Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

Ordenação inicial de BDDs para a verificação automática de sistemas de transição finita

Jorgiano Márcio Bruno Vidal

Natal - RN
2002

Ordenação inicial de BDDs para a verificação automática de sistemas de transição finita

Jorgiano Márcio Bruno Vidal

Dissertação submetida à Banca Examinadora do Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos necessários para a obtenção do grau de Mestre em Sistemas e Computação.

Apresentada aos Srs.:

_____

Prof. David Boris Paul Déharbe, D. Sc.
Presidente

_____

Profa. Anamaria Martins Moreira, D. Sc.

_____

Prof. André Inácio Reis, D. Sc.

Natal, Rio Grande do Norte, Brasil
Fevereiro de 2002

JORGIANO MÁRCIO BRUNO VIDAL

Ordenação inicial de BDDs para a verificação automática de sistemas de transição finita

Dissertação apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, para obtenção do grau de Mestre em Sistemas e Computação.

ORIENTADOR

Prof. David Boris Paul Déharbe

Natal - RN

2002

Agradecimentos

Agradeço a todos aqueles que contribuíram direta ou indiretamente à realização deste trabalho.

Aos meus pais, irmãos, avós, toda a minha família e a minha namorada.

Aos meus professores, especialmente ao meu orientador, pela paciência e por acreditar no meu trabalho.

A todos os amigos e colegas.

Obrigado

Resumo

Um BDD (*Binary Decision Diagram*) é um grafo acíclico dirigido que permite representar e manipular expressões booleanas. Através da utilização de BDDs é possível reduzir o espaço alocado para a representação das expressões em comparação com os métodos tradicionais, como árvores de decisão ou tabelas verdade.

A utilização de BDDs na verificação formal tem permitido a verificação automática de sistemas digitais.

A verificação de modelos é uma técnica de verificação formal na qual modela-se o sistema a ser verificado e faz-se a verificação de propriedades de interesse no modelo. Estruturas de Kripke são máquinas de estados finitos usadas para modelar sistema na verificação formal. A verificação simbólica de modelos usa BDDs para representar o modelo e a verificação é feita diretamente sobre os BDDs. As funções de verificação são aplicadas diretamente sobre os BDDs, fazendo com que chamadas às funções de manipulação de BDDs sejam realizadas intensivamente. A performance da verificação depende diretamente da eficiência das funções de manipulação dos BDDs.

Apesar de representar expressões booleanas usando menos espaço, o tamanho dos BDDs pode mudar, dependendo da ordem em que as variáveis estão dispostas, tendo um impacto direto sobre o desempenho das funções de manipulação. Este trabalho pretende introduzir uma nova abordagem para a ordenação de BDDs voltada para a verificação simbólica de modelos.

Esta nova abordagem, chamada de *algoritmo dos pesos*, se baseia na dependência das funções de transição do sistema para atribuir pesos a variáveis e determinar a posição das variáveis baseadas nos pesos obtidos.

O algoritmo dos pesos foi implementado na ferramenta de verificação VIS e comparado com outros métodos de ordenação de BDDs já implementados nesta ferramenta. Para os testes, foram usados descrições de sistemas que acompanham a ferramenta na qual o algoritmo foi implementado.

Resultados obtidos com o algoritmo dos pesos mostram que esta nova abordagem obtém boas ordens de variáveis BDDs. Houve casos em que o algoritmo dos pesos foi bastante superior aos outros, mas também houve casos em que foi inferior aos demais.

Abstract

A BDD (Binary Decision Diagram) is a direct acyclic graph for representation and manipulation of boolean expressions. Moreover, the use of BDDs makes it possible to reduce significantly the space needed to represent boolean expressions.

The use of BDDs in formal verification allows a fully automatic verification of digital systems.

Model checking is a formal verification technique where the system to be verified is represented as a model and the property checking is done on the model. Symbolic model checking uses BDDs for representing the model and the verification is done over the BDDs. Verification functions are performed directly over BDDs, making an intensive use of the corresponding handling functions. Verification performance is therefore directly dependent of the efficiency of the BDD-manipulating functions.

Although BDDs represent boolean expressions using fewer space, BDD size can change depending on the variable order, with a great impact on the performance of the handling functions. The work presented in this thesis introduces a new approach to the BDD ordering variable problem, focused on symbolic model checking.

This new approach, called *weighing algorithm*, is based on system transitions functions dependencies to assign weight to the variables, and defines the variable position based on variables weights.

The weighing algorithm was implemented in the model checking tool VIS and was compared with others BDDs variable ordering algorithms that are implemented in that tool. The proposed algorithm was benchmarked against the more traditional approaches using a set of system descriptions distributed with VIS.

Results have shown that our approach gets good BDD variable orders. There were some cases where the weighing algorithm was much better than the others, however there were cases where it was worse.

Sumário

1 Introdução

Os BDDs [4] (*Binary Decision Diagram*) são estruturas de dados usadas para representar expressões booleanas. Os BDDs são grafos acíclicos dirigidos com vértices não terminais e dois vértices terminais. Cada vértice representa uma expressão diferente. Assim, os dois vértices terminais representam os valores booleanos **verdadeiro** e **falso**. Os vértices não-terminais são rotulados por uma variável da expressão e os dois filhos correspondem à expressão com a variável valorada em falso e verdadeiro, respectivamente. O resultado da expressão é encontrado percorrendo os vértices do grafo, que são os argumentos da função, até um vértice terminal, que é o valor da função. Porém, uma limitação dos BDDs é que, apesar de ocuparem menos espaço do que outras formas de representação de expressões booleanas, como tabelas verdade ou árvores de decisão, o espaço utilizado pode variar de acordo com a ordem na qual as variáveis aparecem no grafo.

Verificação formal é uma técnica de se verificar se uma descrição obedece uma dada propriedade, com garantia de 100% de acerto. A verificação simbólica de modelos [7] usa BDDs para representar simbolicamente o sistema a ser verificado, diminuindo o espaço necessário para representar o sistema. Porém, como o espaço alocado para os BDDs depende da ordem em que as variáveis BDDs estão dispostas, se torna necessário o estudo e desenvolvimento de algoritmos para encontrar boas ordens de variáveis, no contexto da verificação simbólica de modelos.

Na verificação simbólica de modelos, o sistema a ser verificado é modelado como uma máquina de estados finitos – MEF – conhecida como estrutura de Kripke. As propriedades são especificadas através de lógica temporal, como CTL ou LTL, e a técnica de verificação consiste em determinar se a fórmula especificada é verdadeira no modelo do sistema. Para se determinar se uma fórmula f é verdadeira em uma estrutura de Kripke K, faz-se uma busca nos estados iniciais de K verificando se a fórmula f é verdadeira em cada estado. Caso a fórmula seja falsa em algum destes estados, o sistema modelado por K não possui a propriedade especificada por f.

A utilização da verificação simbólica de modelos na verificação de circuitos tem

se mostrado bastante eficiente na identificação de erros de projeto, pois o método de verificação tradicionalmente utilizado é a simulação . Simulações tomam muito tempo para serem feitas, principalmente em circuitos com muitos elementos de entrada, saída e registradores, além de não garantirem que uma propriedade seja verdadeira no circuito, pois não conseguem alcançar uma taxa de cobertura de 100%.

A utilização de BDDs para representar o modelo do sistema permite representar estados e conjuntos de estados ocupando menos espaço que outras formas. Isto é possível porque estados e conjuntos de estados são representados através de suas funções características, onde vários estados podem ser representados através de um único BDD. Porém, como dito anteriormente, o espaço ocupado pelos BDDs pode variar dependendo da ordem em que as variáveis BDDs aparecem. Pesquisas sobre como encontrar a melhor ordem das variáveis BDDs têm sido feitas nos últimos anos e vários métodos de ordenação têm sido apresentados.

Existem dois tipos de ordenação, ordenação estática [29, 16, 18, 13], que define a ordem das variáveis antes da construção dos BDDs, e ordenação dinâmica [28, 24, 26, 31, 20], ou reordenação, que muda a ordem das variáveis após os BDDs serem construídos. Algoritmos de ordenação estática têm sido desenvolvidos analisando as funções booleanas, ou circuitos combinatórios, e encontras boas ordens. Os algoritmos de reordenação têm se mostrado bastante eficientes, porém a qualidade da ordem encontrada é diretamente proporcional à qualidade da ordem das variáveis antes da aplicação do algoritmo.

Este trabalho pretende introduzir uma nova abordagem para a ordenação estática de variáveis BDDs que representam circuitos sequenciais, ou máquinas de estados finitos. Esta nova abordagem consiste em analisar a dependência das variáveis em relação às funções de transição dos estados da máquina. Esta análise visa determinar o **peso** de uma variável x , onde o peso é o número de funções de transição do modelo do circuito que possuem x como argumento. Variáveis de maior peso devem ficar no início, ou topo, da ordem de variáveis, isto acontece porque uma variável com maior peso tem uma influência muito forte no comportamento do circuito.

A qualidade de uma ordem pode variar de acordo com o estado do gerenciador de BDDs, onde o estado do gerenciador é o conjunto de variáveis e as funções representadas. Na verificação simbólica de modelos, o estado do gerenciador de BDDs muda no decorrer do processo, dessa forma, se torna necessário que a ordenação inicial das variáveis seja, além de uma boa ordem inicial, uma ordem que otimize os métodos de reordenação quando o estado mudar.

Este documento está disposto da seguinte forma, no capítulo 2 apresentaremos

os conceitos fundamentais sobre BDDs, além de métodos de ordenação estática e dinâmica. No capítulo 3 mostraremos o que é verificação simbólica de modelos e como representar sistema de estados finitos usando BDDs e os algoritmos de verificação. Logo a seguir, no capítulo 4, apresentaremos uma heurística para a ordenação inicial de BDDs e os resultados obtidos até o momento. Por fim, no capítulo 5, falaremos sobre as conclusões dos resultados obtidos e o que pode ser estudado com objetivo de melhorar a heurística apresentada.

2 *Binary Decision Diagrams*

Os diagramas de decisão binária [4] – BDDs – formam uma estrutura de dados eficiente na representação e operação de funções da lógica proposicional booleana. Os BDDs são grafos acíclicos dirigidos, onde as funções booleanas são representadas de forma canônica. Assim, se *bdd(f)* denota o BDD representando a função f, duas fórmulas f e g equivalentes são representadas pelo mesmo BDD:

$$f \Leftrightarrow g \text{ sss } bdd(f) = bdd(g)$$

Um fator importante na utilização de BDDs é que eles ocupam, geralmente, menos espaço que representações tradicionais, como tabelas verdades ou árvores de decisão.

Um problema na utilização de BDDs para representar funções booleanas é que, apesar dos BDDs serem representações canônicas, eles são canônicos dada uma determinada ordem em que as variáveis estão dispostas e uma mudança na ordem de disposição das variáveis geralmente ocasiona uma mudança no tamanho do BDD. Dessa forma torna-se importante definir uma ordem de variáveis que evite um crescimento exagerado do espaço utilizado. O problema de calcular a **melhor** ordem das variáveis é NP-completo [1], mas um ser humano com um bom conhecimento do problema pode, geralmente, escolher uma **boa** ordem das variáveis sem maiores dificuldades. Porém, com o aumento do número de variáveis, esta tarefa se torna bastante demorada e é interessante que as ferramentas possam escolher uma ordem sem ajuda externa, o que facilita a automação do processo.

Heurísticas para ordenação de variáveis têm sido desenvolvidas [29, 16, 18, 13, 28, 24, 26, 31, 20]. Elas são divididas em dois tipos:

ordenação estática Algoritmos que definem a ordem em que deverão ficar as variáveis antes da construção dos BDDs.

ordenação dinâmica Algoritmos que mudam a ordem das variáveis depois da con-

strução dos BDDs.

Mesmo que, de forma geral, a alteração da ordem das variáveis BDDs mudem o tamanho do BDD, existem casos em que a qualquer ordem correspondem BDDs que ocupam o mesmo espaço. Como o caso do circuito multiplicador, onde *Bryant* [4] mostrou que o número de nós BDDs do bit do meio é uma função exponencial do número de entradas do circuito.

Neste capítulo, iremos conhecer os BDDs e alguns métodos de ordenação de variáveis, tanto estáticos como dinâmicos. Na seção 2.1 mostraremos os conceitos gerais relativos aos BDDs, como são estruturados e manipulados. Logo a seguir, na seção 2.2 falaremos sobre alguns métodos de ordenação estática que estão sendo utilizados e suas vantagens e desvantagens. Por fim, na seção 2.3 conheceremos os conceitos de ordenação dinâmica, abordaremos alguns métodos e veremos vantagens e desvantagens.

2.1 Conceitos gerais

Esta seção apresenta os conceitos básicos sobre funções booleanas, BDDs e sua manipulação. Primeiramente, na seção 2.1.1 será feita uma breve introdução às funções booleanas. Logo a seguir, na seção 2.1.2 mostraremos como representar funções booleanas usando os BDDs. Na seção 2.1.3 mostraremos como está definida a valoração dos argumentos de uma função sendo representada através de BDDs, computando, assim, o valor da função. A ordem em que as variáveis de uma função estão dispostas no BDD podem alterar o espaço alocado para a representaçaõ deste, isto está mostrado na seção 2.1.4. A seguir, na seção 2.1.5 veremos como aplicar operações sobre BDDs que representam funções booleanas. Por fim, na seção 2.1.6 mostraremos como pode ser feita uma implementação de BDDs em linguagens de programação.

2.1.1 Funções booleanas

Uma variável booleana, ou proposição booleana, possui um valor pertencente ao conjunto dos valores booleanos $B = \{F, V\}$, onde F e V denotam os valores FALSO e VERDADEIRO, respectivamente (também usamos 0 e 1 para F e V). Uma função booleana f é uma função sobre valores booleanos para um valor booleano:

$$f : B^n \rightarrow B$$

Considere o conjunto $X = \{x_1, \ldots, x_n\}$ o conjunto de variáveis booleanas, ou proposições booleanas, e $f(x_1, \ldots, x_n)$ uma função booleana sobre X. Uma variável $x \in X$ possui uma determinada posição nos argumentos de f, ou seja, existe uma ordem das variáveis. A função *índice:* $X \rightarrow [1,n]$ é uma função que mapeia as variáveis pertencentes a $f(X)$ para o intervalo $[1,n]$ e retorna a posição da variável na ordem dos argumentos. Esta função é usada para representar uma determinada variável através de um identificador único pertencente ao intervalo.

A função que resulta quando um argumento x_i de uma função f é substituído pela constante $b \in B$ é chamado de **restrição** de f (também conhecido como cofator) e denotado por $f|_{x_i=b}$. Ou seja, para quaisquer argumentos $x_1, \ldots, x_n$,

$$f|_{x_i=b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Usando esta notação, a expansão de Shannon [30] de uma função sobre uma variável x_i é dada por:

$$f = x_i \bullet f|_{x_i=1} + \overline{x}_i \bullet f|_{x_i=0} \tag{2.1}$$

O resultado de uma função f pode então ser encontrado através da aplicação sucessiva da expansão de Shannon, substituindo as variáveis pelos seus respectivos valores assumidos na expressão.

Para mostrar a utilização da expansão de Shannon, consideremos a função:

$$f(x_0, x_1) = x_0 + x_1$$

A expansão aplicada a variável x_0 obtém a seguinte expressão:

$$\begin{aligned} f &= x_0 \bullet f|_{x_0=1} + \overline{x}_0 \bullet f|_{x_0=0} \\ f &= x_0 \bullet (1 + x_1) + \overline{x}_0 \bullet (0 + x_1) \end{aligned}$$

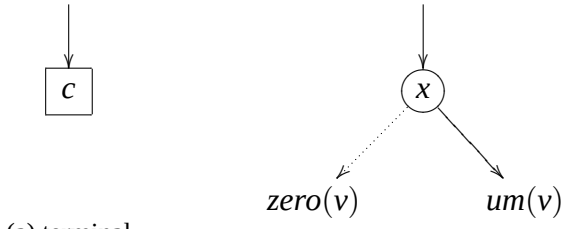
Caso x_0 receba o valor 1, a expressão se torna:

$$1 + x_1$$

o que faz com que o resultado da expressão seja 1, porém se x_0 recebe 0, a expressão fica:

$$0 + x_1$$

e o valor da expressão vai depender da expansão de Shannon sobre x_1 aplicada à



(a) terminal

(b) não-terminal

Figura 2.1: Representação gráfica de vértices de BDD

$0+x_1$:

$$\begin{aligned} f &= x_1 \bullet f|_{x_1=1} + \bar{x}_1 \bullet f|_{x_1=0} \\ f &= x_1 \bullet (0+1) + \bar{x}_1 \bullet (0+0) \end{aligned}$$

e o resultado da expressão será 1 se x_1 assumir 1 ou 0 caso contrário.

2.1.2 Representação de funções boolenas com BDDs

BDDs são gráfos acíclicos dirigidos formados por um conjunto V de vértices. Estes vértices podem ser de dois tipos: terminais e não-terminais. Um BDD possui apenas dois vértices terminais, onde cada um contém um valor booleano. Então, sendo v um vértice terminal, seu valor é denotado pela função *valor:* $V \rightarrow B$.

Seja X o conjunto de variáveis de uma função booleana f. Um vértice não-terminal v é rotulado com *var(v)=x,* onde $x \in X$, é uma variável de f. Cada vértice não-terminal possui pelo menos um arco de chegada e exatamente dois arcos de saída. Os arcos de saída têm como destino os BDDs representando as restrições $f|_{x=F}$ e $f|_{x=V}$. $f|_{x=F}$ e $f|_{x=V}$ são denotados por *zero(v)* e *um(v)*, respectivamente.

As figuras 2.1(a) 2.1(b) mostram representações gráficas de vértices terminal e não-terminal, respectivamente, e *valor(v)*$= c$ ($c \in \{0,1\}$) no gráfico 2.1(a) e *var(v)*$= x$ no gráfico 2.1(b). Os dois arcos de saída do não-terminal representam as funções *um(v)* e *zero(v)*, denotadas pelas linhas contínua e tracejada, respectivamente.

A função *nível(v)* denota o nível do vértice v no grafo. O grafo possui $n+1$ níveis, onde $n = |X|$ é o número de variváveis de X. Todos os vértices rotulados por uma mesma variável estão em um mesmo nível. O nível $n+1$ é formado pelos dois vértices terminais. O nível i, onde $1 \leq i \leq n$, é composto pelos vértices tais que *índice(var(v))* $=$

i. Sejam duas variáveis x e $y \in X$, se *nível(x)* $<$ *nível(y)* então a variável x está antes de y no nível das variáveis BDDs, o que é também denotado por $x < y$.

