UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Um Estudo de Técnicas de Aceleração para Algoritmos de Análise de *Timing* Funcional baseados em Geração Automática de Teste.

por

ANA CRISTINA MEDINA PINTO

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Ricardo Reis
Orientador

Porto Alegre, dezembro 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pinto, Ana Cristina Medina

Um Estudo de Técnicas de Aceleração para Algoritmos de Análise de *Timing* Funcional baseados em Geração Automática de Teste / por Ana Cristina Medina Pinto. – Porto Alegre : PPGC da UFRGS, 2002.

85 f. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2002. Orientador: Reis, Ricardo Augusto da Luz.

1. Microeletrônica. 2. Ferramentas de CAD para Microeletrônica. 3. Verificação de Timing. 4. Análise de Timing Funcional. 5. Geração automática de teste (ATPG). 6. Portas Lógicas Complexas. I. Reis, Ricardo Augusto da Luz. II. Título.

AGRADECIMENTOS

Agradecer…

Inicialmente gostaria de dizer que seria uma tarefa impossível chegar aqui sem o apoio, a compreensão e o empenho dos seguintes companheiros :

Ao meu orientador Ricardo Reis por seu exemplo na busca incansável de novas oportunidades. O gosto por seu trabalho demonstrado diariamente é admirável. Obrigada pela oportunidade de fazer parte deste grupo de trabalho. Agradeço a sua amizade e compreensão. E sem dúvida agradeço pelas tentativas incansáveis para que eu terminasse este trabalho.

Ao meu co-orientador José Luis Güntzel por sua inesgotável energia na busca do conhecimento e por compartilhar isso comigo. Meu testemunho de que apesar de minhas dificuldades sempre me apoiou e me incentivou. A ele devo meu crescimento acadêmico sempre me oportunizando ser melhor a cada dia ou então me mostrando o caminho para que eu fosse melhor superando minhas limitações. Aqui expresso toda a minha admiração por seu profissionalismo. Agradeço sua amizade, obrigada.

Ao grupo de microeletrônica, todos sem excessões, por seus exemplos de profissionais de qualidade e excelência. Um agradecimento especial aos seguintes colegas que sempre estiveram mais próximos: Fernanda Lima, Choi Yung, Lisane Brisolara, Luciano Agostini, Débora Bertasi, Sérgio Ito, André Reis, Tatiane Campos, Érica Cota, e Marcelo Johann.

Ao grupo TIC-TAC, aos colegas Gustavo Wilke e Márcio Bystronski meus sinceros agradecimentos por suas importantes contribuições neste trabalho. Meu reconhecimento e admiração por sua dedicação.

Não poderia de deixar de reconhecer o trabalho realizado pelos funcionários do Instituto de informática, desempenhando suas funções nos laboratórios, na biblioteca e na secretaria: dos seus esforços dependem o bom andamento do nosso trabalho. Como também o suporte fornecido pelo CNPq ao conceder-me uma bolsa para dedicar-me a este trabalho.

A Deus, minha Fortaleza.

Ao meu amor, André, pela compreensão e apoio em todos os momentos, acreditando que eu posso mudar sempre.

Aos meus pais, Braúlio e Noeli, pelo amor e carinho.

Aos amigos, de todos os tempos, sempre próximos, estão guardados comigo em minhas melhores lembranças.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ATPG	Geração Automática de Padrões de Teste
CMOS	*Complementary Metal-Oxyde Silicon*
DAG	Grafo Acíclico Direto
FTA	Análise de *Timing* Funcional
mdt	Atraso máximo até o nodo **t**
SSF	Falhas de Colagem Simples (*Single Stuck Fault)*
SCCG	*Static CMOS Complex Gate*
TTA	Análise de Timing Topológica
PODEM	*Path-Oriented Decision Make*
FAN	*Fanout-Oriented Test Generation*
DETA	*Delay Enumeration-Based Timing Analisys*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem como objetivo estudar e avaliar técnicas para a aceleração de algoritmos de análise de *timing* funcional (FTA - *Functional Timing Analysis*) baseados em geração automática de testes (ATPG – *Automatic Test Generation*). Para tanto, são abordados três algoritmos conhecidos : algoritmo-D, o PODEM e o FAN. Após a análise dos algoritmos e o estudo de algumas técnicas de aceleração, é proposto o algoritmo DETA (*Delay Enumeration-Based Timing Analysis*) que determina o atraso crítico de circuitos que contêm portas complexas. O DETA está definido como um algoritmo baseado em ATPG com sensibilização concorrente de caminhos.

Na implementação do algoritmo, foi possível validar o modelo de computação de atrasos para circuitos que contêm portas complexas utilizando a abordagem de macro-expansão implícita. Além disso, alguns resultados parciais demonstram que, para alguns circuitos, o DETA apresenta uma pequena dependência do número de entradas quando comparado com a dependência no procedimento de simulação. Desta forma, é possível evitar uma pesquisa extensa antes de se encontrar o teste e assim, obter sucesso na aplicação de métodos para aceleração do algoritmo.

Palavras-chave: Análise de Timing Funcional (FTA), Sensibilização de Caminhos, Geração Automática de Padrões de Vetores de Teste (ATPG).

TITLE: "STUDY OF ACCELERATION TECHNIQUES FOR FUNCTIONAL TIMING ANALYSIS BASED ON AUTOMATIC TEST GENERATION ALGORITHMS "

ABSTRACT

This work has as objective to study and evaluate techniques for the acceleration of algorithms of Functional Timing Analysis (FTA) based on Automatic Test Patterns Generation (ATPG). Three widely known ATPG algorithms are used in this study: D-algorithm, PODEM and FAN. After the analysis of the algorithms and the study of some acceleration techniques, a new algorithm for functional timing analysis is proposed (DETA - *Delay Enumeration-Based Timing Analysis*). The proposed algorithm determines the critical delay of circuits that contain complex gates and can be classified as an algorithm based on ATPG with concurrent path sensitization.

In this work the proposed algorithm is implemented and validated for circuits that contain complex gates using the implicit macro-expansion approach. Experimental results demonstrate that, for some circuits, DETA has presented a lower dependency on the number of inputs when compared to the dependency presented by the simulation process. With this strategy, it is possible to reduce the expensive search for a test input, which is the main bottleneck of ATPG-based timing analysers.

Keywords: Functional Timing Analysis (FTA), path sensitization, Automatic Test Pattern Generation (ATPG)

1 INTRODUÇÃO

Desde os anos 80, o aumento da complexidade dos sistemas eletrônicos conduz à necessidade de ferramentas que automatizem o trabalho do projetista de circuitos integrados. Atualmente existem ferramentas que auxiliam o projeto em suas diversas fases. Algoritmos capazes de tratar sistemas com milhares de portas fazem parte destas ferramentas. Entretanto, com o advento das tecnologias submicrônicas, os modelos utilizados pelas ferramentas não mais atendem às necessidades de precisão requeridas para avaliar com segurança as restrições impostas ao projeto. Desde então tem sido empreendido um esforço no sentido de desenvolverem-se modelos físicos e computacionais mais precisos.

O objetivo principal da verificação de *timing* é determinar se as restrições de tempo impostas ao projeto podem ser satisfeitas ou não. Mais especificamente, a avaliação do desempenho temporal de circuitos digitais está associada à determinação da **máxima freqüência de funcionamento**, a qual é estimada a partir do **atraso máximo ou crítico**, no caso de blocos combinacionais puros.

Dentre as técnicas existentes, a simulação elétrica é aquela capaz de oferecer avaliações mais precisas das restrições temporais em circuitos CMOS. Os simuladores elétricos tal como o Spice [NAG75], representam o circuito como uma rede de elementos e solucionam o sistema de equações diferenciais ordinárias que modelam o comportamento do circuito para cada passo de simulação. Outra técnica passível de ser usada na avaliação temporal de circuitos é a simulação de *timing*, que segue a mesma filosofia da simulação elétrica, porém usando modelos de atraso simplificados. Em ambos os casos, uma avaliação do desempenho temporal do circuito pode não ser factível na prática. Mesmo para circuitos pequenos, a simulação pode demandar muito tempo de execução. Outro problema é garantir que o conjunto de vetores escolhido ativa o ou os caminhos que definem o atraso crítico do circuito. Teoricamente, esse problema seria solucionado ao se exercitarem todas as situações possíveis de (seqüências) de vetores de entrada. Porém, considerando-se somente pares de vetores, para um circuito com n entradas, seria necessário simular 2^{2n-1} combinações de entrada. Portanto, determinar o conjunto de vetores de entrada que garante encontrar o atraso do circuito não é um processo trivial.

Uma alternativa para superar as dificuldades encontradas na simulação é a técnica chamada **análise de *timing***. A análise de *timing* é uma técnica de avaliação do desempenho temporal de circuitos que usa o atraso dos caminhos que compõem o circuito para estimar seu atraso crítico. Na análise de *timing*, cada bloco combinacional é representado como um grafo acíclico direto (DAG - *Direct Acyclic Graph*) onde os nodos representam as portas e as arestas representam as conexões. Os pesos atribuídos aos nodos e às portas representam seus atrasos, respectivamente. Dado o DAG que

representa um bloco combinacional, a solução mais simples consiste em identificar o caminho de maior atraso deste bloco como sendo seu caminho crítico. Este problema pode ser resolvido em tempo linear pelo algoritmo conhecido como *topological sort* [COR90]. Tal técnica é referenciada como análise de *timing* topológica (TTA - *Topological Timing Analysis*). Contudo, pode não haver um vetor que ative o caminho mais longo do circuito. E neste caso, o atraso crítico pode ser menor do que o atraso do caminho topologicamente mais longo. Os caminhos que não transmitem alguma transição são chamados de caminhos falsos ou caminhos não sensibilizáveis. Um circuito pode conter inúmeros caminhos falsos. Como conseqüência, a análise de *timing* topológica pode gerar uma estimativa pessimista do atraso do circuito.

A fim de melhorar a precisão da estimativa do atraso, as ferramentas devem identificar a ocorrência de caminhos falsos. O problema de determinar se um caminho pode ser ativado ou não é conhecido como o problema de sensibilização de caminhos ou o problema de caminhos falsos [DU89]. Nos últimos anos, um grande número de trabalhos dedicaram-se ao desenvolvimento de algoritmos eficientes de análise de *timing* que considerem o problema da sensibilização de caminhos. A abordagem que estuda estes algoritmos é chamada de Análise de *Timing* Funcional (FTA - *Functional Timing Analysis*). Infelizmente, o problema de sensibilização de caminhos é NP-Completo e muitas considerações devem ser feitas a fim de se obter uma estimativa segura para o atraso crítico.

A precisão das ferramentas de verificação de *timing* é dependente do modelo computacional que descreve o comportamento do circuito. Portanto, é necessário um modelo de cálculo capaz de estimar de maneira precisa o atraso do circuito. Baseado na assertiva de que o atraso do circuito depende da natureza de suas entradas, o modelo é determinado através da observação dos tipos de entradas do circuito, que podem assumir **pares de vetores** ou **seqüências de vetores** [LAM94].

Ao determinar-se o modelo de cálculo em função dos tipos de entradas do circuito, observa-se a equivalência com o modelo de funcionamento de circuitos digitais. Quando o circuito opera de modo síncrono, assume-se que cada novo vetor de entrada é aplicado após um período T em que todas as saídas estão estáveis. Também assume-se que todas as entradas estão alinhadas (ou seja, mudam ao mesmo tempo). Então, neste caso, o modelo de cálculo é pares de vetores e este tipo de operação é referenciado como **modo de transição** (*transition mode*). Porém, na prática é muito difícil assegurar-se que todas as entradas mudam ao mesmo tempo. Na realidade, é comum ocorrer um desalinhamento das entradas, de modo que, o funcionamento de cada bloco combinacional é assíncrono. Assim, o modelo de cálculo de atraso mais realista é aquele que considera seqüências de vetores de entrada.

A ausência de uma implementação eficiente dos modelos baseados em seqüências de vetores motivou o uso de um modelo de cálculo que determina o atraso por meio de um único vetor de entrada, como uma aproximação do cálculo do atraso por seqüências de vetores. Este modelo assume que os nodos do circuito estão "flutuando", ou seja, os valores lógicos dos nodos do circuito são desconhecidos até que sejam determinados pela aplicação de um vetor de entrada. Assim, assume-se o pior caso para os valores dos nodos antes da chegada do vetor que determina o atraso do circuito. Então, o estado final dos nodos do circuito fica determinado por meio de um único vetor e este tipo de abordagem é conhecida por **modo de operação flutuante** (*floating mode*). Atualmente, todas as ferramentas para análise de *timing* funcional assumem o modo de operação flutuante.

Para determinar se um caminho pode ser ativado ou não, as ferramentas podem

identificar a ocorrência de caminhos falsos através da definição de um conjunto de condições que permitem decidir se um dado caminho é sensibilizável ou não. A este conjunto de condições denomina-se **critério de sensibilização**. O estudo das condições de sensibilização e a observação de diversos compromissos entre a complexidade computacional dos algoritmos para FTA e a precisão do resultado determinou a razão para a existência de mais de um critério de sensibilização. Assim, pode-se afirmar que, quanto mais abrangente o critério, mais preciso será o resultado. Porém, a conseqüência imediata é o aumento do custo computacional.

Os critérios de sensibilização podem ser classificados como **dependentes de atraso** e **independentes de atraso**. Os critérios dependentes de atraso são aqueles que consideram informações sobre o tempo de estabilização dos sinais. Por outro lado, denominam-se critérios independentes de atraso aqueles que não consideram as informações sobre o tempo de estabilização. A maioria dos critérios de sensibilização estão definidos no contexto do modo de operação flutuante. Pode-se citar como exemplos de critérios de sensibilização independentes de atraso a sensibilização estática [BEN90], co-sensibilização estática [DEV91], e também o critério apresentado por Brand e Iyengar [BRA88]. Como critérios dependentes de atraso, pode-se citar: viabilidade [MCG89], critério de sensibilização exata do modo flutuante ou simplesmente critério exato [CHE91], critério *Loose* [CHE91], critério dinâmico [CHE91] e o critério apresentado por Perremans et al. em [PER89].

A análise e desenvolvimento de algoritmos de FTA compreendem diversas possibilidades que diferenciam-se pelos seguintes aspectos: os modelos de computação de atraso para as portas e para os circuitos, o conjunto de condições usado para testar a sensibilização de caminhos e o método usado para testar se as condições de sensibilização foram satisfeitas ou não. O desenvolvimento de atividades de pesquisa nestes temas apresentou uma série de dificuldades, apesar do grande número de trabalhos publicados, devido à ausência de uma terminologia padrão. Portanto, o trabalho da dissertação adota o método de análise de *timing* funcional proposto em [GÜN2000].

Em [GÜN2000] é apresentada uma taxonomia para a classificação dos algoritmos de computação do atraso usados em FTA que sugere uma classificação dos algoritmos de FTA sob dois aspectos, além do critério de sensibilização. O primeiro aspecto diz respeito ao número de caminhos que podem ser verificados simultaneamente. Por esta ótica os algoritmos de FTA podem adotar a **sensibilização individual de caminhos**, a **sensibilização concorrente de caminhos** ou uma **abordagem mista**. De acordo com método utilizado para determinar se as condições de sensibilização são satisfeitas ou não, os algoritmos podem ser baseados em **geração automática de testes** (ATPG - *Automatic Test Generation*) ou baseados **em solvabilidade** (SAT - *Satisfiability*).

Fazendo uso da taxonomia proposta em [GÜN2000], pode-se citar três algoritmos de FTA mais representativos:

- sensibilização de um único caminho baseada em ATPG [CHE93];
- sensibilização concorrente de caminhos baseada em ATPG [DEV93];
- sensibilização concorrente de caminhos baseada em SAT [MCG93].

Então é proposto um método de análise de *timing* funcional, assumindo o modo de operação flutuante e baseado em ATPG. Tal método corresponde a uma extensão do algoritmo TrueD-F de Devadas et al. [DEV93a], a qual inclui a possibilidade de tratamento de circuitos contendo portas complexas. A computação do atraso de um circuito é feita usando o critério exato [CHE91]. Partindo do atraso topológico do

circuito, T, todos os caminhos de atraso maior igual a T-ε_0 são testados simultaneamente usando uma técnica derivada do algoritmo PODEM [GOE81]. Caso não haja nenhum caminho **sensibilizável,** um novo valor $\varepsilon_1 > \varepsilon_0$ é adotado, dando origem a um novo conjunto de caminhos (agora com atraso maior igual a T-ε_1). Os caminhos deste novo conjunto são também testados. O algoritmo termina quando é encontrado pelo menos um caminho **sensibilizável** pertencente a um conjunto de caminhos com atraso maior igual a T-ε_i. Então, o atraso do circuito é declarado estar entre T-ε_i e T-ε_{i-1}. A rotina que testa as condições de sensibilização usa um conjunto de regras conhecido como **timed calculus** [DEV94], que determinam o tempo de estabilização dos sinais na saída de cada porta de acordo com as assertivas do modo de operação flutuante, em confluência com as regras de dominância do critério de sensibilização exata.

Outro aspecto abordado por [GÜN2000] diz respeito ao procedimento para determinar se as condições de sensibilização são satisfeitas ou não. Neste caso, os algoritmos são baseados em ATPG e utiliza técnicas para a geração de testes baseadas no modelo de falhas de colagem simples (*single stuck-at faults*). Segundo este modelo, dado um circuito combinacional, o procedimento para a geração de um teste para uma falha propõe: **ativar a falha** e **propagar o erro** resultante até alguma saída primária (*primary output* - PO). Ao ativar a falha deve-se encontrar um conjunto de valores que aplicados às entradas primárias faz aparecer em l (nó do circuito) o valor lógico $\bar{v}$ (v negado). Este problema é chamado de **justificação de linha**.

A fim de solucionar o problema de justificação de linha, deve-se atribuir valores às linhas do circuito gerados como resultado de uma decisão tomada durante a execução do algoritmo. Além disso, uma seqüência de decisões tomadas pode incorrer em uma inconsistência, que advém da necessidade de se atribuir dois valores diferentes a uma mesma linha do circuito. Assim, o algoritmo de geração de testes deve permitir uma exploração do espaço de possíveis soluções, recuperando valores atribuídos às linhas do circuito imediatamente anteriores a uma tomada de decisão, em caso desta mostrar-se incorreta. Para tanto, os algoritmos de geração de testes incorporam uma "estratégia de retrocesso" (*backtracking*)**,** que permite uma exploração sistemática do espaço de soluções, recuperando estados anteriores do circuito no caso de decisões incorretas.

De acordo com [ABR90] o principal fator para o controle da complexidade de um algoritmo de geração de testes é minimizar o número de decisões incorretas, em outras palavras, significa reduzir o número de ocorrências de retrocessos. O princípio básico para atingir este objetivo é reduzir o número de problemas que requerem decisões, de modo que o algoritmo tenha menos oportunidades de tomar decisões erradas.As técnicas usualmente utilizadas em algoritmos de teste incorporam procedimentos que reduzem a ocorrência de retrocessos e assim, aceleram o processo da geração de vetores de teste. O algoritmo-D [ROT66], é um algoritmo clássico que aborda sensibilização de caminhos e propaga um sinal por caminhos utilizando o mecanismo de retrocesso. O PODEM [GOE81] e o FAN [FUJ83] são apresentados como novas técnicas que melhoram o desempenho do algoritmo-D, pois abordam o tratamento de problemas decorridos deste algoritmo. Além disso, o uso de heurísticas em conjunto com medidas de testabilidade [GOL79], [CHA89], [BRG84], [BEN84] influenciam o desempenho do algoritmo PODEM na medida que buscam reduzir as chances de ocorrência de retrocessos durante o processo de geração de vetores de teste. Além do uso de medidas de testabilidade, existem técnicas de aceleração mais sofisticadas que operam no retrocesso, tais como *backtracking* não-cronológico [SIL94] e *recursive learning* [SIL99].

A proposta deste trabalho focaliza o estudo dos algoritmos de ATPG (algoritmo-D, PODEM e FAN) objetivando identificar possíveis pontos de aceleração que operam no retrocesso. Assim, obtêm-se elementos suficientes para avaliar a aplicabilidade dessas técnicas de aceleração aos algoritmos de análise de *timing* funcional (FTA) baseados em ATPG. Além disso, o procedimento adotado para acelerar o processo de busca de vetores de teste é validado através da implementação do algoritmo **DETA** (*Delay Enumeration-Based Timing Analysis*) que propõe uma solução para acelerar o processo de busca.

1.1 Organização da Dissertação

A dissertação está organizada da seguinte forma:

O capítulo 2 revisa os conceitos básicos que fazem parte dos algoritmos de geração de vetores de teste. O estudo é complementado por uma apresentação detalhada dos algoritmos Podem [GOE81] e FAN [FUJ83], identificando os possíveis pontos de aceleração.

O capítulo 3 aborda a análise de *timing* funcional. Em particular, o algoritmo TrueD-F [DEV93a] é detalhado, uma vez que este representa o exemplo mais característico de algoritmos de FTA baseados em ATPG. Além disso, são apresentadas as regras para o cálculo do atraso em circuitos que contenham portas complexas [GÜN2000].

O capítulo 4, apresenta um estudo da aplicabilidade das técnicas de aceleração para os algoritmos de análise de *timing* baseados em ATPG. Ao avaliar o comportamento do algoritmos de FTA em circuitos que contém portas complexas, em particular, foi desenvolvido um estudo para a comparação entre cinco medidas de testabilidade aplicadas a circuitos que contém portas complexas. Os resultados dos dados práticos são apresentado em um artigo [GÜN02] enviado e publicado no Workshop Latino Americano de Testes (*LATW2002*).

No capítulo 5 é proposto o algoritmo **DETA** que utiliza as técnicas de ATPG baseado em enumeração de atrasos para acelerar o processo de busca de um conjunto de vetores de teste.

2 MÉTODOS PARA GERAÇÃO DE TESTES PARA FALHAS DE COLAGEM SIMPLES (*SINGLE STUCK FAULTS* - SSFS)

Neste capítulo será apresentada a teoria básica sobre os procedimentos para a geração de vetores de testes para circuitos combinacionais contendo portas simples. Na seção 2.1, o modelo de falhas de colagem simples é discutido. Uma conseqüência imediata é a definição da nomenclatura, denominada notação D. Estes conceitos básicos são utilizados no procedimento para a geração de um teste para falhas de colagem simples. Na seção 2.2 a estrutura do algoritmo básico para a geração do teste é apresentada. Os conceitos comuns aos algoritmos para a geração de testes também são mostrados. Na seção 2.3 o algoritmo-D e os algoritmos PODEM e FAN são discutidos.

2.1 Conceitos Básicos

A maior parte dos métodos para a geração de testes para circuitos combinacionais utiliza o modelo de falhas de colagem simples (*single stuck-at faults*). Segundo este modelo, uma linha (nó) ***l*** qualquer do circuito é suposta estar fixa ("colada") num valor lógico v, onde $v \in \{0,1\}$. Uma tal falha é comumente representada por ***l*** *s-a-v* (***l*** *stuck at v*).

Dado um circuito combinacional e admitindo-se que a linha ***l*** está colada com v, o procedimento para a geração de um teste para essa falha é composto por dois passos: **ativar a falha** e **propagar o erro** resultante até alguma saída primária (*primary output* - PO).

Ativar a falha significa encontrar um conjunto de valores que aplicados às entradas primárias (*primary inputs* - PIs) faz aparecer em ***l*** o valor lógico $\bar{v}$. Este problema é chamado de **justificação de linha** (*line-justification*)**.** O procedimento de propagação do erro, por sua vez, exige a definição de uma nomenclatura denominada notação D. Segundo esta notação, a variável D resulta da composição de dois valores lógicos v/v_f , que representam respectivamente o valor no circuito sem falha e o valor no circuito com falha. Assim, o espaço Booleano é estendido, conforme mostrado na tabela 2.1. As operações lógicas entre variáveis compostas (v/v_f) devem ser efetuadas operando separadamente os valores v e v_f . O resultado obtido de cada operação é novamente composto para o resultado final. Por exemplo, $\bar{D}+0=0/1+0/0=(0+0)/(1+0)=0/1=\bar{D}$. Aos valores da Tabela 2.1 acrescenta-se um quinto valor (X) para denotar o valor composto não especificado. As tabelas 2.2 e 2.3 mostram respectivamente os resultados das operações E e OU sobre as variáveis compostas $\{0,1,D,\bar{D}\}$.

TABELA 2.1 - valores lógicos referentes a notação D.

v/v_f	
0/0	0
1/1	1
1/0	D
0/1	$\overline{D}$

TABELA 2.2 - Tabela verdade da operação lógica E.

E	***0***	***1***	***D***	$\overline{D}$	***X***
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

TABELA 2.3 - Tabela Verdade da operação lógica OU.

OU	***0***	***1***	***D***	$\overline{D}$	***X***
0	0	1	D	$\overline{D}$	X
1	1	1	1	1	1
D	D	1	D	1	X
$\overline{D}$	$\overline{D}$	1	1	$\overline{D}$	X
X	X	1	X	X	X

2.2Estrutura do Algoritmo Básico para Geração de Testes para Falhas de Colagem Simples

Neste item, são apresentadas as estruturas básicas que fazem parte dos algoritmos de geração de teste para falhas de colagem simples. A divisão em duas seções, uma para circuitos com *fanout* unitário e outra para circuitos com *fanout* não unitário, se justifica, pois os circuitos com *fanout* unitário não necessitam de estratégias de *backtracking*. A estratégia é necessária quando existem conflitos lógicos ocorrendo durante o teste. Entretanto, conflitos não acontecem em circuitos com *fanout* unitário,

pois não comportam caminhos reconvergentes.

2.2.1 Circuitos com Fanout Unitário (*Fanout-Free*)

Em circuitos com fanout unitário não existe reconvergência de caminhos. Isto significa que, dada uma linha ***l*** em um ponto qualquer do circuito, existe um único caminho entre ***l*** e alguma das saídas primárias.

Na figura 2.1 vê-se a estrutura de um algoritmo que gera um teste para a falha ***l*** *s-a-v*, para circuitos com fanout unitário. Nele os passos de justificação e propagação são realizados pelas rotinas *Justifica* e *Propaga*.

```
Início
    inicializa todas as linhas com X
    Justifica( l , v̄ )
    se v=0
       então Propaga ( l , D )
             senão Propaga ( l , D̄ )
fim
```

FIGURA 2.1 - Geração de teste para a falha *l s-a-v*.

