 0 - ele será utilizado para armazenar tanto os produtos parciais quanto o resultado final.
- Se os bits do multiplicador a serem testados forem "10", subtraia o multiplicando do produto parcial.
- Se os bits do multiplicador a serem testados forem "01", adicione o multiplicando do produto parcial.
- Se os bits do multiplicador a serem testados forem "00"ou "11", não faça nada.
- Realize um shift aritmético à direita do produto parcial.
- Verifique os próximos 2 bits adjacentes do multiplicador.
- Se ainda houverem bits a serem verificados, retorne ao segundo passo.

A Figura 5.3 apresenta a codificação de Booth para a multiplicação $11 * -3$.

Testes realizados

Nesta seção serão apresentados os resultados da verificação da descrição de dois multiplicadores que implementam o esquema de propagação de *carry* CLA e da verificação de um multiplicador implementando CLA e com um multiplicador de Booth. Em todos os casos, os multiplicadores foram verificados pela metodologia de verificação tradicional e pela verificação proposta nesta dissertação. A metodologia tradicional referida é a verificação, através do resolvedor ZChaff, sem a utilização das similaridades estruturais encontradas. Já a verificação com

Multiplicando (Y)						0	1	0	1	1	($+11_{10}$)
Multiplicador (X)						1	1	1	0	1	0 (-3_{10})
Resultado (R)											Ação
Passo	0	0	0	0	0	0	0	0	0	0	Inicialização
	0	1	0	1	1						10 => SUB (R, Y)
	1	0	1	0	1	0	0	0	0	0	A. Shift a direita
①	1	1	0	1	0	1	0	0	0	0	
	0	1	0	1	1						01 => ADD (R, Y)
	0	0	1	0	1	1	0	0	0	0	A. Shift a direita
②	0	0	0	1	0	1	1	0	0	0	
	0	1	0	1	1						10 => SUB (R, Y)
	1	0	1	1	1	1	1	0	0	0	A. Shift a direita
③	1	1	0	1	1	1	1	1	0	0	
	1	1	0	1	1	1	1	1	0	0	11 => NOP
④	1	1	1	0	1	1	1	1	1	0	A. Shift a direita
	1	1	1	0	1	1	1	1	1	0	11 => NOP
⑤	1	1	1	1	0	1	1	1	1	1	A. Shift a direita

Figura 5.3: Exemplo de codificação de Booth

a metodologia proposta realiza a realimentação das partições a partir das similaridades estruturais encontradas em partições anteriores. As partições que não possuíam intersecção foram processadas de forma distribuída.

Os circuitos foram particionados segundo critério de largura, ou seja, foram verificados circuitos incrementais em termos do número de bits.

Para o caso da metodologia tradicional, a verificação foi realizada até que o problema esgotasse os recursos da máquina. Já na metodologia proposta foi estabelecido um limite de verificação de 32 bits.

Tamanho dos multiplicadores CLA

Para a geração dos circuitos multiplicadores CLA que foram utilizados para a verificação, foi utilizado um algoritmo implementado para este propósito.

A Tabela apresenta o número de cláusulas e variáveis para cada circuito CLA gerado.

Bits	Portas Lógicas	Cláusulas
1	102	240
2	246	668
3	454	1304
4	726	2148
5	1062	3200
6	1462	4460
7	1926	5928
8	2454	7604
9	3046	9488
10	3702	11580
11	4422	13880
12	5206	16388
13	6054	19104
14	6966	22028
15	7942	25160
16	8982	28500
17	10086	32048
18	11254	35804
19	12486	39768
20	13782	43940
21	15142	48320
22	16566	52908
23	18054	57704
24	19606	62708
25	21222	67920
26	22902	73340
27	24646	78968
28	26454	84804
29	28326	90848
30	30262	97100
31	32262	103560
32	34326	110228

Verificação de dois multiplicadores CLA

Nesta seção serão apresentados os resultados da verificação de dois multiplicadores CLA. Os dois multiplicadores que estão sendo verificados são cópias idênticas e foram particionados utilizando o critério de largura com o valor 1.

Tempos de Resposta

Na Figura 5.4 é mostrado os tempos de resposta obtidos pelos dois métodos em escala logarítmica. Neste gráfico é interessante observar que até o bit 5 não é identificada nenhuma similaridade estrutural, como pode ser visto no Gráfico 5.5. Desta forma, como as partições não são realimentadas, os tempos da metodologia tradicional e da metodologia proposta são idênticos. A partir do bit 5 os resultados da metodologia proposta se sobrepõe aos resultados da metodologia tradicional.

Os resultados de tempo da metodologia proposta apresentam alguns picos. É interessante comparar estes picos com a número de cláusulas de conflito geradas, apresentada no Gráfico 5.7. O desenho dos dois gráficos é bem parecido, o que é justificado: a geração de cláusulas de conflito significa que houve buscas pelo espaço de soluções, consumindo tempo. Portanto, quanto maior a busca, mais cláusulas de conflito são geradas e mais tempo de processamento é necessário. O inverso também é verdadeiro.

Analisando o gráfico de tempo com relação ao número de similaridades estruturais encontradas, apresentado no Gráfico 5.5, é possível perceber que sempre que há um crescimento no número de similaridades estruturais identificadas no bit i, imediatamente no bit posterior existe uma queda no tempo de processamento. Isto acontece porque o circuito é realimentado com as similaridades encontradas no bit anterior. Este comportamento reforça a validade da metodologia proposta, uma vez com a realimentação das cláusulas, partes similares dos circuitos são descartadas, diminuindo a complexidade do problema a ser resolvido.