2.1.3 Valor de uma função

Considere o conjunto $X = \{x_1, \ldots, x_n\}$, o conjunto de variáveis de um BDD, e $V = \{v_1, \ldots, v_m\}$ o conjunto de vértices desse BDD. Seja $v_k \in V$ um não-terminal, *zero*(v_k) e *um*(v_k) dois outros vértices não-terminais v_l e v_m, então *nível*(v_k) $<$ *nível*(v_l) e *nível*(v_k) $<$ *nível*(v_m).

Dado um grafo G correspondente à função $f(x_1, \ldots, x_n)$, um grafo de função G_v tendo como raiz o vértice v representa uma função f_v da seguinte forma:

1. Se v é um vértice terminal:

   (a) Se *valor(v)*=1 então $f = 1$

   (b) Se *valor(v)*=0 então $f = 0$

2. Se v é um vértice não-terminal com *índice(var(v))*=i então f_v é a função obtida através da expansão de Shannon:

   $$f_v(x_1, \ldots, x_n) = \overline{x_i} \bullet f_{zero(v)}(x_1, \ldots, x_n) + x_i \bullet f_{um(v)}(x_1, \ldots, x_n),$$

   onde: $f_{zero(v)}(x_1, \ldots, x_n) = f_v(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ e $f_{um(v)}(x_1, \ldots, x_n) = f_v(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$

Dessa forma, podemos ver uma valoração do conjunto de argumentos $x_1, \ldots, x_n$ como descrevendo um caminho no grafo começando pela raiz, onde: dado um vértice v com *nível(v)*$= i$, o caminho continua para o vértice indicado por *um(v)* se *var(v)*$= 1$, onde *nível(um(v))* $> i$, ou continua para o vértice indicado por *zero(v)* se *var(v)*$= 0$, onde *nível(zero(v))* $> i$.

O valor da função para este conjunto de argumentos é o valor do vértice terminal no fim do caminho. Podemos observar que o caminho definido por uma valoração dos argumentos é único.

Para exemplificar a representação de funções booleanas utilizando BDDs consideremos a função $f = x_1 \bullet (x_2 + x_3)$. A figura 2.2 mostra a representação gráfica do BDD desta função.

Para mostrar como o grafo é percorrido, consideremos a valoração dos argumentos como $x_1 = 1$, $x_2 = 0$ e $x_3 = 1$, que também será representada por $< 1, 0, 1 >$. Para

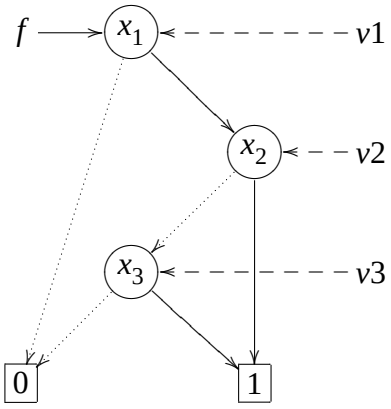


Figura 2.2: BDD para a função $f = x_1 \wedge (x_2 \vee x_3)$

encontrarmos o valor da função f com esse argumentos, seguimos as equações abaixo.

$$\begin{aligned}
f(1,0,1) &= f_{v_1}(x_1,x_2,x_3) \\
&= \overline{x_1} \bullet f_{zero(v_1)}(x_1,x_2,x_3) + x_1 \bullet f_{um(v_1)}(x_1,x_2,x_3) \\
&= 0 \bullet f_{zero(v_1)}(x_1,x_2,x_3) + 1 \bullet f_{um(v_1)}(x_1,x_2,x_3) \\
&= f_{v_2}(x_1,x_2,x_3) \\
&= \overline{x_2} \bullet f_{zero(v_2)}(x_1,x_2,x_3) + x_2 \bullet f_{um(v_2)}(x_1,x_2,x_3) \\
&= 1 \bullet f_{zero(v_2)}(x_1,x_2,x_3) + 0 \bullet f_{um(v_2)}(x_1,x_2,x_3) \\
&= f_{v_3}(x_1,x_2,x_3) \\
&= \overline{x_3} \bullet f_{zero(v_3)}(x_1,x_2,x_3) + x_3 \bullet f_{um(v_3)}(x_1,x_2,x_3) \\
&= 0 \bullet 0 + 1 \bullet 1 \\
&= 1 \bullet \mathit{valor(1)} \\
&= 1;
\end{aligned}$$

2.1.4 Influência da ordem das variáveis no tamanho do BDD

Apesar dos BDDs ocuparem menos espaço para representar funções booleanas do que outros métodos, como árvores de decisão e tabelas verdade, a ordem em que as variáveis estão dispostas pode alterar o tamanho do BDD. A figura 2.3 mostra a diferença do tamanho de BDDs que representam a função $(a \wedge b) \vee (c \wedge d)$ com a diferença apenas na ordenação das variáveis. Primeiramente, quando a ordem é $a < b < c < d$ o número total de nós BDDs é 4, enquanto que com a ordem $a < c < b < d$ o número sobe para 6, ou seja, há um acréscimo de 50% na utilização de memória para

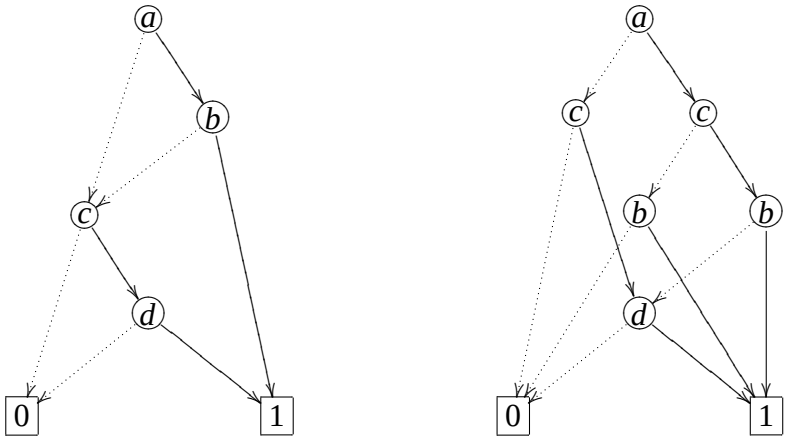


Figura 2.3: A influência da ordem de variáveis no tamanho de um BDD

representar a mesma função, além dos 2 vértices terminais.

2.1.5 Operações sobre BDDs

Bryant [4] definiu um conjunto de operações para a manipulação de BDDs. A operação para a aplicação de operadores booleanos com 2 argumentos, $f_1\ op\ f_2$, pode ser definida como:

$$[f_1 op\, f_2](x_1,\ldots,x_n) = f_1(x_1,\ldots,x_n)\ op\ f_2(x_1,\ldots,x_n).$$

O algoritmo começa nos vértices raízes dos dois argumentos e vai descendo, criando vértices no grafo resultado nos pontos de ramificação dos argumentos. A estrutura de controle do algoritmo é baseada na recursão seguinte, derivada da expansão de Shannon (equação 2.1)

$$f_1\ op\ f_2 = \overline{x}_i \bullet (f_1|_{x_i=0}\ op\ f_2|_{x_i=0}) + x_i \bullet (f_1|_{x_i=1}\ op\ f_2|_{x_i=1})$$

Dessa forma qualquer operação pode ser aplicada através do mesmo algoritmo. Este algoritmo possui complexidade $O(|G_1|.|G_2|)$, onde G_1 é o grafo que representa a função f_1 e G_2 é o grafo que representa função f_2. $|G|$ é o número de vértices no grafo G.

Campo	Terminal	Não-terminal
zero	null	*zero(v)*
um	null	*um(v)*
indice	$n+1$	*indice(v)*
val	*valor(v)*	X

Tabela 2.1: Valoração dos campos para vértices terminais e não-terminais

2.1.6 Implementação

Consideraremos a estrutura de um nó BDD como a descrita abaixo, em pseudo-pascal:

```
type vertice = record
     zero,um: vertice;
     indice: 1..n+1;
     val: (0,1,X);
     id: integer;
     marcado: boolean;
end;
```

Tanto vértices terminais como não-terminais são representados pelo mesmo tipo de registro. Porém os valores dos campos em um vértice dependem do tipo do vértice, como mostrado na tabela 2.1. O campo **marcado** é usado para identificar os nós que já foram visitados em uma travessia do grafo.

Uma tabela de dispersão [2] impõe uma forma canônica forte nos nós BDDs. Funções lógicas equivalentes são representadas por um único nó BDD. Assim sendo, esta tabela é chamada tabela única (*unique-table*).

A tabela única mapeia cada tripla (x,G,H) em um nó BDD $F = (x,G,H)$, tal que *var(F)* $= x$, *zero(F)* $= G$ e *um(F)* $= H$. Cada nó do BDD possui uma entrada na tabela única. Antes de um novo nó *n*, que representa um função f, ser acrescentado, uma busca na tabela determina se um nó que representa a função f já existe. Caso o nó exista, ele é usado, se não, uma nova entrada na tabela é criada. A tabela única permite a um grafo simples de múltiplas raízes representar todas as fórmulas do usuário simultaneamente.

Um outro fator na implementação de um pacote BDD é a utilização de arcos complementados [2]. Imagine dois nós BDDs que representam G e $\overline{G}$, que são similares exceto pelo fato de intercambiarem os arcos de saída, ou seja, *um(G) = zero(*$\overline{G}$*)*,

zero(G) = um($\overline{G}$) e *var(G) = var($\overline{G}$).* Esta similaridade é explorada com a utilização de arcos complementados. Um arco complementado é um arco comum com um bit a mais que indica que a fórmula que o conecta é para ser interpretada de forma complementada. Dessa forma, $\overline{G}$ pode ser representada através de arco complementado do nó *G*, reduzindo assim a quantidade de memória necessária e tornando possível a implementação da negação lógica em tempo constante (por inversão do bit de complementação).

2.2 Ordenação estática

A ordenação estática consiste em definir uma ordem para as variáveis antes de se construir o BDD [29, 16, 18]. Esta definição se dá a partir da análise estrutural das funções a serem representadas. Existem várias técnicas para ordenação estáticas de variáveis e elas dependem do tipo de problema a ser investigado.

A ordenação estática pode ser automática ou manual. Na ordenação automática, o sistema utiliza algoritmos que encontram uma ordem de variáveis analizando a estrutura do problema em questão. Na ordenação manual o usuário informa a ordem das variáveis.

Os algoritmos de ordenação estática que abordaremos aqui foram desenvolvidos, inicialmente, para ordenar variáveis BDDs que representam circuitos combinatórios, mas podem ser adaptados para ordenar BDDs que representam circuitos sequenciais, como mostraremos nas explicações dos algoritmos. Entende-se como circuito combinatório como sendo um ou mais elementos combinatórios combinados de forma acíclica. Um elemento combinatório é uma porta lógica, e as seis portas lógicas básicas são: *negação*, *e*, *ou*, *ou exclusivo*, *negação do e* e *negação do ou*. As interconexões são chamadas de **ligações** (*wires*).

Uma ligação pode conectar um valor de saída de um elemento combinatório a um valor de entrada de outro. Uma ligação pode ter apenas um valor de saída conectado a ela, enquanto pode servir como valor de entrada para mais de um elemento. Se nenhum valor de saída de qualquer elemento é conectado a uma ligação, então esta ligação é uma entrada do circuito. Se uma ligação não serve como valor de entrada para qualquer elemento, então esta ligação é uma saída do circuito. Estas saídas proveêm os resultados da computação do circuito. Uma ligação interna entre dois elementos pode também ser uma saída do circuito. Circuitos combinatórios não possuem ciclos e não possuem elementos de memória. Chama-se **rede multi-nível**, ou simplesmente **rede** um circuito que possui mais de um elemento combinatório, e que estes estão inter-

conectados. Cada entrada, saída ou elemento combinatório de uma rede é chamado de **nó** da rede. Cada nó pode possuir entradas e saídas, as entradas de um nó são também conhecidas como *fanins* do nó e as saídas são os *fanouts* do nó.

Um circuito sequencial é um circuito que possui como nós elementos de memória, como registradores e *latches*. Para a utilização destes algoritmos em ordenação de circuitos sequenciais, se considera saída de registrador como sendo uma entrada do circuito e entrada de registrador como sendo uma saída do circuito, dessa forma se faz a ordenação normalmente com os algoritmos como se o circuito fosse um circuito combinatório.

2.2.1 Busca em profundidade (*appending*)

O algoritmo de busca em profundidade [29] tem sido bastante utilizado e serviu como base para outros métodos de ordenação. Este algoritmo se baseia em fazer uma travessia em pós-ordem[1] de todos os nós do circuito a partir do nó de saída. A idéia do algoritmo é a de atribuir níveis para cada nó do circuito e ordenar estes níveis em ordem decrescente, portanto, o nó com maior nível fica no início da ordem. Todas as saídas do circuito começam com nível 0 e para todos os outros nós, o nível é dado por:

$$\text{nível}(n_i) = \max_j \ (\text{nível}(n_j)) + 1, n_j \text{ é um } \textit{fanout} \text{ de } n_i$$

Após a identificação dos níveis de uma rede, o algoritmo percorre os nós em ordem de profundidade de nó. A profundidade de um nó é o nível do nó de maior nível alcançado a partir de suas entradas (*fanins*) seguindo-se a rede no sentido inverso da propagação dos sinais. O algoritmo 1 mostra os passos para se encontrar a ordem para uma rede multi-nível.

Após todos os nós estarem ordenados, elimina-se da lista todos os nós que não são entradas do circuito, já que não serão criadas variáveis BDDs para os outros nós (ligações). Para circuitos com mais de uma saída, é criado um nó virtual tendo como entrada todas as saídas do circuito.

Para exemplificar o funcionamento deste algoritmo, consideramos o circuito da figura 2.4.

A tabela 2.2 mostra os passos para se encontrar a ordem do circuito 2.4 pelo

[1]Na travessia pós-ordem, para cada nó, primeiramente, faz-se o processamentos dos nós que são entradas para este, para então processar o nó.

Algoritmo 1 Ordenação através de busca em profundidade

```
1  begin
2     order_list = null; comment: Lista com os nós da rede ordenados
3     faninOrder(n, order_list); comment: n = nó saída do circuito
4  end
5  proc faninOrder(node, order_list) ≡
6     if node ∉ order_list
7        foreach fanin de node
8           calcule profundidade;
9        endsorted_fanin_list = fanins em ordem decrescente de profundidade;
10       foreach fanin em sorted_fanin_list
11          faninOrder(fanin, orderList);
12       end
13       append(order_list, node);
14    fi
15 end
```

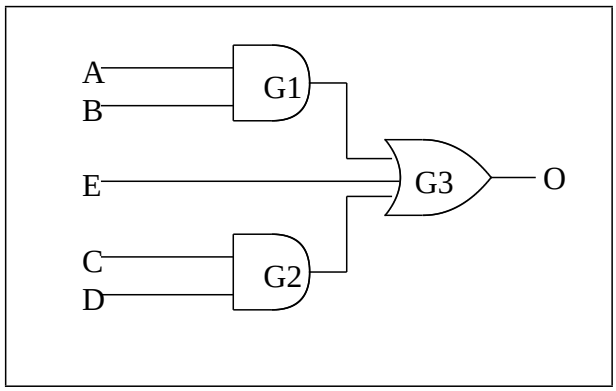


Figura 2.4: Circuito $O = (A \wedge B) \vee (C \wedge D) \vee E$

Passo	Nó visitado	Ordem
1	O	null
2	G3	null
3	G1	null
4	A	A
5	B	A,B
6	G1	A,B,G1
7	G2	A,B,G1
8	C	A,B,G1,C
9	D	A,B,G1,C,D
10	G2	A,B,G1,C,D,G2
11	E	A,B,G1,C,D,G2,E
12	G3	A,B,G1,C,D,G2,E,G3
13	O	A,B,G1,C,D,G2,E,G3,O
14	Remover	A,B,C,D,E

Tabela 2.2: Passos para encontrar a ordem de variáveis usando Busca em profundidade

algoritmo de busca em profundidade. Os níveis encontrados para cada nó são:

$$\{A = 3, B = 3, C = 3, D = 3, E = 2, G1 = 2, G2 = 2, G3 = 1, O = 0\}$$

Este método encontra ordens muito boas para circuitos combinatórios com apenas uma porta de saída, porém para circuitos com mais de uma saída ou sequenciais ele não é tão eficiente. Este fato ocorre porque, quando o circuito possui mais do que uma saída, estas saídas podem compartilhar estes nós, e o nó ficará ordenado de acordo com sua profundidade baseado na primeira saída percorrida. Esta ordenação pode ser boa para a primeira saída, contudo pode não ser boa para a outra saída, e como o nó já está ordenado, ele é ignorado, ficando numa posição ruim para a ordenação do BDD que representa esta última saída.

2.2.2 Algoritmo de entrelaçamento (*Interleaving based algorithm*)

Em *Fujii et al* [16], utilizou-se como base o método de busca em profundidade para desenvolver um algorimo que encontra uma boa ordem inicial para as variáveis de BDDs que representam circuitos. O algoritmo, chamado de *interleaving based algorithm*, percorre todo o circuito, a partir das saídas e, caso o nó ainda não esteja na lista, o acrescenta à lista e o marca como visitado. Em seguida, ele elimina da lista todos os nós que não são entradas, ou seja, remove todos os nós para os quais não serão

criadas variáveis BDDs.