2.2.1.1 Rotina *Justifica*

O problema da justificação de linha é um processo recursivo no qual o valor na saída de uma porta simples é justificado por valores nas entradas da porta, e assim por diante, até que as entradas primárias sejam alcançadas. Por exemplo, considere uma porta NAND com k entradas. Existe apenas uma possibilidade para justificar o valor 0 na saída da porta (figura 2.2a). Por outro lado, para justificar o valor 1 na saída dessa porta, pode-se selecionar qualquer uma dentre as 2^k-1 possíveis combinações de entrada que resultam no valor 1 na saída da porta. Assim, a maneira mais simples de justificar o valor 1 na saída desta porta é atribuir 0 a somente uma de suas entradas, escolhida arbitrariamente, e deixar as demais entradas não especificadas (figura 2.2b). Isto corresponde a selecionar um cubo dentre os k-cubos primitivos que justificam o valor 1 na saída da porta.

(a) (b)

FIGURA 2.2 - Justifica o valor 0 ou 1 na saída de uma porta NAND com k entradas.

A rotina *Justifica* ativa a falha ***l*** *s-a-v*, o que é feito determinando-se os valores que aplicados nas entradas da porta justificam o valor $\bar{v}$ na sua saída.

A figura 2.3 mostra a rotina *justifica*, onde *val* é o valor que deve ser justificado no nó ***l*** do circuito, *C* é o valor controlante da porta G cuja saída é ***l*** e *i* representa a polaridade da porta. Caso ***l*** $\notin$ PI, então ***l*** é saída de uma porta G. Note que no pseudocódigo da figura 2.3 ***l*** se refere tanto ao nó que é saída de G quanto à própria porta G. Este procedimento baseia-se no fato de que, para portas de polaridade 0 (ANDs e ORs), o valor controlado é igual ao valor controlante da porta. Por exemplo, o valor

controlante da porta AND é 0, que é igual ao valor controlado. No caso de portas com polaridade 1 (NANDs e NORs), a variável *inval* irá corresponder à negação de *val*, de modo que a solução utilizada para portas com polaridade 0 permanece válida.

```
Justifica(l, val)              /*  na primeira chamada, val recebe v̄  */
    início
    inicializa l com val
    se l ∉ PI então retorna
    /*  l = saída da porta G */
    C=  valor controlante de l /*  valor controlante da porta cuja saída é l  */
    i= inversão de l /*  polaridade da porta cuja saída é l  */
    inval=val ⊕ i/* se a polaridade de l for 1, inval= val̄  */
    se (inval=C)
        então
            início
                seleciona uma entrada (j) de l
                Justifica(j,inval)
            fim
        senão para cada entrada j de l
        Justifica( j , inval )
    fim
```

FIGURA 2.3 - Pseudocódigo da rotina *Justifica.*

TABELA 2.4 - rotina *Justifica* aplicada a portas simples (AND, NAND,OR ,NOR).

(porta , *val* ***)***	*C*	***i***	$\overline{C}$	*Inval*	***Justifica cada_entrada (inval=*** $\overline{C}$ ***)***	***Justifica uma_entrada (inval=C)***
(AND , 0)	0	0	1	0		✓
(AND , 1)	0	0	1	1	✓	
(NAND , 0)	0	1	1	1	✓	
(NAND , 1)	0	1	1	0		✓
(OR , 0)	1	0	0	0	✓	
(OR , 1)	1	0	0	1		✓
(NOR , 0)	1	1	0	1		✓
(NOR , 1)	1	1	0	0	✓	

A tabela 2.4 apresenta os valores atribuídos às variáveis contidas na rotina *Justifica* quando aplicada a portas simples. De acordo com a funcionalidade da porta e o valor ***l*** (0 e 1) que deverá ser justificado na saída a rotina fica condicionada a dois tipos de

justificativa: para uma entrada da porta ou cada uma delas. Por exemplo, considere a porta NAND representada na Figura 2.2: a tabela 2.4 mostra como a rotina se comporta ao justificar os valores 0 e 1 na saída da porta. Para a entrada da rotina (NAND, 0), obtemos: c=0, pois 0 é o valor controlante da porta; i=1 e inval=1, pois a polaridade da porta NAND é 1 portanto *inval* nega o valor de *val*. O algoritmo pergunta se inval=c, como a resposta é negativa então justificar o valor 0 na saída da porta NAND fica condicionado a justificar o valor j (identificado devido a funcionalidade da porta) em cada entrada da porta.

2.2.1.2 Rotina Propaga

O erro é propagado do ponto do circuito onde se supõe que ocorreu a falha até a saída primária (PO – *primary output*). Para tanto, é necessário sensibilizar o único caminho de ***l*** até a saída primária. Cada porta no caminho deve ter uma única entrada sensibilizável, a qual faz parte do caminho que vai propagar o erro (figura 2.4). Assim, deve-se justificar o valor não controlante para todas as entradas laterais do caminho, o que significa que o problema de propagação do erro é equivalente ao problema de justificação de linha. O pseudocódigo da rotina *Propaga* é visto na figura 2.5.

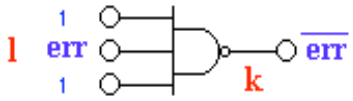


FIGURA 2.4 - Propagação do erro.

```
Propaga (l , erro)
      /* erro é D ou D̄ */
      início
       inicializa l com erro
       se l é PO então retorna
       k = fanout de l
       C = valor controlante de k
       i = inversão de k
       Para cada entrada j de k diferente de l
           Justifica( j , C̄ )
       Propaga( k , erro ⊕ i )
        fim
```

FIGURA 2.5 - Pseudocódigo da rotina *Propaga.*

2.2.2 Circuitos com Fanout não Unitário

Para circuitos combinacionais, com fanout unitário ou não, mas que não apresentam caminhos reconvergentes, qualquer seqüência de decisões referentes à justificação de linha ou propagação do erro conduz a uma solução válida para a geração do teste.

Porém, no caso de circuitos que apresentam caminhos reconvergentes, os problemas de justificação de linha não são independentes, e podem resultar em inconsistências, quando da justificação de valores. Uma inconsistência advém da necessidade de se atribuir dois valores diferentes a uma mesma linha do circuito. Para

esta classe de circuitos, as decisões a serem tomadas por um algoritmo de geração de teste são de dois tipos: escolha dos valores de entrada que justificam a saída de uma porta (exatamente como no caso de circuitos com fanout unitário) e escolha do caminho para propagar o erro.

Como uma seqüência de decisões pode resultar em inconsistência, o algoritmo de geração de teste deve permitir uma exploração do espaço de possíveis soluções, recuperando os valores atribuídos às linhas do circuito anteriores a uma decisão, caso esta se mostre incorreta. Para tanto, os algoritmos de geração de teste incorporam uma "**estratégia de retrocesso**" *(backtracking)*, que permite uma exploração sistemática do espaço de soluções.

Os valores atribuídos às linhas do circuito são o resultado de decisões tomadas que "implicam" em outros valores. O processo de cálculo desses valores e a verificação da consistência destes é normalmente referenciado como **implicação**.

Considere a falha ctrl_n s-a-0 no circuito da figura 2.6. A seguir, será descrita uma possível seqüência de passos para a geração de um teste para essa falha. Essa seqüência está resumida na tabela 2.5.

Para ativar a falha em questão, é necessário justificar o valor 1 na linha crtl_n. Para tanto, existem três possibilidades de valores a serem justificados nas linhas (p0,p1): (0,1), (1,0) e (0,0). Supondo que se tenha escolhido (0,1), então os valores para as entradas (a0, b0, a1, b1) que justificam p0=0 e p1=1 são: (1101), (1110), (0001), (0010). Assumindo-se o vetor (1101), por implicação, conclui-se que g0=0 e g1=1.

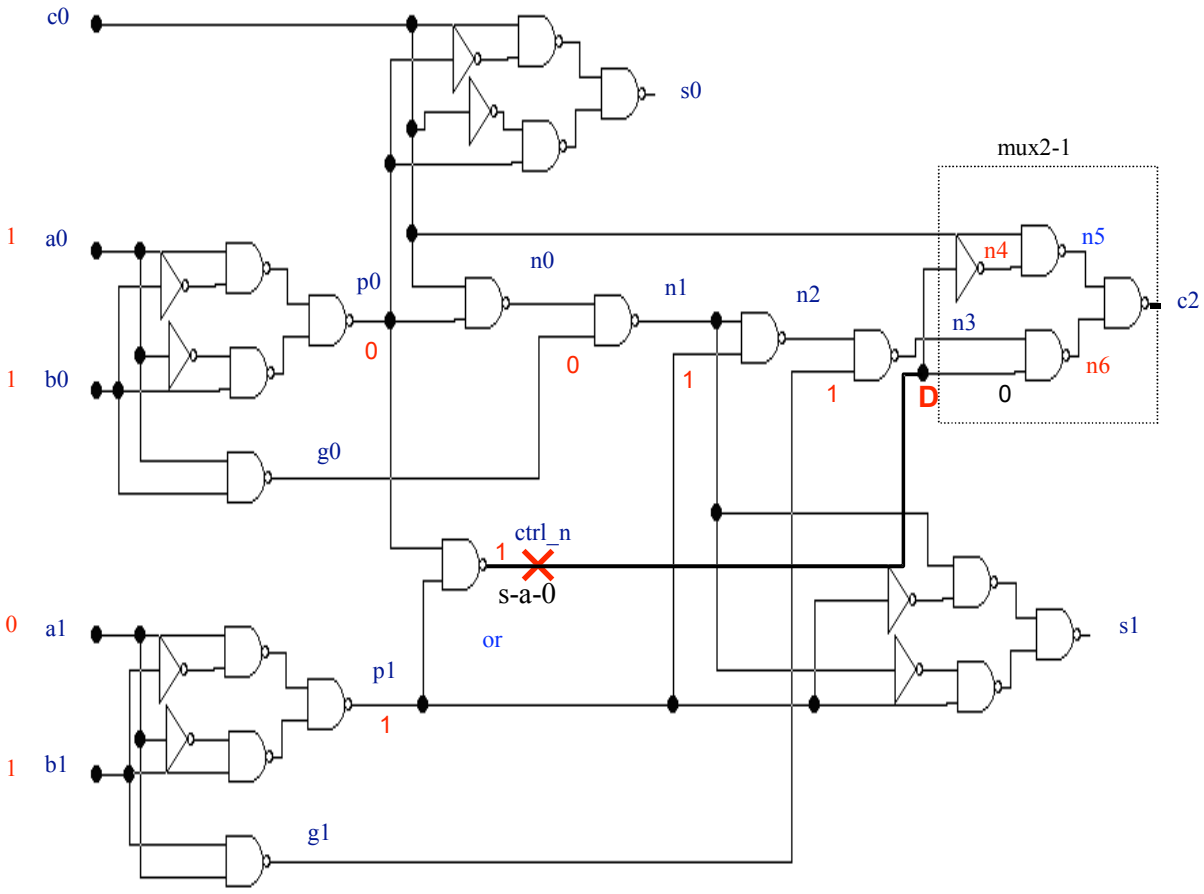


FIGURA 2.6 - Propagação do erro em circuitos com fanout não unitário.

Para propagar-se o erro há duas alternativas de caminhos: por n4 e por n6. Supondo que se decida propagar por n4, é necessário justificar c0=1, que já é uma entrada primária, e n6=1. Para justificar n6=1, são necessárias as seguintes justificações: n3=0, n2=1, n1=0 e n0=1. n0=1 implica que g0=1. Porém, a decisão 2 implicou em g0=0, o que se caracteriza como inconsistência. Essa inconsistência demonstra que a seqüência de decisões até aqui tomadas foi equivocada, sendo necessário mudá-la. Por questão de simplicidade, suponhamos que o procedimento de *backtracking* adotado desfaça a última decisão tomada (decisão 3), que foi tentar propagar o erro por n4. Assim, é tomada a decisão 4, propagar por n6, em lugar da decisão 3. Para propagar o erro por n6 serão necessárias as seguintes justificações: n3=1, n2=0 e n1=1. Para justificar n1=1 existem duas possibilidades, n0=0 e n0=1, uma vez que g0=0. Isso dá origem à decisão 5, n0=0, que implica em c0=1 e p0=1. Porém, p0=0, como decorrência da decisão 1, de modo que surge nova inconsistência. Então, voltando atrás na decisão 5, segue a decisão 6, n0=1. Desta não decorre implicação alguma, uma vez que c0 pode valer 0 ou 1, pois p0=0. Então, resta apenas justificar n5=1, o que implica que c0=0.

TABELA 2.5 – Possível seqüência de passos para a geração de teste para a falha ctrl_n s-a-0.

Decisões	Implicações	Comentários
	ctrl_n = 1	Ativar a falha ctrl_n s-a-0 Leva à decisão **1**
1 (p0,p1) = (0,1)		Justificar o valor 1 Leva à decisão **2**
2 (a0,b0,a1,b1) = (1,1,0,1)	g0=0 g1=1	Leva à decisão **3** (escolha do caminho)
3 propagar a falha por n4	c0=1	
	Justifica n6=1 *Justifica* n3=0 *Justifica* n2=1 *Justifica* n1=0 *Justifica* n0=1 g0=1	Inconsistência (pois g0=0) ***Backtracking!*** (desfaz decisão **3**)
4 propagar a falha por n6	*Justifica* n3=1	
	Justifica n2=0 *Justifica* n1=1	
	n0=0 ou n0=1	Leva à decisão **5**
5 n0=0	c0=1	
	p0=1	Inconsistência (pois p0=0) ***Backtracking!*** (desfaz decisão **5**)
6 n0=1	c0=1 ou c0=0 (i.e., c0=X)	
	Justifica n5=1 c0=0	**ok,** propaga o erro até a saída

A figura 2.7 mostra a **árvore de decisão** relacionada ao procedimento de geração de teste descrito anteriormente. Nela, a seqüência das decisões tomadas corresponde a um caminhamento à esquerda. A seqüência hachureada corresponde ao sucesso na geração do teste. No exemplo descrito, assumiu-se uma seqüência aleatória para as decisões. Entretanto, um algoritmo de geração de teste geralmente incorpora uma estratégia para tomada de decisões.

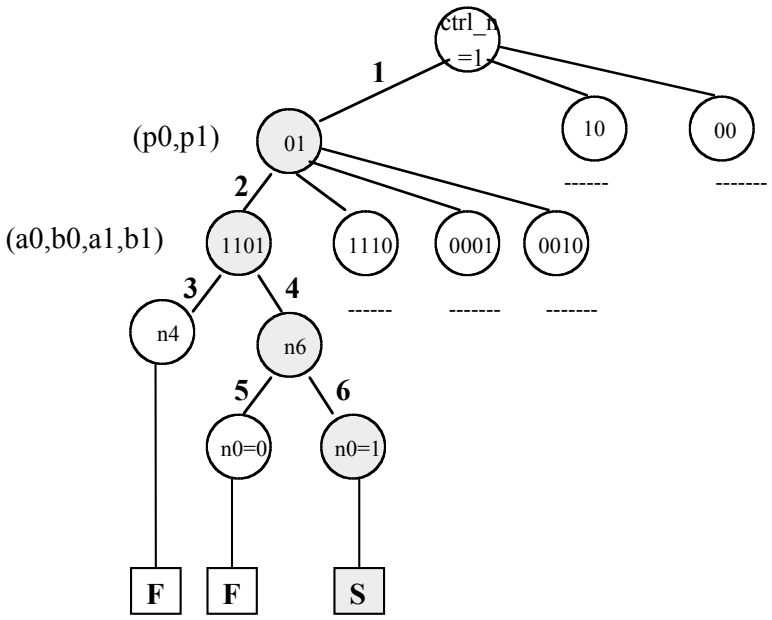


FIGURA 2.7 - Árvore de Decisões.

A rotina Resolve() (figura 2.8), mostra um esquema recursivo de *backtracking* para a geração de um teste para uma falha s-a-*v*. O problema original de justificar o valor $\bar{v}$ na linha ***l*** e propagar o erro até uma de suas saídas primárias é transformado recursivamente em subproblemas, pois no caso de impedimento para resolvê-lo diretamente os subproblemas são resolvidos em primeiro lugar. O procedimento apresenta dois tipos de representação para a verificação dos resultados durante a execução do algoritmo: FRACASSO ou SUCESSO. O FRACASSO representa a ocorrência de uma inconsistência. O SUCESSO significa que o teste para uma falha *s-a-v* foi encontrado.

Quando a rotina não consegue determinar imediatamente FRACASSO ou SUCESSO então um dos problemas não solucionados (justificação de linha ou propagação do erro) é selecionado. Neste momento, podem existir várias alternativas para a solução. O algoritmo seleciona uma das alternativas e tenta resolver o problema até o próximo nível de recursão. O processo continua até que a solução seja encontrada ou tenham-se esgotado todas as alternativas. algoritmos.

```
  Resolve()
início
se Implica_e_verifica()= FRACASSO então retorna FRACASSO
se (erro foi propagado até alguma PO e todas as linha foram justificadas)
    então retorna SUCESSO
se( erro não pode ser propagado até alguma PO)
    então retorna FRACASSO
seleciona um problema não resolvido
repete
    início
        seleciona uma alternativa não escolhida para solucionar
        se Resolve()=SUCESSO então retorna SUCESSO
    fim
até que todas as alternativas tenham sido escolhidas
retorna FRACASSO
fim
```

FIGURA 2.8- Pseudocódigo do algoritmo para geração de teste

Uma parte dos algoritmos para a geração de teste para falhas de colagem simples apresentam estrutura similar ao Resolve(). O item 2.3.3 descreve e analisa os conceitos comuns utilizados nos algoritmos de ATPG.

2.2.3 Conceitos Comuns aos Algoritmos para a Geração de Testes para Falhas de Colagem Simples

A execução do algoritmo que implementa a estratégia de *backtracking* pode ser representada como uma árvore de decisão. Desta forma, cada nodo chamado nodo de decisão, representa o problema a ser resolvido. As arestas de cada nodo correspondem às decisões e determinam os caminhos alternativos para a solução do problema. O fracasso, representado pela letra **F** em um nodo terminal da árvore, indica que uma inconsistência foi detectada ou encontrou-se um estado que impede a propagação do erro. Por outro lado, o sucesso, representado pela letra **S** no nodo terminal, indica que o teste foi encontrado. A execução do algoritmo corresponde a um caminhamento em profundidade na árvore (no caso da figura 2.7, caminhamento à esquerda).

Durante a execução do algoritmo, a rotina *Resolve()* utiliza o processo exaustivo para encontrar a solução do problema. O processo garante encontrar o teste para a falha *s-a-v* caso ele exista. De outra forma, pode-se dizer que na hipótese do algoritmo não encontrar o teste para uma falha específica, ela **não pode ser detectada**. O algoritmo garante encontrar a solução, caso ela existir, porque é capaz de **enumerar implicitamente** todas as possíveis soluções. Enquanto a **enumeração explícita** gera todos os vetores de entrada e verifica a detecção da falha, na enumeração implícita a pesquisa de conjuntos de vetores que detectam a falha é baseada em encontrar os vetores que satisfazem o conjunto de restrições impostas pelos valores que devem ser justificados nas linhas do circuito. Assim, o espaço de soluções do problema torna-se menor à medida que o conjunto de restrições aumenta durante a execução do algoritmo, reduzindo o número de vetores que as satisfazem.

Ao analisar a complexidade do algoritmo *Resolve()*, pode-se dizer que, em decorrência da natureza exaustiva do processo de busca, a complexidade do algoritmo

no pior caso é de ordem exponencial. O pior caso ocorre quando um número muito grande de decisões devem ser desfeitas (*backtrackings*) como conseqüência da realização de uma pesquisa extensa antes de encontrar o teste ou então quando ocorrer o caso de falha não detectável. Para minimizar o tempo de execução de um algoritmo de geração de testes, utiliza-se algum parâmetro que limite a pesquisa. Por exemplo, pode-se abandonar a pesquisa quando o número de decisões incorretas atingir um dado tempo de CPU. Por outro lado, o melhor caso ocorre quando o teste é encontrado sem a necessidade de *backtracking*, o que ocorre em circuitos com fanout unitário. e em circuitos sem caminhos reconvergentes.

Ao analisarem-se as circunstâncias em que ocorrem o pior caso, quando um número grande de decisões devem ser desfeitas, e o melhor caso, quando o teste do circuito é encontrado sem que exista a necessidade de *backtracking*, pode-se concluir que o principal fator para o controle da complexidade de um algoritmo de geração de teste é **minimizar o número de decisões incorretas**. O princípio básico para atingir este objetivo é reduzir o número de problemas que requerem decisões, de modo que o algoritmo tenha menos oportunidades de tomar decisões erradas. Este é o chamado **princípio das implicações máximas**, que usa como parâmetro de seleção para as decisões a maximização do número de implicações. A heurística de maximizar o número de implicações reduz o número de problemas que necessitariam de uma tomada de decisões ou permite chegar mais cedo a uma inconsistência.

A execução de um algoritmo de geração de teste faz uso de dois conjuntos chamados de **fronteira-*D*** e **fronteira-*J***. O conjunto **fronteira-*D*** contém todas as portas cujo o valor na saída é desconhecido. Então o valor X é atribuído a saída está assinalada com o valor X e ao menos uma das entradas está assinalada com D ou $\overline{D}$. A propagação do erro consiste em selecionar uma das portas e assinalar valores para as entradas assinaladas com X de modo que a saída seja igual a D ou $\overline{D}$. Este procedimento é referenciado como operação **avança-*D***. Se fronteira-*D* retorna vazio durante a execução do algoritmo, então o erro não pode ser propagado até alguma saída primária, indicando que o *backtracking* deve ocorrer.

O segundo conjunto, **fronteira-*J***, é utilizado para controlar a situação dos problemas de justificação de linha não solucionados. Assim, fronteira-*J* contém todas as portas cujo valor na saída é conhecido mas não foi implicado por valores nas entradas da porta. Seja a porta simples G, com paridade i(G) e cujo valor controlante é c(G). Como a saída de G é c⊕i, ao menos duas de suas entradas devem estar assinaladas com X. Além disso, nenhuma entrada pode estar assinalada com c(G).

No procedimento *Resolve()* (figura 2.8) o passo de implicação é realizado pela rotina *Implica_e_verifica*, que trata o espaço de problemas que somente são resolvidos por implicação, verificando a consistência dos valores implicados. Então, as tarefas que fazem parte desse passo são: cálculo dos valores que podem ser unicamente determinados por implicação, verificação da consistência e atribuição de valores e manuntenção dos conjuntos fronteira-*D* e Froteira-*J*.

No processo de implicação, todos os valores atribuídos são processados segundo uma fila de assinalamentos. Os valores podem ser propagados em direção às saídas primárias (***forward implication***) e em direção às entradas primárias (***backward implication***). Os dados da fila tem o seguinte formato: (***l***, *v*', **direção**), onde *v*' é o valor atribuído para a linha ***l*** e **direção** ∈{*backward, forward*}. Por exemplo, ao gerar um teste para falha ***l*** *s-a-1* as duas entradas iniciais na fila de assinalamentos são: (***l***, 0, *backward*) e (***l***, $\overline{D}$, *forward*).

A rotina *Implica_e_verifica* recupera cada entrada na fila de assinalamentos.

Para atribuir o valor v' a linha ***l*** é necessário verificar a consistência com o valor v que está em ***l*** (inicialmente todos os valores são iguais a X). Uma inconsistência é detectada se v é diferente de X ou de v'. A exceção ocorre quando no caso de se estar na linha em que a falha é detectada. Neste caso, se o valor é binário, deve ser propagado na direção das entradas primárias, e o erro, na direção das saídas primárias. O valor consistente é atribuído e então processado de acordo com **direção**.

A figura 2.9 ilustra dois exemplos de propagação de valores na direção das entradas primárias (*backward*). A coluna à direita mostra os resultados das atribuições feitas na coluna da esquerda. As setas indicam em que direção os valores foram propagados. Observa-se que na segunda situação a atribuição a=0 adiciona a porta ao conjunto fronteira-*J*.

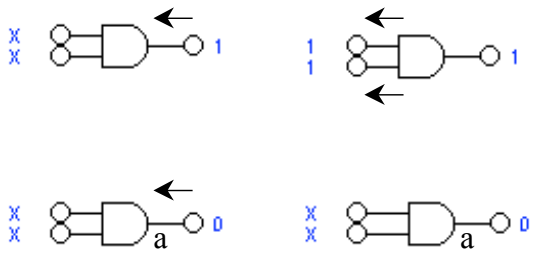


FIGURA 2.9 - Propagação de valores na **direção** *backward*.

De forma similar, a figura 2.10 mostra dois exemplos de propagação de valores na direção das saídas primárias (*forward*). Observa-se como, na primeira situação, a propagação *forward* pode induzir a uma propagação *backward* em outra entrada da mesma porta.

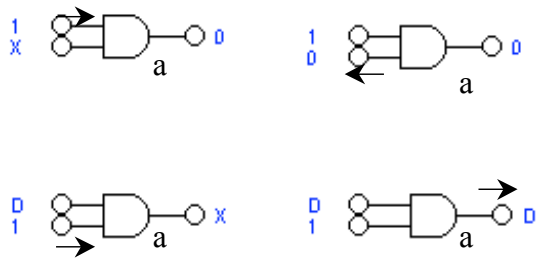


FIGURA 2.10 - Propagação de valores na **direção** *forward.*

Outros dois tipos de operações que podem ser realizadas durante a execução do algoritmo são: a **reversão de decisões incorretas** e a **avaliação da propagação do erro** (*error-propagation look-ahead*).

Existem implicações que estão fortemente relacionadas entre si. Por exemplo, considere que se deseja justificar o valor 0 na saída de uma porta AND, cujas entradas a, b e c possuem o valor X. Suponha que a primeira decisão tomada, a=0, mostrou-se incorreta. Isto significa que, independentemente dos valores de b e c, a não pode ser 0. Assim, antes de tentar justificar o valor 0 para b, é necessário atribuir 1 à entrada a e implicar este valor. Similarmente, se b=0 falhar, será necessário assinalar 1 às entradas a

e b (e implicando estes valores) antes de tentar c=0. O benefício desta técnica de reversão de decisões incorretas é aumentar o número de implicações.

Outra operação que pode ser realizada durante a execução do algoritmo é a avaliação da propagação do erro. Dado um caminho, busca-se identificar uma situação com maior probabilidade de propagar o erro, a fim de minimizar a ocorrência de *backtracking*. Define-se o **caminho-X** como o caminho cujas linhas está atribuído o valor X. Supondo G uma porta pertencente à fronteira-*D*, os valores D ou $\overline{D}$ somente podem ser propagados de G até uma saída primária Z se existir pelo menos um caminho-X, que contém a porta G, e vai até a saída primária Z. Caso contrário, os valores atribuídos as linhas no caminho podem levar a não ocorrência da propagação do erro. Neste caso, é necessário realizar o *backtracking*. Utilizando esta técnica pode-se podar a árvore de decisão, de modo a acelerar o sucesso da geração do teste, evitando um maior número de *backtrackings*.