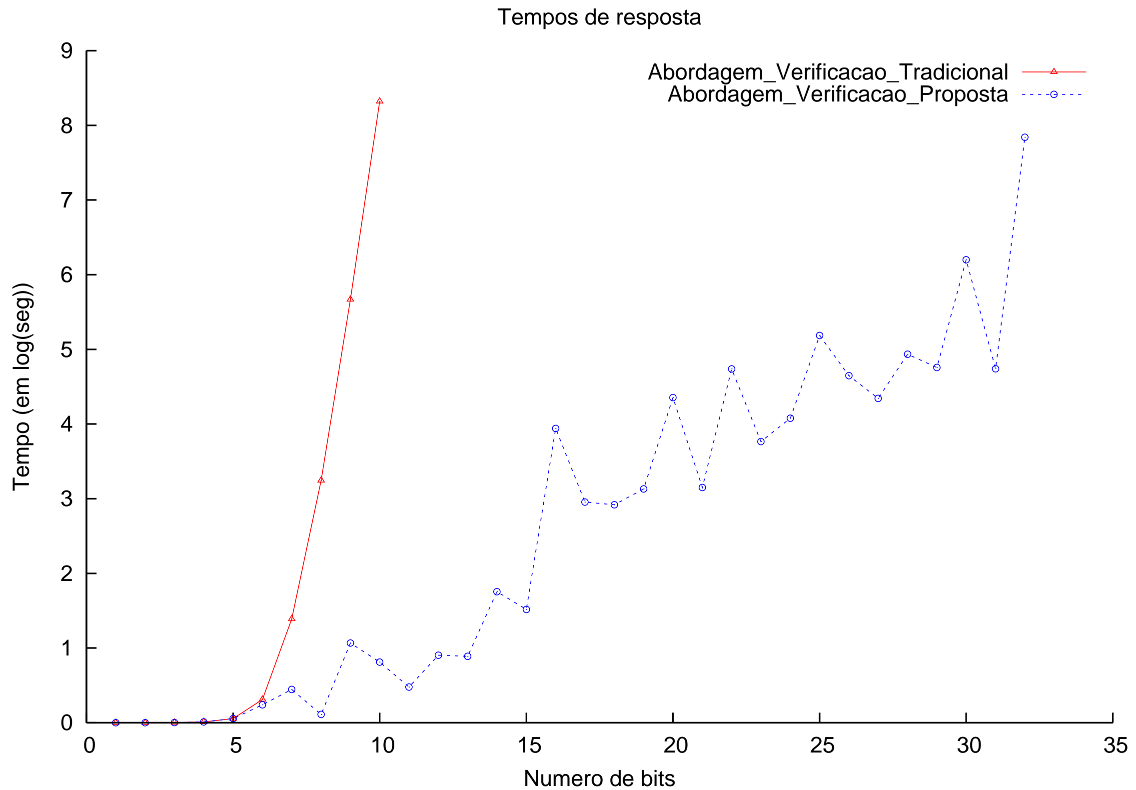


Figura 5.4: Tempos de resposta para a verificação de duas descrições de multiplicadores idênticos

Número de similaridades estruturais encontradas

Na Figura 5.5 é mostrado o número de similaridades estruturais identificadas entre os dois circuitos. O gráfico é apresentado em uma escala logarítmica. Naturalmente a curva do gráfico de identificação de similaridades acompanha de forma assintótica a curva do gráfico de cláusulas de conflito geradas, apresentada no Gráfico 5.7.

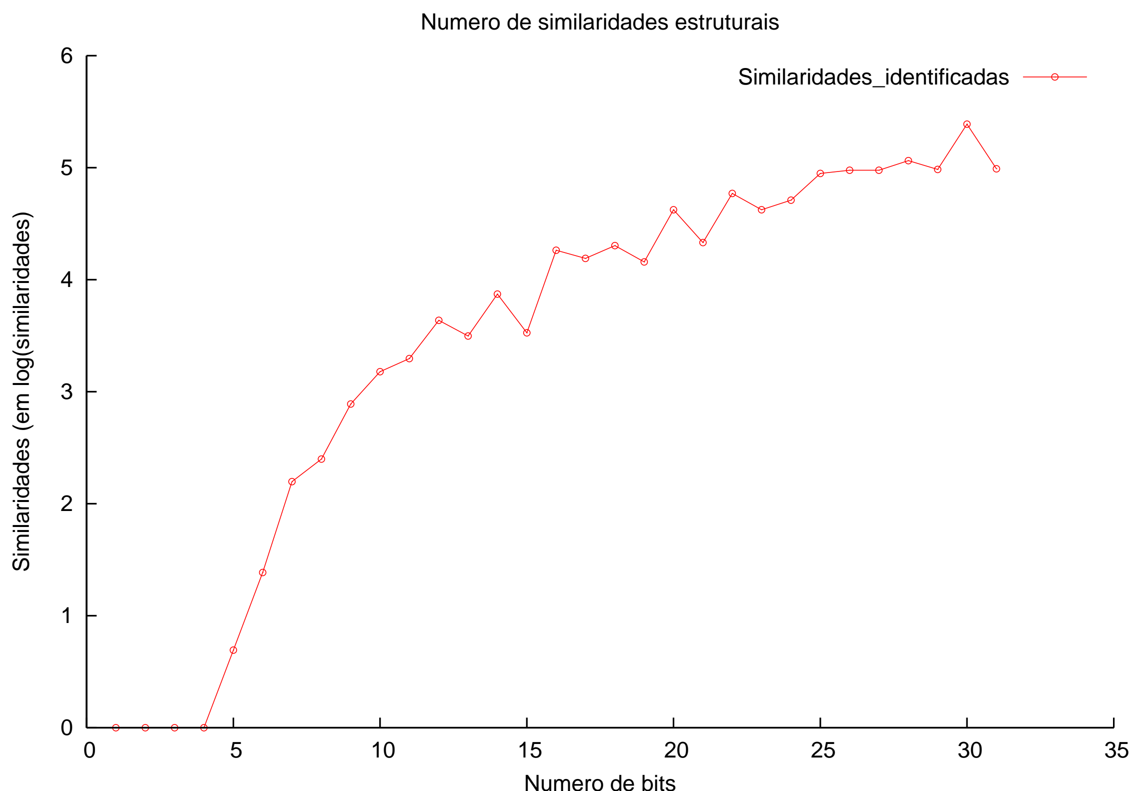


Figura 5.5: Similaridades encontradas na verificação de duas descrições de multiplicadores idênticos

Memória

Na Figura 5.6 é mostrado a utilização de memória pelo ZChaff, em escala logarítmica, durante a verificação dos multiplicadores pelas duas metodologias. Novamente até o bit 5 não houve uma diferença significativa de uso de memória, já que até este ponto não foram identificadas similaridades estruturais entre os circuitos. A partir do bit 5, as partições são realimentadas com as similaridades estruturais identificadas. Apesar de ter havido um aumento no número de cláusulas iniciais (cláusulas originais mais as cláusulas que representam as similaridades encontradas), o problema se torna menor, já que partes similares dos circuito são eliminadas. Esta diminuição do tamanho do problema reflete diretamente no número de cláusulas de conflito geradas, como pode ser visto no gráfico anterior. Por sua vez, a utilização de memória também é afetada, já que ela é proporcional ao número de cláusulas de conflito geradas e que devem ser armazenadas durante o processo de resolução SAT.