Este algoritmo se baseia na idéia de que diferentes nós compartilham mesmos nós de entradas. Dessa forma, quando se passa para outra saída do circuito, os nós encontrados são colocados no início da lista e não no fim, como acontece no algoritmo de busca em profundidade. O algoritmo 2 detalha o *interleaving* e a tabela 2.3 mostra os passos para encontrar a ordem de variáveis BDDs para o circuito da figura 2.5

Algoritmo 2 Ordenação através de *Interleaving*

```
1  begin
2     VL ← ∅; comment: Lista ordenada com os nós do circuito
3     for (cada nó g)
4        g.from ← null; comment: from indica a partir de qual nó g foi visitado
5     end
6     foreach (saída o em ordem de prioridade)
7        last ← null; comment: último nó visitado
8        ordene(o)
9     end
10    remova todos os nós exceto entradas de VL;
12 end
13 proc ordene(g) ≡
14    if (g está marcado)
15       if (g.from != o)
16          last ← g;
17          g.from ← o;
18       end
19       return
20    end
21    marque g como visitado;
22    g.from ← o;
23    for (cada fan-in f de g)
24       ordene(f);
25    end
26    if (last == null)
27       insira g no topo de VL;
28 else
29       insira g após last em VL;
30    end
31    last ← g;
32 end
```

Este algoritmo pode ainda ser incrementado para melhorar a ordenação de BDDs que representam circuitos combinatórios de apenas uma saída. Para isso, calcula-se o

Passo	Nó vis.	Saída	ponto de inserção	Ordem
1	Y	Y	null	∅
2	G3	Y	null	∅
3	G1	Y	null	∅
4	A	Y	A	A
5	B	Y	B	A,B
6	G1	Y	G1	A,B,G1
7	G3	Y	G1	A,B,G1
8	G2	Y	G1	A,B,G1
9	C	Y	C	A,B,G1,C
10	D	Y	D	A,B,G1,C,D
11	G2	Y	G2	A,B,G1,C,D,G2
12	G3	Y	G3	A,B,G1,C,D,G2,G3
13	Y	Y	Y	A,B,G1,C,D,G2,G3,Y
14	Z	Z	null	A,B,G1,C,D,G2,G3,Y
15	G6	Z	null	A,B,G1,C,D,G2,G3,Y
16	G4	Z	null	A,B,G1,C,D,G2,G3,Y
17	B	Z	B	A,B,G1,C,D,G2,G3,Y
18	E	Z	E	A,B,E,G1,C,D,G2,G3,Y
19	G4	Z	G4	A,B,E,G4,G1,C,D,G2,G3,Y
20	G6	Z	G4	A,B,E,G4,G1,C,D,G2,G3,Y
21	G5	Z	G4	A,B,E,G4,G1,C,D,G2,G3,Y
22	D	Z	D	A,B,E,G4,G1,C,D,G2,G3,Y
23	F	Z	F	A,B,E,G4,G1,C,D,F,G2,G3,Y
24	G5	Z	G5	A,B,E,G4,G1,C,D,F,G5,G2,G3,Y
25	G6	Z	G6	A,B,E,G4,G1,C,D,F,G5,G6,G2,G3,Y
26	Z	Z	Z	A,B,E,G4,G1,C,D,F,G5,G6,Z,G2,G3,Y
27	remover nós			A,B,E,C,D,F

Tabela 2.3: Passos para encontrar a ordem de variáveis usando *interleaving*

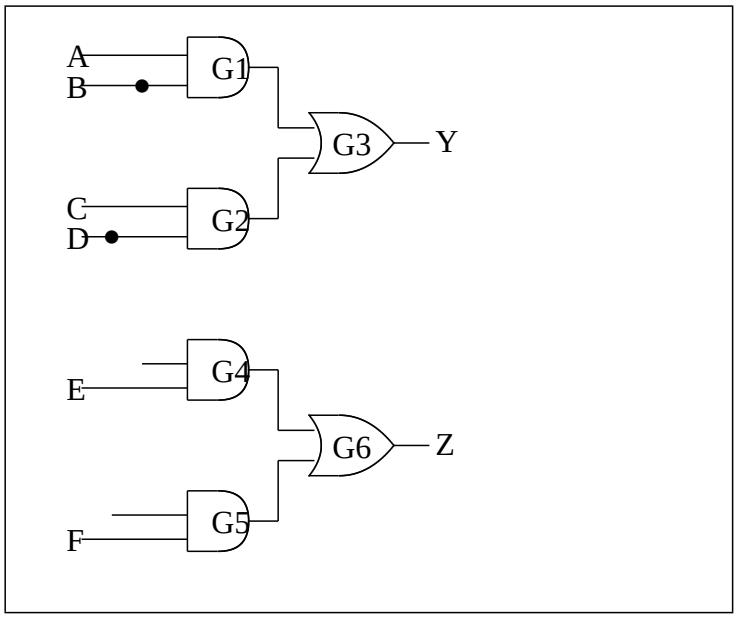


Figura 2.5: Circuito combinatório $Y = (A \wedge B) \vee (C \wedge D)$ e $Z = (B \wedge E) \vee (D \wedge F)$

número de dependências[2] de uma saída e se esta saída tiver uma dependência alta, o que gera um BDD com bastante nós então esta saída é subdivida e cada entrada deste nó é então calculada separadamente como uma saída independente (ver seção 3.2 de [16]).

Como exemplo da melhoria da divisão de uma saída em várias, consideremos o circuito descrito na figura 2.6. Este circuito possui apenas uma saída. Dessa forma, tanto o algoritmo de busca em profundidade, quanto o de *interleaving* irão encontrar a ordem $A0 < B0 < A1 < B1 < C0 < C1$. Esta ordem, apesar de ser boa para o nó X, é ruim para nó Y, tornando-se, consequentemente, ruim para o nó Z, que é a saída do circuito. Esta ordem em um gerenciador de BDDs que utiliza arcos complementados [2] contém 14 nós.

Fazendo-se a subdivisão do nó (saída) Z em dois nós (saídas) do circuito e aplicando-se o algoritmo de *interleaving*, encontramos a ordem total de variáveis como sendo $A0 < C0 < B0 < A1 < C1 < B1$, que é a melhor ordem para o circuito. Esta abordagem constrói um BDD com 9 nós, que é menor que a anterior.

[2] número de dependências é o número de nós entradas das quais este nó depende.

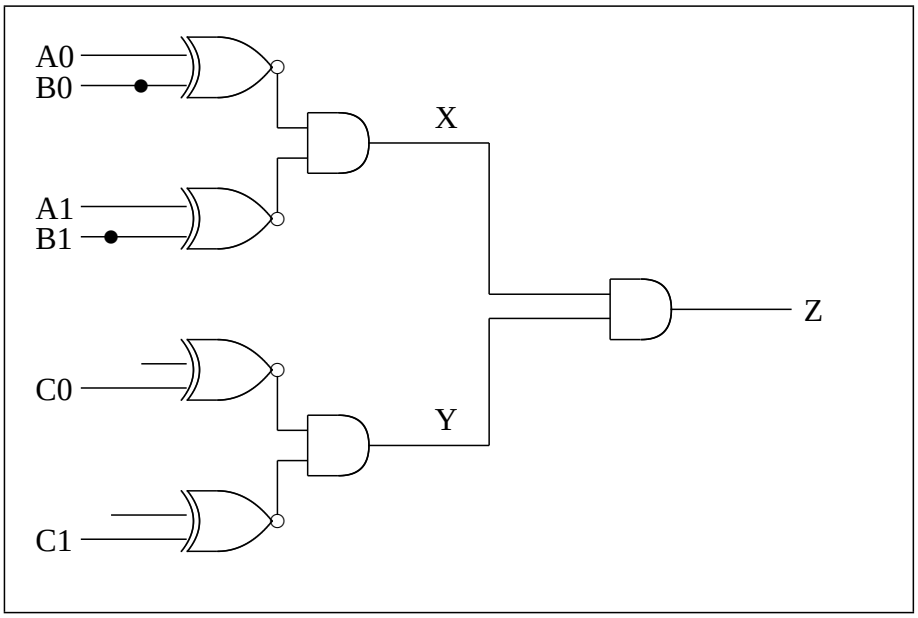


Figura 2.6: Circuito simples de apenas uma saída

2.2.3 Esquema de amostras

O método de amostragem (*sampling Scheme*) [18] tem obtido bons resultados na ordenação inicial das variáveis. Este método baseia-se na análise das informações funcionais da estrutura, ao invés da estrutura em si. Dessa forma, o método consiste em escolher partes das funções, as amostras, e construir BDDs para estas amostras, ordenando-os com métodos de reordenação, como *sifting*, e escolhendo uma melhor ordem global para as variáveis daquela função. Este método propõe, consequentemente, aplicar métodos mais caros, e mais precisos, a porções reduzidas do sistema.

Este método é dividido em quatro fases: amostra por estimação, amostra por candidatos de determinação da ordem, amostra por teste e amostra por evolução.

Amostra por estimação Esta fase decide heuristicamente os parâmetros relacionados com o tamanho dos subespaços usados para calcular os candidatos a ordem das variáveis primeiramente examinando o comportamento da função dada em algum subespaço bastante pequeno. Esta fase decide o número de variáveis usadas nas amostras e o número de amostras a serem utilizadas.

Amostra por candidatos à determinação de ordem Um conjunto de ordens de variáveis candidatas é gerado. Essencialmente, gera-se t números de funções simples e, para cada função, independentemente, chama um algoritmo de ordenação

(*sifting*, por exemplo, ver seção 2.3.2). Isto provê até *t* candidatos diferentes de ordem de variáveis.

Amostra por teste Esta fase elimina ordens ruins usando um espaço de amostra gerado independentemente, onde cada ordem gerada na fase anterior é usada para criar BDDs. As ordens candidatas com performance ruim são rejeitadas.

Amostra por evolução Esta fase permite a um dado candidato ser melhorado, reordenando-o em subespaços obtidos independentemente, para aumentar suas chances de ser apropriado para o espaço booleano completo. Dadas *m* ordens de variáveis como entrada para esta fase, a ordem de variáveis que tiver o menor grafo no subespaço escolhido para evolução é selecionada como a ordem de variáveis computadas por amostra.

Testes realizados com o método de amostragem têm se mostrados bastante eficientes quanto ao tamanho do grafo gerado, porém o tempo para se calcular a ordem inicial das variáveis é, na média, maior que o método de busca em profundidade [29]. Porém o tempo pode ser diminuido com a redução do número de amostras, com potencial redução na qualidade do resultado. Valores ideais para o tamanho da amostra devem ser avaliados em cada siatuação.

2.2.4 Algoritmo genético paralelo

Em [13] foi desenvolvido um método de ordenação estática que usa a idéia dos algoritmos genéticos e paralelismo para encontrar uma ordem inicial para os BDDs.

Algoritmo genético é uma meta-heurística para a busca de uma solução ótima de problemas. Para este algoritmo, considera-se um indivíduo como sendo uma solução provável para o problema. Este algoritmo se baseia na seleção natural, e, a partir de um conjunto de indivíduos, que formam um população inicial, vai evoluindo os indivíduos da população até que um indivíduo de uma população alcance um nível de exigência ou haja *n* iterações sem melhoria nos indivíduos.

No caso de ordenação de BDDs, um indivíduo é uma ordem de variáveis BDDs e sua qualidade é o número de nós, sendo melhor qualidade um número menor de nós. A idéia do algoritmo genético paralelo é a de criar várias populações iniciais independentes e evoluir cada população independentemente, em paralelo. Em determinados pontos há um intercâmbio entre as populações, onde indivíduos de uma população mudam, ou são copiados, para outra população.

Testes iniciais mostraram que o método, apesar de tomar bastante tempo, encontra ordens de variáveis boas. Os testes foram feitos sobre circuito ISCAS85 e comparados com o algoritmo de entrelaçamento (ver seção 2.2.2). O algoritmo de entrelaçamento é bastante rápido, porém o método genético paralelo encontra ordens iniciais melhores que o primeiro.

2.3 Ordenação dinâmica

A ordenação dinâmica, ou reordenação, consiste em alterar a ordem das variáveis BDDs depois da construção dos mesmos [28, 24, 26, 31, 20]. Métodos de ordenação dinâmicos Os métodos de reordenação mais simples tentam diminuir o número de nós BDDs alterando a posição das variáveis e identificando em que posição aquela variável estava quando o número de nós era o menor possível.

A manipulação de funções booleanas está sempre criando novos nós BDDs, o que aumenta o espaço alocado. Através das técnicas de reordenação, pode-se diminuir o espaço alocado e ainda acelerar e otimizar os algoritmos de manipulação dos BDDs.

As técnicas de reordenação apresentadas aqui fazem a reordenação independente da aplicação, ou seja, a reordenação é feita internamente nas estruturas de manipulação dos BDDs e fica totalmente transparente ao usuário. Alguns métodos apresentados aceitam parâmetros passados pela aplicação que são usados para fazer a reordenação de forma dirigida [26, 31, 20]. Estes métodos, apesar de serem independentes da aplicação, podem ser guiados pelas mesmas.

2.3.1 Permutação de variáveis adjacentes

Um fator importante nas técnicas de ordenação dinâmica é o fato de que permutar duas variáveis adjacentes tem um custo baixo [28]. Porém, existem dois pontos que devem ser considerados para este problema. O primeiro é que deve-se encontrar todos os vértices no nível i sem ter que percorrer todo o grafo. O segundo é que cada nó no grafo deve representar a mesma função antes e depois da permutação.

Um esquema eficiente de memória é usado para encontrar todos os nós de um determinado nível. Usa-se um esquema de vetores de tabelas de dispersão ao invés da tabela única, com uma tabela por nível do grafo.

Um nó F no nível i pode ser apontado por outros nós acima no grafo e qualquer função que seja retornada ao usuário. Por questão de redução de memória, não se mantém ponteiros inversos, dessa forma torna-se impossível encontrar todas as referências

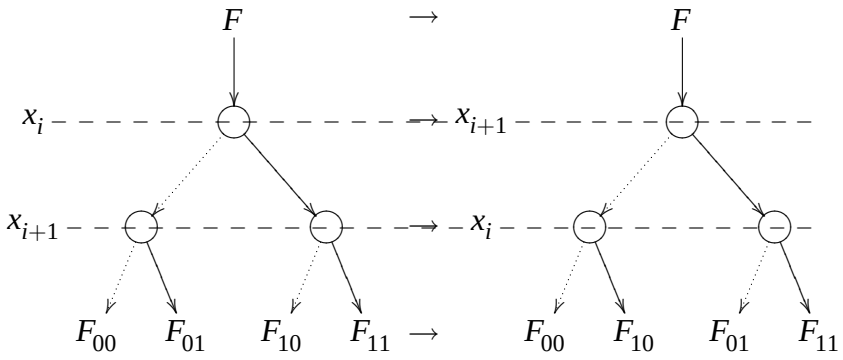


Figura 2.7: Permutação de duas variáveis adjacentes

ao nó F sem ter que percorrer todo o grafo. Entretanto, para se fazer uma permutação de variáveis locais, é necessário manter uma função lógica idêntica em cada nó. Isto é feito sobrescrevendo o nó que representa F pelo novo nó que resulta da modificação da ordem das variáveis, da seguinte maneira.

Seja $F = (x_i, F_1, F_0)$ um nó no nível i. Seja F_{11} o cofator positivo de F_1 em relação a x_{i+1}. O cofator é o nó apontado pelo *um(v)* de F_1 se x_{i+1} é a variável no topo de F_1 ou F_1 nos outros casos. Similarmente, seja F_{10} o cofator negativo de F_1 e seja F_{01} e F_{00} os dois cofatores de F_0. O nó F é reescrito sobre a tupla $(x_{i+1}, (x_i, F_{11}, F_{01}), (x_i, F_{10}, F_{00}))$. A expansão dessa fórmula mostra que ela preserva a função do nó F e inspeções garantem que a nova ordem de variáveis (x_{i+1} está antes de x_i) é estabelecida para todos os caminhos sobre F:

$$
\begin{aligned}
& F(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) = \\
& = x_i \bullet F_1(x_1, \ldots, x_n) + \overline{x}_i \bullet F_0(x_1, \ldots, x_n) \\
& = x_i \bullet x_{i+1} \bullet F_{11}(x_1, \ldots, x_n) + x_i \bullet \overline{x}_{i+1} \bullet F_{10}(x_1, \ldots, x_n) + \\
& \qquad \overline{x}_i \bullet x_{i+1} \bullet F_{01}(x_1, \ldots, x_n) + \overline{x}_i \bullet \overline{x}_{i+1} \bullet F_{00}(x_1, \ldots, x_n) \\
& = x_{i+1} \bullet [x_i \bullet F_{11}(x_1, \ldots, x_n) + \overline{x}_i \bullet F_{01}(x_1, \ldots, x_n)] + \\
& \qquad \overline{x}_{i+1} \bullet [x_i \bullet F_{01}(x_1, \ldots, x_n) + \overline{x}_i \bullet F_{00}(x_1, \ldots, x_n)]
\end{aligned}
$$

A figura 2.7 ilusta essa mudança.

Os novos nós requeridos no nível i ((x_i, F_{11}, F_{01}) e (x_i, F_{10}, F_{00})) podem ser nós degenerados (por exemplo, quando $F_{11} = F_{01}$), ou podem já existir no grafo quando requeridos para implementar outras funções. Quando F é re-expressado como resultado de permutação de variáveis, os grafos cujas raizes são F_1 e F_0 podem ser liberados, caso não estejam sendo referenciados.

Um pacote de BDD eficiente utiliza apontadores para nós de BDDs, o que torna

$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	inicial
$x_1, x_2, x_3, x_5, x_4, x_6, x_7$	permutar(x_4, x_5)
$x_1, x_2, x_3, x_5, x_6, x_4, x_7$	permutar(x_4, x_6)
$x_1, x_2, x_3, x_5, x_6, x_7, x_4$	permutar(x_4, x_7)
$x_1, x_2, x_3, x_5, x_6, x_4, x_7$	permutar(x_7, x_4)
$x_1, x_2, x_3, x_5, x_4, x_6, x_7$	permutar(x_6, x_4)
$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	permutar(x_5, x_4)
$x_1, x_2, x_4, x_3, x_5, x_6, x_7$	permutar(x_3, x_4)
$x_1, x_4, x_2, x_3, x_5, x_6, x_7$	permutar(x_2, x_4)
$x_4, x_1, x_2, x_3, x_5, x_6, x_7$	permutar(x_1, x_4)
$x_1, x_4, x_2, x_3, x_5, x_6, x_7$	permutar(x_4, x_1)
$x_1, x_2, x_4, x_3, x_5, x_6, x_7$	permutar(x_4, x_2)
$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	permutar(x_4, x_3)
$x_1, x_2, x_3, x_5, x_4, x_6, x_7$	permutar(x_4, x_5)
$x_1, x_2, x_3, x_5, x_6, x_4, x_7$	permutar(x_4, x_6)
$x_1, x_2, x_3, x_5, x_6, x_7, x_4$	permutar(x_4, x_7)

Tabela 2.4: Exemplo do algoritmo de *sifting*

essa troca basicamente como mudança de apontadores, ou seja, ela é bastante rápida.

2.3.2 *Sifting*

Dentre os vários métodos de reordenação, o método de *sifting* [28] tornou-se bastante popular, pois permite encontrar boas ordens em tempos razoáveis. Este algoritmo se utiliza das características da permutação de variáveis adjacentes (ver seção 2.3.1) para procurar a melhor posição de cada variável no grafo. Dessa forma o algoritmo consiste em, para cada variável v_i do grafo em ordem decrescente de número de nós no nível, permutá-la em todas as posições do grafo e armazenar a posição dessa variável em que o grafo ficou com o menor número de BDDs, depois mover a variável em questão para essa posição.

Este algoritmo requer $O(n^2)$ permutações de níveis adjacentes no grafo. Cada uma dessas permutações possui complexidade proporcional ao tamanho do grafo. Para controlar o pior caso, a busca em uma determinada direção é terminada se o tamanho do grafo cresce duas vezes o tamanho original. A tabela 2.4 mostra a permutação de uma variável BDD (x_4) e seu posicionamento. As 7 possíveis posições da variável são exploradas usando 9 permutações adjacentes e sua posição ótima é restaurada usando 6 permutações (pior caso).