2.3 Algoritmos

Os conceitos apresentados na seção anterior são comuns à classe de algoritmos para a geração de testes. Nesta seção serão apresentados três algoritmos específicos que fazem parte desta classe. A seguir são apresentados os seguintes algoritmos: **Algoritmo-D**, **PODEM** (*Path-Oriented Decision Make*) e **FAN** (*Fanout-Oriented Test Generation*).

2.3.1 Algoritmo D

O Algoritmo-D [ROT66], [ROT67] é conhecido como um algoritmo clássico. A figura 2.11 mostra o pseudocódigo do algoritmo D. Note-se que este código é uma generalização da rotina *Resolve()*, apresentada na subseção 2.2.2 [Figura 2.8] . Neste caso, os conceitos envolvidos já encontram-se descritos e analisados. A principal característica do algoritmo-D é admitir a propagação dos erros por caminhos reconvergentes. Esta característica, referenciada por **sensibilização de múltiplos caminhos**, é utilizada na detecção de falhas que não podem ser detectadas através da sensibilização de um único caminho [SCH67].

Na figura 2.11, o termo “atribui” refere-se à ação de adicionar o “valor” que será atribuído a linha ***l*** à fila de assinalamentos.

```
início
se Implica_e_verifica () = FRACASSO então retorna FRACASSO
se  (erro não está em PO) então
    início
        se Fronteira-D = ∅ então retorna FRACASSO
        repete
            início
                seleciona uma porta (G) na lista Fronteira-D
                C = valor controlante da porta G
                Atribui C̄ a todas as entradas de G que tem o valor x
                se alg-D() =  SUCESSO então retorna SUCESSO
            fim
        até todas as portas de Fronteira-D tenham sido testadas
        retorna FRACASSO
    fim
/*propaga o erro até PO*/
se Fronteira-J =  ∅  então retorna SUCESSO
seleciona uma porta (G) da lista Fronteira-J
C = valor controlante da porta G
repete
    início
          seleciona a entrada (J) de G com o valor X
          assinala C para j
          se alg-D() = SUCESSO então retorna SUCESSO
          atribui  C̄  para j/decisão revertida/
    fim
até todas as entradas de G estão especificadas
retorna FRACASSO
fim
```

FIGURA 2.11 - Pseudocódigo do algoritmo-D.

A fila de assinalamentos armazena os valores, ao invés de atribuí-los definitivamente às linhas do circuito.

Durante a execução da rotina *Implica_e_verifica()*, cada entrada da fila de assinalamentos é processada. É necessário verificar a consistência com o valor ‘v’ que está em ***l*** .

2.3.2 Algoritmo PODEM (*Path-Oriented Decision Make*)

Experimentos relatados em [GOE81] demostraram que o algoritmo-D não é eficiente para circuitos do tipo ECAT (*error-correction-and-translation*). Tais circuitos são caracterizados por possuir um número significativo de portas do tipo ou-exclusivo e por apresentar reconvergências. O algoritmo PODEM, desenvolvido por Goel [GOE81], demonstrou ser mais eficiente que o algoritmo-D quando trata de circuitos do tipo ECAT.

No algoritmo PODEM valores lógicos são assinalados às entradas primárias do circuito, uma entrada por vez. A seguir, os valores lógicos assinalados às entradas são propagados em direção às saídas primárias do circuito pelo processo de implicação. A característica marcante do algoritmo determina o que processo de retrocesso (*backtracking*) restringe sua ocorrência somente às entradas primárias do circuito.

A figura 2.12 apresenta o pseudocódigo do PODEM(). Inicialmente, o procedimento *backtrace()* calcula valores nas linhas do circuito até que sejam atingidas as entradas primárias. Assim que um valor lógico é atribuído à uma entrada primária, a função *implica()* encarrega-se de satisfazer um **objetivo**, assinalando um valor lógico v_k à linha ***k*** do circuito (*objetivo(k ,*v_k*)*). Entretanto, quando um objetivo não é satisfeito, ou um número maior de entradas do circuito devem ser assinaladas, ou então, ocorreu um conflito. Para que um número maior de entradas sejam assinaladas o algoritmo chama a função *backtrace()*. Em caso de conflito, o *backtracking()* desfaz o último assinalamento v_j atribuído a uma entrada, e assinala o valor lógico v'_j (complemento) à entrada primária e uma nova implicação é realizada. O PODEM busca satisfazer um único objetivo de cada vez.

Algoritmo PODEM (com backtrace)

Início
(k , v_k*) = objetivo()*
(j , v_j*) = backtrace(k ,* v_k*)* /* j representa uma entrada primária */
implica(j , v_j*)*
se (*implica()* produz um conflito) **então**
backtracking()
se *backtracking()* esgotou todas possibilidades então retorna FRACASSO
senão /*falha ativada */
enquanto (*erro* não foi propagado até alguma PO)
selecionar uma porta G do conjunto fronteira-D
backtrace(entrada X da porta G, valor não controlante)
implica()
se (implicação gerou conflito) **então**
backtracking()
fim

FIGURA 2.12 - Pseudocódigo do algoritmo PODEM.

O exemplo de circuito apresentado na figura 2.13 ilustra como o algoritmo se comporta para uma falha do tipo s-a-1 na aresta **h**. Para ativar a falha em questão a função objetivo identifica que o valor lógico 0 deve ser assinalado à aresta **h** (*objetivo* (**h**,0)). A seguir, a função *backtrace***(h,0)** calcula os valores lógicos para as arestas **f** e **c** (**f** =1 e **c**=0), conforme mostrado na figura 2.13(a). Como resultado do *backtrace***(h,0)** o valor lógico 0 é assinalado à entrada **c** e implicado em direção às saídas primárias pela função *implica*(**c**,0). Entretanto, conforme ilustrado na figura 2.13(b), o valor atribuído à **c** não controla a aresta **h**, pois não foi possível satisfazer o *backtrace***(h,0)**. Então, uma ou mais outras entradas primárias devem ser assinaladas. Para tanto, a função *backtrace***(h,0)**, será novamente invocada. Nesta nova chamada *backtrace***(h,0)** calcula **g**=1, **d**=0 e **a**=1. Ao assinalar o valor lógico 1 à entrada primária **a**, o *objetivo*(**h**,0) é satisfeito. Além disso, em conseqüência do assinalamento **e**=1, o valor lógico1 é assinalado à aresta **g**, o que causa a propagação do erro (D) até a saída primária **i,** como

mostra a figura 2.13(d). Neste caso, o algoritmo PODEM terminou com sucesso sem que tenha ocorrido inconsistência.

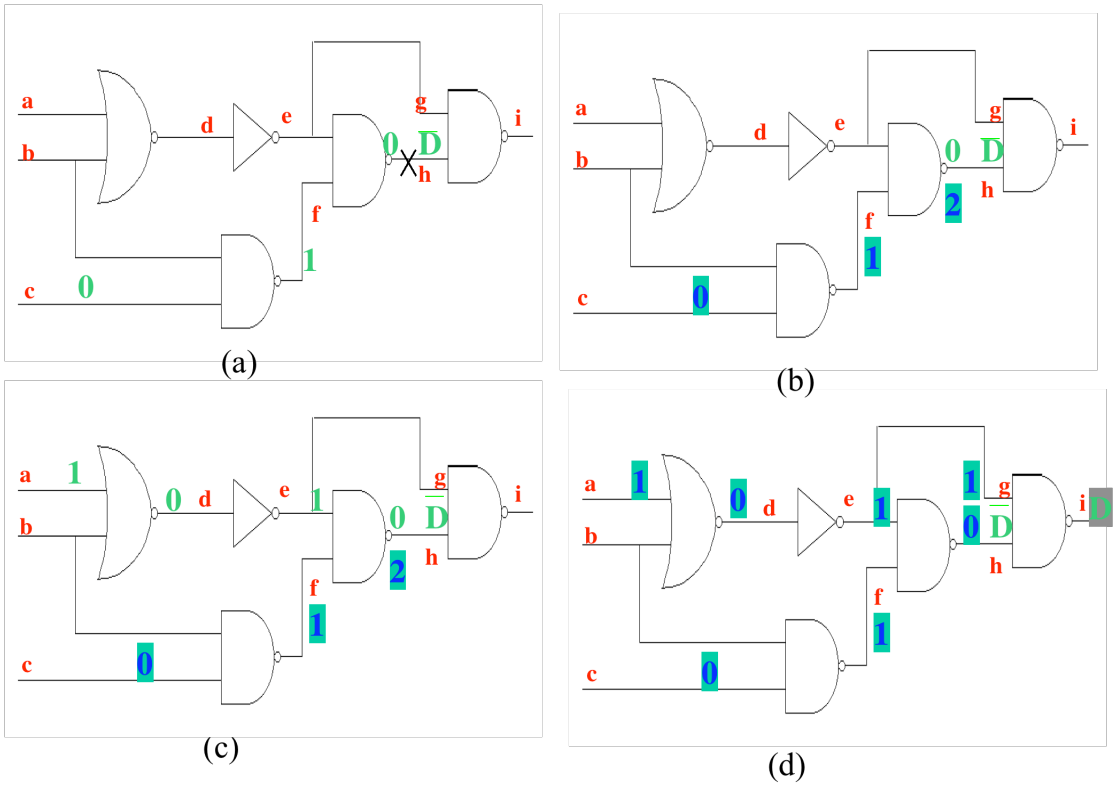


FIGURA 2.13 - Exemplo do algoritmo PODEM com sucesso sem ocorrência de *bakctracking*.

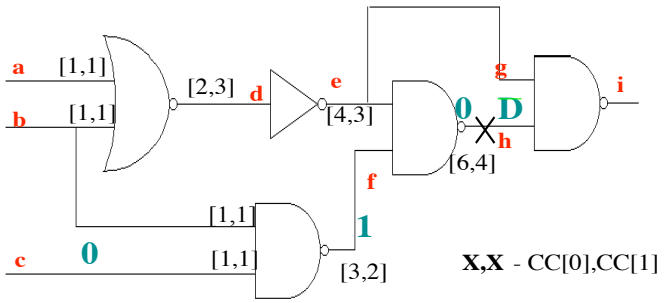
No caso de ocorrer inconsistência, será necessário chamar o procedimento de retrocesso (*backtracking)*. Como no PODEM o procedimento de *backtracking* ocorre somente nas entradas primárias, então, o último assinalamento atribuído a uma entrada primária é complementado. Após, a modificação da entrada é implicada em direção às saídas primárias do circuito. No caso em que o novo assinalamento não satisfaz o objetivo, diz-se que o *backtracking* não obteve sucesso.

Considere novamente o circuito da figura 2.13(a). Para assinalar o valor lógico 0 à aresta **h** (*objetivo*(**h**,0)) a única combinação possível de entrada que resulta no valor lógico 0 na saída da porta é calcular **e**=1 e **f**=1. O PODEM emprega heurísticas em cada passo da função *backtrace*(**h**,0) toda vez que o valor lógico deve ser transferido da saída de uma porta lógica para a entrada da porta na direção das entradas primárias. A heurística, chamada *easy/hard heurístic*, considera que o valor lógico mostrado na saída da porta é reflexo do tipo de valor lógico colocado na entrada da porta. Assim, quando a entrada da porta apresenta um valor controlante, significa que a heurística escolhe a entrada em que o valor lógico é mais fácil de ser calculado, como por exemplo, uma porta AND/NAND cujo o valor lógico colocado numa das entradas é 0. No caso em que todas as entradas da porta foram colocados em valor não controlante, a heurística escolhe a entrada cujo o valor lógico é mais difícil de ser colocado. Além disso, medidas de testabilidade quantificam os valores assinalados as arestas do circuito para

avaliar qual o melhor caminho para seguir. Segundo [CHA89] grande parte das medidas de testabilidades utilizam os conceitos de **controlabilidade** e **observabilidade**. A controlabilidade avalia a facilidade de controlar um determinado valor lógico em uma determinada aresta do circuito. Enquanto a observabilidade avalia com que facilidade pode-se observar o valor lógico de uma determinada aresta na saída do circuito. Na literatura, pode-se citar as medidas de testabilidade apresentadas por Goldstein [GOL79], as medidas baseadas em probabilidade [BRG84].

A figura 2.14 ilustra um exemplo de *bakctrace*() orientado pela heurística *easy/hard* em conjunto com as medidas de testabilidade apresentadas por Goldstein [GOL79]. O cálculo da controlabilidade é mostrado na figura 2.14(a) conforme a descrição das equações para as portas lógicas AND/OR. De acordo com a figura 2.14(b), todas as entradas primárias são inicializadas com valor igual a 1 para as variáveis CC[0], CC[1] que representam respectivamente controlabilidade para o valor lógico 0 e controlabilidade para o valor lógico 1. Após o cálculo da controlabilidade para cada aresta do circuito, a função *backtrace*(), encarregada de trazer um valor lógico até as entradas primárias, utiliza a heurística para escolher o caminho a percorrer. Por exemplo, entre as arestas **e**=1 e **f**=1, deve-se escolher a entrada mais difícil de se colocar o valor lógico 1. Então é necessário comparar o valor CC[1] de cada aresta e escolher a maior dentre eles (e=1, CC[1]=3). Assim, a heurística é aplicada para percorrer o caminho até as entradas primárias. No exemplo, o *bakctrace*() escolhe o caminho que contém as arestas **e**, **d**, **a**.

função AND entrada - n : $CC^0[O] = \min(CC^0[I_1], CC^0[I_2], ..., CC^0[I_N]) + 1$ $CC^1[O] = CC^1[I_1] + CC^1[I_2] + ... + CC^1[I_N] + 1$
Função OR entrada - n: $CC^0[O] = CC^0[I_1] + CC^0[I_2] + ... + CC^0[I_N] + 1$ $CC^1[O] = \min(CC^1[I_1], CC^1[I_2], ..., CC^1[I_N]) + 1$



(a) (b)

FIGURA 2.14 - Exemplo do cálculo da controlabilidade

Uma vez detalhado, o PODEM, difere dos demais algoritmos para geração de testes, baseados na rotina *resolve()* - figura 2.8, sob alguns aspectos. Um dos aspectos diz respeito aos valores atribuídos às linhas internas do circuito durante a execução do algoritmo. Estes valores são assinalados durante o processo de **implicação,** na direção das saídas primárias(*forward implication*), como função dos valores assinalados às entradas primárias do circuito. Por outro lado, o passo de implicação na direção das entradas primárias não é realizado, pois não existem problemas de justificação de linhas para serem solucionados. Assim, o conjunto fronteira-*J* , que contém problemas de justificação de linha não é utilizado.

O algoritmo PODEM é considerado um procedimento simples quando comparado com outros algoritmos de geração de teste. Os resultados experimentais

apresentados em [GOE81] mostram que, em geral, o PODEM apresenta um tempo de processamento menor do que o tempo de processamento do algoritmo-D. Conforme os resultados apresentados, o procedimento mostra-se mais eficiente quando utilizado na geração de teste para circuitos com maior número de portas.

2.3.3 Algoritmo FAN (*Fanout-Oriented Test Generation*)

Os experimentos relatados por Goel [GOE81] demonstraram que o algoritmo PODEM obteve êxito ao reduzir o número de ocorrências de retrocessos quando comparado com o algoritmo-D. Entretanto, observa-se que existem inúmeras possibilidades de reduzir o número de retrocessos no algoritmo. Um estudo realizado por Fujiwara [FUJ83] focalizou pontos específicos do PODEM, em que foi possível propor técnicas como forma de eliminar a ocorrência de retrocessos. Ao validar algumas destas técnicas como forma de acelerar o processo da geração do teste (redução do número de retrocessos), Fujiwara [FUJ83] apresentou um novo algoritmo, o FAN (*Fanout-Oriented Test Generation*).

O algoritmo FAN encarrega-se de satisfazer o **objetivo**, assinalando e implicando um valor lógico v_k a linha k do circuito (*objetivo(k , v_k)*). A seguir, como resultado desta implicação, um conjunto de novos objetivos é gerado. O procedimento *backtrace* múltiplo contribui para satisfazer um conjunto de objetivos simultaneamente. Assim, escolhem-se valores que serão assinalados às várias linhas do circuito. A seguir, estes valores são implicados até que o conjunto de objetivos tenham sido completamente satisfeitos. Por fim, o algoritmo trata de justificar os valores às linhas do circuito.

Além de satisfazer um conjunto de objetivos simultaneamente, outra característica marcante do algoritmo determina que o processo de *backtrace* pode parar em linhas internas do circuito, em vez de parar somente quando são atingidas as entradas primárias. O FAN introduziu uma nova nomenclatura para identificar as linhas internas nas quais pode-se parar o *backtrace*.

- **linha limite** (*bound line*) é a linha que pode ser alcançada por um ponto em que o *fanout* é não unitário;
- **linha livre** (*free line*), são as linhas que podem ser atingidas pelas entradas primárias apenas por caminhos de *fanout* unitário;
- **linha principal** (*head lines*) é uma linha livre que é adjacente a uma linha limite.

A figura 2.15 ilustra um exemplo em que A, B, C, E, F, G, H e J correspondem a linhas livres. Além disso, as linhas *K*, *L, M* são definidas como linhas limite. Por fim, H e J correspondem a linhas principais. A linha na qual a função *backtrace*() para é chamada de linha principal. Neste caso, o problema de justificação de linha é adiado para o último estágio na geração do teste. Desta forma, pode-se assinalar valores às entradas primárias que justifiquem valores nas linhas principais, sem a ocorrência de retrocessos. Assim, mesmo antes de atingir o último estágio na geração do teste, pode-se resolver o problema de justificação de linha e reduzir o número de *backtrackings*.

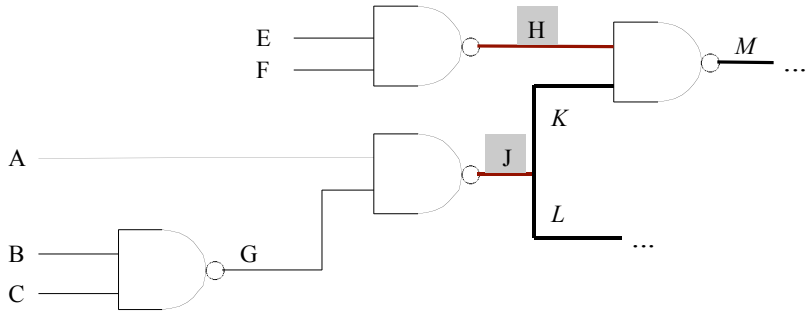


FIGURA 2.15 - Exemplo de linhas principais. [ABR90]

Considere o circuito mostrado na figura 2.15, suponhamos que em algum ponto, durante a execução do algoritmo PODEM, deseja-se assinalar o valor lógico 0 a linha **J**. Além disso, por hipótese, assume-se que os valores assinalados previamente às entradas A,B e C não determinam o valor J=0. Assim, ocorre uma inconsistência e é necessário realizar um retrocesso para alterar o valor lógico assinalado a uma das entradas primárias. A figura 2.16(a) representa a árvore de decisões que corresponde ao processo de *backtracking* durante a busca de valores para as entradas A, B e C. Entretanto, como a linha J é identificada como uma linha principal, a função *backtrace*() do FAN para em **J**. Assim, o assinalamento J=0 é identificado e a justificação da linha neste caso é adiada para ser solucionada num próximo estágio. Por fim, pode-se reduzir o número de retrocessos como mostra a figura 2.16(b).

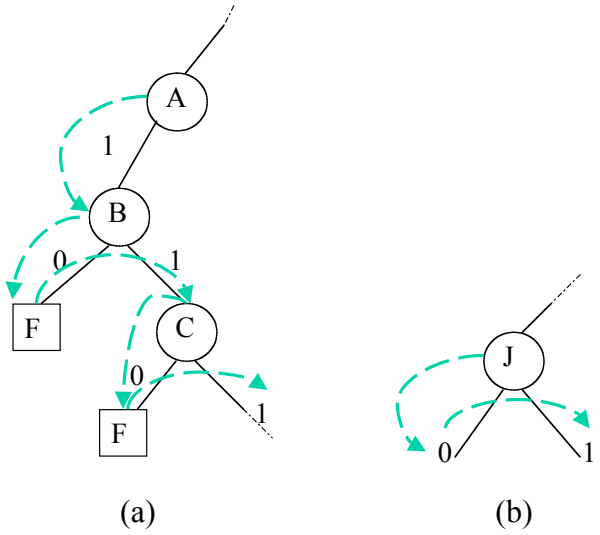


FIGURA 2.16 - Comportamento da operação *Backtraking*. (a) PODEM. (b) Linha principal do FAN.

A figura 2.19 apresenta o pseudocódigo do FAN(). Inicialmente, o procedimento *implica_e_verifica* assinala e implica o valor do objetivo principal. Como resultado desta implicação um conjunto de novos objetivos são gerados. A função *implica_e_verifica* é composta de duas sub-funções, o *backward imply* e *forward imply*. O procedimento ***forward imply*** é equivalente a função **implica**() do algoritmo PODEM que encarrega-se de satisfazer um **objetivo**, assinalando um valor lógico v_k à linha ***k*** do circuito. A função ***backward imply*** assinala valores as entradas (*fanins*) de uma porta quando estes valores são obrigatórios. O procedimento *implica_e_verifica* também testa se algum objetivo foi satisfeito durante a implicação. A sub-função *forward imply* é

utilizada apenas para satisfazer os objetivos. Por outro lado, o *backward imply* pode gerar novos objetivos. Além disso, o **objetivo** no algoritmo FAN é diferente do objetivo do PODEM. O objetivo no FAN é uma linha *k* cujo valor lógico não foi justificado e que a cada passo do *backtrace* pode-se gerar um novo objetivo. A seguir, o conjunto de objetivos é tratado pelo procedimento *backtrace* múltiplo (***Mbacktrace***(objetivos)).

Ao contrário do PODEM, em que a função *backtrace* calcula os valores nas linhas do circuito ao longo de um único caminho, o procedimento *Mbacktrace*(), do FAN, é capaz de escolher várias linhas do circuito para serem setadas ao mesmo tempo. Ao identificar os objetivos, deve-se representá-los por uma 3-upla: (**s**, $n_0(s)$, $n_1(s)$). Assim, **s** corresponde à linha do circuito que será assinalado o valor lógico 0 ou 1. A variável $n_0(s)$ representa o número de vezes que o valor lógico 0 é assinalado à linha **s**, também $n_1(s)$ identifica o número de vezes que o valor lógico 1 é assinalado à linha **s**. Assim, ao assinalar o valor lógico 0 a uma linha **s** do circuito obtemos a seguinte representação : (s, $n_0(s)$, $n_1(s)$) = **(s, 1, 0)**. Além disso, o valor 1 determina que o valor lógico 0 é assinalado uma vez à linha **s**. Por outro lado, um objetivo inicial que deve assinalar o valor 1 a uma linha **s** está representado por (s, n_0 (s), $n_1(s)$) = **(s, 0, 1)**. A partir da identificação dos objetivos iniciais, o múltiplo *backtrace* gera os próximos objetivos, chamados **objetivos correntes,** até que as linhas principais sejam atingidas. Então, o próximo objetivo é determinado assinalando o valor lógico 0 ou 1 à linha **s** e pelo cálculo do número de vezes que cada valor lógico deve ser assinalado a cada linha do circuito.

A figura 2.17 ilustra como são computados o número de vezes que cada valor lógico deve ser assinalado à linha **s**, de acordo com os critérios descritos abaixo.

1. Porta *AND* (fig2.17(a))

A entrada **X** é mais fácil de controlar para o valor lógico 0. Neste caso, o número de vezes que o valor 0 é assinalado à aresta X é igual ao número de vezes que este valor foi assinalado a aresta **Y**, assim $\mathbf{[n_0(X) = n_0(Y)]}$. Da mesma forma, para o valor lógico 1 temos $\mathbf{[n_1(X) = n_1(Y)]}$. Para as demais entradas X_i, no caso do valor lógico ser igual a 0, estes valores não são computados, $\mathbf{[n_0(Xi) = 0]}$. Por outro lado, o valor 1 é calculado $\mathbf{[n_1(Xi) = n_1(Y)]}$.

2. Porta *OR* (figura2.17(b))

A entrada **X** é mais fácil de controlar para o valor lógico 1. Neste caso, o número de vezes que o valor 1 é assinalado à aresta X é igual ao número de vezes que este valor foi assinalado a aresta **Y**, assim $\mathbf{[n_1(X) = n_1(Y)]}$. Da mesma forma, para o valor lógico 0 temos $\mathbf{[n_0(X) = n_0(Y)]}$. Para as demais entradas X_i, no caso do valor lógico ser igual a 1, estes valores não são computados, $\mathbf{[n_1(Xi) = 0]}$. Por outro lado, o valor 0 é calculado $\mathbf{[n_0(Xi) = n_0(Y)]}$.

3. Porta *NAND*.

A entrada **X** é mais facilmente controlável para o valor lógico 0. Assim, o número de vezes que o valor 0 é assinalado à aresta **X** é igual ao número de vezes que o valor lógico 1 foi assinaldo à aresta **Y**, deste modo temos $\mathbf{[n_0 (X) = n_1(Y)]}$. De modo idêntico, para o valor 1 na aresta **X**, é computado o número de vezes que valor lógico 0 foi assinalado à aresta **Y**. Assim, $\mathbf{[n_1(X) = n_0(Y)\ ]}$. Para as demais entradas Xi, no caso

do valor lógico ser igual a 0, estes valores não são computados, $\mathbf{[n_0(Xi) = 0]}$. Por outro lado, o valor 1 é calculado como sendo $\mathbf{[n_1(Xi) = n_1(Y)]}$.

4. Porta NOR.

A entrada **X** é mais facilmente controlável para o valor lógico 1. Assim, o número de vezes que o valor 1 é assinalado à aresta **X** é igual ao número de vezes que o valor lógico 0 foi assinaldo à aresta **Y**, deste modo temos $\mathbf{[n_1 (X) = n_0(Y)]}$. De modo idêntico, para o valor 0 na aresta **X**, é computado o número de vezes que valor lógico 1 foi assinalado à aresta **Y**. Assim, $\mathbf{[n_0(X) = n_1(Y)\ ]}$. Para as demais entradas Xi, no caso do valor lógico ser igual a 1, estes valores não são computados, $\mathbf{[n_1(Xi) = 0]}$. Por outro lado, o valor 0 é calculado como sendo $\mathbf{[n_0(Xi) = n_1(Y)]}$.