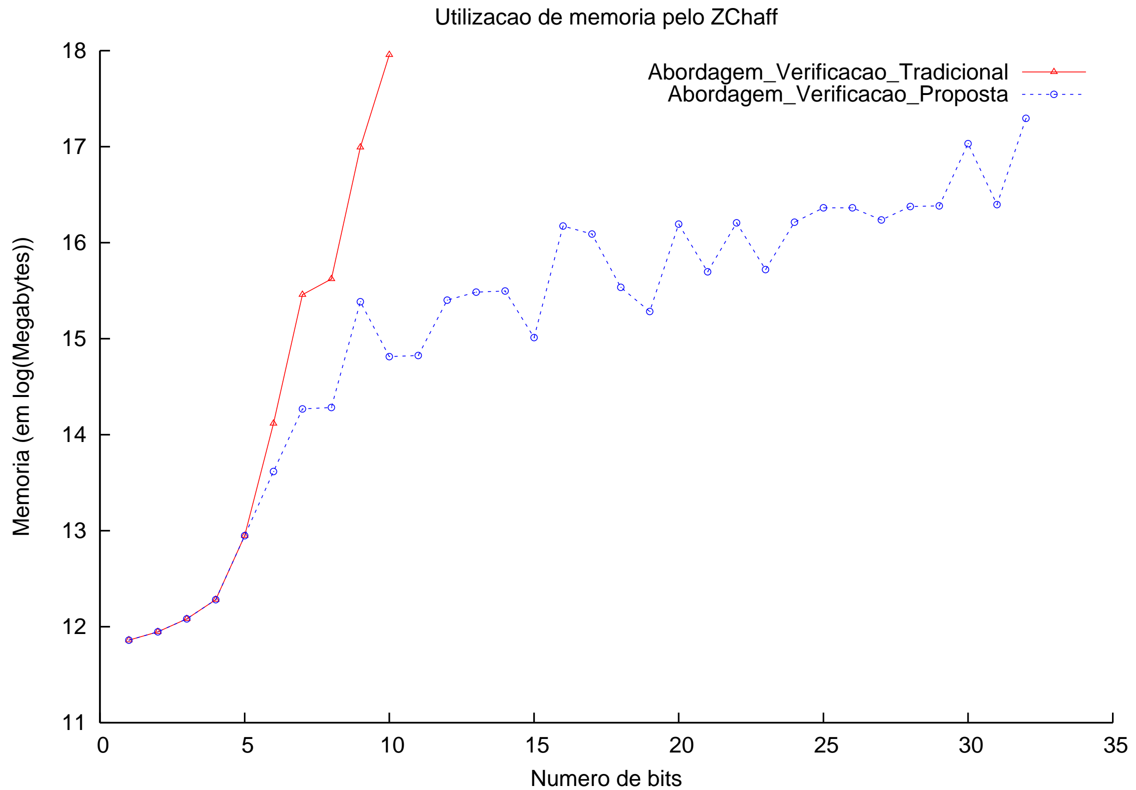


Figura 5.6: Utilização de memória pelo ZChaff para a verificação de duas descrições de multiplicadores idênticos

Número de cláusulas

Na Figura 5.7 é mostrado o número de cláusulas de conflito geradas durante o processo de verificação. É interessante notar que quando as cláusulas representando as similaridades estruturais começam a ser inseridas nos circuitos, existe um esforço bem menor de resolução, que pode ser visto na diminuição do número de cláusulas de conflito geradas.

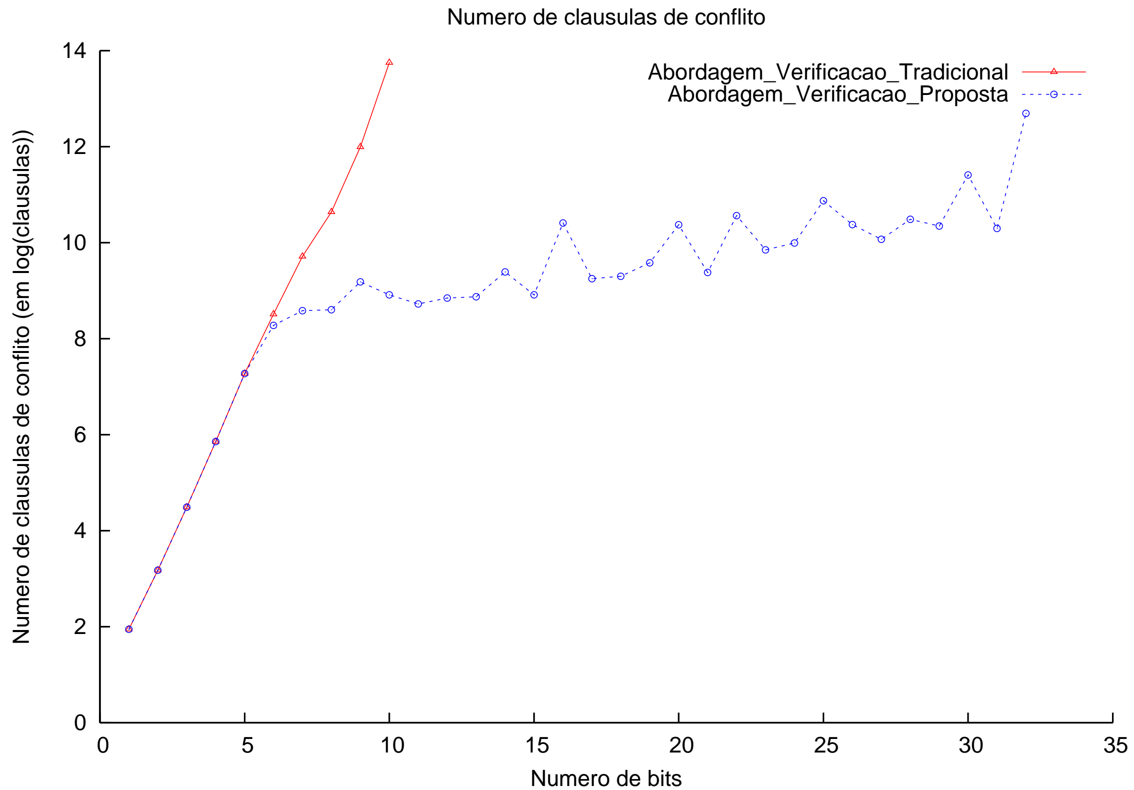


Figura 5.7: Número de cláusulas de conflito geradas para a verificação de duas descrições de multiplicadores idênticos

Verificação de dois multiplicadores CLA com inserção de erros aleatórios

Nesta seção serão apresentados os resultados da verificação de dois multiplicadores CLA. Foram inseridos erros aleatórios em uma das descrições do circuito. Os erros aleatórios introduzidos representaram 1% de alteração do circuito. Este tipo de cenário ocorre quando são realizadas pequenas alterações manuais pela equipe de projetistas dos circuitos.

Tempos de Resposta

Na Figura 5.8 é mostrado os tempos de resposta obtidos pelos dois métodos em escala logarítmica. É interessante notar a curva de tempos, quando comparada à verificação de dois circuitos idênticas, não apresenta grandes diferenças. Para multiplicadores com até 17 bits, os erros aleatórios aumentaram um pouco o tempo necessário para verificação, sendo que em seguida os tempos ficaram bastante parecidos com os tempos da verificação de circuitos idênticos. Este comportamento pode ser explicado pela dimensão da inserção de erros aleatórios:

apenas 1% do circuito total teve a sua estrutura modifica.