Este método de reordenação é implementado diretamente na biblioteca de BDDs

$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	inicial
$x_1, x_3, x_2, x_4, x_5, x_6, x_7$	permutar(x_2, x_3)
$x_1, x_3, x_4, x_2, x_5, x_6, x_7$	permutar(x_2, x_4)
$x_1, x_4, x_4, x_2, x_5, x_6, x_7$	permutar(x_3, x_4)
$x_1, x_4, x_2, x_3, x_5, x_6, x_7$	permutar(x_3, x_2)
$x_1, x_2, x_4, x_3, x_5, x_6, x_7$	permutar(x_4, x_2)
$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	permutar(x_4, x_3)
$x_1, x_3, x_2, x_4, x_5, x_6, x_7$	permutar(x_2, x_3)
$x_1, x_3, x_4, x_2, x_5, x_6, x_7$	permutar(x_2, x_4)

Tabela 2.5: Exemplo de permutação em *Windowing*

e fica funcionalmente transparente para o usuário. Normalmente é definido um valor limite de utilização de memória (número de nós BDDs) para que o processamento seja parado, seja feita a reordenação e o processamento continue normalmente. O usuário também pode invocá-lo explicitamente quando achar necessário uma reordenação sem que seja atingido o valor limite.

2.3.3 *Windowing*

Outro método que ficou bastante conhecido e utiliza a permutação de variáveis como base é o método de *Windowing* [24]. A idéia do algoritmo consiste em criar janelas (*windows*) de k variáveis e fazer todas as possíveis $k!$ combinações permutando as variáveis dessas janelas. Isto é feito fazendo-se todas as $k! - 1$ permutações entre as variáveis, armazenando a melhor ordem, e depois fazendo-se, no pior caso, mais $k(k-1)/2$ permutações para restaurar o BDD da melhor ordem.

Uma janela de tamanho k no nível i é composta das variáveis do nível i até o nível $i+k$. As janelas são definidas como sendo do nível 0 até o nível $n-k$. A tabela 2.5 mostra a permutação de uma janela de tamanho 3 começando na variável x_2. 6 (seis) permutações adjacentes são necessárias para todas as combinações e mais 3 (três) permutações adjacentes adicionais (pior caso) para restaurar a melhor permutação.

Como este método utiliza a idéia de permutação de variáveis adjacentes, ele é bastante rápido. Inclusive mais rápido que o método de *sifting* (ver seção 2.3.2), porém como está limitado a utilização de janelas de tamanho k, ele não encontra ordens ótimas e o comportamento geral da biblioteca de BDDs pode se degradar rapidamente. Testes realizados mostram que o tamanho da janela deve ficar entre 3 e 6.

2.3.4 *Block Restricted Sifting*

O método de *Block Restricted Sifting* [26] consiste em criar blocos no BDD e realizar o *Sifting* apenas sobre os blocos. Este método é particularmente interessante a se utilizar em domínios de aplicação onde se pode determinar automaticamente quando variáveis devem permanecer adjacentes na ordem. Este método se baseia em que mudanças na ordem das variáveis em um bloco não tem qualquer influência em variáveis fora desse bloco, estejam estas variáveis acima ou abaixo do bloco.

O número de blocos e o tamanho de cada bloco pode ser definido de forma manual ou ser calculado por uma função, definida por *Meinel et Slobodova* [26], que calcula o número de subfunções da raiz do OBDD e baseado nestes valores, define quantos blocos existem e qual o tamanho de cada bloco.

Resultados de reordenação utilizando este método mostraram uma melhoria considerável no tempo para construção de OBDDs que representam circuitos combinatórios com um pequeno acréscimo no espaço utilizado em relação ao uso do *sifting*.

2.3.5 Método de amostras

O método de amostras [31] consiste em escolher um subconjunto de variáveis, as amostras, e fazer a reordenação nessas amostras.

A primeira questão relativa a esse método é a de como encontrar uma boa amostra. Uma boa amostra traduz-se como uma amostra que tendo uma ordem ótima, esta ordem seja também ótima para todo o grafo. Outra questão é a de como encontrar uma ordenação ótima para a amostra. E a terceira questão é como fazer a adaptação da amostra para o BDD inteiro.

A primeira questão pode ser resolvida fazendo-se amostras randomicamente. Esta abordagem tem a vantagem de poder ser usada em qualquer aplicação. Experimentos mostraram que o tamanho da amostra deve ser maior do que a metade do grafo para se obter bons resultados. *Meinel et Slobodova* [31] realizou testes em circuitos combinatórios usando como amostra as raízes (saídas) do circuito.

Para reordenar a amostra pode-se utilizar qualquer tipo de reordenação. Nos testes realizados foi utilizado o método de *sifting* para fazer a reordenação da amostra. O método de *sifting* aplicado diretamente em todo o grafo é bastante lento, enquanto que aplicado nas amostras encontra ordens de variáveis aceitáveis em tempos melhores que aplicando-se a todas as variáveis.

A solução trivial para a terceira questão é a de manter as variáveis que não estão na amostra em suas respectivas posições e fazer a reordenação das outras para suas

novas posições, definidas pela reordenação da amostra. Esta abordagem tem a vantagem de verificar se houve, na transição das variáveis, uma ordem melhor do que a final encontrada e voltar para a melhor ordem após a reordenação.

2.3.6 *Adaptive Variable Reordering*

Baseado na idéia de que a manipulação de BDDs altera bastante o tamanho do grafo e de que o gargalo principal na métrica utilizada não é o espaço alocado e sim o tempo de computação, *Kahmi* [20] desenvolveu um algoritmo que busca manter o BDD em tamanhos aceitáveis para o término do processamento em andamento.

Conhecido como *adaptive variable ordering*, este método se baseia em informações específicas da aplicação para fazer a reordenação das variáveis, são elas:

- O grafo de conectividade das funções sendo avaliadas e
- A fase da verificação no momento da chamada à reordenação.

Além dessas observações, este método faz uso de fase da verificação em andamento para chamar explicitamente a reordenação.

Para aumentar a localidade das variáveis sendo reordenadas, *adaptive reordering* computa uma ordem de avaliação para as funções de próximo estado que minimiza o suporte cumulativo de elementos sequenciais no modelo sendo verificado. Isto é feito usando uma algoritmo de busca que minimiza a função custo seguinte da variável de permutação α:

$$cost(\alpha) = \sum_{i \leq j \leq n} \left| \bigcup_{i \leq i \leq j} supp(f_{\alpha i}) \right|$$

Onde $|A|$ denota a cardinalidade de A, $f_{\alpha i}$ denota a função de próximo estado do registro (*latch*) α_i e $supp(f_{\alpha i})$ corresponde ao conjunto de variáveis no suporte de $f_{\alpha i}$, e n é o número de elementos sequenciais.

A utilização desta heurística no domínio de reordenação durante a verificação de modelos pode ser feita como descrito abaixo:

1. Encontrar uma ordem inicial para as variáveis BDDs utilizando algum método de ordenação estática (ver seção 2.2);
2. Encontrar uma ordem *ord* para as variáveis de estado que minimizem o suporte cumulativo. Avaliar as funções de próximo estado descritas por *ord* na fase de construção da relação de transição da verificação;

3. Guardar quais funções de importância estão sendo avaliadas durante a verificação;

4. Caso a reordenação seja chamada (seja automaticamente por atingir valores limites ou explicitamente) durante a avaliação de uma função de importância, reordenar apenas as variáveis que dão suporte a essa função.

Uma outra característica percebida é que forçando a reordenação em certas fases da verificação, aumenta-se drasticamente a performance da verificação. Isto acontece por que chamando a reordenação com mais nós BDDs alocados, o tempo de reordenação será maior. Dessa forma, forçando a reordenação em certas fases, pode-se evitar a aplicação de atingir valores limites.

Teste feitos no SMV [21] comparando este método como método de *sifting*, onde o *sifting* só é chamado quando atingido valores limites, mostrou que este método é eficiente para a verificação de modelos. Apesar do tempo de reordenação ser maior do que no *sifting*, o tempo total de computação é menor do que usando-se apenas *sifting* e houve casos em que a computação estourou o tempo limite quando usando apenas o *sifting* e terminou normalmente usando o *adaptive variable reordering*.

3 Verificação automática de sistemas de transição finita

A verificação simbólica de modelos [7] (*Symbolic Model Checking* – SMC) tem sido bastante utilizada para se fazer verificação automática de sistemas que possam ser modelados como máquinas de estados finitos. Para se fazer a verificação é realizada a especificação da propriedade a ser verificada através de um formalismo adequado, como uma lógica temporal, e determina-se se a máquina de estados finitos que representa o sistema descrito satisfaz a propriedade especificada. Em outras palavras, verifica-se se uma fórmula f é verdadeira em um grafo G de transição de estados.

Uma máquina de estados finitos – MEF – pode ser descrita como uma máquina abstrata contendo estados e transições entre estados. Este modelo é bastante usado na verificação de sistemas computacionais, como circuitos, protocolos, programas e outros. Estas MEFs são conhecidas como **Estruturas de Kripke**.

O termo *Symbolic Model Checking* surgiu a partir da utilização de BDDs para a representação do sistema, onde o sistema é modelado como uma estrutura de Kripke e representado simbolicamente através de BDDs. Dessa forma, faz-se a verificação no modelo do sistema, que são estruturas de Kripke representadas através de BDDs.

Uma MEF é um modelo de uma expressão lógica quando esta expressão é verdadeira nos estados iniciais da máquina. A Verificação Simbólica de Modelos – VSM – permite verificar se um determinado grafo, que representa uma MEF, é um modelo de uma determinada expressão lógica [10]. Através dela é possível verificar sistemas relativamente grandes, pois são representados e manipulados conjuntos de estados, que são representados através de suas funções características (expressadas em lógica proposicional), reduzindo o tamanho da representação do espaço de estados, já que a representação por enumeração tende a ocupar muito espaço em sistemas com muitos estados [5, 8].

Neste capítulo iremos conhecer a verificação simbólica de modelos. Primeiramente, na seção 3.1, iremos abordar os conceitos das estruturas de Kripke, máquinas

de estados usadas para representar sistemas. Na seção 3.2 veremos como modelar sistemas de transições finitas através de estruturas de Kripke e na seção 3.3 mostraremos como representar estas estruturas usando BDDs. Logo a seguir, na seção 3.4 introduziremos como representar máquinas de estados finitos usando BDDs. A seção 3.5 faz uma introdução aos conceitos de reticulado e ponto fixo. Na seção 3.6 veremos como encontrar todos os estados alcançáveis da máquina representada a partir dos estados iniciais. Na seção 3.7 veremos como são descritas as propriedades para a verificação automática. E, por fim, na seção 3.8 conheceremos os algoritmos usados para a verificação.

3.1 Estruturas de Kripke

As estruturas de Kripke vêm sendo largamente utilizadas na verificação simbólica de modelos com lógica temporal. Estruturas de Kripke são modelos de sistemas de transição não-determinísticos baseados em estados e que servem para representar várias classes de sistemas. Podem ser definidas como: um conjunto S de estados, um conjunto R de transições entre estados, um conjunto I de estados iniciais e uma função que associa a cada estado um conjunto de proposições verdadeiras naquele estado.

Definição 1 (Estrutura de Kripke) *Seja P um conjunto finito de proposições booleanas. Uma estrutura de Kripke sobre P é uma quádrupla* $K = (S, R, I, L)$ *onde:*

- *S é um conjunto de estados (quando S é finito, então K é uma* estrutura de Kripke finita*);*
- $R \subseteq S \times S$ *é uma relação de transição, tal que* $\forall s \in S \bullet \exists s' \in S, (s, s') \in T$;
- $I \subseteq S$ *é o conjunto de estados iniciais;*
- $L : S \rightarrow 2^P$ *é uma função de rotulamento. Para cada estado, L associa um conjunto de proposições booleanas verdadeiras naquele estado.*

Um caminho válido na estrutura K a partir de um estado $s \in S$ é uma sequência infinita de estados $\pi = s_0, s_1, s_2, \ldots$ tal que $s_0 = s$ e $(s_i, s_{i+1}) \in R$, para todo $i \geq 0$.

Consideremos $V = \{v_1, \ldots, v_n\}$ o conjunto de variáveis de um sistema e $D = D_1 \times D_2 \ldots D_n$ o domínio. Uma valoração para os elementos de V é considerado um estado do sistema. Um estado do sistema é um mapeamento $V \rightarrow D$ descrito através de uma fórmula booleana que é verdadeira no estado. Para exemplificar, considere

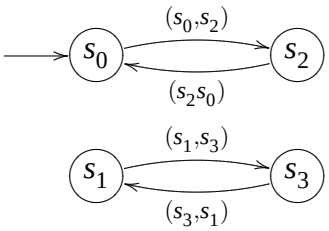


Figura 3.1: Grafo de transição de estados

$V = \{v_1, v_2, v_3\}$ e $D = B \times B \times B$, onde $B = \{0,1\}$, sabendo-se que $v_1 \leftarrow 1$, $v_2 \leftarrow 0$ e $v_3 \leftarrow 1$ em um determinado estado, este estado pode ser descrito através da expressão $v_1 \wedge \neg v_2 \wedge v_3$. Podemos observar também que uma fórmula pode representar um conjunto de estados, ou seja, representa todos os estados nos quais ela é verdadeira. Por exemplo, considerando a fórmula $v_1 \wedge \neg v_2$, ela representa 2 estados, pois v_3 pode assumir qualquer valor booleano.

As mudanças de estados em um sistema são chamadas de transições. Transições são pares ordenados de estados (e_1, e_2), onde e_1 é o estado atual e e_2 é o próximo estado.

Consideremos o conjunto $V = \{x,y\}$, $D = \{0,1\} \times \{0,1\}$ e uma única função de transição $x \leftarrow (x+1) mod 2$. A estrutura de Kripke abaixo descreve este sistema:

- $S = D \times D$, ou seja, $S = \{s_0, s_1, s_2, s_3\}$
- $R = \{(s_0, s_2), (s_2, s_0), (s_1, s_3), (s_3, s_1)\}$
- $I = \{s_0\}$
- $L(s_0) = \{\}, L(s_2) = \{p_0\}, L(s_1) = \{p_1\}, L(s_3) = \{p_0, p_1\}$, onde $p_0 \equiv x = 1$ e $p_1 \equiv y = 1$

A figura 3.1 ilustra esta estrutura. Observe que os estados s_1 e s_3 não são alcançáveis, pois partindo-se do conjunto de estados iniciais $I = \{s_0\}$ não há transições para estes estados. Esta estrutura possui apenas um caminho partindo-se dos estados iniciais, que é $s_0, s_2, s_0, s_2, s_0, s_2 \ldots$ Esta é a única computação desse sistema.

Veremos na próxima seção como circuitos sequenciais podem ser modelados por estruturas de Kripke, e nas seções 3.7 e 3.8 como especificar e verificar automaticamente propriedades sobre o comportamento de estruturas de Kripke.

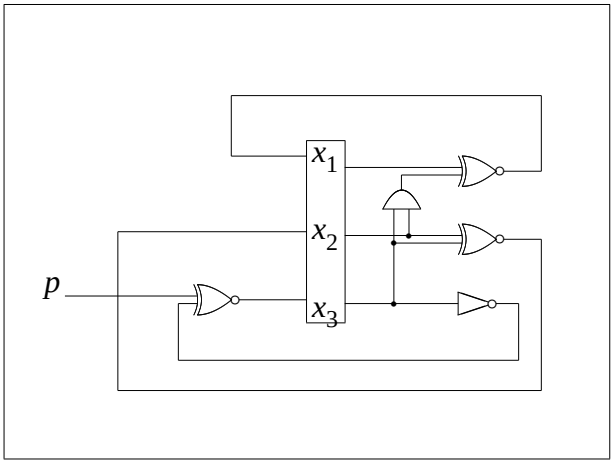


Figura 3.2: Contador módulo 8

3.2 Modelagem de circuitos sequenciais com Estruturas de Kripke

Circuitos sequenciais são circuitos cujos valores das portas de saída dependem dos valores das portas de entradas e dos valores armazenados nos registradores. Podemos dizer que um circuito sequencial é composto de cinco tipos de elementos: entradas, registradores, saídas, elementos combinatórios e ligações. Um estado do circuito é descrito a partir dos valores de seus registradores e de suas entradas. Consideramos que um circuito sequencial seja modelado como uma máquina de estados finitos quando considerarmos as valorações das entradas e dos registradores como sendo um estado da máquina.

Seja $E = \{e_1, \ldots, e_n\}$ as entradas do circuito e $X = \{x_1, \ldots, x_m\}$ seus registradores, todos valores booleanos. O número de estados da estrutura pode ser definido como $2^{|E|+|X|}$, onde $|Y|$ é o número de elementos de Y, ou também podemos escrever 2^{n+m}.

As transições são obtidas a partir da identificação de dois estados subsequentes. Se dois estados e_x e e_y podem acontecer no circuito com uma mudança da valoração das variáveis de E e X diretamente de um para o outro, isto caracteriza uma transição $(e_x, e_y) \in R$.

Consideremos o circuito da figura 3.2. Para que ele seja modelado como uma estrutura de Kripke, observamos que $E = \{p\}$ e $X = \{x_1, x_2, x_3\}$. O número de estados da estrutura de Kripke é:

$$2^{1+3} = 2^4 = 16$$

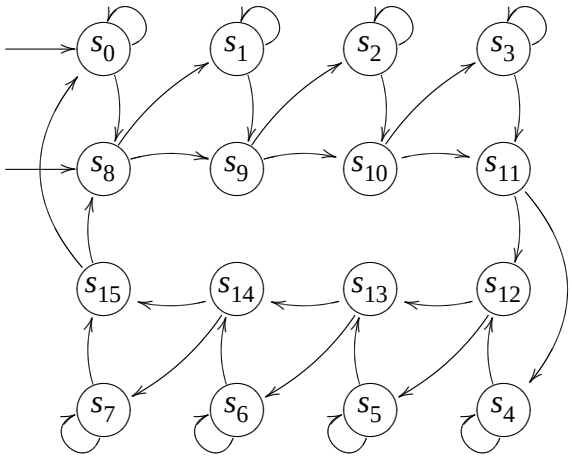


Figura 3.3: Estrutura de Kripke do circuito Contador Módulo 8

Na estrutura de Kripke que modela este circuito, $V = \{p, x_1, x_2, x_3\}$ e $D = \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\}$. A seguir temos esta estrutura e sua representação gráfica na figura 3.3

- $S = D \times D \times D \times D$, ou seja, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}\}$
- $R = \{(s_0, s_0), (s_0, s_8), (s_1, s_1), (s_1, s_9), (s_2, s_2), (s_2, s_{10}), (s_3, s_3), (s_3, s_{11}), (s_4, s_4), (s_4, s_{12}), (s_5, s_5), (s_5, s_{13}), (s_6, s_6), (s_6, s_{14}), (s_7, s_7), (s_7, s_{15}), (s_8, s_1), (s_8, s_9), (s_9, s_2), (s_9, s_{10}), (s_{10}, s_3), (s_{10}, s_{11}), (s_{11}, s_4), (s_{11}, s_{12}), (s_{12}, s_5), (s_{12}, s_{13}), (s_{13}, s_6), (s_{13}, s_{14}), (s_{14}, s_7), (s_{14}, s_{15}), (s_{15}, s_0), (s_{15}, s_8)\}$
- $I = \{s_0, s_8\}$
- $L(s_0) = \{\}, L(s_1) = \{x_3\}, L(s_2) = \{x_2\}, L(s_3) = \{x_2, x_3\}, L(s_4) = \{x_1\}, L(s_5) = \{x_1, x_3\}, L(s_6) = \{x_1, x_2\}, L(s_7) = \{x_1, x_2, x_3\}, L(s_8) = \{p\}, L(s_9) = \{p, x_3\}, L(s_{10}) = \{p, x_2\}, L(s_{11}) = \{p, x_2, x_3\}, L(s_{12}) = \{p, x_1\}, L(s_{13}) = \{p, x_1, x_3\}, L(s_{14}) = \{p, x_1, x_2\}, L(s_{15}) = \{p, x_1, x_2, x_3\}$

3.3 Representação de Estrutura de Kripke com BDDs

Estruturas de Kripke são usadas como modelo para a verificação. Elas são representadas simbolicamente através de BDDs. Para que possamos representar uma estrutura de Kripke através de BDDs, primeiramente devemos considerar os elementos do conjunto V. Os elementos de V que são do tipo booleano são mapeados diretamente através de variáveis BDDs, uma vez que estas são variáveis booleanas. Consideraremos que as variáveis são todas booleanas[1].