5. Porta Inversor (figura2.17(c))

Nesta porta, o número de vezes que o valor 0 é assinalado à aresta **X** é igual ao número de vezes que o valor lógico 1 foi assinaldo à aresta **Y**, deste modo temos $\mathbf{[n_0 (X) = n_1(Y)]}$. De modo idêntico, para o valor 1 na aresta **X**, é computado o número de vezes que valor lógico 0 foi assinalado à aresta **Y**. Assim, $\mathbf{[n_1(X) = n_0(Y)\ ]}$

6. Pontos de fanout não unitário (figura 2.17(d))

Neste caso, o cálculo do número de vezes que o valor lógico 0 é computado na aresta **X** é igual ao somatório de i=1 até o *fanout* da aresta (k), do número de vezes que o valor lógico 0 foi assinalado as demais arestas X_i. Assim, $\mathbf{[n_0(X) = \sum_{i=1...k} n_0(Xi)]}$. Da mesma forma, o número de vezes que o valor lógico 1 é computado na aresta **X** corresponde à $\mathbf{[n_1(X) = \sum_{i=1...k} n_1(Xi)]}$.

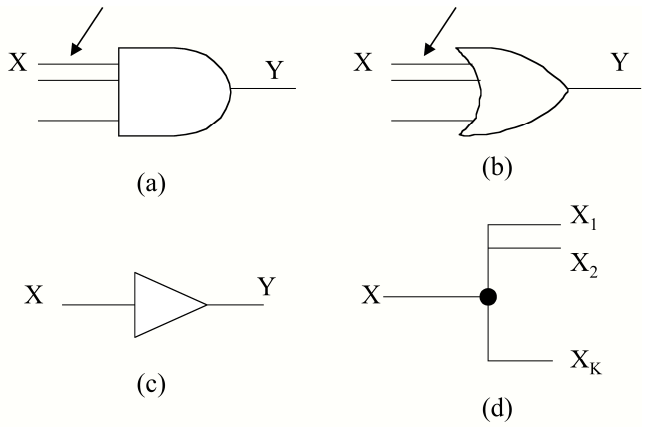


FIGURA 2.17 - Portas simples e Pontos de *fanout* não unitário. (a) AND. (b) OR. (c) NOT. (d) *fanout*.

A figura 2.18 mostra um exemplo de cálculo das variáveis $n_0(s)$ e $n_1(s)$. Os objetivos iniciais são (Q,0,1) e (R,1,0), i.e., as arestas Q e R devem assinalar os valores lógicos 1 e 0 respectivamente. Os próximos objetivos derivados dos objetivos setados,

são computados de acordo com os critérios descritos do item 1 ao item 6, representados na figura 2.16. Particularmente, observe a aresta com *fanout* não unitário, **H**, o cálculo de n_1(H) é a soma de n_1(K) e n_1(L). A 3-upla correspondente à aresta H é igual a (**H**, 0, 2).

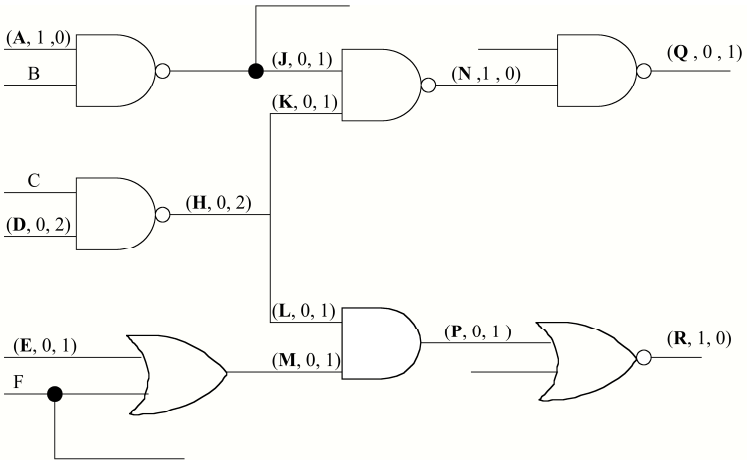


FIGURA 2.18 - Cálculo das variáveis n_0 e n_1.

O múltiplo *backtrace* inicializa dois conjuntos de objetivos simultaneamente, o **conjunto de objetivos primários (ou iniciais)** e o **conjunto de objetivos correntes**. Ao mesmo tempo que armazena os objetivo iniciais, o procedimento gera os próximos objetivos que devem ser satisfeitos. Além disso, são incorporados dois novos conjuntos ao procedimento. O conjunto de objetivos principais, cuja tarefa é armazenar os objetivos assinalados as linhas principais, e o conjunto chamado de objetivos de pontos de *fanout* que contém os objetivos setados às arestas com *fanout* não unitário. Cada objetivo que atinge um ponto de *fanout* faz parar o *backtrace* enquanto espera que todos os objetivos correntes tenham sido gerados. Se o ponto de *fanout* ***p*** não é atingível por uma linha que contém uma falha e ambos os valores $n_0(p)$ e $n_1(p)$ são diferentes de zero, então, um conflito é encontrado. Neste caso, será assinalado ao ponto *p* o valor lógico 0 se o número de vezes em que é necessário setar o valor 0 for maior que para setar o valor 1 ($n_0(p) \geq n_1(p)$).Caso contrário, o valor 1 é setado se $n_0(p) < n_1(p)$. Além disso, depois de assinalado o valor lógico a *p* este será implicado. Quando os objetivos em pontos de fanout são não conflitantes, i.e., ou $n_0(p) = 0$ ou $n_1(p) = 0$, o procedimento continua até que sejam atingidas as linhas principais. Desta forma, se todos os objetivos atingem as linhas principais, o **conjunto de objetivos primários (ou iniciais)** e o **conjunto de objetivos correntes** encontram-se vazios e o procedimento termina com sucesso.

No algoritmo PODEM, o assinalamento de um valor lógico e o procedimento de *backtracking* acontece apenas nas entradas primárias. De outra forma, no FAN, o *backtracking* pode ocorrer nos pontos de *fanout* ou nas linhas principais do circuito antes que sejam atingidas as entradas primárias.

```
Início
    se Implica_e_verifica () = FRACASSO então retorna FRACASSO
    se (erro foi propagado até alguma PO e todas as linhas limite são
    justificadas ) então
      Início
          Justifica todas as linhas principais não justificadas
          retorna SUCESSO
      fim
    se (erro não foi propagado até alguma PO e fronteira-D = ∅)
    então retorna FRACASSO
    /* inicializa objetivos */
    adiciona cada linha limite não justificada ao conjunto de objetivos
    seleciona uma porta (G) do conjunto fronteira-D
    c= valor controlante de G
    para cada entrada (j) de G com valor X
        adiciona (j, c) a conjunto de objetivos
    /* backtrace múltiplo*/
    (i ,v_i) = Mbakctrace(objetivos)
    atribui (i,v_i)
    se FAN() = SUCCESS então retorna SUCESSO
    atribui( j , v̄_j ) /* valor negado da decisão */
    se FAN() = SUCESSO então retorna SUCESSO
    atribui( i , X )
    retorna FRACASSO
fim
```

FIGURA 2.19 - Pseudocódigo do algoritmo FAN.

O exemplo de circuito apresentado na figura 2.20 ilustra como o algoritmo se comporta para uma falha do tipo s-a-1 na aresta **h**. Para ativar a falha em questão a função objetivo identifica que o valor lógico 0 deve ser assinalado à aresta **h** (*objetivo* (**h**,0)). Inicialmente, a função *implica*(**h**,**D**) gera o *objetivo*(**h**,0) (figura 2.6).

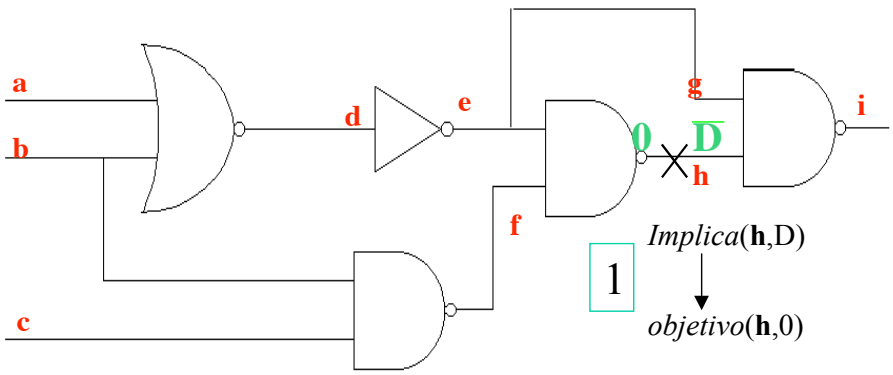


FIGURA 2.20 – A função *implica*(**h**,**D**) gera o *objetivo*(**h**,0).

A tabela 2.6 apresenta o conjunto de novos objetivos gerados como resultado das implicações nas direção das saídas e entradas primárias. Ao implicar o valor lógico 1 à aresta **g**, *implica*(**g**,1), o objetivo *objetivo*(**g**,1) é gerado. O processo de implicação e

geração de novos objetivos pode satisfazer objetivos decorrentes de outras implicações e assim, retirá-los da lista correntes. Na tabela 2.6, ao implicar o valor lógico 0 à aresta **d**, *implica*(**d**,0) e gerar o *objetivo*(**d**,0), os objetivos *objetivo*(**g**,0) e *objetivo*(**e**,0) são satisfeitos e retirados do conjunto de objetivos. Outra implicação *implica*(**h**,0) satisfaz o *objetivo*(**h**,0). Por fim, a falha é propagada até a saída primária, pois, *implica*(**i**, $\overline{D}$) resulta em setar o valor $\overline{D}$ à aresta **i**.

TABELA 2.6 – Conjunto de novos objetivos gerados por implicações.

Implicações	Objetivos gerados
h = D	*objetivo* (**h**,0)
g = 1	*objetivo* (**g**,1)
f = 1	*objetivo* (**h**,1)
e = 1	*objetivo* (**e**,1)
d = 0	*objetivo* (**d**,0)
	retira *objetivo* (**g**,0) da lista
	retira *objetivo* (**e**,1) da lista
h = 0	retira *objetivo* (**h**,0) da lista
i = $\overline{D}$	-

O procedimento *backtrace* múltiplo contribui para satisfazer o conjunto de objetivos simultaneamente. Assim, neste exemplo o múltiplo *backtrace* deve setar ao mesmo tempo *implica*(**d**,1) e *implica*(**f**,0). Este procedimente atinge a linha principal **b**. Estas implicações resultam na retirada dos seguintes objetivos das linhas principais: *objetivo*(**d**,0) e *objetivo*(**f**,0). Então a lista de objetivos está vazia e o erro encontra-se na saída primária. O resultado é que. O algoritmo FAN obteve **sucesso.** Neste exemplo o múltiplo bakctrace não atinge pontos de fanout antes de encontrar as linhas principais. Assim, não é necessária a avaliação para conflitos lógicos.

2.4 Análise Comparativa dos algoritmos para a geração de testes.

Os algoritmos descritos nas seções 2.3.1 à 2.3.3, algoritmo-D, PODEM e FAN, representam os algoritmos clássicos na geração de vetores de teste. Assume-se que inúmeras técnicas de geração de vetores de teste abordam características de cada um dos algoritmos. Estes algoritmos abrangem rotinas básicas, como estratégias de retrocesso, árvore de decisões, implicações de valores, determinação de inconsistências. A estratégia de retrocesso é utilizada para tratar problemas de inconsistências de valores em circuitos que contém portas com *fanout* não unitário e que apresentam caminhos reconvergentes. Outra rotina básica, a implicação, verifica a consistência de valores lógicos na direção das saídas primárias ou então na direção das entradas primárias. Todas estas estratégias buscam resolver o problema de justificação de linhas. A

complexidade derivada da busca de soluções de problemas de justificação de linha está centrada no número de ocorrências de inconsistências que podem ser reduzidas ou não. Então, pode-se concluir que para tratar o problema deve-se reduzir as chances de acontecer um retrocesso.

O algoritmo clássico que utiliza todas as propriedades descritas como básicas é o algoritmo-D. Entretanto, um número elevado de reconvergências pode comprometer o tempo de execução do algoritmo. Assim, encontrar os vetores de teste pode ser possível somente em tempo exponencial. Alternativamente, o algoritmo PODEM implementa uma técnica capaz de reduzir o número de ocorrências de retrocessos (*backtrackings*), utilizando a assertiva de que o retrocesso ocorre somente nas entradas primárias do circuito. A forma que o PODEM encontra para assinalar às entradas primárias um valor lógico sem que sejam implicados em cada ponto do circuito é incluir a rotina de *backtrace*. A rotina calcula os valores nas arestas do circuito até o momento que uma entrada primária é encontrada. A seguir, é disparado o processo de implicação, assinalando valores às arestas até que um objetivo seja satisfeito. A fim de satisfazer um objetivo é necessário excitar uma falha. Além disso, a falha é propagada até que uma saída primária seja atingida. Segundo [GOE81], limitar a ocorrência de *backtrackings* às entradas primárias acelera o processo de busca dos vetores de teste. Entretanto, um único objetivo de cada vez é satisfeito. Desta forma, ao satisfazer um objetivo após o outro pode-se tentar assinalar um valor lógico a uma aresta que tem um valor assinalado e assim gerar um conflito, i.é., satisfazer um objetivo pode anular um objetivo já satisfeito.

Assim surge o FAN como forma de reduzir o número de conflitos durante o *backtrace*. Para tanto, o FAN focaliza dois pontos. Um deles é observar os pontos de *fanout* pois, podem ocorrer conflitos lógicos durante o *bakctrace*. Para tratar este problema um novo procedimento foi proposto, o múltiplo *backtrace*, que procura satisfazer mais de um objetivo por vez. O FAN reutiliza as funções do algoritmo-D, como implicação na direção das entradas primárias e na direção das saídas primárias. Porém, identifica pontos de *fanout* (conjunto que contém os objetivos setados às arestas com *fanout* não unitário) assim que os encontra a fim de diminuir a ocorrência de *backtrackings*. Em último caso, quando o *backtracking* ocorre, através do cálculo de uma 3-upla, que identifica o conflito lógico nos pontos de *fanout*, pode-se simplesmente trocar o valor lógico na aresta e implicá-lo sem que seja necessário atingir as entradas primárias. Assim, o algoritmo pode reduzir ainda mais o número de ocorrência de *backtrackings*. Existem outras técnicas como *recursive learning* [SIL99], *backtracking* não cronológico [SIL94] que não foram abordadas neste capítulo pois, o objetivo deste capítulo é fornecer os subsídios necessários para o estudo do algoritmo TrueD-F. Este algoritmo incorpora técnicas do algoritmo PODEM e do algoritmo FAN. O TrueD-F é um algoritmo que, além da informação lógica, agrega informações de tempo. Desta forma, busca-se identificar pontos de aceleração no algoritmo, inicialmente desenvolvido para tratar portas simples, para estudar sua aplicação para circuitos que contém portas complexas.

A característica que fomentou o estudo dos três algoritmos descritos anteriormente foi a intenção de, ao detalhar e reconhecer suas propriedades, seja possível identificá-las no algoritmo TrueD-F, para acelerar o processo da geração de vetores de teste. Todos os algoritmos englobam as rotinas básicas de geração de vetores de teste, encontrar a solução significa resolver problemas de implicação de linhas. O principal foco de problemas neste tratamento é o numero de ocorrências de

backtrackings, o que exige o estudo de técnicas que reduzem as chances destes ocorrerem.

O capítulo 3 aborda a análise de *timing* funcional e fornece subsídios para a aplicação de técnicas de aceleração para circuitos que contém portas complexas.

3 ANÁLISE DE *TIMING* FUNCIONAL BASEADA EM ATPG

As primeiras ferramentas para análise de *timing* utilizavam as informações topológicas do circuito para determinar uma estimativa do seu atraso máximo. Assim, o caminho de maior atraso, declarado como **caminho crítico**, assume que o seu atraso é o **atraso crítico** do circuito. Entretanto, assumir este modelo de cálculo do atraso, significa obter uma estimativa pessimista do atraso do circuito, pois é importante considerar que alguns caminhos nunca são ativados, de modo que seus atrasos nunca ocorrem. Estes caminhos são considerados **não sensibilizáveis** ou também denominados **falsos**. Tão logo foi percebida essa limitação da análise de *timing*, as pesquisas passaram a focalizar a incorporação de critérios de teste de **sensibilização de caminhos**.

Os estudos na área de análise de *timing,* sob um ponto de vista mais abrangente, dividem-se em dois grandes grupos. A análise de *timing* que não considera a sensibilização chamada de **análise de *timing* estática** (*static timing analysis*), ou ainda **análise de *timing* topológica** (*Topological Timing Analysis* – TTA) [DEV94], e a **análise de *timing* funcional** (*Functional Timing Analysis* – FTA) [KUK97][KUK98] que focaliza a incorporação de critérios de teste de sensibilização de caminhos.

3.1 Algoritmos de Análise de *Timing* Funcional

Os algoritmos de análise de *timing* funcional modelam o problema de sensibilização de caminhos considerando não apenas a topologia do circuito, como também as relações temporais e funcionais entre os elementos do circuito. Os critérios de sensibilização representam uma parte dos algoritmos de FTA, a outra parte, concentra-se no próprio algoritmo de cálculo do atraso.

Uma nova taxonomia proposta para a classificação dos algoritmos de computação do atraso em [GÜN2000] permite classificar os algoritmos de análise de *timing* funcional sob outros dois aspectos, além dos critérios de sensibilização. O primeiro aspecto diz respeito ao número de caminhos que podem ser verificados simultaneamente: **sensibilização individual de caminhos** ou **sensibilização concorrente de caminhos**. O outro aspecto refere-se ao método utilizado para determinar se as condições de sensibilização são satisfeitas ou não. Neste caso, destacam-se os algoritmos baseados na **geração automática de testes** (*Automatic Teste Generation-**ATPG -based***) ou **baseados em solvabilidade** (*Satisfiability-**SAT-based***). Além disso, as duas classificações podem ser agrupadas, pois as condições de sensibilização podem ser testadas para um único caminho ou para conjuntos de caminhos simultaneamente. Faz-se necessária, ainda, uma observação final sobre

terminologia. De acordo com [GÜN2000]: "Na literatura, a designação "baseada em ATPG" está associada aos algoritmos baseados em ATPG que usam sensibilização concorrente. Isto ocorre por razões históricas, pois na realidade a sensibilização individual, que é derivada do algoritmo D, surgiu antes da sensibilização concorrente. De modo similar, a designação "baseado em SAT" assume implicitamente a sensibilização concorrente."

Os primeiros algoritmos de FTA testavam a sensibilização dos caminhos, um após o outro, usando adaptações do algoritmo D [ROT66]. Este era o caso dos trabalhos apresentados em [BEN90], [BRA88], [CHE91] e [CHE93]. Nos dois últimos, os critérios de sensibilização usados eram dependentes de atraso. Esta característica de testar um caminho por vez corresponde ao conceito de **sensibilização individual de caminhos** existente em ATPG. Porém, logo alguns autores reconheceram ser bastante comum a ocorrência de circuitos com centenas de milhares de falsos caminhos com atraso maior do que o atraso do caminho crítico (verdadeiro) [KEU91][DEV94][LAM94], de modo que o teste de sensibilização caminho por caminho foi reconhecido ser de uso limitado, na prática. Por outro lado, surgiram diversas propostas de algoritmos de FTA capazes de tratar simultaneamente a sensibilização de conjuntos de caminhos, habilidade esta que em ATPG é conhecida por **sensibilização concorrente de caminhos**. Dentre tais algoritmos merecem destaque o **procedimento de geração de teste temporal** (*timed-test generation procedure*), de Devadas et al. [DEV91] [DEV93a] e o trabalho de Silva e Sakallah [SIL94] [SIL94a]. Existe ainda uma abordagem **mista**, proposta em [CHA93], na qual um conjunto de potenciais caminhos críticos é identificado usando sensibilização concorrente de caminhos. Após, sensibilização individual é aplicada ao conjunto.

A seguir, serão discutidas as seguintes classes de algoritmos: baseados em ATPG com sensibilização individual e baseados em ATPG com sensibilização concorrente. A discussão se utilizará de um exemplo para cada um dos tipos de algoritmos.

3.1.1 Algoritmos Baseados em ATPG com Sensibilização Individual

Os algoritmos baseados em ATPG com sensibilização individual testam as condições de sensibilização para um único caminho por vez. Um algoritmo de análise de *timing* baseado em caminhos opera da seguinte maneira: traça o caminho de maior atraso topológico e verifica se as condições de sensibilização são satisfeitas ou não. Caso sejam satisfeitas (i.e., o caminho é sensibilizável), seu atraso será assumido como sendo o atraso (crítico) do circuito. Caso não sejam satisfeitas (i.e., o caminho não é sensibilizável), o próximo caminho de maior atraso é traçado e sua sensibilização é testada. Este procedimento de traçar o próximo caminho de maior atraso e testar a sensibilização continua até que um caminho sensibilizável seja encontrado. Isto significa que o atraso do circuito corresponderá ao atraso do primeiro caminho sensibilizável encontrado.

Ao traçar um único caminho por vez, os algoritmos de sensibilização individual necessitam utilizar um procedimento capaz de enumerar caminhos segundo a ordem não decrescente dos atrasos. Os algoritmos de enumeração de caminhos mais eficientes apresentam complexidade de execução $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$, com $\boldsymbol{n}$ igual ao número de nodos do grafo [PIN98]. O procedimento pode ser ligeiramente acelerado se o teste de sensibilização for sendo realizado à medida que o caminho for sendo traçado. Neste

caso, os passos para o teste da sensibilização são os mesmos do algoritmo D: **implicação** e **justificação** [ROT66]. Para cada nova porta que é acrescida ao caminho que está sendo traçado, valores lógicos são assinalados às entradas laterais. Então, tais valores são propagados para trás, em direção às entradas primárias, e para frente, em direção às saídas primárias. No caso de condições de sensibilização dependentes de atraso, os tempos de estabilização dos valores lógicos são também considerados. Os nós não avaliados permanecem com valores desconhecidos (don't cares). A figura 3.1 ilustra este procedimento.

Como pode existir mais de um conjunto de valores lógicos capazes de justificar um conjunto de condições de propagação para uma porta, os conjuntos de valores possíveis são armazenados numa lista. No caso de ocorrer alguma inconsistência quando da propagação das condições para as demais portas do caminho, os últimos valores assinalados devem ser desfeitos. Então, um novo conjunto de valores deve ser escolhido da lista, e propagado para o resto do circuito. Se, ao final do processo, for encontrado um assinalamento de valores lógicos que satisfaça às condições de sensibilização, então o caminho será declarado sensibilizável (com respeito ao critério de sensibilização e aos modelos de atraso adotados). Por outro lado, um caminho não pode ser declarado não-sensibilizável até que todas as possibilidades de assinalamentos de valores lógicos tenham sido testadas.

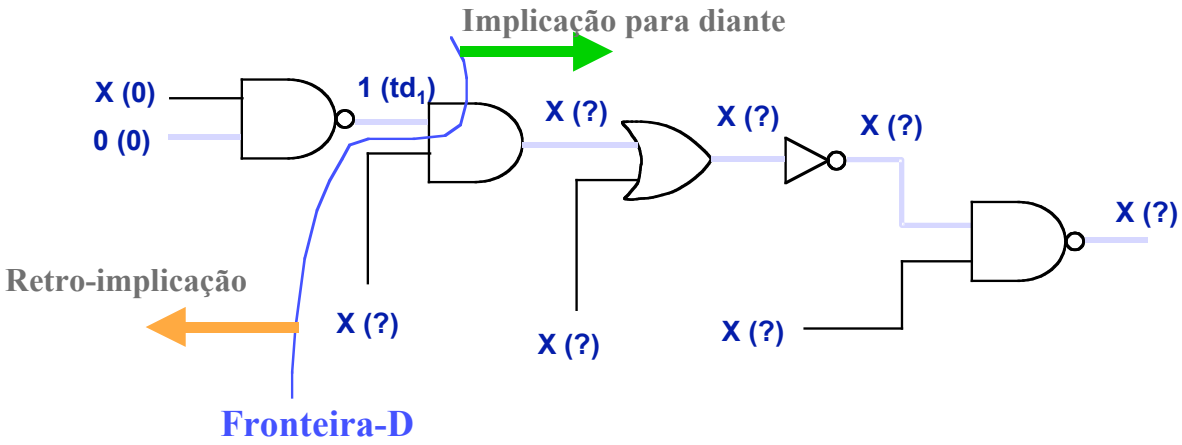


FIGURA 3.1 – Procedimento de sensibilização individual de caminho.

A fim de reduzir o tempo de execução, alguns algoritmos baseados em ATPG com sensibilização individual (e.g., [DU89] e [CHE93]) não realizam o passo de justificação, uma vez que seus autores alegam que a maioria dos caminhos falsos podem ser detectados no passo de implicação. Entretanto, de acordo com Peset Llopis, o passo de implicação não é capaz de detectar todos os caminhos falsos [PES94] e assim, quando a justificação não é realizada, um caminho não-sensibilizável pode ser declarado sensibilizável, resultando numa sobrestimativa do atraso do circuito.

Conforme já mencionado, a fase de **enumeração de caminhos** é extremamente importante para qualquer algoritmo de sensibilização individual. O problema de traçar caminhos em circuitos combinacionais tem sido estudado desde o início dos anos 80, quando vários algoritmos de enumeração foram propostos. Alguns deles eram implementações diretas dos procedimentos clássicos para percorrer grafos, a busca "primeiro em largura" (*Breadth-First Search BFS*) e a busca "primeiro em profundidade" (*Depth-First Search DFS*). Exemplos são encontrados em [YEN88] e

[OUS85]. Entretanto, tais procedimentos não eram capazes de traçar os caminhos de forma ordenada, pois não consideravam informações pré-anotadas no grafo. Assim, para poder ser aplicada, havia a necessidade de armazenar os caminhos numa lista, que deveria ser posteriormente ordenada. Tal procedimento logo se mostrou infactível em função do grande número de caminhos que um circuito pode apresentar. Para solucionar este problema, Yen e Du propuseram em [YEN89] (e com mais detalhes em [YEN91]) o uso do procedimento de busca "primeiro o melhor" (*Best-First Search*), também conhecido por A* (A-star) [WIS84], o qual foi extensivamente utilizado pelos algoritmos de sensibilização individual que sucederam.