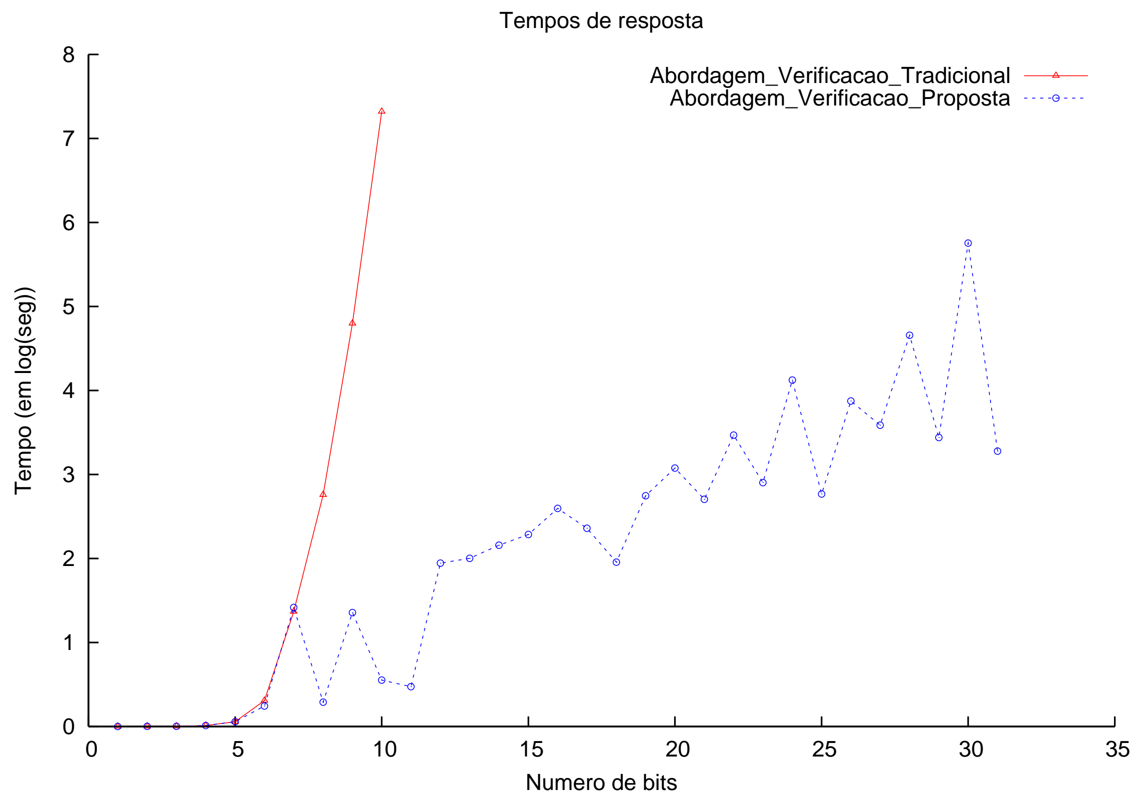


Figura 5.8: Tempos de resposta para a verificação de duas descrições de multiplicadores com inserção de erros aleatórios

Número de similaridades estruturais encontradas

Na Figura 5.9 é apresentado o número de similaridades estruturais, em escala logarítmica, identificadas nos circuitos que foram verificados. Apesar dos erros aleatórios inseridos, a relação do número de similaridades estruturais encontradas com o tempo de processamento apresentado na seção anterior é mantida.

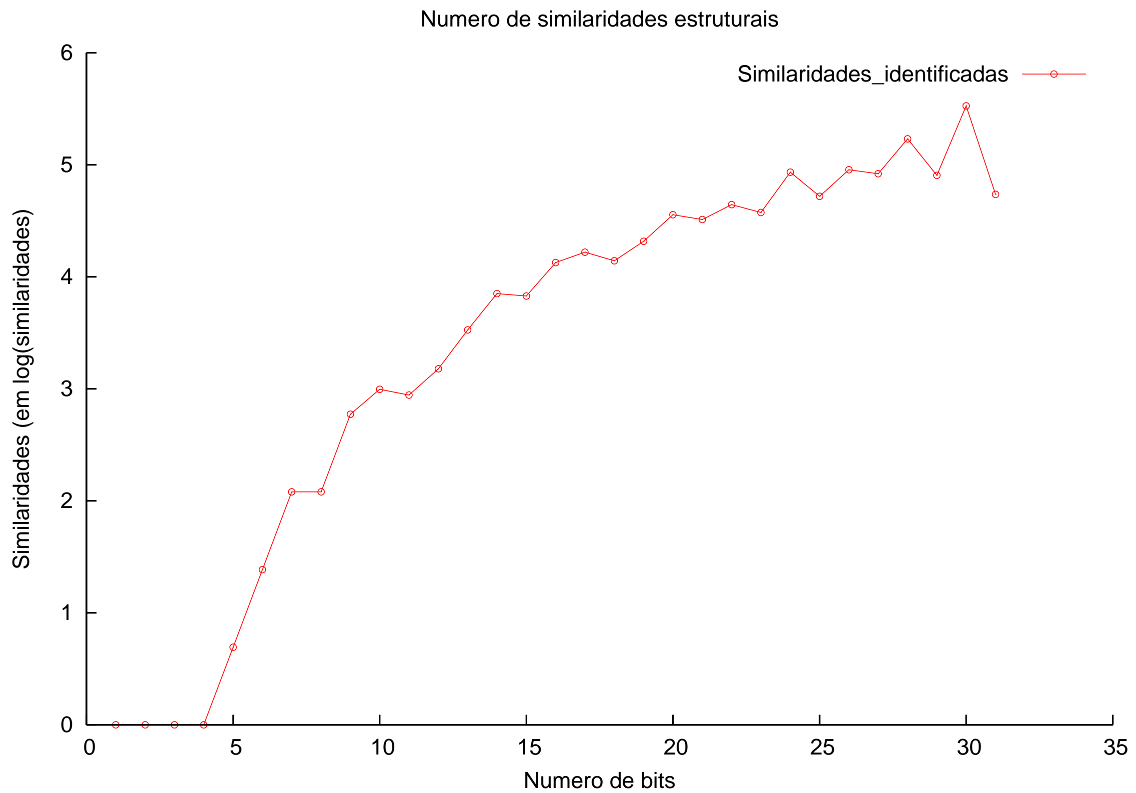


Figura 5.9: Similaridades encontradas na verificação de duas descrições de multiplicadores com inserção de erros aleatórios

Memória

Na Figura 5.10 é apresentada a utilização de memória durante a verificação dos circuitos multiplicadores com inserção de erros aleatórios. As discussões sobre o relacionamento da memória com o número de cláusulas de conflito e com o número de similaridades encontradas apresentadas na seção anterior continuam válidas.