Para que possamos representar estruturas de Kripke através de BDDs, consideraremos, primeiramente, como expressar uma estrutura de Kripke através de expressões da lógica proposicional. Cada estado é representado através de sua função característica. Seja $M = (S, R, I, L)$ uma estrutura de Kripke sobre o conjunto de proposições booleanas $P = \{v_1, \ldots, v_n\}$. Consideremos v denotando o valor $(v_1, \ldots, v_n)$. A função característica de um estado $s \in S$, denotado por $[s]$, é definida como:

$$[s](v) = \left(\left(\bigwedge_{v_i \in L(s)} v_i \right) \wedge \left(\bigwedge_{v_i \notin L(s)} \neg v_i \right) \right)$$

A definição de funções características pode ser extendida para conjuntos de estados a partir das seguintes definições:

$$[\{\}](v) = 0$$

$$[\{x\} \cup X](v) = [x](v) \vee [X](v)$$

Utilizando estas definições, é possível expressar estados e conjuntos de estados através de suas funções características. Essas definições se limitam a representar os estados da estrutura separadamente. Para que possamos expressar as transições entre estados, primeiramente utilizaremos um conjunto $P' = \{v'_1, \ldots, v'_n\}$, que é uma cópia das variáveis de estado e expressa o próximo estado. A função característica de uma transição $t = (e_1, e_2) \in R$, denotado por $[t]$, é definida como:

$$[t](v, v') = [e_1](v) \wedge [e_2](v')$$

Analogamente a representação de estados, podemos extender esta definição para

[1] Ná prática, um circuito é formado de valores booleanos, onde qualquer outro tipo de valor é mapeado para variáveis booleanas.

representar conjunto de transições.

As funções características da estrutura de Kripke na figura 3.3 são:

- $[S] = \{(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)\}$
- $[(e_0, e_1)] = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \wedge (\neg x'_1 \wedge \neg x'_2 \wedge x'_3)$
- $[R] = ((\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \wedge (\neg x'_1 \wedge \neg x'_2 \wedge x'_3)) \vee$
  $((\neg x_1 \wedge \neg x_2 \wedge x_3) \wedge (\neg x'_1 \wedge x'_2 \wedge \neg x'_3)) \vee$
  $((\neg x_1 \wedge x_2 \wedge \neg x_3) \wedge (\neg x'_1 \wedge x'_2 \wedge x'_3)) \vee$
  $((\neg x_1 \wedge x_2 \wedge x_3) \wedge (x'_1 \wedge \neg x'_2 \wedge \neg x'_3)) \vee$
  $((x_1 \wedge \neg x_2 \wedge \neg x_3) \wedge (x'_1 \wedge \neg x'_2 \wedge x'_3)) \vee$
  $((x_1 \wedge \neg x_2 \wedge x_3) \wedge (x'_1 \wedge x'_2 \wedge \neg x'_3)) \vee$
  $((x_1 \wedge x_2 \wedge \neg x_3) \wedge (x'_1 \wedge x'_2 \wedge x'_3)) \vee$
  $((x_1 \wedge x_2 \wedge x_3) \wedge (\neg x'_1 \wedge \neg x'_2 \wedge \neg x'_3))$
- $[I] = \neg x_1 \wedge \neg x_2 \wedge \neg x_3$

3.4 Representando circuitos diretamente com BDDs

Apesar de usarmos estruturas de Kripke para modelar os circuitos, na prática os circuitos são mapeados diretamente para BDDs. As ferramentas de verificação, como CV [14] e VIS [3], constróem os BDDs a partir de descrições feitas em linguagens de descrição de *hardware* (*Hardware Description Language* – HDL), como VHDL, VerilogHDL ou formatos como BLIF-MV. Estas linguagens descrevem circuitos a partir de definições de variáveis e atribuições sobre estas variáveis, além de estruturas de controle. Nesta seção mostraremos como criar as relações de transição diretamente a partir das atribuições feitas aos registradores em um circuito sequencial.

Através de uma leitura sobre uma descrição em HDL é possível definir quais os possíveis valores de um registrador no próximo estado a partir de seus *fanins*. A função f_i determina o valor do registrador i no próximo estado do circuito. Seja $X = \{x_1, x_2, x_3, \ldots, x_n\}$, $E = \{e_1, e_2, e_3, \ldots, e_l\}$, $X' = \{x'_1, x'_2, x'_3, \ldots, x'_n\}$ e $W = X \cup E$, e seja $f_k(W)$ a função que computa o próximo estado de x_k, o valor de cada variável do circuito no próximo estado pode ser encontrado através da relação $x'_k \Leftrightarrow f_k(W)$ e a

relação de transição do circuito como um todo pode ser obtida pela fórmula

$$R(W,X') \quad = \quad \bigwedge_{i=0}^{n} v_i' \Leftrightarrow f_i(W).$$

A função $f_i(W)$ é chamada função de transição da variável v_i.

Consideremos agora o circuito da figura 3.2. Seja $X = \{x_1, x_2, x_3\}$ o conjunto de variáveis de estado deste circuito, $E = \{p\}$ as entradas do circuito, e seja $X' = \{x_1', x_2', x_3'\}$ uma cópia destas variáveis de estado. Então $W = X \cup E = \{p, x_1, x_2, x_3\}$. A transição do contador módulo 8 é dada por:

$$\begin{aligned} x_1' &= ((x_3 \wedge x_2) \oplus x_1) \\ x_2' &= x_3 \oplus x_2 \\ x_3' &= \neg x_3 \oplus p \end{aligned}$$

Como p representa uma entrada do circuito, seu valor é não-determinístico, podendo assumir qualquer valor do domínio. As seguintes equações são usadas para definir as relações

$$\begin{aligned} R_1(W,X') &= (x_1' \Leftrightarrow ((x_3 \wedge x_2) \oplus x_1)), \\ R_2(W,X') &= (x_2' \Leftrightarrow (x_3 \oplus x_2), \\ R_3(W,X') &= (x_3' \Leftrightarrow \neg x_3 \oplus p), \end{aligned}$$

que descrevem a restrição onde cada x_i' deve satisfazer em uma transição legal. Estas restrições podem ser combinadas através de suas conjunções para formar a relação de transição:

$$R(W,X') = R_1(W,X') \wedge R_2(W,X') \wedge R_3(W,X').$$

No caso geral, em circuitos com n variáveis de estados, $X = \{x_1, \ldots, x_n\}$ e $X' = \{x_1', \ldots, x_n'\}$. De forma análoga ao contador módulo 8, para cada x_i' há uma função f_i tal que

$$x_i' = f_i(W).$$

Estas equações são usadas para definir as relações

$$R_i(W,X') = (x_i' \Leftrightarrow f_i(W)).$$

E a conjunção dessas relações formam a relação de transição

$$R(W,X') = R_1(W,X') \wedge \cdots \wedge R_n(W,X').$$

Dessa forma, a relação de transição de um circuito pode ser expressada como sendo uma conjunção das relações de transição atômicas.

Dado um BDD para cada função f_i, é fácil computar o BDD que representa R. Esta relação de transição é chamada de **monolítica**, pois ela é representada através de um único BDD. O gargalo na utilização de relações de transição monolítica está no tamanho do BDD que representa a relação. O particionamento da relação de transição tem sido usado para minimizar o número de nós BDDs. Não faz parte do escopo deste trabalho o detalhamento das relações de transição particionadas, mais informações adicionais podem ser encontradas em [19, 12, 17, 25].

3.5 Reticulado e Ponto fixo

Um reticulado completo é um conjunto que possui uma ordem parcial em seus elementos, um maior elemento $\top$ e um menor elemento $\bot$.

Consideremos P um conjunto finito não vazio de proposições atômicas e $M = (S,R,I,L)$ um estrutura de Kripke finita sobre P. O conjunto formado pelos subconjuntos de S (2^S) e o operador de **está contido** ($\subseteq$) formam um reticulado completo. O conjunto vazio $\{\}$ e S são o menor ($\bot$) e maior ($\top$) elemento do reticulado. Como podemos representar cada subconjunto de S através de suas funções características, este reticulado pode também ser interpretado como o reticulado das funções características, com a implicação booleana ($\Rightarrow$) como ordenação, FALSO como menor elemento e a função característica de S como maior elemento.

A função $\tau : 2^S \rightarrow 2^S$ é chamada de **transformador de predicados**. τ é monotônica se e somente se $P \subseteq Q$ implica em $\tau(P) \subseteq \tau(Q)$. τ é também $\cup$-contínua quando $P_1 \subseteq P_2 \subseteq \ldots$ implica que $\tau(\cup_i P_i) = \cup_i \tau(P_i)$. τ é também $\cap$-contínua quando $P_1 \supseteq P_2 \supseteq \ldots$ implica que $\tau(\cap_i P_i) = \cap_i \tau(P_i)$. *Tarski* [32] mostrou que se τ é monotônica então τ tem um menor e um maior ponto fixo, denotado por **lfp**$Z[\tau(Z)]$ e **gfp**$Z[\tau(Z)]$, respectivamente.

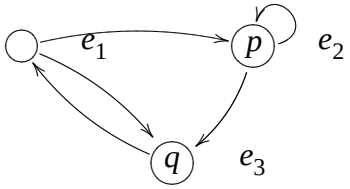


Figura 3.4: Estrutura de Kripke

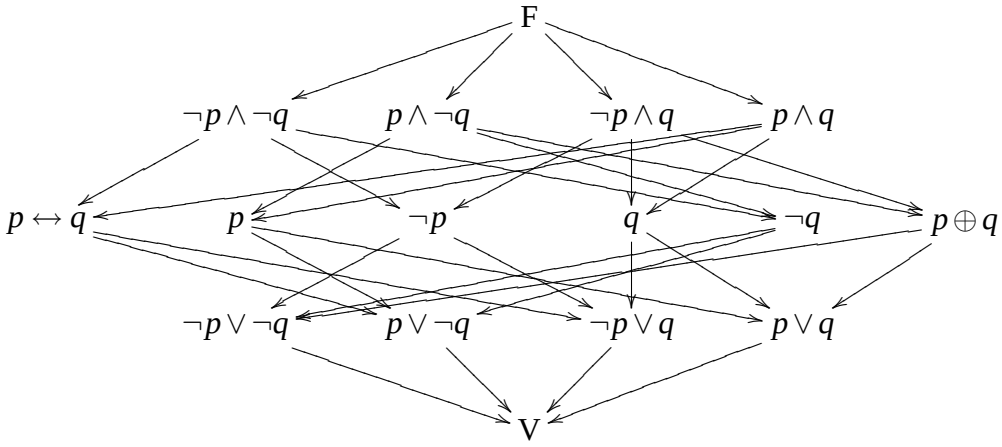


Figura 3.5: Reticulado das fórmulas proposicionais

Um ponto fixo de uma função τ é encontrado quando $\tau(Z) = Z$. O menor ponto fixo de uma função τ é encontrado quando esta é aplicada a partir do menor elemento $\perp$ do reticulado sucessivamente até encontrar o ponto fixo. O maior ponto fixo, por analogia, é encontrado quando τ é aplicada sucessivamente a partir do maior elemento $\top$ até encontrar o ponto fixo.

A figura 3.4 descreve uma estrutura de Kripke que possui três estados, duas proposições p e q são usadas para representar estes estados. Esta máquina possui 4 estados possíveis, mas apenas três deles são alcançáveis.

Considerando a representação da estrutura de Kripke usando as funções características, a figura 3.5 mostra um reticulado das expressões booleanas sobre $\{p,q\}$ e do operador $\rightarrow$ (implica).

A figura 3.6 mostra o reticulado formado a partir dos subconjuntos dos estados reais da máquina. Observamos que este reticulado está inserido no anterior (figura 3.5). O reticulado é construído a partir dos subconjuntos de S e do operador $\subseteq$ (está contido).

A importância do conhecimento do reticulado completo construído através das funções características é que este será usado na verificação simbólica de modelos através de operadores CTL– **EG** e **E**[**U**] – e na busca dos estados alcançáveis (ver

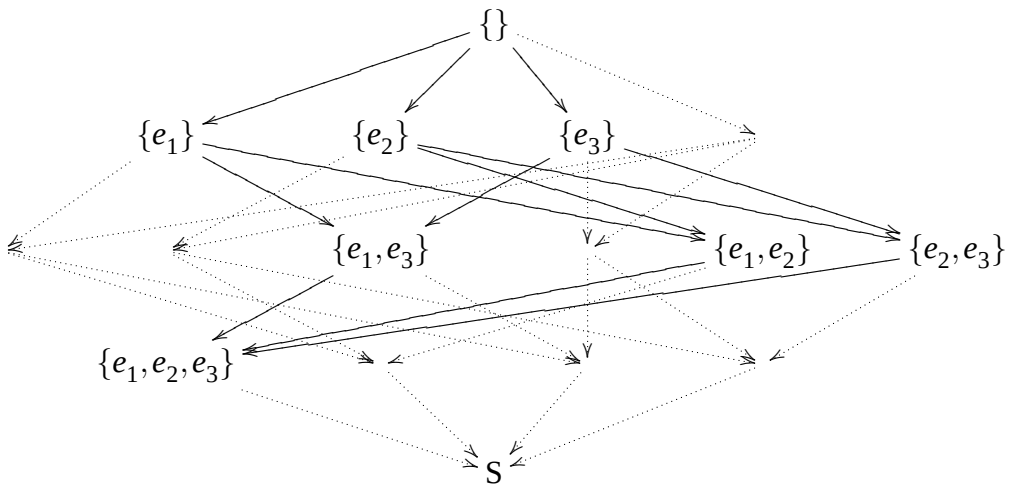


Figura 3.6: Reticulado do conjunto de estados

seções 3.6 e 3.8).

3.6 Alcançabilidade

A técnica de verificação automática de sistemas consiste em fazer uma busca sobre todos os estados do modelo do sistema e determinar se o modelo satisfaz propriedades especificadas para o mesmo. Isto é, dada uma estrutura de Kripke $M = (S, R, L)$, que é um modelo de um sistema de transição finita, e uma fórmula de lógica temporal f, verificar se M satisfaz f significa determinar o conjunto de estados em que f é verdadeiro.

Portanto, é importante encontrar o conjunto de estados alcançáveis do sistema para que os algoritmos possam manipulá-los. Entende-se como estados alcançáveis o conjunto de estados que podem ser atingidos a partir dos estados iniciais. Dessa forma, partindo-se do conjunto de estados iniciais S_0 e utilizando a relação de transição R, temos que computar o conjunto de estados alcançáveis S.

Primeiramente, consideramos o problema de encontrar o conjunto de estados alcançáveis a partir de S_0 em até uma transição. Este conjunto é dado por:

$$S_1 = S_0 \cup \{s' | \exists s[s \in S_0 \wedge (s, s') \in R]\}.$$

Dados os BDDs de $S_0(X)$ e $R(X, X')$ podemos computar o BDD representando S_1

realizando a operação lógica correspondente a expressão acima:

$$S_1(X') = S_0(X') \vee \exists_{x \in X}[S_0(X) \wedge R(X,X')].$$

Similarmente, o conjunto de estados alcançáveis em até dois passos a partir de S_0 é representado por:

$$S_2(X') = S_0(X') \vee \exists_{x \in V}[S_1(X) \wedge R(X,X')].$$

De forma geral, o conjunto dos estados alcançáveis em pelo meno $k+1$ passos pode ser definido como:

$$S_{k+1}(X') = S_0(X') \vee \exists_{x \in X}[S_k(X) \wedge R(X,X')].$$

Podemos observar que cada conjunto de estados é um superconjunto do anterior. Como o número de estados é finito, em algum ponto teremos que $S_{k+1} = S_k$, ou seja, nenhum novo estado é alcançável, então o conjunto de estados alcançáveis é representado por $S(V) = S_k(V)$.

A computação acima pode ser definida como um **menor ponto fixo** [6] do seguinte transformador de predicados:

$$S_0(x') \vee \exists x(K(x) \wedge R(x,x'))$$

3.7 CTL

CTL [9] é uma lógica temporal de tempo ramificado. Uma lógica de tempo ramificado assume que um estado de um sistema, num determinado instante de tempo pode ter mais de um sucessor, possuindo uma estrutura de tipo árvore, que é infinita no tempo. Dado um certo caminho na árvore, sua linha de tempo é isomórfica com o conjunto dos naturais: o tempo é discreto. CTL é uma lógica sobre estruturas de Kripke e é interpretada sobre suas árvores de computação.

Através de CTL é possível relacionar seqüências de eventos no tempo. Assim, é possível expressar que depois de um determinado evento e_1, o evento e_2 ocorrerá.

3.7.1 Sintaxe

O conjunto $T_{\mathsf{CTL}}(P)$ de formulas CTL sobre um conjunto de proposiçoes P é o menor conjunto tal que $P \subseteq T_{\mathsf{CTL}}(P)$ e se f e g estão em $T_{\mathsf{CTL}}(P)$, então $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{AX}f$, $\mathbf{EG}f$, $\mathbf{AG}f$, $\mathbf{EF}f$, $\mathbf{AF}f$, $\mathbf{E}[f\mathbf{U}g]$, $\mathbf{A}[f\mathbf{U}g]$ estão em $T_{\mathsf{CTL}}(P)$.

Cada operador lógico temporal é composto por:

- um quantificador de caminho: **E**, para algum caminho, ou **A**, para todos os caminhos;
- seguido por um quantificador de estado: **X**, próximo estado no caminho, **U**, até, **G**, globalmente, ou **F**, futuramente.