Na enumeração explícita, à medida que um caminho vai sendo expandido, sua sensibilização vai sendo testada mediante a aplicação de um conjunto de testes. Estes testes, definidos pelo chamado **critério de sensibilização**, geram um conjunto de restrições no valor e no tempo de estabilização dos sinais ao longo do circuito. Caso estes testes conduzam à determinação de um vetor de entrada capaz de sensibilizar o caminho, então o caminho é declarado como sensibilizável para as condições de atraso consideradas. Caso contrário, o caminho é declarado não sensibilizável para as condições de atraso consideradas. (Note-se que o conceito de sensibilização é dependente das condições de sensibilização e das condições de atrasos dos elementos do circuito.) Exemplos de abordagens que utilizam enumeração explícita são [CHE91] (e [CHE93]), [MCG91a], e [HSU98].

Como o objetivo é a determinação do caminho sensibilizável de maior atraso, a enumeração explícita tem que ser capaz de enumerar os caminhos em ordem não decrescente dos atrasos. Além disso, caso os primeiros caminhos não sejam sensibilizáveis, o número de caminhos que devem ser listados é potencialmente grande, de modo que além de ter que ser capaz de listar caminhos em ordem não decrescente de atrasos, é altamente desejável que o algoritmo apresente um tempo de execução proporcional ao número de caminhos listados.

Em [GUN98] é apresentado um método de enumeração explícita que considera atrasos de subida e de descida distintos para as portas. Neste artigo, e também em [GUN98a], são comparados os resultados dos atrasos de caminhos com três casos em que apenas um atraso por porta é utilizado. Já em [GUN99] é discutido o compromisso entre tempo de execução e memória utilizada na implementação de algoritmos de enumeração explícita de caminhos para FTA.

Os algoritmos apresentados nesses trabalhos citados anteriormente se baseiam no procedimento conhecido como **best-first search** [YEN89], o qual permite a exploração do espaço de busca (i.e., o conjunto de caminhos a serem traçados) de forma ordenada e com complexidade **O**(nlogn) [MCG91a].

O procedimento best-first search apresenta três principais passos: criação de um grafo acíclico direto (*Direct Acyclic Graph* – DAG, em inglês), pré-processamento do grafo e enumeração dos caminhos propriamente dita. No DAG criado no primeiro passo, cada porta CMOS é representada por um nodo e cada conexão (fio) é representada por uma aresta. Nodos especiais (dummy), com atraso zero, são utilizados para representar as entradas primárias e as saídas primárias. O grafo é polarizado por meio de um nodo fonte **s** (source) e de um nodo terminal **t**, ambos com atraso zero. Cada nodo de entrada é conectado ao nodo **s** por meio de uma aresta especial (dummy) e cada nodo de saída é conectado ao nodo **t** também por meio de uma aresta especial. Tendo representado o circuito combinacional por um DAG polarizado, qualquer caminho do circuito assume a forma $P = (v_0, v_1, v_2, v_3, ...v_n, v_{n+1})$, onde v_n é um nodo do grafo. Se $v_0 = \mathbf{s}$ e $v_{n+1} = \mathbf{t}$, então P é dito ser um caminho **completo**. Se $v_0 \neq \mathbf{s}$ ou $v_{n+1} \neq$

t , então P é dito ser um caminho **parcial**. A figura 3.2 mostra um exemplo de DAG.

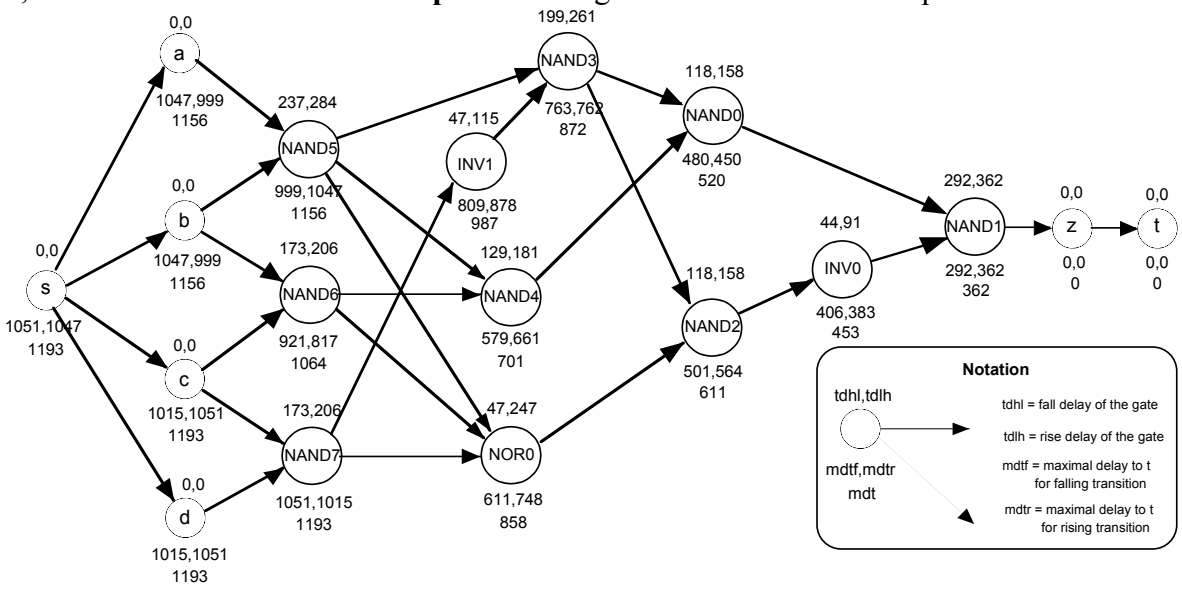


FIGURA 3.2 - Exemplo de DAG.

Foram implementados dois métodos de cálculo de atraso de caminhos, cada qual considerando: um único atraso por porta (*single gate delay* - **sgd**) e um par de atrasos por porta (*single pair gate delay* - **spgd**), sendo um atraso para a descida e outro para a subida do sinal na saída. O método **sgd** computa o atraso do caminho somando o atraso único anotado em cada porta, de modo que nenhuma consideração é feita sobre a polaridade dos sinais. Já no método **spgd**, cada caminho topológico é encarado como dois caminhos lógicos distintos, considerando-se o tipo de transição que ocorre na entrada primária. Desta forma, o atraso do caminho parcial P = (Q, v_{n+1}), com Q = (**s**, v_1, v_2, v_3, ...v_n) é calculado por:

$$d(P) = d(Q) + tdhl(v_{n+1}) \quad (1)$$

se na saída de v_{n+1} estiver ocorrendo uma transição de descida e

$$d(P) = d(Q) + tdlh(v_{n+1}) \quad (2)$$

se na saída de v_{n+1} estiver ocorrendo uma transição de subida. tdhl e tdlh são o atraso de descida e de subida da porta, respectivamente. A consideração das polaridades das transições ao longo de cada caminho equivale a uma forma primitiva de considerar as relações Booleanas entre as portas do circuito.

A fim de permitir a enumeração ordenada dos caminhos, o procedimento best-first search passa por uma fase de pre-processamento na qual o **máximo atraso até o nodo t** (*maximal delay to node* **t** - **mdt**) é calculado para cada nodo e armazenado na estrutura de dados. Entretanto, no caso do método spgd, dois valores de mdt devem ser calculados: **mdtf** e **mdtr**. Para um dado nodo v, **mdtf**(v) e **mdtr**(v) fornece o máximo dos atrasos dentre todos os caminhos lógicos parciais que iniciam no nodo v e terminam no nodo **t**, para uma transição de descida e de subida na saída de v, respectivamente, e são calculados por:

$$mdtf(v) = tdhl(v) + \max\{mdtr(u_i)\} \quad (3)$$

$$mdtr(v) = tdlh(v) + \max\{mdtf(w_i)\} \quad (4)$$

com $u_i \in$ **adjf**(v) (lista de sucessores de v para uma transição de descida) e $w_i \in$

adjr(v) (lista de sucessores de v para uma transição de subida).

O procedimento para calcular os mdts de todos os nodos do grafo é baseado no algoritmo conhecido como *topological sort* [COR90] e inicia pelo nodo **t**, seguindo em direção ao nodo **s**. O mdtf e o mdtr de um nodo v somente será calculado depois que todos os seus sucessores tiverem sido computados. Este procedimento termina quando o mdtf e o mdtr do nodo **s** tiver sido calculado. Ao mesmo tempo em que o mdtf e o mdtr de um nodo v são calculados, suas duas listas de nodos sucessores, **adjf**(v) e **adjr**(v) são ordenadas segundo uma ordem não decrescente dos mdtfs e mdtrs dos sucessores, respectivamente.

O valor mdt evita que a procura siga por ramos que não resultarão em caminhos de atraso maior do que o menor atraso dentre os caminhos já descobertos. Este valor também permite manter os caminhos lógicos parciais candidatos ordenados pelos valores de suas esperanças. A esperança de um caminho lógico parcial $P = (Q, v_{n+1})$, obtido pela anexação do nodo v_{n+1} ao caminho lógico parcial $Q = (\mathbf{s}, v_1, v_2, v_3, ...v_n)$, corresponde ao máximo atraso entre todos os caminhos completos que têm P como prefixo [BEN87] e é calculado por:

$$e(P) = d(Q) + mdtf(v_{n+1}) \quad (5)$$

se na saída de v_{n+1} está ocorrendo uma transição de descida e

$$e(P) = d(Q) + mdtr(v_{n+1}) \quad (6)$$

se na saída de v_{n+1} está ocorrendo uma transição de subida.

A fase de enumeração pode ainda ser sub-dividida em **inicialização** e **extensão de caminhos**. Na inicialização, uma lista para o armazenamento de caminhos parciais (caminhos lógicos parciais, no caso do método spgd) é inicializada pela inserção dos primeiros **fo(s)** caminhos parciais (2**fo(s)** caminhos lógicos parciais, no caso do spgd) cada caminho constituído somente de dois nodos: **s** e um dos sucessores de **s**, obtido a partir **adj**(s). **fo(s)** é o grau da saída (outdegree) de **s**, i.e., o número de entradas primárias. No caso do método spgd, os nodos sucessores de s são obtidos a partir das listas **adjf**(s) e **adjr**(s) (ao invés de **adj**(s)), tomando sempre o maior elemento entre ambas as listas.

Foram implementadas duas versões do algoritmo best-first search, as quais se distinguem pela estrutura de dados utilizada para armazenar os caminhos parciais: uma usa uma lista linear dinâmica enquanto que a outra usa uma árvore binária. Durante o processo de enumeração de caminhos, a lista de caminhos parciais é mantida ordenada usando o valor da esperança. No caso da árvore binária, um procedimento do tipo *heapsort* [COR90] é utilizado. Assim, a principal característica desta árvore binária ("heapified") é que o caminho parcial de maior esperança está armazenado na raiz. A duas versões serão referenciadas por **linear** e **tree**, respectivamente.

Uma segunda lista linear chamada **list_of_paths** é usada para armazenar os caminhos completos (já estendidos). Esta lista não requer nenhum tipo de esquema especial de inserção uma vez que o procedimento best-first assegura que os caminhos vão sendo traçados em ordem não decrescente de atrasos. A figura 3.3 ilustra as estruturas para armazenamento de caminhos (completos e parciais) existentes.

Após a inicialização da lista de caminhos parciais, cada caminho (lógico) parcial é estendido, seguindo a ordem não decrescente da esperança. Um caminho (lógico) parcial torna-se um caminho (lógico) completo quando o processo de extensão atinge o nodo **t**. Um vez que o caminho foi totalmente estendido, ele é removido para list_of_paths e o próximo caminho (lógico) parcial inicia a ser estendido. Quando a lista tree é usada ao invés da lista linear, o próximo caminho a ser estendido já se encontrará

na raiz da árvore, pois esta é “heapificada” toda vez que um caminho completo é removido para list_of_paths.

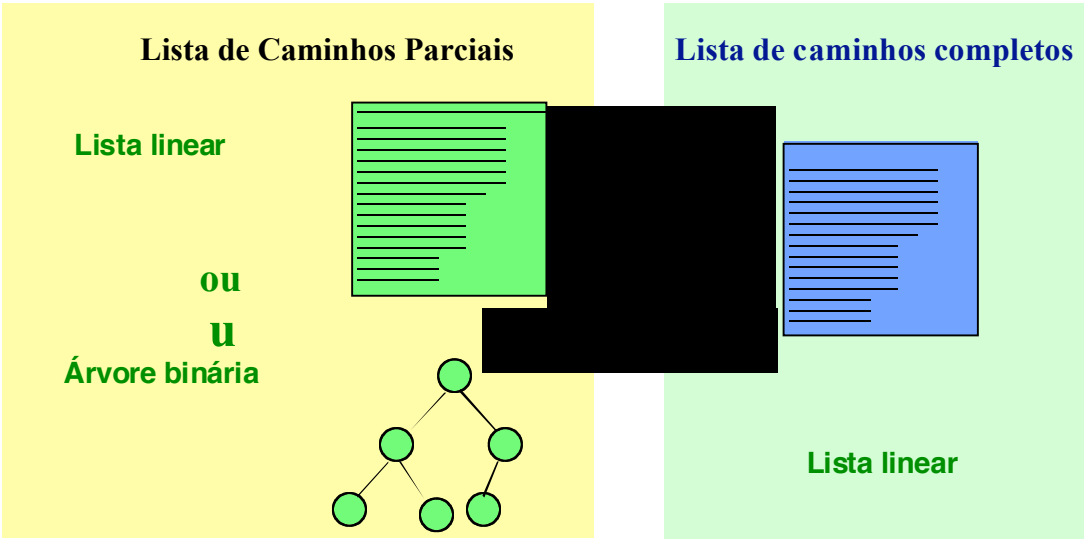


FIGURA 3.3 - Estruturas para armazenamento de caminhos.

Cada posição da lista de caminhos parciais armazena os nodos do grafo correspondentes ao caminho lógico parcial, o atraso do caminho, d(P), sua esperança, e(P), o número de nodos do caminho, level(P) e o tipo de transição considerada na entrada primária do caminho type(P). No caso do método sgd, os dois últimos parâmetros não são necessários.

A função **extend_and_insert** é a responsável por estender um caminho (lógico) parcial. Considere que o caminho lógico parcial P = (Q, v_{n+1}), com Q = (**s**, v_1, v_2, v_3, ...v_n), apresenta uma transição na entrada primária tal que na saída da porta v_{n+1} está ocorrendo uma transição de descida. Assuma também que **adjf**(v_{n+1}) = { u_0, u_1, u_2, u_3, ...} é a lista ordenada de sucessores de v_{n+1}para uma transição de descida em sua saída. Estendendo P através de u_0 origina um caminho P_0, com esperança igual àquela de P (isso porque u_0 é o primeiro sucessor de v_n). Assim, P_0 substitui P. Entretanto, estendendo P através do nodo u_1 origina um novo caminho parcial P_1 com esperança:

$$e(P_1) = d(P) + mdtf(u_1) \quad (7)$$

De modo similar, estendendo P através do nodo u_2 origina um novo caminho P_2 com esperança:

$$e(P_2) = d(P) + mdtf(u_2) \quad (8)$$

e assim por diante. Antes de estender o caminho P através de u_0, a esperança de cada novo caminho parcial, exceto P_0, é calculada. A esperança irá guiar a inserção de cada novo caminho parcial na lista de caminhos parciais.

No caso da lista linear, o número total de caminhos **T** (**T** = caminhos (lógicos) completos + caminhos (lógico) parciais) é controlada de modo a nunca exceder **k**, onde **k** é o número total de caminhos (lógicos) a serem traçados, fornecido pelo usuário. (Essa limitação somente ocorre quando esse número é estabelecido no início do programa.). Desta forma, enquanto **T** < **k**, qualquer novo caminho (lógico) parcial é inserido na lista linear respeitando a ordem fornecida pela esperança. Entretanto, a partir do momento em que **T** = **k**, um novo caminho parcial somente é inserido se sua esperança for maior do que a esperança do caminho armazenado no final da lista (i.e.,

do caminho parcial de menor esperança da lista). E neste caso, o caminho (lógico) parcial armazenado na última posição da lista é descartado (deletado). No caso da versão tree, o número de caminhos parciais não é limitado.

O atraso de cada novo caminho parcial P_i que é inserido na lista de caminhos parciais será calculado ou pela equação (1) ou pela (2). É interessante notar que, para um caminho completo P, d(P) = esperance(P).

Uma vez que o caminho P foi totalmente estendido, a função **extend_and_insert** será novamente chamada para estender o próximo caminho (lógico) parcial. A fase de enumeração termina ou quando o k-ésimo caminho for estendido ou quando algum critério de parada for (por exemplo, um caminho sensibilizável for encontrado). A figura 3.4 mostra o conteúdo da lista de caminhos parciais para o circuito representado pelo DAG da figura 3.2 (no caso, uma lista linear), quando o primeiro caminho da lista está sendo estendido.

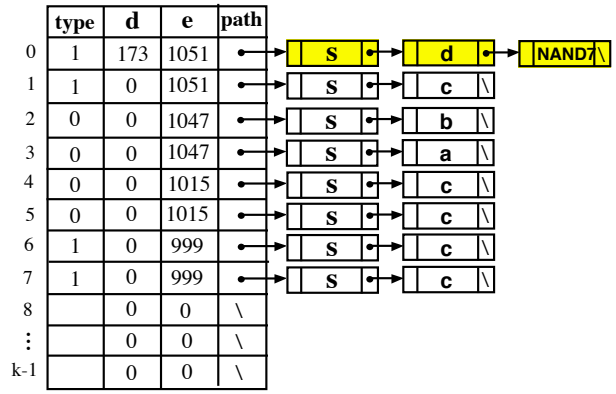


FIGURA 3.4 - Lista linear para armazenamento de caminhos lógicos parciais (método spgd), inicializada para o circuito cujo DAG é mostrado na figura 3.2.

Os algoritmos descritos foram implementados em linguagem C e ambas versões permitem o uso do método spg e do método spgd para o cálculo do atraso de caminhos.

A avaliação dos algoritmos de enumeração explícita discutidos foi realizada utilizando-se os circuitos do banco de benchmarks ISCAS'85. Esses circuitos foram mapeados usando-se somente portas CMOS simples (não foram usadas SCCGs nem transmission gates). A tabela 3.1 mostra as complexidades dos circuitos e dos DAGs que os representam.

TABELA 3.1 - complexidade dos circuitos ISCAS'85.

Circuito	# de portas	# de redes	# de nodos	# de arestas
C432	182	222	231	448
C499	364	405	439	883
C880	529	589	617	987
C1355	604	645	679	1227
C1908	955	988	1015	1656
C2670	1605	1762	1828	2773
C3540	2307	2357	2381	3671
C5315	3249	3427	3552	5752
C6288	2672	2704	2738	5152
C7552	4556	4762	4871	7608

A complexidade do procedimento best-first é basicamente dominado pela inserção de caminhos parciais na lista de caminhos parciais. Assim, reduzindo esse tempo obtém-se significativa redução no tempo de execução total. Com efeito, as curvas *tempo de execução* x *número de caminhos traçados* mostrados pela figura 3.5 confirma as assertivas anteriores. Pode-se ver que a complexidade de tempo da versão linear é exponencial com relação a **k**, enquanto que a complexidade de tempo da versão tree é quase linear com relação a **k**. Como resultado, a variação no tempo de execução para o método spgd é significativment reduzida quando a versão tree do algoritmo é usada.

A figura 3.6 mostra as curvas *memória* x *número de caminhos traçados* para os mesmos casos. Pode-se notar que em termos de uso de memória os comportamentos se invertem: a versão linear tem comportamento linear em relação a **k** enquanto que a versão tree tem comportamento exponencial.

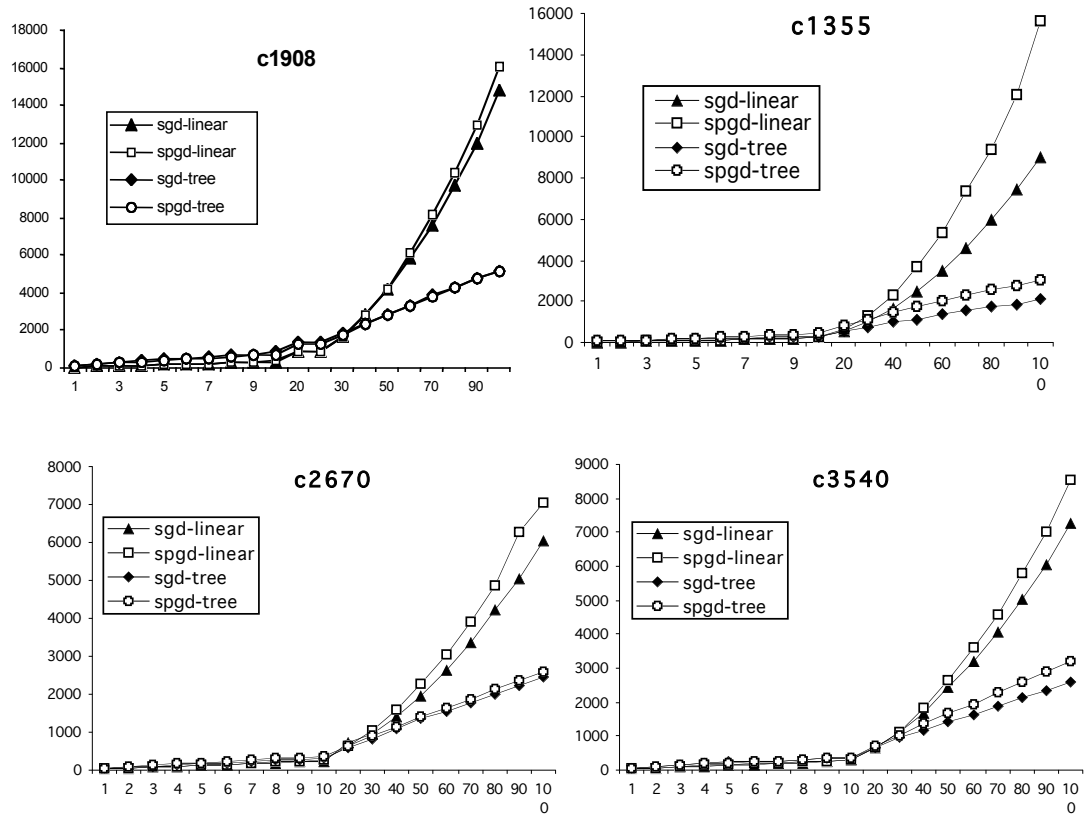


FIGURA 3.5 - curvas tempo de execução(ms) x número de caminhos traçados para alguns circuitos ISCAS85.

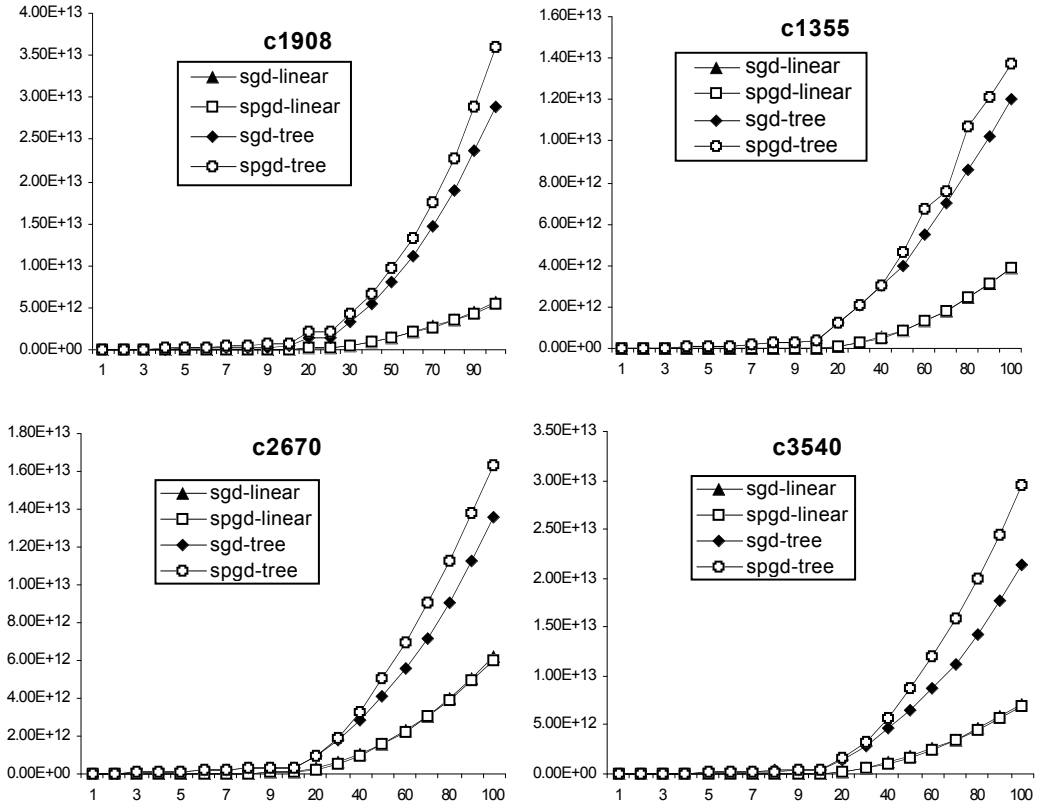


FIGURA 3.6 - curvas *memória (bytes) x número de caminhos traçados* para alguns circuitos ISCAS85.

Considerando-se as duas versões do algoritmo, pode-se observar que durante a enumeração de caminhos, em momentos distintos, cada implementação assume um comportamento exponencial. Observe que a implementação não considera ainda a sensibilização individual de caminhos. Assim, pode-se concluir que testar a sensibilização de caminhos durante o procedimento de enumeração explícita de caminhos é um processo inviável.

3.1.2 Algoritmos Baseados em ATPG com Sensibilização Concorrente

Os algoritmos baseados em ATPG com sensibilização concorrente de caminhos fazem uso de uma estratégia baseada na não enumeração de caminhos. Esta abordagem surgiu como forma de amenizar as dificuldades encontradas relacionadas à sensibilização individual de caminhos. Esta estratégia visa estabelecer intervalos de tempo a fim de, testar se as saídas primárias do circuito estão estáveis (ou em 0 ou em 1). A cada resposta positiva, um novo intervalo é submetido ao teste. Entretanto, se alguma saída primária não estiver estável, então o atraso do circuito pertence ao intervalo de tempo anterior ao teste. Assim, ao testar uma saída, o algoritmo estará considerando implícitamente um conjunto de caminhos que podem influenciar a saída em questão e estará, portanto, realizando sensibilização concorrente de caminhos. A estratégia utilizada visa à **enumeração de atrasos**.