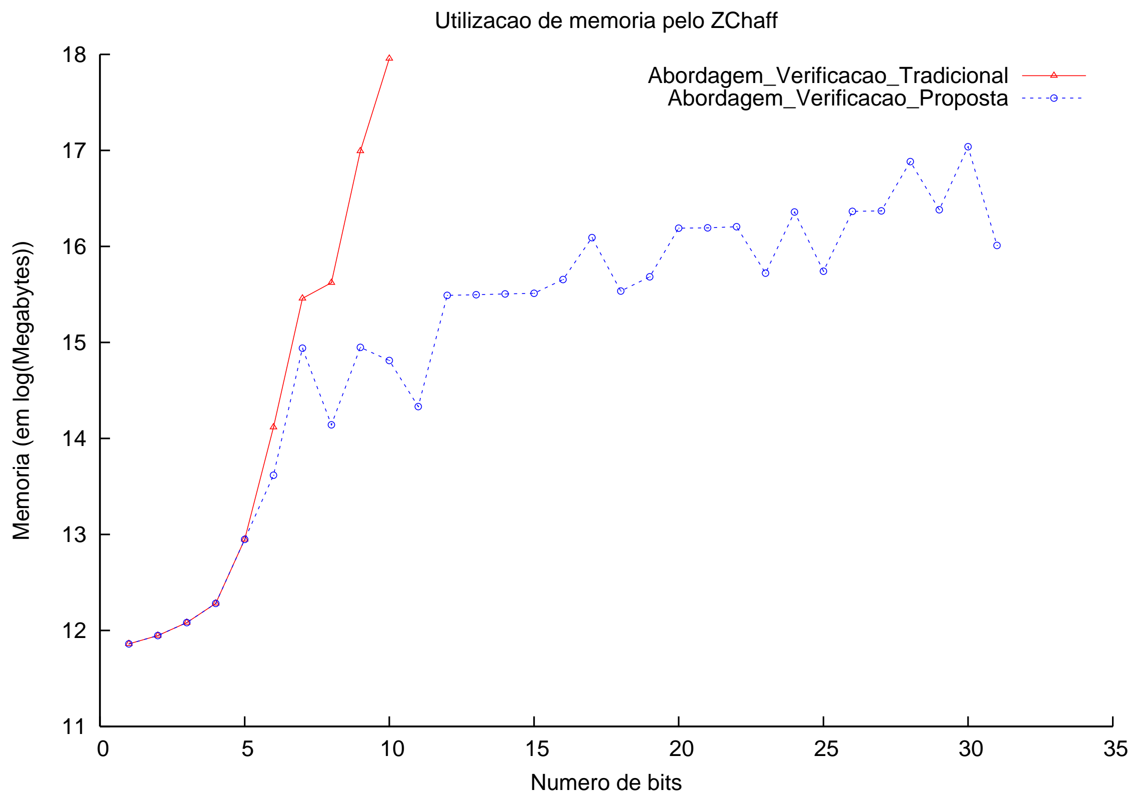


Figura 5.10: Utilização de memória pelo ZChaff na verificação de duas descrições de multiplicadores com inserção de erros aleatórios

Número de cláusulas

Na Figura 5.11 é mostrado, em escala logarítmica, o número de cláusulas de conflito geradas durante a verificação dos circuitos com erros aleatótios pelos dois métodos.

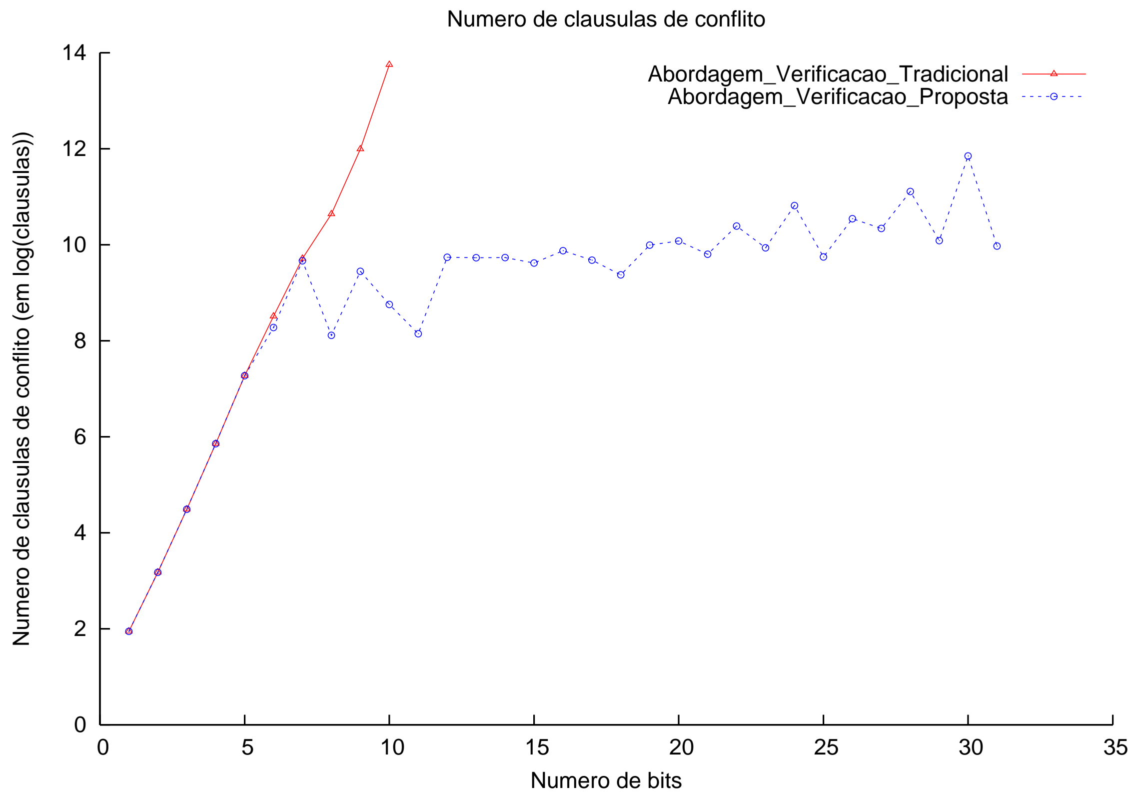


Figura 5.11: Cláusulas de conflito geradas na verificação de duas descrições de multiplicadores com inserção de erros aleatórios

Verificação de dois multiplicadores CLA de 3 em 3 bits.

Nesta seção serão apresentados os resultados da verificação da descrição de dois multiplicadores com particionamento baseado em largura de 3 bits.

Tempos de Resposta

Na Figura 5.12 é mostrado os tempos de resposta obtidos durante a verificação dos multiplicadores com particionamento de 3 bits. Os valores apresentados estão em escala logarítmica. É interessante notar neste gráfico que a tendência oscilatória apresentada nos outros gráficos de tempo de verificação é mantida. A diferença neste caso é que a oscilação possui valores absolutos muito maiores do que os apresentados na verificação de multiplicadores com particionamento de largura 1. Isto ocorre porque como a lógica envolvida em cada partição é muito maior, o impacto das similaridades encontradas que realimentam as partições também é muito maior, tanto no aspecto positivo (quando muitas similaridades são identificadas) quanto no aspecto negativo (quando poucas similaridades são identificadas).

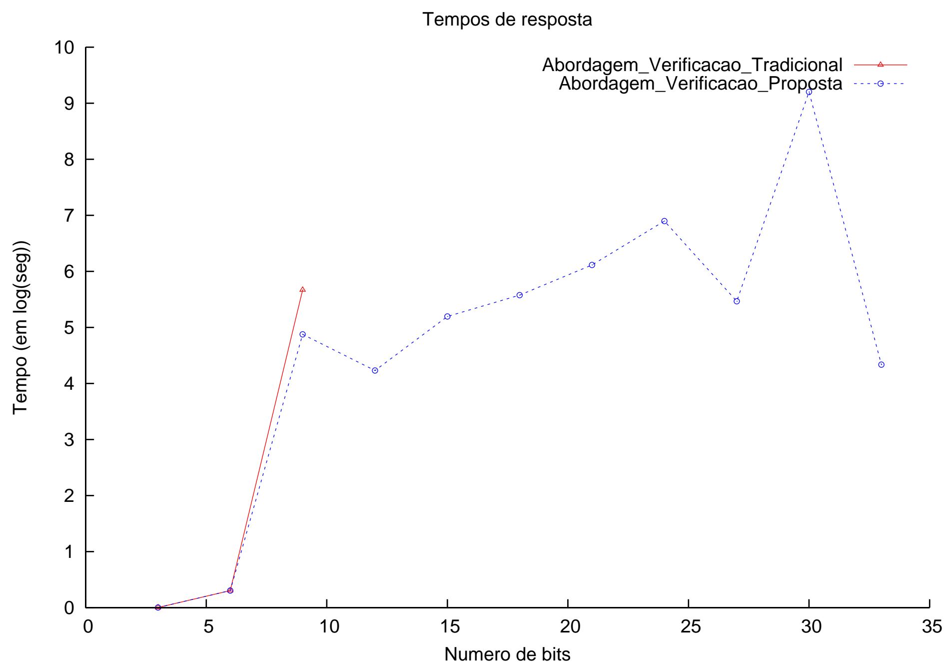


Figura 5.12: Tempos de resposta para a verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits

Número de similaridades estruturais encontradas

Na Figura 5.13 é apresentado o número de similaridades estruturais, em escala logarítmica, identificadas nos circuitos que foram verificados. A curva apresentada neste gráfico é bastante próxima a curva apresentada na verificação de multiplicadores com particionamento de largura 1.