3.7.2 Semântica

A semântica de CTL é definida através da estrutura de Kripke $M = (S,T,I,L)$ sobre um conjunto de proposições atômicas P. Se f está em $T_{\mathsf{CTL}}(P)$, $M,s \models f$ significa que f é verdadeiro no estado s de M.

Estejam f e g em $T_{\mathsf{CTL}}(P)$, então

1. $M,s \models p$ sss $p \in L(s)$.
2. $M,s \models \neg f$ sss $M,s \not\models f$.
3. $M,s \models f \wedge g$ sss $M,s \models f$ e $M,s \models g$.
4. $M,s \models \mathbf{EX}f$ sss existe um estado s' de M tal que $(s,s') \in R$ e $s' \models f$. I.e., s tem um successor onde f é válido.
5. $M,s \models \mathbf{EG}f$ sss existe um caminho π de M tal que $\pi(1) = s$ e $\forall i \geq 1 \bullet M,\pi(i) \models f$. I.e., s está no começo do caminho onde f é verdadeiro globalmente.
6. $M,s \models \mathbf{E}[f\mathbf{U}g]$ sss existe um caminho π de M tal que $\pi(1) = s$ e $\exists i \geq 1 \bullet (M,\pi(i) \models g \wedge \forall j, i > j \geq 1 \bullet M,\pi(j) \models f)$. I.e., s está no começo do caminho onde g eventualmente se tornará verdadeiro e f é verdadeiro até g se tornar válido (nada pode ser dito a respeito de f quando g se torna válido).

Os outros operadores lógico-temporais podem ser definidos em função de **EX**, **EG** e **E**[**U**].

$$
\begin{aligned}
\mathbf{AX}f &= \neg\mathbf{EX}\neg f \\
\mathbf{AG}f &= \neg\mathbf{EF}\neg f \\
\mathbf{AF}f &= \neg\mathbf{EG}\neg f \\
\mathbf{EF}f &= \mathbf{E}[\mathbf{true}\mathbf{U}f] \\
\mathbf{A}[f\mathbf{U}g] &= \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g
\end{aligned}
$$

Para a verificação simbólica de modelos, as propriedades a serem verificadas são especificadas usando CTL.

3.8 Algoritmos de verificação

A verificação simbólica utiliza a representação das funções características de conjuntos de estados e transições através de BDDs. O algoritmo de verificação se baseia na caracterização de ponto fixo de diferentes operadores CTL [9].

O algoritmo de verificação – algoritmo 3 –, chamado de *smc*, é um predicado que recebe dois argumentos, uma estrutura de Kripke M e uma fórmula CTL f. O algoritmo verifica se a fórmula f é verdadeira nos estados iniciais de M. Para isso, uma função *smcAux* retorna o conjunto de estados de M que satisfazem f.

Algoritmo 3 Verificação de modelos simbólicos com fórmulas CTL

```
1  funct smc(M : Kripke structure, f : CTL) : boolean ≡
2     return bddImplies(M.I, smcAux(M, f))
3  .
5  funct smcAux(M : Kripke structure, f : CTL) : BDD ≡
6     if (f é uma proposição booleana) then return bddAtom(f)
7     elsif (f = ¬f1) then return bddNot(smcAux(M, f1))
8     elsif (f = EXf1) then return smcEX(M, f1)
9     elsif (f = EGf1) then return smcEG(M, f1)
10    elsif (f = E[f1 U f2]) then return smcEU(M, f1, f2)
11    fi
12 .
```

A função $smcEX(M, f)$, algoritmo 4, computa os estados de M onde $\mathbf{EX}f$ é válida. Primeiramente, é computado F, que é o BDD para a função característica do conjunto de estados de M onde f é verdadeira, e retorna a imagem inversa de F, ou

seja, todos os estados que tem pelo menos um sucessor onde f é verdadeira.

Algoritmo 4 Cálculo de $\mathbf{EX}f$

```
1  funct smcEX(M : Kripke structure, f : CTL) : BDD ≡
2     F : BDD
3     F ← smcAux(M, f)
4     return backward(M,F)
5  .
```

A fórmula $\mathbf{E}[f\mathbf{U}g]$ consiste em calcular o menor ponto fixo do reticulado formado pelos subconjuntos de todos os estados de M [6].

$$\mathbf{E}[f\mathbf{U}g] = \mathbf{lfp}Z[g \vee (f \wedge \mathbf{EX}Z)]$$

O algoritmo 5 mostra como isto é feito. Primeiramente ele computa F e G, que são os estados onde f e g são válidos, respectivamente, e depois calcula o ponto fixo.

Algoritmo 5 Cálculo de $\mathbf{E}[f\mathbf{U}g]$

```
1  funct smcEU(M : Kripke structure, f : CTL, g : CTL) : BDD ≡
2     Q, Q', F, G : BDD
3     F ← smcAux(M, f)
4     G ← smcAux(M, g)
5     Q ← bddFalse
6     Q' ← bddOr(G, bddAnd(F, backward(M, Q)))
7     while Q ≠ Q' do
8           Q ← Q'
9           Q' ← bddOr(G, bddAnd(F, backward(M, Q)))
10    od
11    return Q
12 .
```

Similarmente, a fórmula $\mathbf{EG}f$ consiste em calcular o maior ponto fixo de reticulado formado pelos subconjuntos dos estados de M [6].

$$\mathbf{EG}f = \mathbf{gfp}Z[f \wedge \mathbf{EX}Z]$$

Da mesma forma que o anterior, o algoritmo 6 computa os estados de M onde f é válida e então computa o maior ponto fixo.

Os outros operadores CTL podem ser definidos a partir de **EX** $\mathbf{E}[\mathbf{U}]$ e **EG**, portanto estes algoritmos definem a verificação de toda e qualquer fórmula CTL.

Algoritmo 6 Cálculo de **EG** f

```
1  funct smcEG(M : Kripke structure, f : CTL) : BDD ≡
2     Q, Q', F : BDD
3     F ← smcAux(M, f)
4     Q ← bddTrue
5     Q' ← bddAnd(F, backward(M, Q))
6     while Q ≠ Q' do
7           Q ← Q'
8           Q' ← bddAnd(F, backward(M, Q))
9     od
10    return Q
11 .
```

4 Heurística dos pesos

A utilização de BDDs para verificação simbólica de modelos (*CTL model checking*) tem tido bastante sucesso para verificar circuitos digitais, sendo adotada nos centros de projeto das principais empresas do ramo (por exemplo: INTEL, IBM, MOTOROLA). Para isso, várias técnicas de ordenação estática e reordenação de BDDs têm sido desenvolvidas. Pretendemos aqui mostrar uma nova abordagem para a ordenação inicial de variáveis de BDDs que representam sistema de transição finita e que pode, portanto, se aplicar a verificação de circuitos. Esta abordagem consiste em analisar a estrutura do sistema de transição finita, atribuindo pesos para as variáveis e colocando as variáveis com maior peso no topo do grafo.

As técnicas de verificação de modelos têm se mostradas bastante "eficientes" na verificação de circuitos sequenciais. Os métodos de ordenação inicial atuais obtêm bons resultados para a ordenação de BDDs que representam circuitos combinatórios, porém para BDDs que representam circuitos sequenciais estes métodos não conseguem ser eficientes para todos os tipos de circuito, tornando, assim, relevante a busca por novas heurísticas. Os métodos de reordenação têm se mostrado bastante eficientes para qualquer tipo de aplicação, pois os métodos foram desenvolvidos para que a reordenação seja transparente ao usuário.

Para que uma ordem seja considerada boa, depende do estado do gerenciador de BDDs. Um estado do gerenciador de BDDs é o conjunto de variáveis existentes e as funções booleanas sendo representadas naquele momento, bem como uma certa quantidade de informações já processadas (nas caches). Na verificação formal, o estado do gerenciador de BDDs muda constantemente, desde a criação do ambiente para a verificação, como a construção da relação de transição ou a computação da alcançabilidade, como na verificação em si, na busca dos estados que satisfazem a propriedade especificada.

Pesquisas anteriores têm mostrado que os métodos de reordenação são eficientes, porém eles se tornam mais rápidos e encontram melhores ordens quando a ordenação de variáveis no momento de sua chamada é já uma ordenação boa. Uma técnica de

reordenação, o *sifting*, é bastante eficiente, evitando uma explosão no número de nós, quando a ordem inicial já é uma boa ordem. Porém, quando a ordem inicial não é uma ordem boa, ou próxima de uma ordem boa, os métodos de reordenação não conseguem reverter para uma boa ordem. Podemos dizer que um método de reordenação funciona melhor quando a ordem das variáveis já é uma boa ordem, ou uma ordem próxima de uma ordem boa. A partir destas observações, podemos concluir que para se obter uma boa ordem de variáveis é necessário ter uma boa ordem inicial, fazendo com que a ordenação estática se torne de fundamental importância para a verificação simbólica de modelos.

O objetivo deste trabalho é apresentar um algoritmo de ordenação estática eficiente para BDDs que representam máquinas de estados finitos, como circuitos sequenciais. A questão em torno da ordenação de variáveis começa com a identificação de uma boa ordem inicial das variáveis. Baseado nestas idéias e estudando a modelagem de sistemas de transição finita através de BDDs, desenvolvemos um algoritmo de ordenação estática que permite identificar a dependência das funções de transição com relação as variáveis. A dependência de uma função é o conjunto de variáveis das quais esta função depende, ou argumentos da função. Entendemos intuitivamente que variáveis que estão no conjunto de dependência de muitas funções são as que mais definem o comportamento do sistema e, colocando essas variáveis no início da ordenação, podemos otimizar o espaço alocado e acelerar os algoritmos que percorrem o grafo. Experiências práticas são necessárias para confirmar, ou não, nossa intuição.

Neste capítulo, na seção 4.1, apresentaremos um algoritmo para a ordenação inicial de BDDs que representam sistemas de transição finita. A seguir, na seção 4.2 descreveremos uma implementação deste algoritmo diretamente sobre BDDs e na ferramenta VIS. Na seção 4.3 apresentaremos os primeiros resultados obtidos com a aplicação da heurística para a ordenação de circuitos sequenciais. Por fim, na seção 4.4, apresentaremos os resultados obtidos com a implementação da heurística em uma ferramenta de verificação já desenvolvida e bastante usada, o VIS.

4.1 Algoritmo

A idéia principal do algoritmo é a de atribuir pesos para as variáveis BDDs, ficando com maior peso as variáveis que devem vir no início (topo) da ordenação. Para isso deve-se avaliar as funções de transição de cada variável de estado e identificar de que outras variáveis ela depende, sejam outras variáveis de estado ou variáveis que rep-

resentam entradas externas que modificam o estado do sistema – no caso de circuitos sequenciais, elas são as entradas do circuito.

A heurística, mostrada no algoritmo 7, consiste em, para cada variável $x \in X$, determinar de quais variáveis sua função de transição depende, ou seja, quais variáveis determinam o próximo valor de x. Sendo assim, consideremos S_i o conjunto de todas as variáveis que são argumentos da função de transição de x_i, chamado de **suporte** de x_i. Quanto mais vezes uma variável aparece suporte de outra, mais ela influenciará o comportamento do sistema. Consequentemente ela deverá se localizar no início (topo) da ordenação. O número de funções que dependem de uma variável determinam o **peso** dessa variável.

O algoritmo 7 calcula os pesos das variáveis: primeiramente cada variável recebe o peso 0, então o suporte S_i para cada função de transição f_i é calculado e os pesos de cada variável x_j em S_i é incrementado.

Algoritmo 7 Ordenação inicial de variáveis em uma estrutura de Kripke

```
1  begin
2    foreach w_i ∈ W do
3      peso(w_i) ← 0
4    od
5    foreach x_i ∈ X do
6      S_i ← suporte(f_i)
7      foreach w_j ∈ S_i do
8        peso(w_j) ← peso(w_j) + 1
9      od
10   od
11   ordena(W, peso)
12 end
```

Após a atribuição dos pesos das variáveis, é determinada uma ordenação inicial para os BDDs que representam estas variáveis. A observação que deve ser feita é que não existe dependência nas funções das variáveis BDDs que representam o próximo estado – variáveis x' – e portanto estas ficariam no fim da ordem. Porém, como os algoritmos de manipulação de BDDs para verificação de modelos estão sempre fazendo o produto relacional das variáveis de estado x com as de próximo estado x', estas devem ficar sempre próximas umas das outras. Para que isto seja obedecido, o algoritmo mantém cada variável x'_i logo após x_i na ordenação. Após a ordem ser definida, agrupa-se as variáveis x_i e x'_i.

Exemplificaremos o funcionamento do algoritmo aplicando-o ao circuito da figura 3.2.

Temos que $X = \{x_1, x_2, x_3\}$, $E = \{p\}$ e $X' = \{x'_1, x'_2, x'_3\}$. Dessa forma temos sete variáveis BDDs para representar este circuito e as seguintes relações de transição:

$$\begin{aligned} f_1(W) &= (x_3 \wedge x_2) \oplus x_1 \\ f_2(W) &= x_3 \oplus x_2 \\ f_3(W) &= \neg x_3 \oplus p \end{aligned}$$

De forma visual, podemos identificar os pesos das variáveis como sendo:

x_1	1
x_2	2
x_3	3
p	1

Observamos que as variáveis de próximo estado não estão nos cálculos dos pesos. Isto acontece porque não existe dependência de funções das variáveis de próximo estado. Conforme explicado anteriormente, as variáveis de próximo estado ficam ordenadas logo após sua correspondente de estado atual. A ordenação inicial das variáveis BDDs do circuito fica:

$$\{x_3, x'_3, x_2, x'_2, x_1, x'_1, p\}$$

4.2 Implementação

Primeiramente, este algoritmo foi implementado em uma ferramenta própria, que lê arquivos VerilogHDL e constrói os BDDs que representam o modelo do circuito descrito. A implementação do algoritmo foi feita diretamente sobre os BDDs, assim que as funções de transição são construídas, uma vez que o algoritmo precisa da informação de dependência de variáveis. Esta ferramenta foi implementada usando o pacote de BDDs de *David Long* [23]. Este pacote possui uma função, chamada `support(f)`, que retorna uma lista contendo as variáveis BDDs das quais `f` depende. Assim sendo, é fácil saber qual a lista de dependência de uma função e, de posse desta lista, incrementar o peso de cada variável na lista. O algoritmo 8 mostra os passos para computar os pesos das variáveis.

Após os primeiros resultados, este algoritmo foi de novo implementado, desta vez na ferramenta de verificação VIS [3]. VIS é uma ferramenta de verificação que lê arquivos em VerilogHDL [33], BLIF [15] ou BLIF-MV [22] desenvolvido na Univer-

Algoritmo 8 Método dos pesos implementado diretamente sobre os BDDs

```
1 proc computaPesos ≡
2   foreach (f_i)
3     dep ← support(f_i)
4     foreach (v ∈ dep)
5       peso(v) ← peso(v) + 1
6     end
7   end
8   ordena(W);
9 .
```

sidade da Califórnia, Berkeley, e é uma ferramenta de referência na área de verificação automática de circuitos digitais. VIS representa internamente os circuitos como redes.

Para a implementação do algoritmo dos pesos, foi acrescentado um novo campo no nó da rede que será usado para armazenar o peso do nó. Abaixo segue o campo acrescentado à estrutura.

```
struct NtkNodeStruct {
  char *name;  /* Nome do nó */
   ⋮
  long peso;   /* Usado pela heurística dos pesos*/
   ⋮
};
```

O algoritmo 9 mostra como é feito o cálculo dos pesos de cada nó.

O cálculo dos pesos dos nós é feito a partir das dependências dos registradores (*latches*) e das saídas. Após o cálculo dos pesos, uma lista com os nós cujos valores serão representados através de variáveis BDDs será criada, composta, apenas, por nós que representam entradas e registradores. Cada variável BDD que mapeia registradores possui uma outra variável BDD que representa o valor desta variável no próximo estado da máquina.

Após a ordenação dos nós, um procedimento de inserção de nós de próximo estado é chamado para que as variáveis que representam o próximo estado da MEF sejam inseridas na lista, cada variável logo após sua correspondente variável. O procedimento de criação efetiva de variáveis é chamado tendo como um dos argumentos uma lista contendo os nós que devem ser representados através de BDDs em ordem de posicionamento.

Algoritmo 9 Computação dos pesos dos nós da rede

```
1  proc ComputaPeso(rede) ≡
2     foreach (nó ∈ rede)
3        peso(nó) ← 0
4     end
5     foreach (nó ∈ rede)
6        if (tipo(nó) ∈ [registrador, saída])
7           suporte = dependencia(f_{nó})
8           foreach (nó_{dep} ∈ suporte)
9              peso(nó_{dep}) ← peso(nó_{dep}) + 1
10          end
11       fi
12    end
13 .
```

fifo2x4			fifo2x8			fifo2x16			fifo4x2			fifo4x4		
ent	est	tot	ent	est	tot	ent	est	tot	ent	est	tot	ent	est	tot
10	38	48	10	58	68	10	94	104	12	42	54	12	62	74

Tabela 4.1: Variáveis booleanas em cada exemplo (entradas, estados e total)

4.3 Experimentos preliminares

Os primeiros resultados foram obtidos a partir da aplicação do algoritmo em uma representação de circuitos diretamente através de BDDs. Foi desenvolvido e utilizado um analisador sintático Verilog que gera diretamente os BDDs.

Um arquivo Verilog contendo a descrição de uma FIFO de um árbitro de barramento para um sistema multiprocessado [11] foi usado como teste. Várias versões da FIFO foram utilizadas para os primeiros testes. As versões variam em tamanho da palavra e número de palavras.

A tabela 4.1 mosta o número de variáveis booleanas presentes nos exemplos usados. A nomenclatura "fifo*xxy*" permite identificar o número de bits e palavras de cada exemplo, onde *x* é o número de bits de cada palavra e *y* é o número de palavras da FIFO. Estes testes foram realizados em um microcomputador com processador Intel Pentium III de 600Mhz e 512Mb de Memória RAM.