O exemplo mais significativo de algoritmo baseado em ATPG com sensibilização concorrente é o **algoritmo TrueD-F**, desenvolvido por Devadas e

colaboradores [DEV93a]. Este algoritmo responde à seguinte pergunta: **o atraso do circuito é maior ou igual a δ?** O valor δ é inicializado com $T-\varepsilon_0$, onde T é o atraso topológico do circuito e ε_0 é um pequeno valor maior que zero. Partindo do atraso T, todos os caminhos de atraso ≥ δ são testados simultaneamente. O método empregado para responder à pergunta está baseado em simulação de cubos de entrada (*input cube simulation*). A técnica deriva de uma versão modificada de algoritmo PODEM [GOE81], denominada de **procedimento de geração de teste temporal** (*timed-test generation procedure*) [DEV93a]. Enquanto a resposta à pergunta for "não" para cada uma das saídas primárias, i.e., o atraso do circuito não é ≥ δ, o procedimento de geração de teste temporal é sucessivamente invocado para $\delta_i = T-\varepsilon_i$, com i=0,1,2,...., $\varepsilon_{i+1} > \varepsilon_i$. Se a resposta for "sim" para uma saída, então esta saída apresenta um atraso entre o valor corrente de δ (digamos $T-\varepsilon_k$) e o valor anterior ($T-\varepsilon_{k-1}$). Assim, $T-\varepsilon_{k-1}$ representa um limite superior seguro para o máximo atraso do circuito. Caso se deseje um valor mais preciso para o atraso, pode-se aplicar pesquisa binária sobre o intervalo [$T-\varepsilon_k$, $T-\varepsilon_{k-1}$].

Como o circuito pode estabilizar com o valor lógico 0 ou com o valor lógico 1, para cada valor δ_i, o procedimento de geração de teste temporal deve ser aplicado duas vezes a cada saída primária (exceto no caso em que a resposta for "sim" para o primeiro valor lógico testado). A figura 3.7 ilustra o procedimento básico do algoritmo TrueD-F.

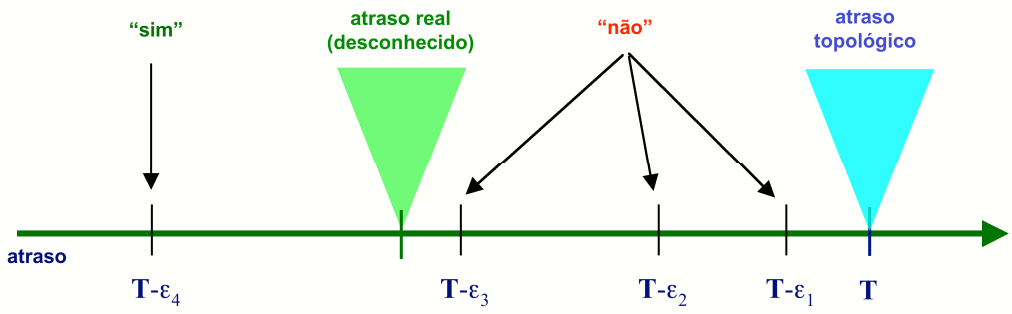


FIGURA 3.7 - Procedimento de geração de teste temporal aplicado a um circuito de uma única saída. [GÜN2000]

A fim de determinar se o atraso máximo (atraso crítico) numa saída primária do circuito é maior ou igual a δ_i para o valor lógico `lvalue`, o procedimento de geração de teste tenta justificar `lvalue` na saída do circuito no tempo maior ou igual a δ_i. Isto é feito por meio de simulação de cubos, de maneira similar à realizada pelo algoritmo PODEM. O algoritmo PODEM permite uma exploração sistemática e exaustiva do espaço das entradas, caso ele falhe em encontrar um cubo de entrada capaz de justificar `lvalue` na saída do circuito para o valor de atraso δ_i considerado, então a resposta à pergunta será "não" (para a saída testada e para o valor lógico `lvalue` considerado). Em outras palavras, o problema de determinar o atraso do circuito é transformado num problema de geração de teste para uma falha de colagem simples localizada na saída primária do circuito.

Como a geração do teste incorpora informações de *timing*, os **valores associados à uma aresta** (entrada/saída) **ou à uma porta lógica**: correspondem a um par de valores do tipo [valor lógico (ou 0 e ou 1)/ atraso (δ)]. A figura 3.8 identifica os campos associados as arestas e a uma porta.

Porta Lógica : value = `gate.timed_value.value`
delay = `gate.timed_value.lower`

Saída da Porta (**PO**) : `v.value`, `v.lower`, `v.upper`

Entrada da Porta : value = `fanin. timed_value.value`
upper = `fanin. timed_value.upper`
lower = `fanin. timed_value.lower`

FIGURA 3.8 - Campos da estrutura de dados associados à uma porta lógica e a suas arestas.

A seguir, a chamada de mais alta hierarquia do procedimento de geração de teste temporal (`timed_test`) é descrita pelo pseudocódigo que segue (figura 3.9). Tal qual no algoritmo PODEM puramente lógico, há uma lista de justificação `jlist` armazenando as linhas do circuito que precisam ser justificadas. Então, a lista contém o conjunto de portas lógicas as quais não foi possível ainda satisfazer os valores nas saídas primárias do circuito. Além disso, os valores nas entradas entradas primárias já foram determinados, porém não foram implicados (*forward implication*). Assim, é importante destacar que pertencer a `jlist` significa ter valor lógico determinado.

O procedimento inicia pela inserção de uma dada saída primária `po` em `jlist` com valor lógico `lvalue` (0 ou 1) e assumindo `delay` como sendo o limite inferior do atraso do circuito a ser testado (δ_i). Então, o procedimento `SEARCH_1` é chamado

```
timed_test(po,delay,lvalue)
{
  v.value = lvalue;
  v.lower = delay;
  v.upper = INFINITY;

  modify_jlist (po,v,jlist);
  backward schedule po;

  status = SEARCH_1(jlist);
  return(status);
```

FIGURA 3.9 – Pseudocódigo para a chamada de mais alta hierarquia do procedimento de geração de teste temporal.

As funções de busca são similares àquelas encontradas no algoritmo PODEM e estão descritas pelos pseudocódigos das figuras 3.10 e 3.11. A função `SEARCH_1` escolhe uma porta pertencente à `jlist` e chama a função `BACKTRACE` para, partindo da saída primária `po`, encontrar uma entrada primária cujo valor lógico ainda é desconhecido. A entrada primária identificada é inicialmente ajustada ao valor lógico 1 e então a função `IMPLY` é chamada. Esta, por sua vez, leva a cabo uma rodada de simulação de cubos para teste temporal considerando as condições de sensibilização do critério exato do modo flutuante (no PODEM original, que a partir daqui será referido

por PODEM lógico, a função IMPLY corresponde à simulação de cubos com três valores lógicos e sem informação de atrasos). A função de implicação pode levar a uma situação de conflito, o qual pode ser de duas naturezas distintas: lógica ou temporal. Se não ocorrer conflito, SEARCH_1 será chamada recursivamente. O procedimento termina com sucesso em SEARCH_1 se a lista de justificação tornar-se vazia. No caso de ocorrência de conflito em SEARCH_1, o algoritmo retrocede ao assinalamento de entrada primária mais recente, assinalando-lhe o valor lógico 0. Então, a função SEARCH_2, mostrada na figura 3.11, é chamada.

O procedimento BACKTRACE falha se suas funções forem incapazes de encontrar uma entrada primária livre de assinalamento ou se o espaço das entradas tiver sido completamente explorado sem sucesso em SEARCH_2. Ocorre conflito se não for possível atribuir a uma linha do circuito o valor lógico que lhe é requerido com o período de tempo necessário. Em caso de conflito ou falha, é necessário desfazer todos os assinalamentos originados do assinalamento de entradas que causou o conflito ou falha.

```
SEARCH_1(jlist)
     {
if(length of jlist is zero) return SUCCEED;

if(BACKTRACE(gate,value,delay,&pi,&pi_value)==FALSE)
  return(FAILED);

if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
{
     search_status = SEARCH_1(jlist);
     if(search_status == FAILED)
       {
          restore the state of the network;
          search_status = SEARCH_2(jlist,pi,1-pi_value);
     }
     }
     else
     {
          restore the state of the network;
          search_status = SEARCH_2(jlist,pi,1-pi_value);
     }
```

FIGURA 3.10 – Pseudocódigo para o primeiro procedimento de busca.

O procedimento de geração de teste temporal assume que cada porta do circuito (e também cada aresta) possui uma "variável lógico-temporal" formada por três campos: um limite inferior e um limite superior de atraso da porta (atrasos no contexto do circuito como um todo) e um valor lógico. Um passo de pré-processamento inicializa os campos de valores lógicos com o valor 2 (o qual indica que o valor lógico ainda não está determinado) e os limites inferior e superior de atraso com os mínimos e máximos atrasos topológicos computados a partir das entradas primárias. Na fase de geração de teste, ao serem assinalados valores lógicos conhecidos às entradas primárias, os limites inferior e superior de atraso vão sendo aproximados durante a simulação para diante, devido à sensibilização ou bloqueio dos caminhos. Os limites inferior e superior são também modificados pela retro-implicação, no momento em que novos valores são

inferidos em algumas portas, como decorrência dos valores lógicos e respectivos tempos requeridos para as saídas primárias.

```
SEARCH_2(jlist,pi,pi_value)
          {
             backtracks = backtracks + 1 ;
             if(backtracks>BACKTRACK_LIMIT)
    return(ABORTED);

             if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
             {
               search_status = SEARCH_1(jlist);
               if(search_status == FAILED)
               restore the state of the network;
             }
       else
       {
               search_status = FAILED;
               restore the state of the network;
             }
```

FIGURA 3.11: Pseudocódigo para o segundo procedimento de busca.

Outro elemento importante do procedimento de geração de teste temporal é a lista de justificação. Portas são incluídas a esta lista durante a retro-implicação e retiradas tanto durante implicação para diante quanto durante a retro-implicação. A qualquer tempo, a lista de justificação contém as portas cujos valores lógicos ou de atraso precisam ser justificados, o que só se concretiza pelo assinalamento de novas entradas primárias. O fato da lista de justificação ficar vazia significa que a busca foi concluída com sucesso. Então, a resposta para a questão "o atraso do circuito é maior ou igual a δ (quando o valor lógico lv é assinalado à saída considerada)?" é "sim". Por outro lado, caso o espaço de busca tiver sido completamente enumerado sem que a lista de justificação tenha sido esvaziada, a busca falhou: a resposta para a pergunta é "não". Uma terceira situação seria o caso em que a busca é abandonada devido ao número excessivo de retrocessos, quando então a pergunta permanece sem resposta.

É importante ressaltar que a variável lógico-temporal é usada para armazenar tanto os valores reais (lógicos e de atraso) quanto aqueles valores inferidos por meio da retro-implicação. A diferença entre esse dois casos reside no fato de que, no segundo caso a porta relacionada estará na lista de justificação. De fato, todas as portas cujos valores de entradas não produzem os valores constantes em sua variável lógico-temporal devem estar na lista de justificação. Por outro lado, qualquer porta cujas entradas produzem os valores constantes em sua variável lógico-temporal não deve estar na lista de justificação. Estas duas assertivas anteriores caracterizam de maneira precisa a lista de justificação.

Tendo apresentado as duas funções de mais alta hierarquia do procedimento, passemos a examinar a função `imply`, a qual é chamada dentro das funções `search1` e `search2`. A função `imply` é detalhada pelo pseudocódigo da figura 3.12. A entrada primária é assinalada com o valor lógico devido e o efeito deste assinalamento é propagado pelo circuito usando a função `forward_set`. Esta função, por sua vez, arrola o tratamento do fanout da entrada primária que foi modificada e então a função

`forward_imply` realiza uma simulação temporal digirida por eventos. `forward_imply` é seguida de uma retro-implicação (função `backward_imply`). Estas duas funções são chamadas de maneira iterativa até que os valores no circuito não mudem mais. Geralmente, o assinalamento de um valor lógico particular a uma porta durante a implicação para diante ou durante a retro-implicação exige a previsão de tratamento diante de todos os seus fanouts e a previsão de tratamento para trás de todos os seus fanins que estão na lista de justificação (forward e backward scheduling).

Os eventuais conflitos são detectados por ambas funções de implicação. Estes conflitos podem ser de natureza lógica (conflitos lógicos) ou de natureza temporal (conflitos temporais).

```
imply(pi,pi_value,jlist)
{
  v = pi.timed_value;
  v.value = pi_value;
  status = forward_set(pi,v,jlist);

  while(status==IMPLY_NORMAL)
  {
  status=forward_imply(jlist);
if(status!=IMPLY_CONFLICT)
status=backward_imply(jlist);
  }
return status;
```

FIGURA 3.12 – Pseudocódigo para o procedimento de implicação.

A chave para determinar se é possível ou não justificar um valor lógico numa saída do circuito para um dado tempo reside na adoção de um cálculo temporal de três valores que leva em consideração as condições de sensibilização do critério exato do modo flutuante. Considere uma porta E de duas entradas com atraso d. Cada uma das entradas desta porta i_i pode apresentar um valor lógico pertencente ao conjunto {0,1,2} com limites inferior e superior de atraso dados por l_i e u_i, respectivamente. O termo "atraso do sinal" será utilizado, ao invés de "tempo de estabilização" [CHE93], pois mesmo portas que apresentam valor lógico não assinalado (i.e., 2) em suas saídas, apresentam valores coerentes para os limites inferior e superior do atraso.

TABELA 3.2 – Cálculo temporal de três valores para uma porta E de duas entradas.

i_2	i_1	0	1	2
0	lv	0	0	0
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	$\min(u_1,u_2)+d$	u_2+d	u_2+d
1	lv	0	1	2
	l_o	l_1+d	$\max(l_1,l_2)+d$	l_1+d
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$
2	lv	0	2	2
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$

Os resultados para uma simulação de cubos usando o cálculo temporal para uma porta E de duas entradas e para uma porta OU de duas entradas são mostrados nas tabelas 3.2 e 3.3. Nestas tabelas, *lv* é o valor lógico na saída da porta, enquanto l_o e u_o representam os limites inferior e superior do atraso na saída da porta, respectivamente.

TABELA 3.3 - Cálculo temporal de três valores para uma porta OU de duas entradas.

i_2	i_1	0	1	2
0	*Lv*	0	1	2
	l_o	$\max(l_1,l_2)+d$	l_1+d	l_1+d
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$
1	*Lv*	1	1	1
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	u_2+d	$\min(u_1,u_2)+d$	u_2+d
2	*Lv*	2	1	2
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$

O fato dos algoritmos de FTA baseados em ATPG (com sensibilização concorrente) serem derivados dos próprios algoritmos de ATPG os torna bastante interessantes, pois que as inúmeras técnicas de aceleração já desenvolvidas para os últimos podem ser aplicadas quase que diretamente aos primeiros.

3.3 Aplicabilidade de Algoritmos de FTA em Circuitos que Contém Portas Complexas.

As primeiras técnicas de FTA realizavam sensibilização individual de caminhos utilizando variações do algoritmo D e critérios de sensibilização simplificados, buscando reduzir o tempo execução. Entretanto, logo a sensibilização individual se mostrou impraticável mesmo para circuitos de complexidade moderada, e a sensibilização concorrente tomou-lhe o lugar. Com a sensibilização concorrente as técnicas de FTA passaram a utilizar uma abordagem baseada em "enumeração de atraso", ao invés de uma abordagem baseada em "enumeração de caminhos". Além de não necessitar da fase de enumeração, a sensibilização concorrente também permite a adoção do critério de sensibilização exato do modo flutuante, o qual é o único capaz de fornecer o atraso exato sob o modo flutuante.

Entretanto, toda a teoria que embasa os métodos de teste de sensibilização de caminhos e também os próprios critérios de sensibilização foram desenvolvidos assumindo circuitos compostos por portas simples, isto é, portas E/NÃO-E, OU/NÃO-OU e inversores. Conseqüentemente, se um circuito combinacional que contenha portas mais complexas deve ser analisado, a ferramenta de FTA a ser utilizada deve estar apta não somente a reconhecer tais portas mas também de tratar coerentemente o circuito de acordo com o modelo computacional de atraso adotado. Isto pode ser realizado ou pela

inclusão de uma fase de pré-processamento ou pela extensão do algoritmo/método de computação do atraso do circuito e das condições de sensibilização do critério adotado.

A disponibilização de geradores de macrocélulas CMOS eficientes tais como os apresentados em [CAD99] e [MOR97], e de ferramentas de mapeamento tecnológico independentes de bibliotecas [REI97] tornou possível o uso extensivo de portas complexas (sobretudo portas CMOS estáticas) no projeto físico de grandes blocos combinacionais. Deste modo, a capacidade de tratar circuitos que contenham portas complexas passa a ser altamente desejável para novas ferramentas de FTA.

Antes de iniciar uma discussão sobre a análise de *timing* funcional de circuitos que contenham portas complexas, é importante prover definições para portas simples e portas complexas. No contexto desta dissertação, uma **porta simples** corresponde a uma implementação física de um operador básico da álgebra Booleana. Assim, são portas simples as portas E, OU, NÃO-E, NÃO-OU (com qualquer número de entradas) e o inversor. Particularmente, **portas simples CMOS** são portas simples que podem ser implementadas diretamente em tecnologia CMOS, ou seja, portas NÃO-E, NÃO-OU e inversor. Uma porta complexa corresponde a uma implementação física de qualquer função Booleana de complexidade maior do que os operadores Booleanos básicos. Especificamente, uma porta complexa CMOS estática (*Static CMOS Complex Gate* - SCCG) corresponde a uma porta complexa implementada em tecnologia CMOS.

Inicialmente, será investigado o caso particular das SCCGs, em função de sua importância. Após, serão feitas considerações sobre a extensão dos modelos e algoritmos propostos para o caso de portas complexas quaisquer.

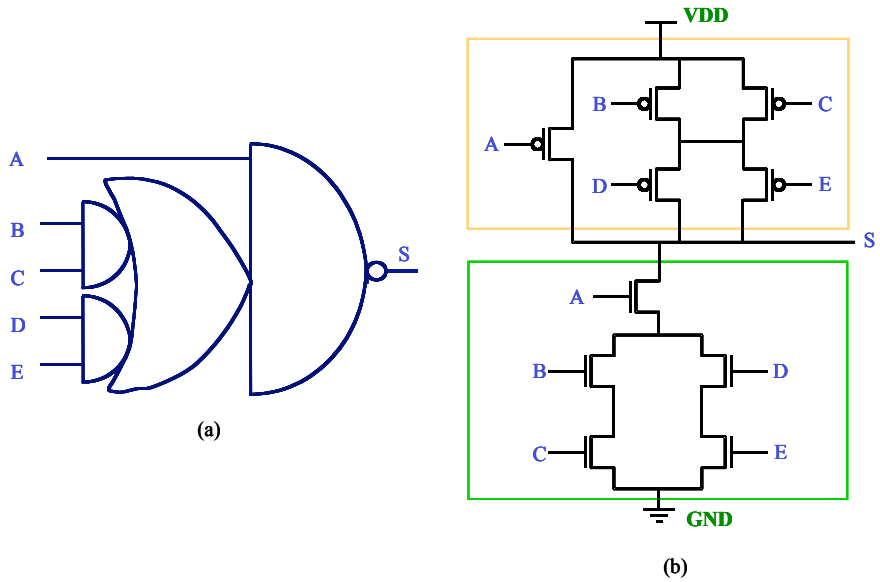


FIGURA 3.13 - Exemplo de SCCG.

Uma porta CMOS estática é implementada por uma rede de transistores CMOS conectados segundo uma topologia de "restauração completa", composta por uma rede PMOS e uma rede NMOS. A rede PMOS é capaz de prover um caminho entre a saída da porta e a massa (Vdd), enquanto que a rede NMOS é capaz de prover um caminho entre a saída da porta e a terra (Gnd). As redes NMOS e PMOS possuem mesmo número de transistores. Por uma questão de simplicidade, iremos assumir que uma

SCCG é uma porta CMOS estática na qual ambas redes de transistores apresentam apenas associações série/paralelo, e sendo rede PMOS o dual da rede NMOS, em termos de associação de transistores. A figura 3.13 mostra um exemplo de SCCG. Note-se que, para uma porta CMOS estática de *n* entradas, há *n* pares NMOS/PMOS conectados pela grade, formando cada par uma entrada da porta.

As portas CMOS estáticas, incluindo as SCCGs, podem ser classificadas de acordo com o número de transistores série/paralelo existentes nas redes NMOS e PMOS. O conjunto das portas CMOS estáticas que apresentam não mais do que *n* (*p*) transistores NMOS (PMOS) em série é definido como sendo uma "biblioteca virtual" [REI98] que pode ser designada por SCG(*n*,*p*). Pode-se também utilizar a notação SCCG(*n*,*p*) para designar o subconjunto de SCG(*n*,*p*) composto apenas por SCCGs. A tabela 3.4, retirada de [DET87], detalha o número de SCGs existentes para bibliotecas virtuais de até 5 transistores série.

TABELA 3.4 – Número de elementos para várias bibliotecas virtuais [DET87].

		número de transistores PMOS em série				
		1	**2**	**3**	**4**	**5**
número de transistores NMOS em série		1	2	3	4	5
		2	7	18	42	90
		3	18	87	396	1677
		4	42	396	3503	28435
		5	90	1677	28435	425803

Muitos algoritmos de FTA foram desenvolvidos na década dos 90. Entretanto, a maioria destes não considera a possibilidade de tratar circuitos que contenham portas complexas. No caso dos algoritmos baseados em ATPG, tal limitação deve-se ao fato da teoria de sensibilização de caminhos ter sido desenvolvida unicamente para portas simples. Obviamente, a extensão de um critério de sensibilização para considerar portas complexas resulta em regras mais complicadas. O uso de tais regras por um algoritmo de sensibilização individual de caminhos tende a piorar significativamente o desempenho deste. Em uma primeira análise, os algoritmos baseados em SAT parecem ser mais promissores, uma vez que as equações características são potencialmente capazes de representar qualquer função Booleana. Entretanto, a fim de reduzir a complexidade das instâncias de SAT a serem resolvidas, alguns algoritmos baseados em SAT assumem que os circuitos combinacionais são compostos unicamente de portas simples.

A solução mais simples para realizar-se FTA de circuitos contendo portas complexas consiste em substituir cada porta complexa por um subcircuito equivalente composto de portas simples. Esta técnica é conhecida como macroexpansão [MCG91][HSU98] e, quando utilizada a título de pré-processamento, viabiliza o uso de qualquer ferramenta de FTA que tenha sido desenvolvida para tratar circuitos constituídos unicamente por portas simples. Entretanto, a macroexpansão apresenta dois inconvenientes [HSU98]. Em primeiro lugar, é muito difícil modelar com precisão o

atraso das portas complexas macroexpandidas. Em segundo lugar, a macroexpansão cria novos nós no circuito. Estes novos nós representam um potencial aumento no número de linhas que devem ser justificadas, no caso de FTA baseada em ATPG, ou num aumento do número de equações características, no caso de FTA baseada em SAT. Em qualquer um destes casos, o aumento no tempo de execução dependerá da complexidade das portas complexas e dos modelos de atraso utilizados para estas.

Uma segunda solução reside em modificar os testes de sensibilização de modo a torná-los aptos a tratar de circuitos que contenham portas complexas. Tal modificação refere-se não apenas ao critério de sensibilização, mas também ao algoritmo que testa a sensibilização. Desde que há mais de um algoritmo para testar a sensibilização de caminhos, esta solução pode ser desdobrada em várias soluções.

Em [HSU98], por exemplo, é apresentado um algoritmo de FTA capaz de operar diretamente sobre circuitos com portas complexas. Com o intuito de evitar a macroexpansão, as condições para sensibilização exata no modo flutuante são estendidas, de modo a considerar portas complexas. Estas condições de sensibilização estendidas são utilizadas por um algoritmo baseado em sensibilização individual derivado do algoritmo D. Os resultados mostrados em [HSU98] permitem comparar modelos de atraso para macroexpansão, bem como comparar o uso da macroexpansão com o algoritmo que testa diretamente portas complexas. Por outro lado, todos os resultados foram obtidos pelo uso de algoritmos baseados em sensibilização individual de caminhos, a qual, sabidamente, não representa o estado-da-arte por sofrer de um problema conhecido por “explosão de caminhos”.

Conforme já apresentado no item 3.2, o procedimento de geração de teste temporal de Devadas et al. [DEV93a] é um algoritmo de sensibilização concorrente de caminhos baseado em ATPG derivado do algoritmo PODEM [GOE81]. No procedimento de geração de teste temporal, o problema de calcular o atraso de um circuito é transformado num conjunto de geração de testes para falhas de colagem “temporais” imaginadas como ocorrendo nas saídas primárias do circuito. O procedimento tem então o objetivo de justificar tais falhas de colagem.

A fim de poder estender o procedimento de geração de teste temporal para circuitos que contenham portas complexas, é necessário generalizar o cálculo temporal para portas E/OU de n entradas. Para tanto, lança-se mão dos conceitos de valor controlante e valor não-controlante, conforme definidos no escopo de teste. Assim, dada uma porta g tipo E/OU de n entradas, podemos classificar seu estado lógico conforme os seguintes casos:

- Casos em que ao menos uma das entradas de g apresenta o valor controlante (c(g)). As demais entradas podem apresentar ou o valor não-controlante (nc(g)) ou o valor 2;
- Caso em que todas as entradas de g apresentam o valor não-controlante (nc(g));
- Casos em que ao menos uma das entradas de g apresenta o valor 2, mas nenhuma entrada apresenta o valor controlante (c(g)). As demais entradas podem apresentar o valor não-controlante (nc(g)).

A partir da identificação dos casos possíveis, chega-se à generalização das regras originalmente expressas nas tabelas 3.2 e 3.3. Também é possível expandir tais regras de modo a se considerar a polaridade de saída das portas, o que permite tratar portas NÃO-E e NÃO-OU. As regras generalizadas são mostradas na tabela 3.5.

TABELA 3.5 – Regras generalizadas para o cálculo temporal de três valores para portas simples de *n* entradas [GÜN2000].

grupo		regras	*Lv*
1	l_o	$\min\{ l_i \mid i= c(g) \text{ or } i=2 \} + d$	$Pol(g) \oplus c(g)$
	u_o	$\min\{ u_j \mid j= c(g) \} + d$	
2	l_o	$\max\{ l_i \} + d$	$pol(g) \oplus nc(g)$
	u_o	$\max\{ u_j \} + d$	
3	l_o	$\min\{ l_i \mid i=2 \} + d$	2
	u_o	$\max\{ u_j \} + d$	

As regras resumidas na tabela 3.5 podem ainda ser representadas de maneira gráfica, conforme mostram as figuras 3.14, 3.15 e 3.16. Por uma questão de simplicidade, tanto na tabela 3.5 como nas figuras 3.14, A3.15 e A3.16 assumiu-se um atraso único por porta. Modelos computacionais de atraso mais sofisticados serão discutidos mais adiante.