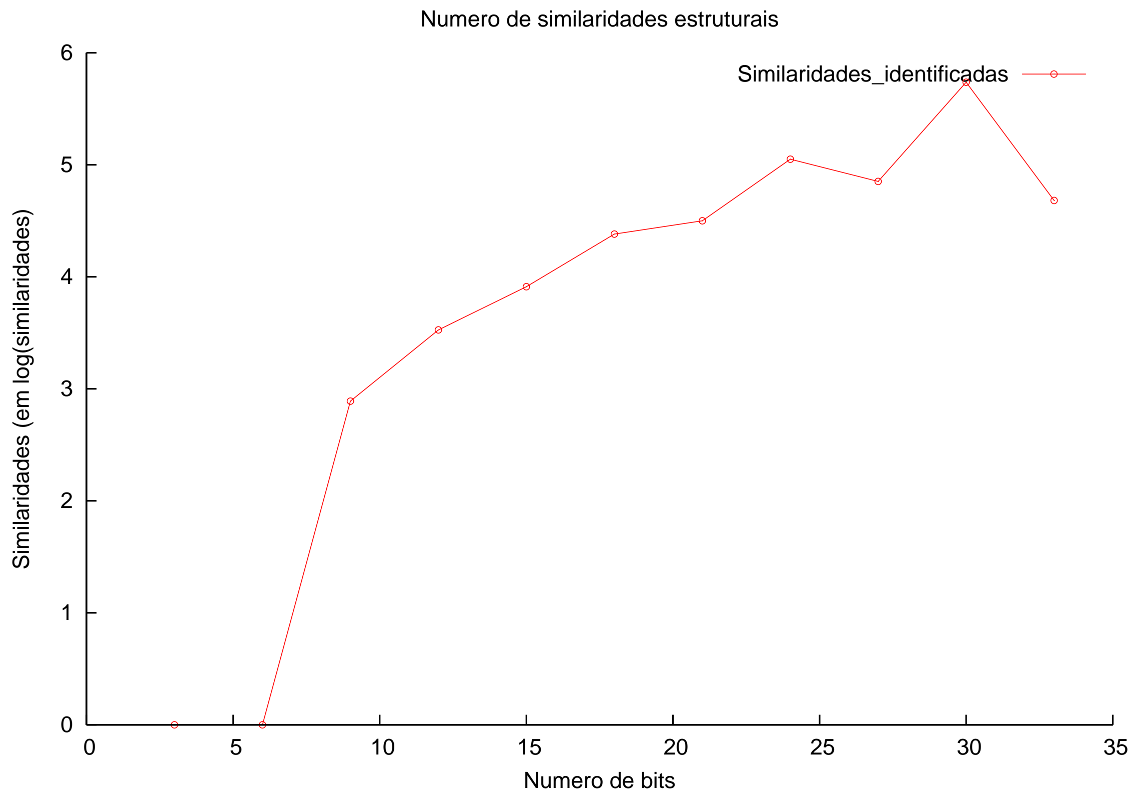


Figura 5.13: Similaridades encontradas na verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits

Memória

Na Figura 5.14 é mostrada a utilização de memória durante a verificação dos circuitos multiplicadores com inserção de erros aleatórios. A utilização de memória também segue a tendência de utilização das verificações anteriores.

Figura 5.14: Utilização de memória pelo ZChaff para a verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits

Número de cláusulas

Na Figura 5.15 é apresentada as cláusulas de conflito geradas durante a verificação de multiplicadores com largura de particionamento de 3 bits.

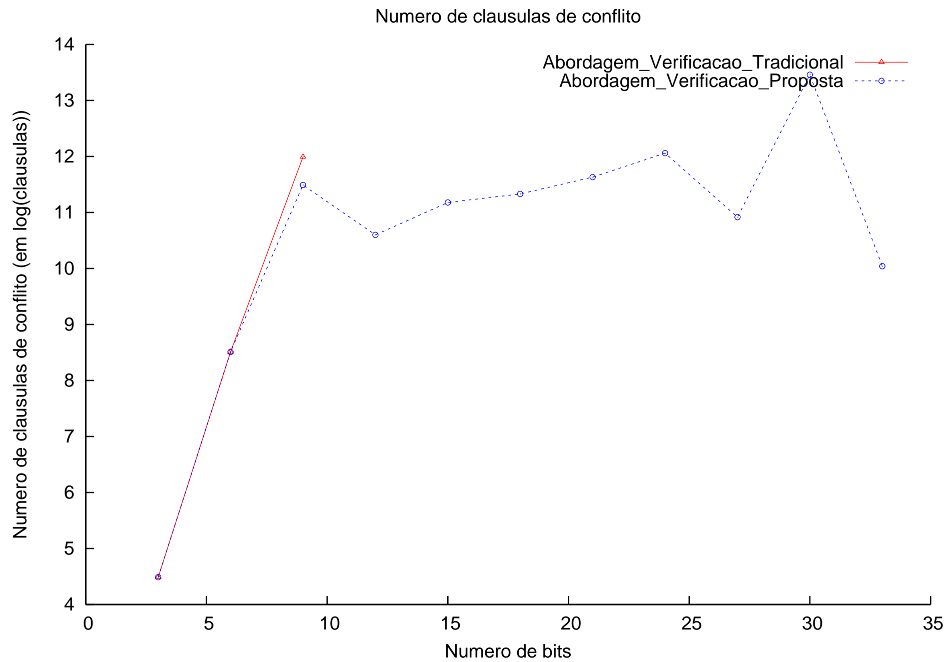


Figura 5.15: Cláusulas de conflito geradas na verificação de duas descrições de multiplicadores com particionamento de 3 em 3 bits

5.2.2 Verificação de um multiplicador CLA com um multiplicador Booth

Nesta seção serão apresentados os resultados da verificação de um multiplicador CLA com um multiplicador Booth. Os dois multiplicadores que estão sendo verificados são implementados corretamente, ou seja, não foram inseridos erros.

É importante notar que neste caso os dois multiplicadores possuem uma estrutura interna diferente, devido a sua forma de cálculo dos produtos parciais, ou seja, eles são circuitos dissimilares.

Na Figura 5.16 é mostrada os tempos de resposta para a verificação dos dois circuitos multiplicadores dissimilares - CLA e Booth. Como pode ser visto, no caso de dois multiplicadores dissimilares, a metodologia não apresenta bons resultados, uma vez que não existe compartilhamento de cláusulas.