Primeiramente, aplicamos esta heurística e vimos os resultados após a construção da relação de transição. A tabela 4.2 mostra o tempo e o número de nós BDDs de cada exemplo. Os testes foram feitos aplicando somente a heurística (linha P na tabela),

Ord.	fifo2x4		fifo2x8		fifo2x16		fifo4x2		fifo4x4	
	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam
P	0	5.342	0	26.230	0	137.092	0	11.040	0	52.451
S	1	3.366	7	13.535	96	59.482	9	2,834	51	24.640
P+S	0	**1.368**	5	**5.292**	51	**20.729**	2	**1.799**	18	**16.545**

Tabela 4.2: Tempo, em segundos, e tamanho, em nós BDDs, para ordenaçao inicial

Blocos	Ordem	fifo2x4	fifo2x8	fifo2x16	fifo4x2	fifo4x4
Nenhum	P	1.390	5.330	20.791	1.809	15.833
	S	2.311	10.664	55.529	1.858	17.008
	P+S	**1.189**	**4.446**	**19.060**	**1.101**	**7.338**
N. Ord.	P	1.083	4.112	23.532	2.366	11.862
	S	2.218	10.696	47.417	2.700	10.516
	P+S	**1.079**	**4.021**	**17.250**	**1.674**	**4.596**
Ord.	P	1.065	4.066	23.455	2.266	11.712
	S	2.186	10.581	47.174	2.636	10.377
	P+S	**1.061**	**3.986**	**17.178**	**1.628**	**4.561**

Tabela 4.3: Número de nós BDDs após cálculo da alcançabilidade

aplicando somente o *sifting* (linha S na tabela) e aplicando a heurística seguida de *sifting* (linha P+S na tabela). Podemos observar que a heurística é bastante rápida, em todos os casos ela terminou em menos de 1 segundo. O método de *sifting* já precisa de mais tempo, porém encontra uma melhor ordem do que a encontrada através da heurística. O fato mais interessante é a aplicação da heurística seguida de *sifting*, onde observamos que o tempo é melhor do que se aplicarmos apenas o *sifting* e foi encontrada uma ordem melhor do que os outros dois testes.

Estes testes também foram aplicados sobre o cálculo da alcançabilidade do sistema. Fizemos as comparações sobre a heurística, o *sifting* e a combinação dos dois. Além disso, fizemos os testes com diferentes agrupamentos de variáveis: sem agrupamento, agrupando as variáveis de estado e próximo estado. Estes grupos de variáveis podem ser ou não reordenáveis. Para todos os casos foi efetivada coleta de lixo (*garbage collection*) e um *sifting* após a conclusão da alcançabilidade.

A tabela 4.3 mostra o número de nós BDDs após o cálculo de alcançabilidade do sistema. Percebemos, mais uma vez, que a combinação da heurística com o *sifting* obtem resultados melhores que aplicando apenas um dos dois. Podemos observar também que o agrupamento de variáveis melhora o resultado após a alcançabilidade,

Blocos	Ordem	fifo2x4	fifo2x8	fifo2x16	fifo4x2	fifo4x4
Nenhum	P	**1**	20	297	4	3.193
	S	3	33	526	15	871
	P+S	**1**	**15**	**209**	**3**	**665**
N. Ord.	P	**1**	19	**289**	4	3.134
	S	3	109	31.682	21	557
	P+S	**1**	**17**	315	**3**	**504**
Ord.	P	**1**	19	**290**	4	3.132
	S	3	108	31.017	21	550
	P+S	**1**	**16**	316	**3**	**465**

Tabela 4.4: Tempo total, em segundos, para alcançabilidade

porém o fato de permitir, ou não, a ordenação interna desses grupos de variáveis não influi nos resultados de maneira significativa. A tabela 4.4 mostra o tempo de cada exemplo. Mais uma vez a combinação heurística e *sifting* é mais eficiente. Houve apenas dois casos em que essa combinação não obteve o melhor tempo (fifo2x16 com blocos), quando o melhor tempo foi o da heurística apenas.

4.4 Experimentos completos

Após os primeiros resultados, percebemos que a heurística tem um bom potencial de melhorar o tempo e a utilização de memória de sistema de verificação automática. O algoritmo foi então implementado no VIS, como descrito na seção 4.2, através de sua adapatação para uma rede de nós que representa um circuito, permitindo que ele seja aplicado antes da criação dos BDDs, tornando-o, assim, efetivamente um método de ordenação estática.

4.4.1 Descrição do experimento

Duas versões do algoritmo foram implementadas no VIS. Na primeira, chamada de peso1, a lista de dependências de um nó é encontrada a partir dos *fanins* do nó e a computação pára quando encontra entradas do circuito ou registradores. Na segunda, chamada peso2, a lista de dependências só pára quando encontra entradas do circuito ou constantes, já que estas não possuem *fanins*.

Os testes foram feitos usando o conjunto de exemplos que acompanham a ferramenta. Como método de ordenação estática, os primeiros testes foram realizados

comparando os resultados com os métodos de ordenação já implementados no VIS: busca em profundidade (*append* – seção 2.2.1), *interleave* (seção 2.2.2), *merge_left* (seção 2.2.2) e *merge_right* (seção 2.2.2).

Os testes consideraram o início do processo quando as redes que representam os circuitos já estão em memória. A partir desse ponto, o primeiro passo foi o de encontrar uma ordem inicial e criar os BDDs que representam esta rede. O segundo passo foi o de computar os BDDs que representam as funções de transição existentes na MEF. O terceiro passo foi o de calcular a alcançabilidade da máquina. Por fim, como último passo, foi aplicada uma chamada a um método de ordenação dinâmica, o método de *sifting* que está implementado no VIS. Em todas as etapas os dados colhidos foram o tempo para a realização da etapa e o número de nós BDDs após a conclusão da mesma.

4.4.2 Casos de teste

Os testes foram realizados sobre descrições que acompanham a ferramenta VIS. Parte dos exemplos não são apresentados, por serem de pequeno porte, e não necessitar de recursos computacionais significativos. Abaixo segue uma breve descrição dos exemplos efetivamente usados para avaliar a heurística.

ABP Este circuito implementa o protocolo `ABP` (*Alternate Bit Protocol*) de forma assíncrona. Ele utiliza 3 módulos, um *sender*, um *receiver* e um árbitro. O árbitro é usado para garantir a assincronissidade do protocolo, o *sender* envia dados para o *receiver*.

arbiter Este circuito é um árbitro de exclusão mútua usado para controlar três controladores, chamados clientes. É uma máquina de estados composta apenas por três estados e possui uma única saída que informa qual dos 3 controladores está selecionado.

bakery Este circuito é composto por um módulo `system`, que representa um sistema operacional, e este sistema possui dois processos, módulos `process`. Estes processos fazem leitura e escrita na memória, mas isso só pode ser feito através do sistema operacional. O arquivo fonte `bakery.v` possui 97 linhas de código VerilogHDL.

coherence Um circuito que faz coerência de cache. Ele possui um módulo principal `COHERENCE` e três outros módulos: `PROC`, que descreve os processos sendo ex-

ecutados, `CACHE_CTRLER`, que descreve o controle de cache de cada processo e `DIRECTORY`, que descreve o acesso à memória através da cache.

dcnew Nenhuma documentação foi encontrada sobre este exemplo. O arquivo possui a descrição de 10 módulos e tem 350 linhas.

eisenberg Este exemplo posssui a descrição de dois módulos, `system`, que representa um sistema operacional com memória compartilhada, e o segundo, `process`, descreve processos que acessam a memória compartilhada. Este exemplo foi testado com 2 processos.

elevator Este exemplo possui um controlador principal de um sistema com dois elevadores em um prédio de 3 andares. Existe um módulo, `elevator`, que descreve o elevador e um módulo que é o controlador principal.

ethernet O exemplo utilizado é uma instanciação do protocolo *ethernet* em um sistema com dois processadores.

exampleS Não foi encontrada documentação sobre estas descrições. `exemploS.v` é um arquivo que possui 170 linhas que descrevem 7 módulos.

fpmpy Este exemplo possui um circuito multiplicador de ponto flutuante, inspirado no IEEE 754.

gcd Circuito que calcula o máximo divisor comum de dois números de 8 bits sem sinal.

prod_cell Este exemplo modela o controle de uma planta com um guindaste, duas esteiras, um braço articulado e uma prensa. O sistema é fechado, não há entradas.

rcnum Este exemplo contém um circuito para a computação dos chamados números 3n+1. Este modelo usa números de 26 bits.

scheduler Este exemplo implementa uma versão do escalonador usado como exemplo no livro de Milner [27] – "Communication and Concurrency", p. 115. A estrutura básica do sistema é a de *token ring*. Cada elemento do anel (*ring*), chamado célula, se comunica com seus *jobs* e suas duas células vizinhas.

slider Este exemplo ilustra a redução do jogo *slider* para alcançabilidade em MEF.

s298 Não há documentação sobre este exemplo. Existe apenas um arquivo do tipo BLIF-MV com 455 linhas de código.

Nome	Variáveis BDDs	Tempo para ordenação inicial					
		peso1	peso2	append	interleave	m_left	m_right
ABP	55	0,1	0,1	0,1	0,1	0,1	0,1
arbiter	49	0,1	0,1	0,1	0,1	0,1	0,1
bakery	45	0,1	0,1	0,1	0,1	0,1	0,1
coherence	82	0,1	0,1	0,1	0,1	0,1	0,1
dcnew	106	0,1	0,1	0,1	0,1	0,1	0,1
eisenberg	39	0,1	0,1	0,1	0,1	0,1	0,1
elevator	84	0,1	0,1	0,1	0,1	0,2	0,2
ethernet	178	0,1	0,2	0,1	0,1	0,2	0,2
exampleS	25	0,1	0,1	0,1	0,1	0,1	0,1
fpmpy	153	0,1	0,1	0,1	0,1	0,1	0,2
gcd	107	0,1	0,1	0,1	0,1	0,2	0,2
production_cell	137	0,3	0,1	0,2	0,3	0,4	0,4
rcnum	75	0,3	0,2	0,2	0,1	0,3	0,2
scheduler	98	0,1	0,1	0,1	0,1	0,1	0,1
slider	74	0,1	0,1	0,1	0,1	0,1	0,1
s298	31	0,1	0,1	0,2	0,2	0,1	0,2
treearbiter	58	0,1	0,1	0,1	0,2	0,3	0,2

Tabela 4.5: Número de variáveis BDDs necessárias para representar os exemplos e o tempo que cada método leva para calcular a ordem inicial

treearbiter Este exemplo descreve um circuito de exclusão mútua em forma de árvore. No exemplo, 4 processadores tentam acessar o mesmo recurso, o acesso é controlado pelo árbitro.

4.4.3 Resultados

A tabela 4.5 mostra os circuitos utilizados e o número de variáveis BDDs que é necessário para representar cada circuito e o tempo que cada heurística leva para ordenar os nós.

Podemos observar que o tempo para calcular a ordem inicial é bastante pequeno para todos os métodos de ordenação. Mesmo em circuitos com um certo número de variáveis, o tempo não chega a meio segundo. Isto mostra que tanto este novo algoritmo como os já existentes são muito eficientes em termos de tempo de computação.

A tabela 4.6 mostra o tempo para calcular a relação de transição do circuito e o número de nós BDDs após o cálculo da mesma. Os melhores resultados para cada exemplo estão em negrito na tabela. O algoritmo que obteve o maior número de melhores resultados foi o de *merge_left*, com 8 circuitos. O algoritmo de *append* obteve

4 melhores resultados. Os algoritmos de Peso2 e *merge_right* obtiveram 2 melhores resultados, o de *interleaving* 1, e o algoritmo de Peso1 não obteve nenhum melhor resultado. Esta grande performance do algoritmo *merge_left* pode ser explicada pela localização das entradas em relação aos registradores, já que pelo algoritmo, as entradas ficam mais no topo do circuito, e as funções de transição são construídas a partir das entradas e das interdependências dos registradores. Entre os dois algoritmo de pesos, vemos que o Peso2 é melhor que o Peso1, pois além do Peso2 ter conseguido dois melhores resultados, em 10 casos, ele foi melhor do que o Peso1, contra 6 em que o Peso1 foi melhor e 1 empate.

A tabela 4.7 mostra o tempo necessário para se computar os estados alcançáveis do sistema e o número de nós BDDs após esta computação. Observamos neste estágio que os algoritmos começam a ficar equivalentes, onde 3 métodos conseguem 4 melhores resultados: Peso2, *append* e *merge_right*. Os métodos de Peso1 e *merge_left* conseguem 2 melhores resultados e o método de *interleave* obtém 1 melhor resultado. Dois exemplos chamam bastante atenção nesta etapa, o `fpmpy` e `scheduler`, onde, no primeiro, os dois algoritmos de peso conseguem ser muito melhores do que os outros e, no segundo, os mesmos algoritmos são muito piores do que os outros. Novamente, entre os dois métodos de peso, o Peso2 foi melhor do que o Peso1, 11 contra 6. Como na etapa anterior, dois exemplos chamam a atenção, o `slider`, onde o Peso1 foi muito melhor e o `gcd`, onde o Peso2 foi muito melhor.

A tabela 4.8 mostra o tempo necessário para se fazer reordenação usando o método de *sifting* após a computação de alcançabilidade e o número de nós BDDs após a reordenação. O método de *sifting* é um excelente algoritmo de reordenação, porém ainda deve ser lembrada a idéia de que para se encontrar uma boa ordem é necessário ter uma boa ordem. O método do Peso2 foi que obteve os melhores resultados, conseguido a melhor ordem em 6 exemplos, contra 4 do *interleave*, 3 dos *append* e *merge_left*, 2 do *merge_right* e 0 do Peso1. É possível obervar que, mesmo nos exemplos onde o método Peso2 não foi o melhor, a ordem encontrada não é muito pior do que os outros. E, mais uma vez, os tempos de reordenar são equivalentes.

A tabela 4.9 mostra o número máximo de nós BDDs alocados durante os processos de computação da relação de transição, computação da alcançabilidade e reordenação usando *sifting*, bem como o tempo necessário para realizar as três tarefas. Esta tabela mostra o resumo do processo da leitura da descrição até que o sistema esteja preparado para fazer a verificação. Aqui observamos que os métodos Peso2, *append* e *interleave* possuem performance semelhante, onde o método *append* obteve 6 melhores resultados e os outros dois métodos, 5 melhores resultados. Os métodos

Nome	Peso1		Peso2		append		inter		left		right	
	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam
ABP	0,0	2.743	0,1	6.258	0,0	**2.465**	0,0	4.264	0,0	4.488	0,0	3.544
arbiter	0,0	633	0,0	593	0,0	**525**	0,0	692	0,0	545	0,0	619
bakery	0,0	1.238	0,0	1.176	0,0	**1.170**	0,0	2.207	0,0	1.275	0,0	1.863
coherence	0,1	5.995	0,1	10.363	0,1	**4.886**	0,1	7.519	0,1	7.009	0,1	7.839
dcnew	0,0	4.571	0,1	5.049	0,0	3.549	0,0	3.794	0,0	**2.973**	0,0	3.241
eisenberg	0,0	2.382	0,1	1.914	0,0	1.928	0,1	2.816	0,0	**1.292**	0,0	2.548
elevator	0,1	5.860	0,1	**5.672**	0,1	6.129	0,2	7.000	0,1	7.128	0,1	6.854
ethernet	0,1	10.906	0,1	13.796	0,1	13.736	0,1	11.954	0,1	**7.514**	0,1	12.180
exampleS	0,0	447	0,0	432	0,1	441	0,0	432	0,0	**359**	0,0	386
fpmpy	0,1	3.816	0,1	3.863	0,1	7.238	0,1	7.050	0,1	**3.030**	0,1	6.746
gcd	7,0	321.034	2,6	**113.786**	4,0	223.819	3,9	223.819	6,7	330.400	4,1	223.819
prod_cell	0,2	8.134	0,1	6.974	0,3	8.932	0,3	8.891	0,2	**4.876**	0,3	9.932
rcnum	0,1	9.924	0,1	9.924	0,1	8.392	0,1	8.392	0,1	**2.244**	0,1	8.392
scheduler	0,1	2.605	0,1	2.574	0,1	2.277	0,1	2.581	0,1	2.507	0,1	**1.907**
slider	0,0	6.147	0,1	10.240	0,0	5.789	0,0	6.069	0,1	6.877	0,1	**5.650**
s298	0,0	325	0,0	295	0,0	291	0,0	**216**	0,0	276	0,0	242
treearbiter	0,1	2.419	0,0	1.100	0,1	1.687	0,1	1,441	0,1	**1.037**	0,1	1.744

Tabela 4.6: Tempo, em segundos, para a construção dos relações de transição e número de nós BDDs após o término do procedimento

Nome	Peso1		Peso2		append		inter		left		right	
	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam
ABP	0,1	**4.925**	0,2	6.977	0,1	13.026	0,2	9.512	0,1	19.868	0,1	11.034
arbiter	0,1	8.727	0,1	13.336	0,0	8.170	0,1	11.048	0,0	9.935	0,1	**7.432**
bakery	0,2	12.844	0,2	12.682	0,2	8.860	0,3	7.869	0,2	**7.824**	0,2	14.964
coherence	26,5	40.072	42,4	57.463	7,5	**16.234**	14,8	46.618	21,8	38.293	12.9	28.256
dcnew	0,2	11.775	0,3	5.783	0,1	20.843	0,1	**4.365**	0,3	18.166	0,1	4.904
eisenberg	0,2	8.515	0,1	**2.888**	0,2	7.968	0,2	12.196	0,2	7.011	0,2	4.831
elevator	3,2	32.075	3,2	20.782	2,2	17.799	7,0	52.419	2,8	29.751	1,4	**17.485**
ethernet	1,1	36.751	0,9	26.805	0,6	62.391	0,7	39.193	0,9	44.590	1,0	**24.734**
exampleS	0,0	2.847	0,0	2.009	0,0	**1.976**	0,0	2.011	0,0	2.212	0,0	2.580
fpmpy	3,8	**42.183**	2,4	44.236	8,4	103.404	17,4	195.203	13,6	136.758	15,3	218.462
gcd	29,7	382.210	4,9	**86.354**	5,5	114.935	6,2	142.990	26,0	368.642	5,7	143.064
prod_cell	4,4	34.737	1,1	**20.347**	2,0	23.663	2,4	34.430	2,9	29.576	2,3	26.693
rcnum	0,1	7.994	0,1	7.993	0,1	15.128	0,1	14.852	0,1	**7.815**	0,1	15.128
scheduler	49,1	262.292	41,4	254.369	0,2	**6.270**	0,1	30.863	1,0	9.832	1,5	14.763
slider	87,2	56.096	64,6	120.015	53,8	**46.032**	90,0	124.071	70,3	105.218	69,8	106.864
s298	0,1	2.412	0,0	**2.133**	0,0	2.507	0,1	5.506	0,1	3.649	0,1	3.727
treearbiter	0,5	13.624	0,2	15.214	0,2	16.501	0,3	18.287	0,3	12.841	0,3	**11.732**

Tabela 4.7: Tempo, em segundos, para o cálculo da alcançabilidade e número de nós BDDs após o término do procedimento