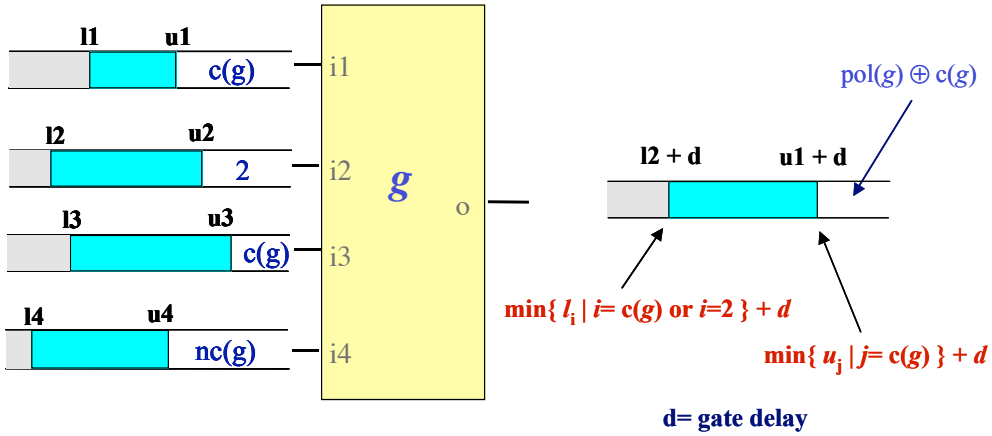


FIGURA 3.14 – Cálculo temporal de três valores para o grupo 1 [GÜN2000].

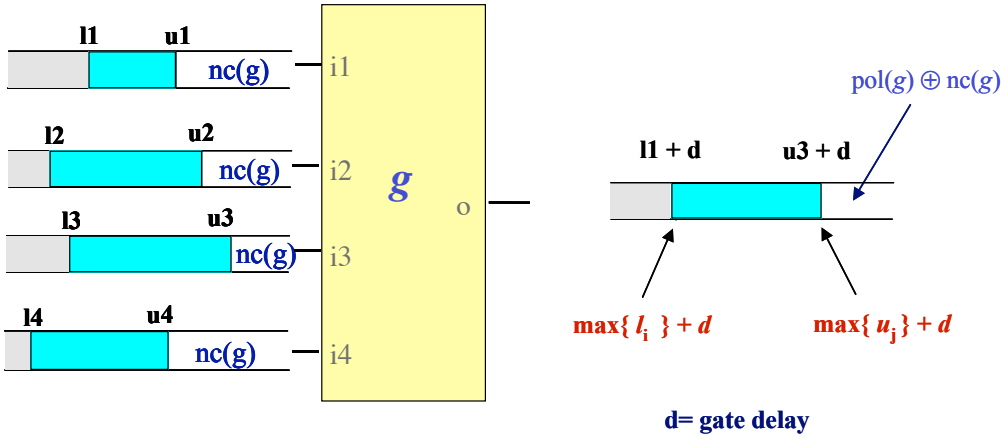


FIGURA 3.15 - Cálculo temporal de três valores para o grupo 2 [GÜN2000].

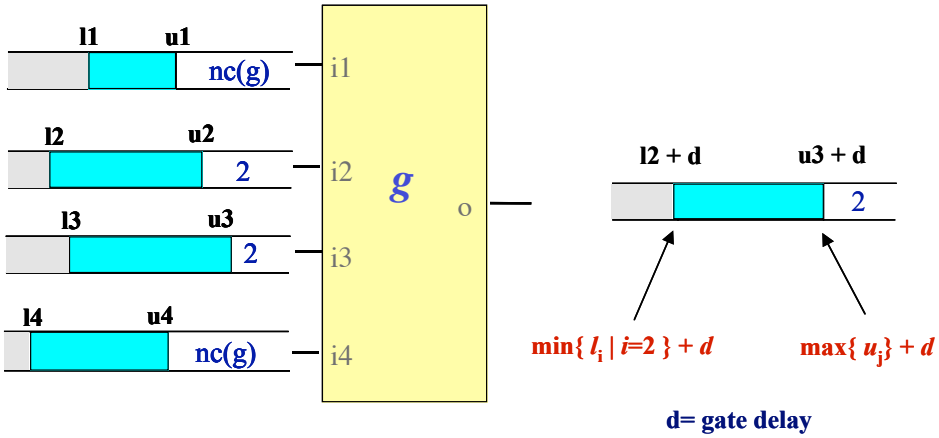


FIGURA 3.16 - Cálculo temporal de três valores para o grupo 3 [GÜN2000].

Assumindo-se que a função lógica de uma porta qualquer *g* esteja na forma fatorada e utilizando uma estrutura de dados apropriada para representá-la, a aplicação das regras generalizadas é direta. Considere, por exemplo, a SCCG mostrada na figura 3.17a. A função lógica desta porta é dada por $S = \overline{A \cdot ((B \cdot C) + (D \cdot E))}$ e pode ser representada por uma "árvore da função" (figura 3.17c). Examinando-se esta árvore da função nota-se que, dado um assinalamento de valores temporais de entrada (valores lógicos e respectivos limites inferior e superior de atraso), os valores temporais na saída podem ser obtidos mediante a aplicação sucessiva das regras da tabela 3.5 para cada subárvore, iniciando-se pela subárvore mais próxima da base, desde que sejam feitas as seguintes assertivas:

- Todas as subárvores, exceto a raiz, apresentam atraso de propagação zero e polaridade igual a zero
- O atraso de propagação da porta é aplicado somente sobre os limites inferior e superior de atraso da saída da porta, i.e., sobre os valores temporais resultantes da avaliação da subárvore de maior hierarquia.
- A polaridade da porta é tratada somente quando a subárvore de maior hierarquia é processada.

A primeira e a segunda assertiva permitem que se divida a avaliação temporal da SCCG em dois passos independentes. No primeiro passo, o valor lógico da saída e os limites inferior e superior de atraso são computados para um atraso de propagação da porta igual a zero. Chamaremos este intervalo de atrasos de "limites de atraso de primeira ordem". Num segundo passo, os limites inferior e superior reais são computados por meio da adição do atraso de propagação da porta aos limites de atraso de primeira ordem. Este segundo passo leva em conta o modelo computacional de atraso adotado para as portas.

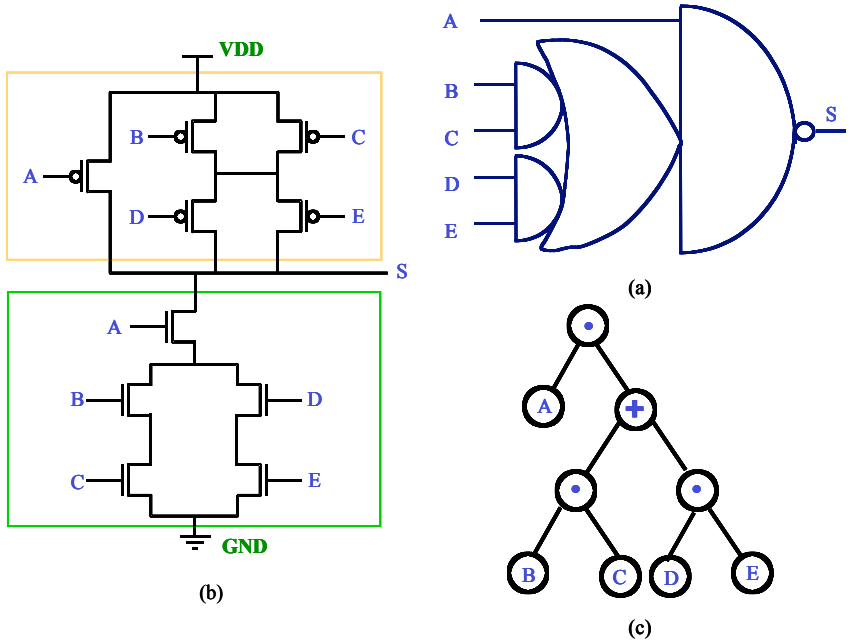


FIGURA 3.17 - Exemplo de SCCG: símbolo para o nível lógico (a), esquemático de transistores (b) árvore da função (c).

A figura 3.18 ilustra a aplicação das regras do cálculo temporal aplicadas à SCCG da figura 3.17.

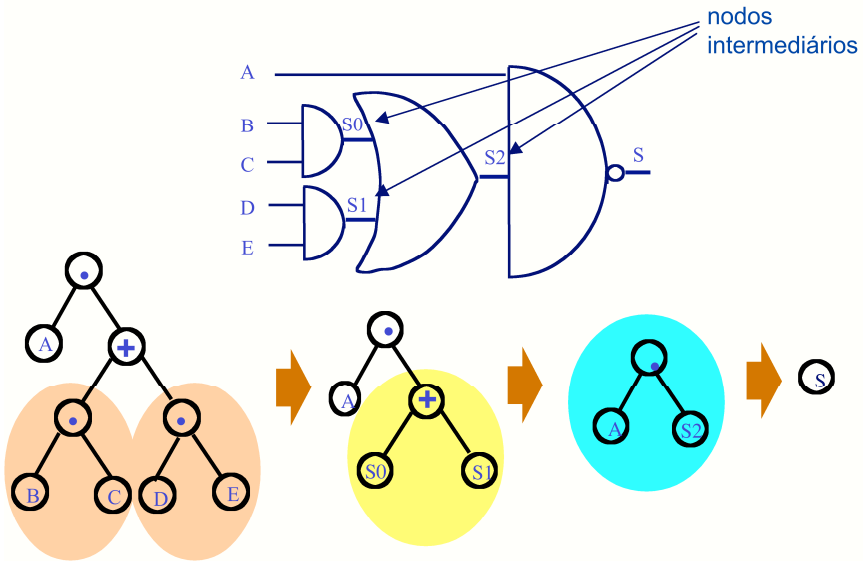


FIGURA 3.18 – Uso do cálculo temporal de três valores para avaliar uma SCCG.

4 ESTUDO SOBRE A ACELERAÇÃO DOS ALGORITMOS DE FTA

O estudo da aceleração de algoritmos de FTA abordado sob o ponto de vista da geração automática de teste empenha-se em encontrar estratégias para a exploração do espaço de busca de vetores de teste de forma a identificar pontos específicos em que o algoritmo pode ser acelerado. O principal fator que influencia diretamente a complexidade dos algoritmos de ATPG é a realização de uma pesquisa extensa, no espaço de possíveis soluções, antes de encontrar o vetor de teste. Assim, buscar mecanismos que diminuam o tempo de exploração do espaço de soluções significa abordar estratégias para encontrar o vetor teste o mais breve possível ou identificar rapidamente a não existência deste, a fim de reduzir a complexidade do processo.

Os algoritmos PODEM [GOE81] e FAN[FUJ83], descrevem técnicas de aceleração do processo de busca de vetores de teste. O algoritmo PODEM implementa uma técnica capaz de reduzir o número de ocorrências de retrocessos (*backtrackings*), utilizando a assertiva de que o retrocesso ocorre somente nas entradas primárias do circuito. A fim de reduzir mais o número de retrocessos do PODEM, o FAN introduz um novo procedimento que faz uma pausa no processo de busca assim que encontra pontos no circuito com fanout maior que um, resolvendo antecipadamente a ocorrência de algum conflito lógico, antes mesmo que as entradas primárias do circuito sejam alcançadas. Assim como no algoritmo FAN, as técnicas utilizadas para acelerar o processo de busca de vetores de teste do algoritmo PODEM buscam satisfazer os valores lógicos (0/1) em nodos internos do circuito.

No item 4.1, é apresentado um trabalho de avaliação do comportamento dos algoritmos de FTA em circuitos que contém portas complexas. Mais especificamente, foram implementadas cinco versões do algoritmo PODEM que em conjunto com medidas de testabilidade fazem uma avaliação dos resultados quando esta técnica é aplicada a circuitos que contém portas complexas. Neste caso, não é considerada o campo da variável temporal do circuito.

Além disso, outras técnicas de aceleração, não exploradas neste trabalho, podem ser citadas *Recursive Learning* [SIL99], *backtracking* não-cronológico [SIL94]. Enfim, como os algoritmos clássicos mais conhecidos já apresentam estas técnicas resolveu-se por um revisão sobre a aplicação destes algoritmos.

4.1 Aceleração de Algoritmos de ATPG Baseada em Medidas de Testabilidade.

Considere os algoritmos baseados em geração automática de teste (ATPG), quando uma técnica de aceleração é desenvolvida visa reduzir o número de *backtrackings* no processo de busca dos vetores de teste. Os trabalhos que apresentam algoritmos de FTA baseados em ATPG, em sua maioria, como SLOCOP [BEN90], TrueD-F [DEV93a], ACPA [CHE93], são descritos e seus resultados experimentados para circuitos que contém portas simples (AND/NAND, OR/NOR, NOT). Então para que os algoritmos tornem-se eficientes também para portas complexas é necessário investigar a aplicabilidade das técnicas para todo o tipo de porta .

Como um primeiro passo para o desenvolvimento de uma ferramenta eficiente de ATPG para circuitos que contém portas complexas, optou-se por avaliar o comportamento do algoritmo PODEM aplicado a portas complexas. O PODEM foi escolhido por apresentar uma técnica de aceleração do processo de *backtracking*. Num primeiro momento, serão apresentadas cinco medidas de testabilidade que são aplicadas durante o procedimento *backtrace*. A seguir, o algoritmo PODEM é aplicado a circuitos que contém redes de portas complexas. Por fim, para estabelecer uma comparação entre medidas de testabilidade, o PODEM foi implementado para quatro diferentes mapeamentos. Um dos mapeamentos com porta simples e os outros três mapeamentos para portas complexas utilizando uma ferramenta de mapeamento tecnológico chamada de TABA [REI97]. Por fim, os resultados obtidos são discutidos e apresentados.

No algoritmo PODEM valores lógicos são assinalados às entradas primárias do circuito, uma entrada por vez. A seguir, os valores lógicos assinalados às entradas são propagados em direção às saídas primárias do circuito pelo processo de implicação. A característica marcante do algoritmo determina que processo de retrocesso (*backtracking*) restringe sua ocorrência somente às entradas primárias do circuito. O procedimento que indica qual a entrada primária será assinalada e qual o valor lógico corresponde a esta entrada é chamado de *bakctrace*. O estudo deste caso, busca mostrar qual medida de testabilidade é mais apropriada para circuitos que contém SCCGs.

4.1.1 Medidas de Testabilidade

As medidas de testabilidade, cuja tarefa é orientar a função *backtrace* na escolha de um caminho até que uma entrada primária seja encontrada, são de suma importância para o algoritmo PODEM pois, tem como objetivo final determinar o teste o mais breve possível. Assim, em cada passo do *bakctrace()*, que é chamado de uma linha L do circuito, o algoritmo deve escolher uma entrada de uma porta para continuar traçando um caminho. A escolha baseia-se na heurística *easy/hard* que utiliza os valores pré calculados pelas medidas de testabilidade para orientar o caminho que será traçado até que sejam atingidas as entradas primárias.

O argumento escolhido pela função *backtrace* (linha,valor lógico), em cada passo da execução do procedimento, influencia diretamente três aspectos do algoritmo. Inicialmente, tem influência sobre o tempo gasto para implicar o valor assinalado a entrada primária. Por outro lado, no caso em que o objetivo não foi satisfeito durante a implicação, outro aspecto diz respeito ao número de entradas que deverão ser testadas

até que o objetivo o seja atingido. E finalmente, se para atingir o objetivo será necessário realizar o *backtracking* ou não. Como conseqüência da medida de testabilidade escolhida para guiar o *backtrace*, serão influenciados um ou mais dos três aspectos citados. Então, para investigar o impacto de algumas medidas de testabilidade sobre o tempo de execução do algoritmo PODEM, operando sobre portas complexas, serão detalhadas cinco medidas escolhidas.

As medidas propostas por Goldestein [GOL79] estão baseadas nas propriedades de controlabilidade e observabilidade. A controlabilidade avalia a facilidade de controlar um determinado valor lógico em uma determinada aresta do circuito. Enquanto a observabilidade avalia com que facilidade pode-se observar o valor lógico de uma determinada aresta na saída do circuito. Nos itens apresentados a seguir são apresentadas as medidas utilizadas.

A.Controlabilidade Combinacional

Neste caso todas as entradas primárias são inicializadas com os valores de controlabilidade iguais a 1, CC[0] = CC[1] = 1. As equações da figura 4.1 são utilizadas para calcular os valores da controlabilidade para as demais linhas do circuito. Neste caso, o cálculo é descrito para as funções AND e OR . Para tal cálculo, utiliza-se o algoritmo *Topological Sort* [COR90]. Observe que a controlabilidade combinacional leva em conta a profundidade lógica das linhas do circuito, o que é representado na equação pelo fator "+1". Para se calcular a controlabilidade para o valor controlado na saída da porta deve-se pegar o menor valor de controlabilidade. Este valor é relativo ao valor controlante dentre todas as entradas e somar 1. Para calcular o valor não-controlado na saída da porta deve-se somar os valores de controlabilidade relativos aos valores não controlantes de todas as entradas mais 1 .

Função AND :

$$CC^0[O] = \min(CC^0[I_1], CC^0[I_2], ..., CC^0[I_N]) + 1$$

$$CC^1[O] = CC^1[I_1] + CC^1[I_2] + ... + CC^1[I_N] + 1$$

Função OR :

$$CC^0[O] = CC^0[I_1] + CC^0[I_2] + ... + CC^0[I_N] + 1$$

$$CC^1[O] = \min(CC^1[I_1], CC^1[I_2], ..., CC^1[I_N]) + 1$$

FIGURA 4.1 - Cálculo da Controlabilidade Combinacional para as funções AND e OR.

B.Controlabilidade Seqüêncial

O cálculo da controlabilidade seqüêncial inicializa igualmente todos os valores nas entradas primárias com valor lógico 1, CC[0]=CC[1]=1. Este cálculo foi proposto por Goldstein [GOL79] e preserva as mesmas propriedades do cálculo da controlabilidade combinacional apresentado anteriormente.

Função AND :

$$SC^0[O] = \min(SC^0[I_1], SC^0[I_2], ..., SC^0[I_N])$$

$$SC^1[O] = SC^1[I_1] + SC^1[I_2] + ... + SC^1[I_N]$$

Função OR :

$$SC^0[O] = SC^0[I_1] + SC^0[I_2] + ... + SC^0[I_N]$$

$$SC^1[O] = \min(SC^1[I_1], SC^1[I_2], ..., SC^1[I_N])$$

FIGURA 4.2: Cálculo da Controlabilidade Seqüêncial para as funções AND e OR.

Observe que o valor 1 não foi adicionado às equações o que significa que a profundidade lógica não é considerada neste cálculo [FIGURA 4.2].

C.Baseada em *Fanout* (*fanout-based*)

Esta medida pode ser vista como uma variação da controlabilidade seqüêncial já que o fator relativo ao *fanout* não considera a profundidade lógica [FIGURA 4.3].

Função AND :

$$C^0[O] = \min(C^0[I_1], C^0[I_2], ..., C^0[I_N]) + Fanout - 1$$

$$C^1[O] = C^1[I_1] + C^1[I_2] + ... + C^1[I_N] + Fanout - 1$$

Função OR :

$$C^0[O] = C^0[I_1] + C^0[I_2] + ... + C^0[I_N] + Fanout - 1$$

$$C^1[O] = \min(C^1[I_1], C^1[I_2], ..., C^1[I_N]) + Fanout - 1$$

FIGURA 4.3: Cálculo da Controlabilidade baseado no *Fanout*.

D.Baseada em *Fanout* e Baseada em distância.

Este cálculo também é baseado nas equações do Goldstein [GOL79]. As equações a seguir levam em conta tanto o fanout quanto a profundidade lógica das linhas do circuito [Figura 4.4].

Função AND :

$$C^0[O] = \min(C^0[I_1], C^0[I_2], ..., C^0[I_N]) + Fanout$$

$$C^1[O] = C^1[I_1] + C^1[I_2] + ... + C^1[I_N] + Fanout$$

Função OR :

$$C^0[O] = C^0[I_1] + C^0[I_2] + ... + CC^0[I_N] + Fanout$$

$$C^1[O] = \min(C^1[I_1], C^1[I_2], ..., C^1[I_N]) + Fanout$$

FIGURA 4.4- Cálculo da Controlabilidade baseda em *Fanout* e em Distância.

E.Probabilística

Neste cálculo todas as entradas primárias são inicializadas com o valor 0.5 e os nodos internos são inicializados com o valor de testabilidade igual a 1.0. Estas

medidas não são relativas a controlabilidade das linhas, mas sim a probabilidade de ter em uma linha um determinado valor lógico. Para o valor não controlado na saída de uma porta o cálculo da probabilidade é o produto das probabilidades entre os valores não controlantes de todas as entradas da porta. A probabilidade do valor controlante na saída da porta é calculada como 1 - $P^1[0]$ [FIGURA 4.5].

Função AND :

$$P^0[O] = 1 - P^1[0]$$

$$P^1[O] = P^1[I_1] * P^1[I_2] * ...* P^1[I_N]$$

Função OR :

$$P^0[O] = P^0[I_1] * P^0[I_2] * ...* P^0[I_N]$$

$$P^1[O] = 1 - P^0[0]$$

FIGURA 4.5: Cálculo da controlabilidade baseado em medidas de Probablidade.

4.1.2 Implementação Considerando Portas Complexas

Uma das técnicas tradicionais para gerar vetores de teste para circuitos com portas complexas é a macro expansão explícita [referencia]. Esta técnica exige uma estrutura de controle complexa para conservar a estrutura lógica do circuito. Como uma alternativa, foi proposto a macro expansão implícita [GÜN02]. Este método consiste em usar uma estrutura para representar a função lógica das portas que permita que as portas complexas sejam processadas por partes. Para tanto utiliza-se a estrutura de árvore que permite que cada subárvore seja processada como uma porta simples [FIGURA 4.6]. Os nodos da árvore foram alterados para que pudéssemos escrever valores lógicos e valores relativos à testabilidade.

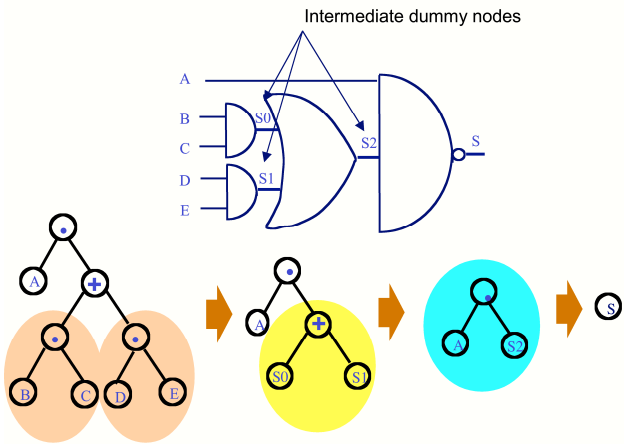


FIGURA 4.6 – Avaliação de uma porta complexa usando macro expansão implícita.

Os experimentos foram realizados utilizando os circuitos de teste do *benchmark* MCNC (ISCAS85 mais alu2, alu4, bw, 9sym, t481). Cada um dos circuitos foi mapeado com a ferramenta TABA [REI97] da seguinte forma:

1. Um mapeamento para portas simples com no máximo dois transistores em série/paralelo
2. Três mapeamentos para portas complexas :
   - um mapeamento com no máximo dois transistores em série/paralelo,
   - um mapeamento com no máximo três transistores em série/paralelo;
   - um mapeamento com no máximo quatro transistores em série/paralelo,

Os testes foram gerados usando uma estação de trabalho UltraSparc10™ com 521MB de memória RAM e 1GB de memória swap. Em cada circuito o algoritmo gerou um teste para stuck-at-faults nas saídas do circuito. O limite de tempo para cada falha foi estabelecido em 2 segundos. Cada valor de tempo correspondente a uma medida de testabilidade é equivalente a soma da média do tempo de execução por falha entre todos os circuitos. Este cálculo foi efetuado para que os resultados fossem mais expressivos.

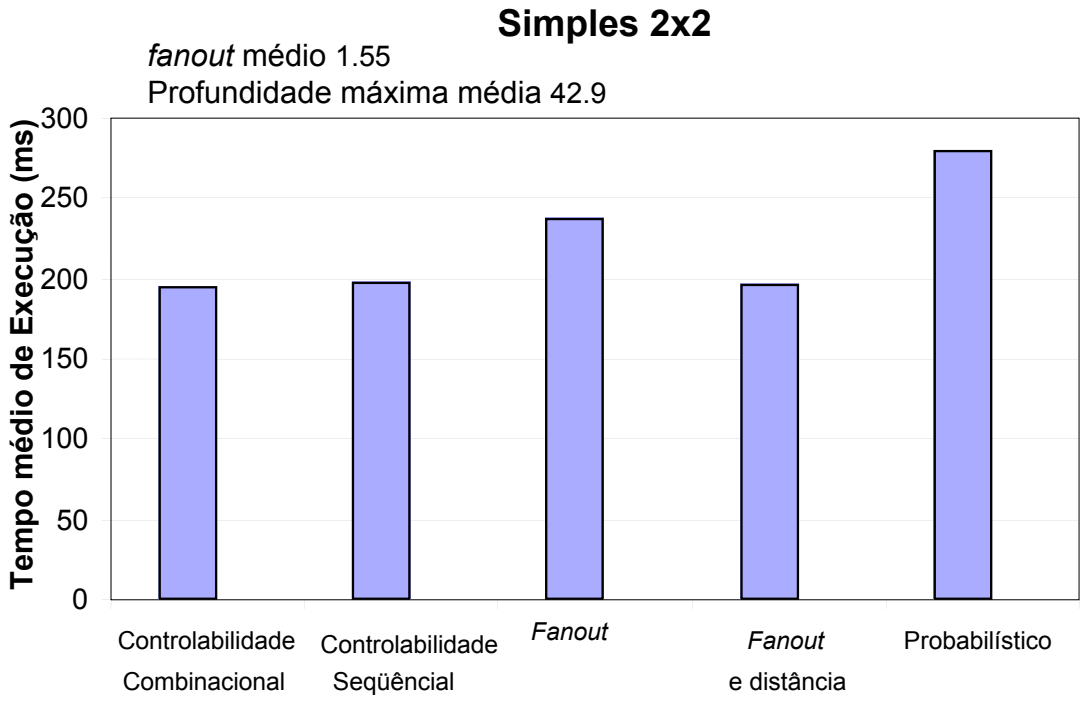


FIGURA 4.7 – Tempo de Execução para Portas Simples 2x2.