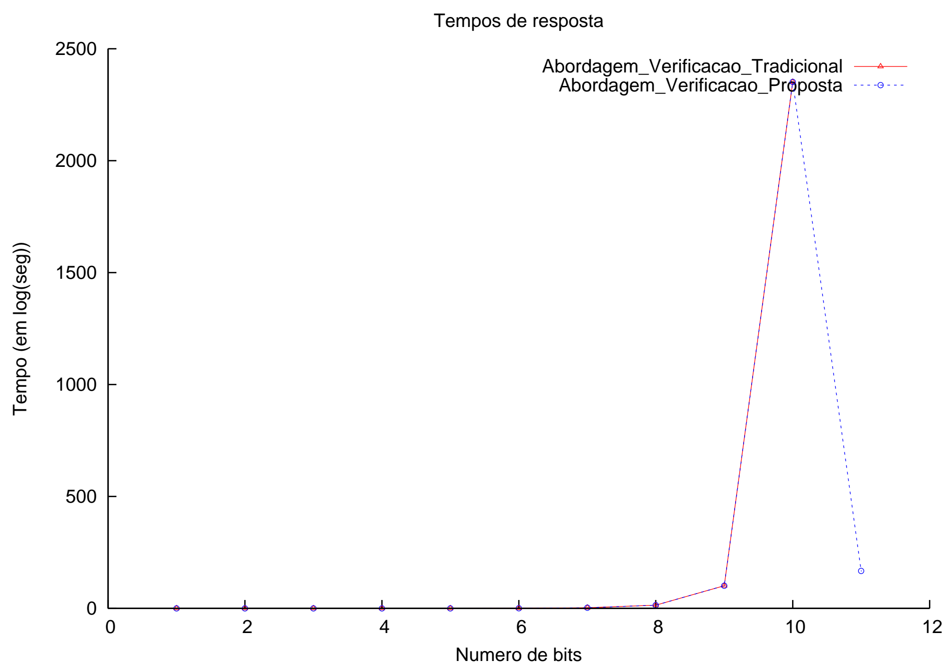


Figura 5.16: Tempos de resposta para a verificação de duas descrições de multiplicadores - CLA e Booth

5.2.3 Processamento Distribuído

Para validar a metodologia de processamento distribuído foi selecionado um conjunto de instâncias de satisfabilidade que não possuem nenhum grau de compartilhamento ou intersecção. Estas instâncias foram distribuídas em até 4 processadores sem que tivessem os seus problemas particionados.

As instâncias selecionadas como caso de teste foram:

1. Instância SAT gerada na verificação formal de um processador superescalar de 64 bits e que executa 4 instruções por cicloo de clock.

2. Instância UNSAT gerada pela verificação invariante da igualdade de implementação de duas filas.

3. Instância UNSAT gerada a partir da verificação de uma cache de 64 bits

4. Instância UNSAT gerada a partir da verificação de um processador superescalar

5. Instância UNSAT gerada a partir da verificação de um processador superescalar com reordenação de buffer.

6. Instância UNSAT gerada a partir da verificação de um multiplicador de 8 bits.

7. Instância UNSAT gerada a partir da verificação de barrier shifter de 8 bits.

A Tabela 5.1 apresenta os tempos de execução para cada uma das instâncias acima quando elas são verificadas de forma isolada em uma máquina, ou seja, sem nenhum grau de paralelismo.

Instância	Tempo (minutos)
1	51,88
2	1,94
3	36,15
4	24,46
5	33,65
6	93,52
7	109,84

Tabela 5.1: Tempo de execução para cada instância isolada

A Tabela 5.2 apresenta os tempos de verificação para as instâncias acima com processamento distribuído em até 4 processadores. É interessante notar que nesta fase da solução as instâncias são submetidas à resolução sem que haja qualquer particionamento de seus circuitos.

Número de Processadores	Tempo (minutos)
1	370,44
2	196,56
3	146,98
4	136,88

Tabela 5.2: Tempo de execução para a verificação das instâncias selecionadas de forma distribuída em até 4 processadores

5.3 Performance da solução adotada

Como foi visto através dos resultados experimentais apresentados na Seção 5.2.1, a metodologia proposta apresenta excelentes resultados para verificação de circuitos que contenham similaridades estruturais e funcionais entre si. No caso da verificação de multiplicadores, um caso patológico para a área de verificação, foi possível verificar multiplicadores de até 32 bits, sendo que os recursos não foram explorados até o limite.

Na Figura 5.17 é apresentado um gráfico com o custo de processamento adicionado pelo módulo de identificação de similaridades, principal responsável pela melhoria da verificação. Como pode ser visto, o atraso em termos de processamento inserido pelo módulo de identificação de similaridades é relativamente baixo em comparação com o benefício que ele agrega à metodologia de verificação proposta.

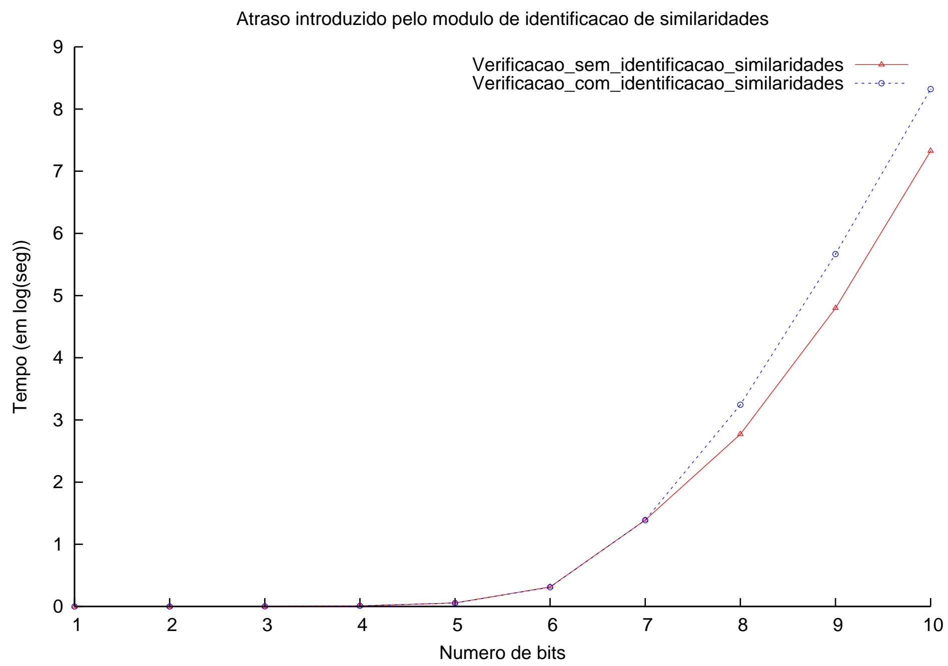


Figura 5.17: Custo da identificação de similaridades dentro da metodologia de verificação

Capítulo 6

Conclusões e Trabalhos Futuros

Verificação por Equivalência é um dos componentes chave da metodologia de verificação formal atual para sistemas digitais. Ela é técnica de Verificação Formal mais utilizada atualmente pela indústria para verificação de igualdade entre duas descrições de um circuito. Apesar das técnicas em Verificação por Equivalência terem tido um avanço significativo nos últimos anos, a complexidade inerente ao problema e aumento exponencial do tamanho e complexidade dos circuitos digitais continuam motivando pesquisas nesta área.