Nome	Peso1		Peso2		append		inter		left		right	
	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam
ABP	0,1	1.285	0,1	1.424	0,1	**1.212**	0,1	1.523	0,1	1.382	0,1	1.295
arbiter	0,1	964	0,1	**652**	0,1	944	0,2	1.621	0,1	887	0,1	798
bakery	0,1	1.604	0,1	1.374	0,1	**1.130**	0,1	1.154	0,1	1.205	0,1	1.151
coherence	2,2	17.961	1,6	10.947	0,7	**7.644**	1,0	9.473	1,6	9.504	1,0	8.759
dcnew	0,2	1.370	0,2	1.447	0,2	1.274	0,2	**1.155**	0,2	1.221	0,2	1.178
eisenberg	0,1	1.094	0,1	1.083	0,1	1.101	0,1	1.080	0,1	1.098	0,1	**1.059**
elevator	1,0	8.631	0,6	8.500	0,7	6.949	0,7	4.874	0,5	**3.224**	0,3	3.297
ethernet	2,9	10.555	1,9	**6.157**	2,0	8.256	1,9	8.257	2,0	9.324	1,9	8.370
exampleS	0,1	358	0,1	**352**	0,1	359	0,1	**352**	0,1	375	0,0	362
fpmpy	1,2	6.754	1,2	7.023	2,2	7.145	3,0	6.872	2,7	**5.200**	3,8	6.972
gcd	18,6	15.314	1,1	**6.273**	1,6	7.246	1,5	7.007	32,5	24.894	1,6	7.056
prod_cell	3,2	8.769	1,1	7.648	2,1	9.394	2,1	**7.087**	2,7	8.567	2,1	8.422
rcnum	0,3	2.008	0,1	2.008	0,2	1.378	0,3	1.391	0,2	**962**	0,2	1.378
scheduler	5,0	1.587	4,5	2.458	0,2	1.966	0,2	1.884	0,3	1.697	0,4	**1.519**
slider	1,9	13.132	2,3	**10.810**	2,0	14.163	1,9	11.953	2,1	11.123	2,0	12.608
s298	0,2	474	1,0	**443**	0,1	453	0,2	611	0,1	573	0,2	481
treearbiter	0,5	2.006	0,2	1.777	0,3	1.829	0,3	**1.055**	0,3	1.699	0,3	1.337

Tabela 4.8: Tempo, em segundos, para o reordenação, usando *sifting*, e número de nós BDDs após o término do procedimento

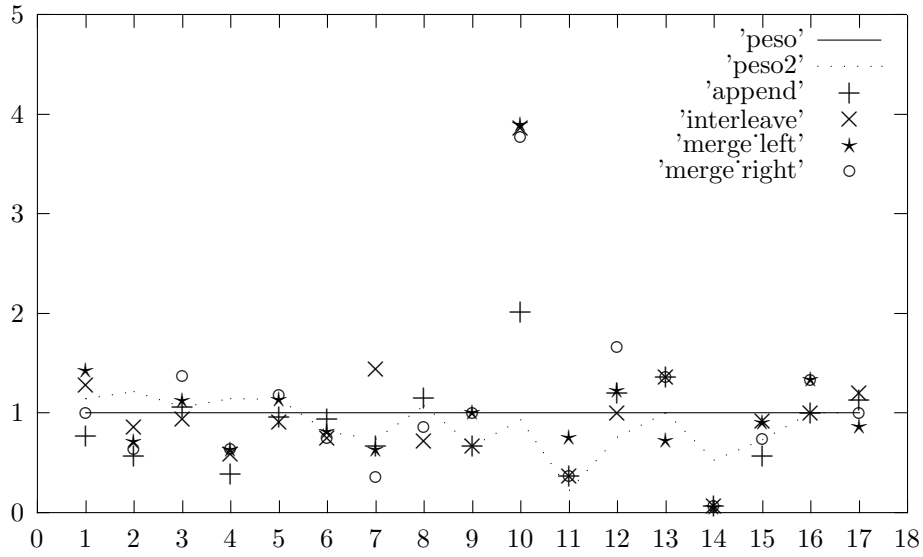


Figura 4.1: Número máximo de nós BDDs utilizado pelos métodos

de *merge_left*, *merge_right* e Peso1 obtiveram 3, 2 e 1 melhores resultados, respectivamente.

A figura 4.1 contém o gráfico com o número máximo de nós BDDs usados nos processos anteriores, normalizados pelo método do `peso1`. Observamos que os métodos de Peso2 e *append* possuem performance um pouco melhor do que os outros. O método de Peso2 é melhor do que o Peso1, com exceção dos exemplos 14 (`scheduler`), onde os métodos de peso são inferiores aos outros, e 10 (`fpmpy`), onde os métodos de peso são superiores, a performance é semelhante entre os métodos.

Nome	Peso1		Peso2		append		inter		left		right	
	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam	tem	tam
ABP	0,3	14.308	0,5	16.352	0,2	**13.286**	0,4	18.396	0,3	20.440	0,4	14.308
arbiter	0,4	14.308	0,4	17.374	0,2	**8.176**	0,4	12.264	0,3	10.220	0,3	9.198
bakery	0,5	16.352	0,5	17.374	0,4	17.374	0,7	**15.330**	0,5	18.396	0,4	22.484
coherence	28,9	157.388	44,1	179.872	8,3	**61.320**	16,0	93.002	23,5	100.156	14,1	77.672
dcnew	0,6	22.484	0,6	25.550	0,4	21.462	0,5	**20.440**	0,6	25.550	0,5	26.572
eisenberg	0,4	16.352	0,4	13.286	0,4	15.330	0,5	**12.264**	0,4	13.286	0,5	**12.264**
elevator	4,4	89.936	3,9	66.430	3,1	60.298	7,0	129.794	3,6	57.232	2,1	**31.682**
ethernet	4,4	55.188	3,1	59.276	2,8	63.364	2,8	**39.858**	3,3	54.166	3,3	47.012
exampleS	0,2	3.066	0,1	**2.044**	0,2	**2.044**	0,1	**2.044**	0,2	3.066	0,2	3.066
fpmpy	5,2	58.254	3,8	**54.166**	10,6	117.530	20,6	224.840	16,5	226.884	19,3	219.730
gcd	55,9	637.728	8,8	**145.124**	11,1	235.060	11,7	235.060	64,3	479.318	11,5	235.060
prod_cell	8,0	45.990	2,4	**34.748**	4,6	55.188	5,0	45.990	6,2	56.210	5,2	49.056
rcnum	0,8	11.242	0,4	11.242	0,6	15.330	0,6	15.330	0,2	**8.176**	0,7	15.330
scheduler	54,3	516.110	46,1	423.108	0,6	36.792	0,4	31.682	1,4	**25.550**	2,0	31.682
slider	89,1	805.336	67,0	589.694	55,1	**457.856**	92.1	739.928	72,5	727.664	72,1	596.848
s298	0,4	**3.066**	0,2	**3.066**	0,4	**3.066**	0,5	6.132	0,4	4.088	0,4	4.088
treearbiter	1,2	15.330	0,6	15.330	0,7	17.374	0,9	18.396	0,9	**13.286**	0,9	15.330

Tabela 4.9: Tempo total, em segundos, e pico de nós BDDs utilizados durante o processo

5 Conclusão

Desde que os BDDs começaram a ser usados como estrutura de dados para representar máquinas de estados finitos, eles passaram a ter grande importância na verificação automática de sistemas. As maquinas de estados finitos podem ser modeladas através de fórmulas da lógica proposicional, podendo ser representadas através de BDDs.

Apesar dos BDDs ocuparem menos espaço do que outras formas de representação de expressões booleanas, o espaço por eles utilizado pode variar de acordo com a ordem em que as variáveis estão dispostas. Torna-se, então, de fundamental importância para verificação automática, a busca por boas ordens de variáveis.

Várias técnicas de ordenação foram desenvolvidas, divididas em dois tipos: ordenação estática e ordenação dinâmica. A primeira é feita antes da construção dos BDDs e a segunda acontece após os BDDs serem construídos, durante a sua manipulação. Mesmo que a ordenação inicial seja boa, no caso da verificação automática, é necessário fazer reordenações durante o processo de verificação, pois o estado do sistema muda, fazendo com que um ordem que era boa antes, em outro momento não mais seja uma ordem boa.

Nesta dissertação, foi proposta uma nova abordagem para se fazer ordenação estática de variáveis BDDs que representam circuitos sequenciais, no contexto da verificação simbólica de modelos. Foi apresentado, baseado nesta abordagem, um algoritmo para a ordenação inicial, chamado de **heurística dos pesos**. Este algoritmo consiste em analisar a máquina de estados finitos que modela o sistema para determinar qual a melhor posição de cada variável. Esta análise se baseia em determinar o peso das variáveis e, para se determinar o peso de uma variável x, calcula-se o número de funções de transição que dependem de x. Variáveis de maior peso devem ficar mais no topo da ordem, isto acontece porque quando um circuito depende muito de uma variável, esta variável tem grande influência no comportamento do circuito.

Um outro fator importante no desenvolvimento de um algoritmo de ordenação estática é que a eficiência dos métodos de reordenação é diretamente proporcional a

qualidade de ordem na chamada ao método. Como consequência dessa observação, temos que a ordem inicial das variáveis tem uma influência muito grande na performance do processo como um todo, nesse caso, a verificação automática. Outra observação importante é o fato da verificação sempre mudar a configuração dos BDDs durante o processo, tornando necessário o uso de reordenação.

O algoritmo dos pesos foi, primeiramente implementado em uma estrutura que representa uma MEF diretamente através de BDDs, onde os pesos de cada variável foi calculado a partir da função `support(f)`, que retorna uma lista de variáveis BDDS que são os argumentos desta função. Duas variáveis com o mesmo peso são consideradas empatadas e, atualmente, variáveis empatadas permanecem na ordem em que estavam na leitura da descrição.

Os resultados preliminares mostram que, quando aplicada em conjunto com um bom método de ordenação dinâmica, esta abordagem obtém ótimos resultados, tanto no tempo, quanto no espaço alocado. O método de ordenação dinâmica utilizado foi o *sifting*. Os testes foram feitos comparando-se os resultados da aplicação do algoritmo na estrutura, do método de ordenação *sifting* e da aplicação dos dois juntos, usando a heurística dos pesos como ordenação inicial e o *sifting* como ordenação dinâmica. Variações de uma FIFO de um árbitro de barramento para um sistema multiprocessado [11] foi utilizada como teste. A aplicação da heurística dos pesos em conjunto com o método de *sifting* foi que obteve melhores resultados, mostrando que a heurística é capaz de colocar as variáveis em posições bem próximas da posição ótima, ou até mesmo na posição ótima.

A primeira versão do algoritmo, implementada diretamente nos BDDs, não analisa o empate de variáveis. A variável que estiver antes na ordem durante a leitura do arquivo continuará antes em caso de empate. Esta abordagem desconsidera um fator importante na ordenação, pois se duas variáveis que representam registradores `a` e `b` possuem o mesmo peso e a função de transição de `a` depende da variável `b`, não acontecendo o inverso, então, intuitivamente, a variável `b` deveria ficar mais no topo. Porém, se `a` está antes de `b` no arquivo de entrada, então `a` vai ficar antes de `b`.

Com os primeiros resultados, pudemos comprovar que esta nova abordagem tem o potencial de melhorar a performance da verificação automática de sistemas, mesmo que o algoritmo dos pesos ainda precise de melhorias, como um critério para efetuar o desempate de variáveis de mesmo peso.

Uma outra versão do algoritmo foi implementa em uma ferramenta de verificação, o VIS. Esta ferramenta lê descrições Verilog, BLIF e BLIF-MV, e verifica propriedades especificadas em lógica CTL. O VIS é uma ferramenta de verificação formal

de circuitos e representa os circuitos internamente como uma rede onde os nós são portas e registradores do circuito.

O algoritmo foi adaptado para sua implementação no VIS. Nesta versão, o algoritmo calcula os pesos das variáveis diretamente sobre a rede de nós, antes da criação dos BDDs. Os pesos das variáveis são determinados a partir da dependência dos nós da rede e as variáveis são criadas de acordo com a posição de cada nó, pois as variáveis são usadas para representar nós registradores e nós entradas. Duas versões do algoritmo foram implementadas, na primeira, a busca de dependências de um nó pára quando encontra uma entrada, uma constante ou um registrador, na outra, a busca só para quando encontra entradas e constantes.

Como foi percebido que deve haver um desempate entre variáveis de mesmo peso, o desempate está sendo feito com a ordenação alfabética. Esta decisão foi tomada por manter bits de menor peso mais no topo da ordem. Por exemplo, `init<0>` fica antes de `init<1>` na ordem das variáveis. Este problema se torna maior quando temos um grande número de variáveis de mesmo peso.

Nos testes realizados no VIS podemos perceber que, enquanto o algoritmo dos pesos é bem superior aos outros em determinados casos, ele é o pior em outros casos, o que nos leva a concluir que, de certa forma, cada algoritmo se adequa a determinadas **classes** de circuitos. Mas, mesmo sem conseguir ser melhor em todos os casos, percebemos que o métodos dos pesos consegue ordenar as variáveis de tal sorte que, quando aplicada uma heurística de reordenação, como o *sifting*, esse método conseguiu obter uma melhor ordem na maioria dos casos.

Através da observação dos resultados, podemos considerar um estudo sobre uma forma de desempate de variáveis de mesmo peso. Uma idéia inicial é a de verificar a interdependência de variáveis de mesmo peso. Por exemplo, se `a` e `b` têm o mesmo peso ambos são registradores, pode-se verificar se `a` depende de `b` e vice-versa, ficando antes na ordem a variável que faz parte da função de dependência da outra. Um outro estudo para o desempate é de tipos de variáveis, quando duas variáveis estiverem empatadas e as duas forem de tipos diferentes, como registrador e entrada, verificar em que ordem se obtém melhor performance.

Além da questão do desempate de variáveis, um outro motivo de estudo para melhorar a performance da verificação é o entendimento de porque determinada heurística se comporta melhor do que outras em um circuito e pior em outros. A intuição nos leva a crer que a estrutura do circuito pode dar uma idéia de qual método de ordenação melhor se aplica a ele. Uma forma de se classificar os circuitos, ou as máquinas de estados, pode ser desenvolvida para entender porque os algoritmos se comportam de

forma diferente em circuitos diferentes. Um primeiro estudo pode tentar entender a relação que o número de entradas, registradores e saídas podem ter nos métodos de ordenação já existentes e, a partir daí, adaptar os métodos para as possíveis variações de números entradas, registradores e saídas.

Um outro estudo, este diretamente sobre o método dos pesos, que pode ser realizado, é a identificação de blocos de variáveis. O agrupamento de variáveis pode aumentar a velocidade da verificação, mesmo que não obtenha as melhores ordens. Isto já foi verificado em outros experimentos [26], sua aplicação à heurística dos pesos pode melhorar ainda mais os resultados aqui obtidos.

Consideramos satisfatórios os primeiros resultados e os resultados realizados no VIS. Formas de melhorias podem ser estudadas no algoritmo. É preciso que mais testes sejam realizados, inclusive combinando o algoritmo dos pesos com outros métodos de reordenação. Também forçar a reordenação em determinadas fases da verificação pode melhorar ainda mais a performance da verificação.

Por fim, concluímos que esta nova abordagem na ordenação inicial de BDDs conseguiu obter bons resultados, e que ainda há o que se estudar e evoluir nesta abordagem, que considera os estados da máquina analisar a dependências das variáveis e encontrar a melhor posição de cada uma.

Referências Bibliográficas

[1] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[2] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEEE Design Automation Conference*, pages 40–45, June 1990.

[3] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, and F. Somenzi. Vis: A system for verification and synthesis, 1996.

[4] R.E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions Computers*, C(35):1035–1044, 1986.

[5] J.R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. Internal report, Carnegie Mellon University, Nov. 1990.

[6] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. Comput.-Aided Design Integrated Circuits and Systems*, 13(4):401–424, Apr. 1994.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference, DAC'90*, 1990.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. 10^{20} states and beyond. In *LICS'90:* 5^{th} *annual IEEE symposium on logic in computer science*, pages 428–439, Philadelphia,PA,USA, June 1990. IEEE.

[9] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[10] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Procs. of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, New-York, Jan. 1983.

[11] I. N. Cota and J. S. Aude. Hardware implementation of multicast communication for interconection networks. In 12^{th} *Symposium on Computer Architecture and High-performance Computing (SBAC-PAD'2000)*, pages 349–356, 2000.

[12] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 299–310, Standford, California, USA, 1994. Springer-Verlag.

[13] Umberto Souza da Costa. Ordenação de variáveis de bdds utilizando algoritmos genéticos paralelos. Master's thesis, UFRN, 2000.

[14] D. Déharbe, S. Shankar, and E. Clarke. Model checking vhdl with cv. In *FM-CAD'98: Formal Methods in Circuit Automation Design*, number 1522 in Lecture Notes in Computer Science. Springer Verlag, 1998.

[15] The VIS Group. Berkeley Logic Interchange Format — BLIF. Technical report, University of California, Berkeley, july 1992. Available on:
bibtex dishttp://vlsi.colorado.edu/~vis/usrDoc.html.

[16] H. Fujii, G. Ootomo, and C. Hori. Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams. In *IEEE /ACM International Conference on CAD*, pages 38–41, Santa Clara, California, 1993. IEEE Computer Society Press.

[17] G. D. Hachtel I. Moon and F. Somenzi. Border-Block Triangular Form and Conjunction Schedule in Image Computation. In *Proc. of Formal Methods in CAD (FMCAD'00)*, LNCS 1954, 2000.

[18] Jawahar Jain, William Adams, and Masahiro Fujita. Sampling schemes for computing OBDD variable orderings. pages 631–638.

[19] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International*

Conference on Very Large Scale Integration, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[20] G. Kahmi and L. Fix. Adaptive variable reordering for symbolic model checking, 1998.

[21] K.L.McMillan. The SMV system, symbolic model checking - an approach to the state-space explosion. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[22] Yuji Kukimoto. BLIF-MV. Technical report, University of California, Berkeley, may 1996. Available on: `http://vlsi.colorado.edu/~vis/usrDoc.html`.

[23] D. Long. bddlib – A Binary Decision Diagram Package. Available on: `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

[24] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis. In *Proceeding of the European Design Automation Conference (EDAC 91)*, pages 50–54, Amsterdam, 1991. IEEE Computer Society Press.

[25] C. Meinel and C. Stangier. Hierarchical Image Computation with Dynamic Conjunction Scheduling. In *Proceedings of the* 10^{th} *International Workshop on Logic and Synthesis (IWLS'01)*, pages 316–321, 2001.

[26] Christoph Meinel and Anna Slobodova. Speeding up variable reordering of OBDDs. In *International Conference on Computer Design*, pages 338–343, 1997.

[27] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[28] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE /ACM International Conference on CAD*, pages 42–47, Santa Clara, California, 1993. IEEE Computer Society Press.

[29] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *International Conference on Computer-Aided Design*, pages 6–9. IEEE, 1988.

[30] C. Shannon. A symbolic analysis of relay and switching circuits, 1938.

[31] A. Slobodov and C. Meinel. Sample method for minimization of obdd. In *Proceedings of the 7th International Workshop on Logic Synthesis (IWLS'98)*, pages 311–316, 1998.

[32] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, pages 285–309, 1955.

[33] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1995.