Nesta dissertação foi proposto, analisado e validado um núcleo inteligente para processamento distribuído de resolvedores SAT em Verificação por Equivalência. O núcleo desenvolvido explora o processamento distribuído de resolvedores SAT e a identificação de similaridades estruturais entre os circuitos que estão sendo verificados. A identificação de similaridades é realizada a partir das cláusulas de conflito geradas durante o processamento dos resolvedores SAT. A metodologia de verificação proposta foi validada a partir da verificação de multiplicadores binários. Os multiplicadores estão inseridos na classe de problemas patológicos da verificação, já que apresentam explosão de estados mesmo para circuitos formado por poucos bits. A metodologia de verificação proposta apresentou um considerável ganho na verificação deste tipo de circuito. Foi possível verificar multiplicadores de até 32 bits enquanto com a metodologia tradicional foi capaz de verificar apenas 11 bits. Os resultados demonstram que a metodologia proposta é correta e eficiente para circuitos que possuam similaridades funcionais e estruturais.

Como trabalhos futuros, pretende-se refinar o módulo de similaridades estruturais construído. Atualmente diversas expressões booleanas de interesse estão sendo identificadas a partir das cláusulas de conflito geradas. No entanto,

a metodologia proposta utiliza apenas as expressões booleanas de igualdade, descartando todo o resto. Desta forma, é necessário analisar o impacto destas novas expressões booleanas nos circuitos a serem verificados e como utilizá-las no contexto da Verificação por Equivalência.

Referências Bibliográficas

[1] Daniel Brand. Verification of large synthesized designs. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993.

[2] A. Kuhlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference, 1997*, 1997.

[3] W. Kunz. An efficient tool for logic verification based on recursive learning. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993.

[4] Eugene Goldberg and Yakov Novikov. On complexity of equivalence checking. Technical report, Cadence Berkeley Labs, 2003.

[5] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for vlsi circuits. In *IEEE/ACM International Conference on Computer-Aided Design*, 1989.

[6] R. E. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.

[7] J. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *IEEE/ACM Design, Automation and Test in Europe*, 1999.

[8] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *IEEE/ACM International Conference on Computer-Aided Design*, 1998.

[9] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society Press, 2001.

[10] Douglas J. Smith. Vhdl & verilog compared & contrasted - plus modeled example written in vhdl, verilog and c. *IEEE/ACM International Conference on Computer-Aided Design*, 1996.

[11] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[12] Randal E. Bryant and Christoph Meinel. Ordered binary decision diagrams in electronic design automation: Foundations, applications and innovations.

[13] Oscar H. Ibarra and Sartaj Sahni. Polynomially complete fault detection problems. *IEEE Trans. Computers*, 24(3):242–249, 1975.

[14] J. P Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 1966.

[15] D. Bhattacharya and J. P. Hayes. *Hierarchical Modeling for VLSI Circuit Testing.* Boston: Kluwer, 1990.

[16] H. Fujiwara and T Shimono. On the acceleration of test generation algorlthms. *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[17] T. Kirkland and M. R. Mercer. A topological search algorithm for atpg. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 502–508, New York, NY, USA, 1987. ACM Press.

[18] M.H. Schulz, E. Trischler, and T.M. Sarfert. SOCRATES: a highly efficient automatic test pattern generation system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7:126–137, 1998.

[19] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to cad problems: Test, verification and optimization. In *IEEE/ACM International Conference on Computer-Aided Design*, 1994.

[20] W. Kunz, D. K. Pradhan, and S.M Reddy. A novel framework for logic verification in a synthesis environment. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1996.

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

[22] Olivier Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.

[23] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, and F. Somenzi. Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 235–241, New York, NY, USA, 1998. ACM Press.

[24] Paulo F. Flores, Horácio C. Neto, and João P. Marques-Silva. An exact solution to the minimum size test pattern problem. *ACM Transactions on Design Automation of Electronic Systems.*, 6(4):629–644, 2001.

[25] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[26] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[27] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Saarbrücken, 1995.

[28] J. Silva and K. Sakallah. GRASP – A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, 10 1996.

[29] Prathima Agrawal, Debashis Bhattacharya, and Vishwani D. Agrawal. Test generation for path delay faults using binary decision diagrams. *IEEE Trans. Comput.*, 44(3):434–447, 1995.

[30] Paul Tafertshofer, Andreas Ganz, and Manfred Henftling. A sat-based implication engine for efficient atpg, equivalence checking, and optimization of netlists. In *International Conference on Computer Aided Design*, 1997.

[31] Wolfgang Kunz and Dhiraj K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test genration in digital circuits. In *Proceedings of the IEEE International Test Conference on Discover the New World of Test and Design*, pages 816–825, Washington, DC, USA, 1992. IEEE Computer Society.

[32] Wolfgang Kunz. Hannibal: an efficient tool for logic verification based on recursive learning. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993.

[33] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C-Y Huang. A circuit sat solver with signal correlation guided learning. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10892, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing & Testable Design*. Wiley-IEEE Press, 1994.

[35] Daniel Sheridan. The optimality of a fast cnf conversion and its use with sat. Technical report, APES Research Group, 2004.

[36] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

[37] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[38] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1633, 1999.

[39] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 454–464, London, UK, 2001. Springer-Verlag.

[40] Nina Amla, Robert P. Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In *TACAS*, pages 34–48, 2003.

[41] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1995.

[42] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. of Conference on Computer-Aided Verification (CAV'02)*, LNCS, 2002.

[43] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated absstraction refinement for model checking large state spaces using sat based conflict analysis.

[44] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.

[45] Hugo Valentim Barros. Um algoritmo distribuído para verificação de modelos com fronteiras. Master's thesis, Departamento de Ciência da Computação - Universidade Federal de Minas Gerais, 2004.

[46] DIMACS Challenge. Satisfiability suggested format, 1993.

[47] Daniel Le Berre and Laurent Simon. The sat 2005 competition. In *Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT05*, 2005.

[48] L. Simon and P. Chatalic. Satex: A web-based framework for sat experimentation, 2001.

[49] Results of the sat 2005 competition. http://www.lri.fr/ simon/contest/results/.

[50] Jawahar Jain, Rajarshi Mukherjee, and Masahiro Fujita. Advanced verification techniques based on learning. In *Proceedings of the 32nd ACM/IEEE conference on Design automation.* ACM Press, 1995.

[51] Yusuke Matsunaga. An efficient equivalence checker for combinational circuits. In *Proceedings of the 33rd annual conference on Design automation.* ACM Press, 1996.

[52] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual conference on Design automation.* ACM Press, 1997.

[53] Allen Wu, Daniel Gajski, Nikil Dutt, and Steve Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[54] Princeton University. Sat research group. http://ee.princeton.edu/ chaff/zchaff.php.

[55] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[56] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996.

[57] ANL/MSU. Mpich - a portable implementation of mpi. Disponível em http://www-unix.mcs.anl.gov/mpi/mpich/.

[58] G. Burns, R. Daoud, and J Vaigl. Lam: An open cluster environment for mpi. In *Supercomputing Symposium*, 1994.

[59] Indiana University. Lam/mpi parallel computing team. Disponível em http://www.lam-mpi.org/.

[60] H. van der Schoot and H. Ural. A uniform approach to tackle state explosion in verifying progress properties for networks of cfsms, 1996.

[61] Justin Hensley, Anselmo Lastra, and Montek Singh. An area and energy-efficient asynchronous booth multiplier for mobile devices. In *International Conference in Computer Design*, 2004.

[62] I. Koren. *Computer Arithmetic Algorithms.* A. K. Peters Ltd, 2002.