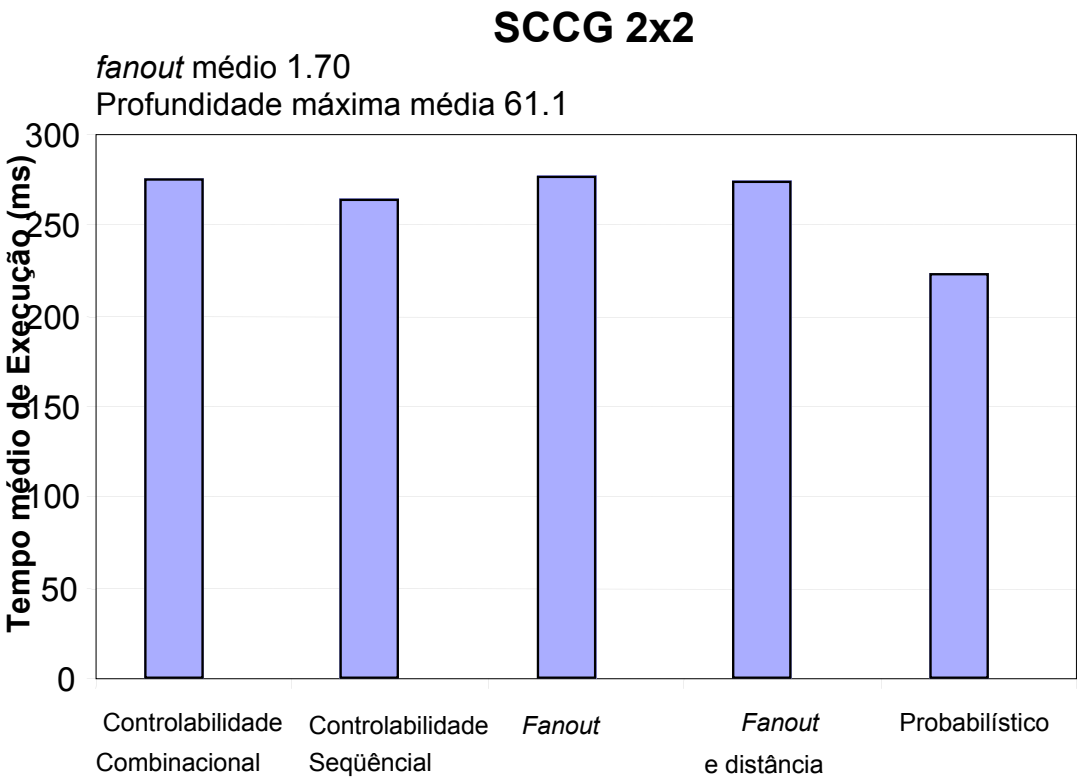


FIGURA 4.8 – Tempo de Execução para Portas Complexas 2x2.

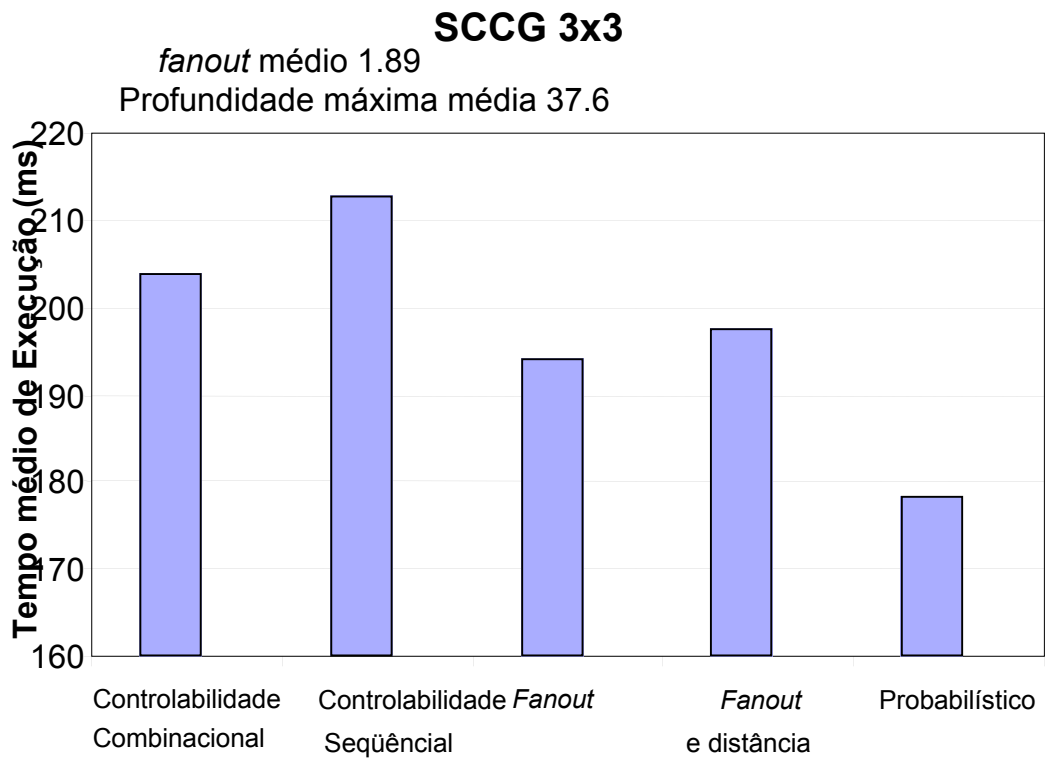


FIGURA 4.9 –Tempo de Execução para Portas Complexas 3x3.

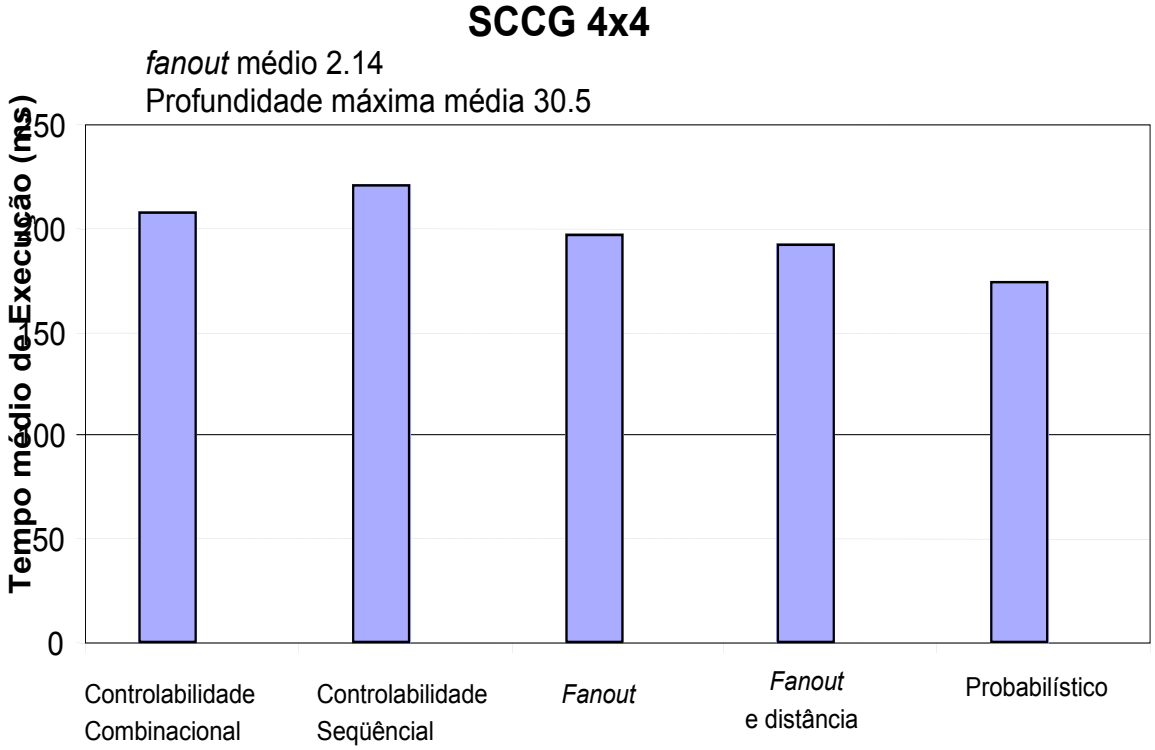


FIGURA 4.10 – Tempo de Execução para Portas Complexas 4x4.

Ao avaliar os resultados, observa-se que as medidas de testabilidade quando aplicadas em circuitos com portas complexas apresentam diferentes desempenhos em relação a circuitos que contenham somente portas simples. Com base nos resultados podemos afirmar que a medida de probabilidade [BRG84]] apresenta melhores resultados que as medidas baseadas no cálculo de Goldstein [GOL79] quando aplicada para portas complexas.

O capítulo 5 apresenta o algoritmo DETA (*Delay Enumeration-Based Timing Analysis*) que descreve um procedimento para acelerar o processo de busca dos vetores de teste. Este procedimento esta baseado na enumeração de atrasos para determinar um valor δ (delta) que representa um limite superior seguro para melhorar a estimativa do atraso do circuito.

5 O ALGORITMO DETA *(DELAY ENUNERATION-BASED TIMING ANALISYS*).

Nesta seção será apresentado o algoritmo DETA (*Delay Enumeration-Based Timing Analysis*). O algoritmo propõe uma solução para acelerar o processo de busca baseado em enumeração de atrasos. Conseqüentemente, provar que o atraso **δ** (delta) determinado na saída de um circuito combinacional é considerado um limite seguro para tratar um conjunto de caminhos reconvergentes na direção de uma saída primária do circuito. Portanto, acelerar o processo de busca significa justificar atrasos em nodos internos do circuito.

A fim de enumerar os atrasos, o algoritmo explora técnicas de aceleração baseadas em algoritmos de ATPG incorporando ao procedimento o fator temporal. Considerando um atraso **δ,** o DETA busca responder à seguinte pergunta: "o atraso do circuito é maior ou igual δ?". Para tanto, o algoritmo explora o espaço de vetores de entrada verificando a existência de um vetor que satisfaça a condição do atraso do circuito ser maior que **δ**. Assim, se existe um vetor de entrada que satisfaça esta condição, pode-se afirmar **δ** não representa um limite seguro para tratar um conjunto de caminhos. Portanto, para provar que **δ** é um limite superior seguro, é necessário explorar todas as possíveis combinações de entradas testando um grande número de vetores ou de cubos de entrada.

Uma forma de reduzir o espaço de busca é encontrar o mais rápido possível um cubo que determina um atraso menor que **δ**. E portanto, garantir que **δ** representa um limite seguro para determinar o atraso do circuito

Uma alternativa para evitar a exploração de todas as possíveis combinações de entradas é assumir que **δ** não representa um limite seguro para determinar o atraso do circuito quando um cubo de entrada estabiliza um sinal na saída em um tempo maior que **δ.** Então, pode-se afirmar que **δ** representa uma estimativa pessimista para o atraso do circuito. Este comportamento, porém não valida a existência de um vetor de entrada cujo atraso é maior que **δ**.

Uma abordagem mais favorável para reduzir a exploração do espaço de busca de vetores de teste, estabelece uma tratativa para deslocar a enumeração de atrasos das saídas primárias por linhas internas do circuito. Inicialmente, é fixado um **objetivo principal** a uma saída primária, do circuito. Para satisfazer este **objetiv**o é necessário encontrar um conjunto de valores que aplicados às entradas atribuem valores às linhas do circuito que justificam o **objetivo principal** na saída. A cada passo do objetivo principal na direção das entradas primárias podem ser criados **objetivos secundários**. Ao criar um objetivo secundário atribui-se um valor às linhas internas do circuito que

são justificados durante a computação do atraso. Quando um dos objetivos secundários é satisfeito, o atraso na linha que está sendo processada é atualizado, conseqüentemente o DETA recalcula os atrasos para todos os objetivos secundários já criados.

Os algoritmos de FTA quase na sua totalidade apresentam três passos básicos : criação do grafo que representa o circuito, pré-processamento do grafo para computar máximos atrasos e a computação do atraso do circuito. De acordo com a taxonomia apresentada em [GÜN00], que classifica os algoritmos baseado na proposta de métodos para computar o atraso, o DETA se classifica como um algoritmo baseado em ATPG com sensibilização concorrente de caminhos.

Inicialmente o algoritmo DETA fixa o objetivo principal em uma saída primária do circuito. Define-se como um *objetivo*(l, v_l, δ) uma terna de valores cujo valor $\boldsymbol{\delta}$ representa um valor de atraso calculado. O elemento δ é o parâmetro utilizado para avaliar se o valor lógico v_l justificado à linha l do circuito estabiliza em um tempo menor que δ. Assim, satisfazer um objetivo significa encontrar um conjunto de valores que entrada que justifica o valor lógico v_l à linha l do circuito em um tempo menor que δ. E portanto, o valor de δ representa um limite superior seguro para estimativa do atraso.

A fim de determinar um δ inicial, o valor deδ corresponde ao mínimo dentre os atrasos máximos de subida (mds_lh) e descida (mds_hl) subtraído de um valor ε, conforme a expressão (1) .

$$\delta_{inicial} = \{\min(\text{mdts\_hl}, \text{mdts\_lh})\} - \varepsilon; \quad (1)$$

Após fixar o **objetivo principal** (l, v_l, $\boldsymbol{\delta_{inicial}}$), novos valores de v_l e $\boldsymbol{\delta}$ são calculados na direção das entradas primárias. Pela análise da funcionalidade das portas que compõem o circuito, a medida que são calculados em suas linhas internas novos valores para v_l e $\boldsymbol{\delta}$, pode-se definir um conjunto de objetivos secundários. Estes objetivos não são criados em todas as linhas do circuito, somente em linhas que são alcançadas imediatamente após uma reconvergência. Além disso, o objetivo secundário será criado nestas linhas somente no caso em que o valor de δ calculado para a porta (g) que está sendo analisada é menor que o atraso máximo do sinal para a transição requerida de acordo com a expressão (2).

$$\delta_g < \{\max(\text{mdts\_hl}, \text{mdts\_lh})\}; \quad (2) \text{ onde}$$

$$\delta_g = \delta_{entrada\_g} = \delta_{saída\_g} - \text{td}_g; \quad (3)$$

onde td corresponde ao atraso de propagação (subida ou descida) da porta g;

O exemplo apresentado na figura 5.1 ilustra como o algoritmo cria o conjunto de objetivos secundários a partir do **objetivo principal**. Para o circuito representado na estrutura de um grafo o algoritmo tenta justificar o valor lógico 1 em um tempo menor que δ =1000 ($\boldsymbol{\delta_{inicial}}$). O valor do novo $\boldsymbol{\delta}$ = 594 apontado no exemplo foi calculado de acordo com a expressão (3). Inicialmente, o $\delta_{inv0} = \delta_{entrada\_nand1} = \delta_{saída\_nand1} - \text{tplh}_{nand1}$ é calculado, seu valor é igual à 638 e satisfaz a condição em (2); porém, esta linha não está imediatamente após uma reconvergência e portanto, não gera algum objetivo secundário.

Então, um próximo δ é calculado de acordo com a expressão (3), o δ = 594 satisfaz a condição expressa em (2). Assim, gera um objetivo secundário, pois a linha está imediatamente após um reconvergência.

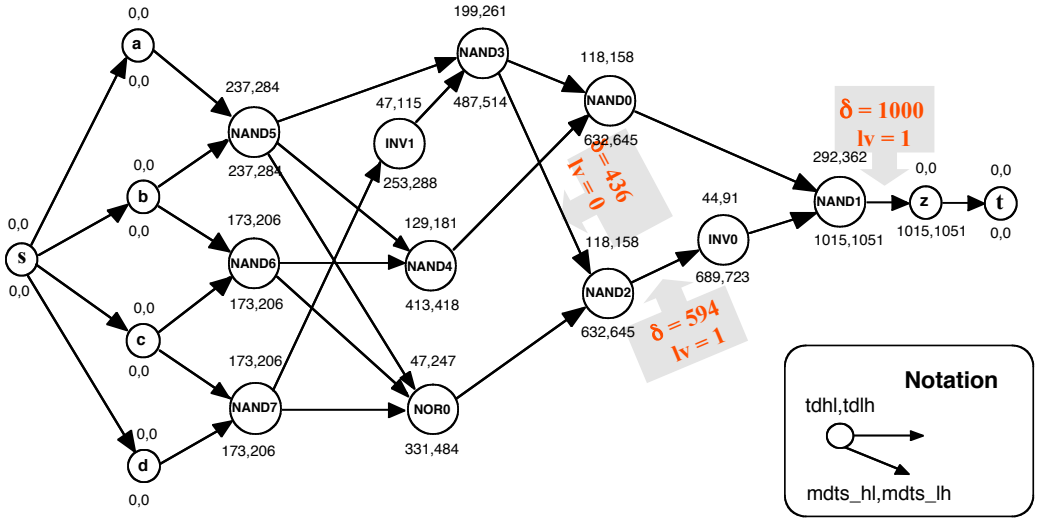


FIGURA 5.1– Conjunto de objetivos Secundários.

A figura 5.2 apresenta o pseudo código do DETA. O procedimento busca_objetivo (*find_objetives*) armazena em uma pilha os objetivos secundários que devem ser satisfeitos. A fim de justificar o valor lógico v_l em um tempo menor que δ e portanto, satisfazer um objetivo (*current_objective*), o algoritmo inicialmente dispara um procedimento temporal (*timed_backtrace()*) que é utilizado para assinalar um valor lógico à uma entrada primária. Após assinalar um valor em uma entrada primária, outra função dispara um processo de implicação temporal (*timed_implication()*) que encarrega-se de assinalar valores lógicos às linhas do circuito e satisfazer o objetivo. Entretanto, quando um objetivo não é satisfeito, ou um número maior de entradas devem ser assinaladas, ou então, não foi possível justificar o valor lógico v_l em um tempo menor que δ.

```
find_objectives(objectives_stack);
while there are objectives in the objectives stack do
current_objective = pop(objective_stack);
while the current_objective is not satisfied do
        decision = timed_backtrace(current_output, δ, logic value);
        timed_implication(decision);
        while there is a conflict in the current objective
                new_decision = backtrack(decision);
                timed_implication(decision);
        end while;
end while;
end while;
```

FIGURA 5.2 – Pseudo-código do algoritmo DETA.

Para que um número maior de entradas sejam assinaladas o algoritmo chama a função (*timed_backtrace()*). Em caso do atraso ser maior que δ, o valor de δ não é

considerado um limite seguro, consequentemente, o objetivo é removido da pilha e um novo objetivo é retirado para ser satisfeito.

De outra forma, quando o algoritmo satisfaz o objetivo (*current_objective*), significa que foi possível justificar o valor lógico à linha *l* do circuito em um tempo menor que δ e portanto, δ determina um limite superior seguro para estimativa do atraso. Ao satisfazer o objetivo é pertinente verificar se o atraso é menor que δ para o complemento (v'_l) do valor lógico. Neste caso, o algoritmo chama uma função que dispara o procedimento de retrocesso (*backtracking()*) e desfaz uma decisão tomada. Os procedimentos de implicação temporal e retrocesso são chamados até que o objetivo(l, v'_l, δ) seja satisfeito.

A fim de analisar o desempenho do algoritmo DETA foi proposto um experimento para medir tempo de execução do algoritmo em determinar o atraso crítico de um conjunto de circuitos. Inicialmente o algoritmo DETA foi rodado para os circuitos de teste do *benchmark* MCNC (C17, alu2, alu4, bw, 9sym, t481). Cada um dos circuitos foi mapeado com a ferramenta TABA [REI97] da seguinte forma:

3. Um mapeamento para portas simples com no máximo dois transistores em série/paralelo
4. Três mapeamentos para portas complexas :
    - um mapeamento com no máximo dois transistores em série/paralelo;
    - um mapeamento com no máximo três transistores em série/paralelo;
    - um mapeamento com no máximo quatro transistores em série/paralelo.

Depois de contabilizados os tempos de execução, foram determinados o número de falsos caminhos e número de entradas de cada circuito. O mesmo comportamento foi medido para um conjunto de vetores de teste através da simulação de um vetor *floating*. Assim como a maioria dos algoritmos de FTA determinam a natureza de suas entradas utilizando o modo *floating*, a simulação computa o atraso do circuito aplicando às entradas do circuito um vetor *floating*.

A figura 5.3 apresenta o gráfico que compara o resultado do algoritmo DETA com a simulação de um vetor *floating*. O gráfico demonstra que em alguns casos o algoritmo proposto apresenta uma baixa dependência do número de entradas quando comparado com a simulação. Por outro lado, o tempo de execução do algoritmo DETA ainda é muito alto, o que demanda o uso de uma técnica apropriada de aceleração para algoritmos de ATPG que deve ser incluída no DETA.

O alto grau de complexidade dos circuitos que contém portas complexas eleva o tempo de execução do algoritmo de enumeração de atrasos, pois tendem a apresentar um aumento do número de redes internas com *fanout* maior que um, conseqüentemente, o circuito incorpora um maior número de reconvergências. O aumento de caminhos reconvergentes tende a gerar um número elevado de objetivos secundários.

O número de caminhos falsos faz aumentar o tempo de execução pois tende a requerer um maior número de iterações até que sejam determinados caminhos falsos.

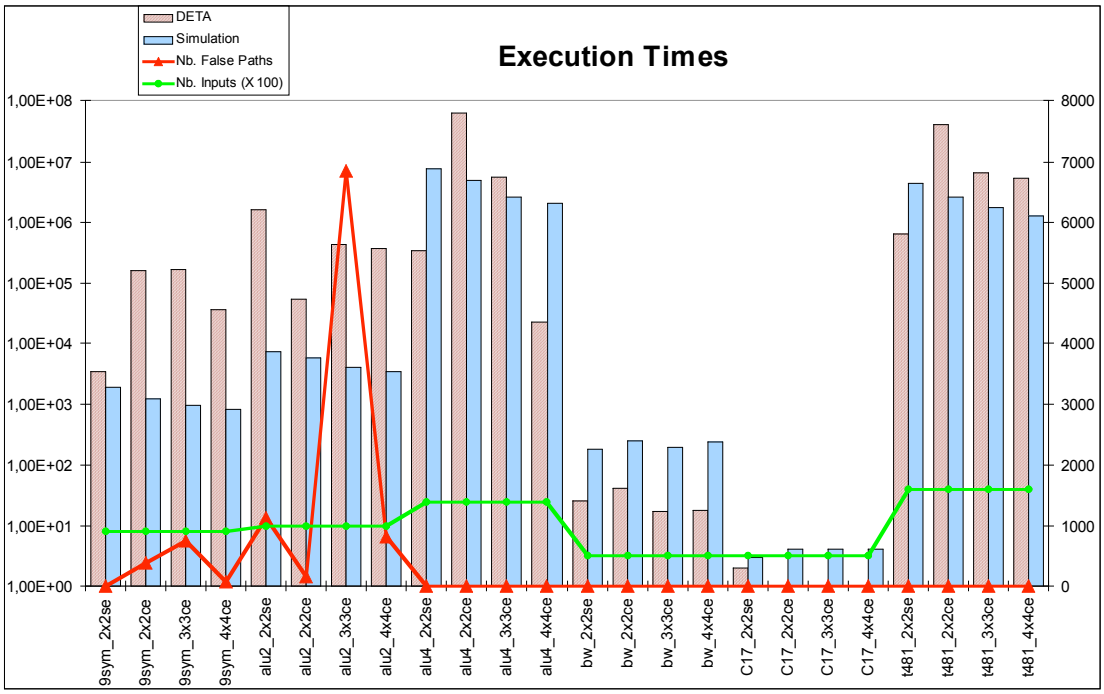


FIGURA 5.3 – DETA x Simulação.

De acordo com os resultados obtidos durantes os experimentos é possível afirmar que o algoritmo DETA pode ser acelerado utilizando-se técnicas de aceleração dos algoritmos de ATPG. Neste caso, por exemplo, pode-se utilizar um procedimento que implementa um múltiplo *backtrace* ou então fazer uso de *headlines* durante a tomada de decisões. Assim, ao aplicar as técnicas de aceleração os resultados obtidos pelo algoritmo podem ser refinados.

6 CONCLUSÕES

O presente trabalho abordou o estudo de algoritmos de ATPG - Geração Automática de Padrões de Teste (*Automatic Test Pattern Generation*) aplicados à Análise de *timing* Funcional. As contribuições deste trabalho dizem respeito a teoria geral de algoritmos de ATPG e a investigação da possibilidade de aplicação das técnicas de aceleração utilizadas nestes algoritmos para a aplicação em algoritmos de FTA.

Durante o estudo de métodos para a geração de testes para falhas de colagem simples foi apresentado uma descrição das funções básicas que fazem parte dos algoritmos de ATPG . O trabalho apresentou a caracterização e descrição de três algoritmos apontados como clássicos de geração de vetores de testes : algoritmo-D, PODEM, e o FAN. Uma contribuição fundamental foi uma análise comparativa dos algoritmos visando a investigação das técnicas utilizadas na aceleração do processo de geração do teste.

No que se refere à Análise de Timing Funcional, fazendo uso da taxonomia proposta em [GÜN2000] foram discutidas duas classes de algoritmos de FTA : baseados em ATPG com sensibilização de um único caminho e baseados em ATPG com sensibilização concorrente de caminhos. Uma extensão dos algoritmos de FTA para circuitos que contém portas complexas descrito por [GÜN2000] foi apresentado.

Outra contribuição foi a abordagem do processo da Análise de *timing* Funcional sobre o espectro do teste, através de um estudo sobre a aceleração dos algoritmos de FTA. Um trabalho derivado deste estudo, foi uma avaliação do comportamento dos algoritmos de FTA em circuitos que contém portas complexas. Mais especificamente a implementação do algoritmo PODEM em conjunto com medidas de testabilidade para uma avaliação do desempenho destes circuitos. Foi possível evidenciar e provar que é possível o uso de medidas de testabilidade aplicadas em portas complexas utilizando a abordagem de macro expansão implícita. Os resultados podem fornecer uma escolha confiável de um caminho para atingir uma entrada primária, reduzindo assim, a possibilidade de se escolher um caminho que aumenta o número de ocorrências de *bakctrackings*.

Finalmente, origina-se deste trabalho, o algoritmo **DETA (Delay Enumeration-Based Timing Analisys)**. Após uma análise de algoritmos baseados em ATPG e o estudo de algumas técnicas de aceleração, foi implementado um algoritmo para FTA, o *DETA (Delay Enumeration-Based Timing Analisys***),** de acordo com a taxonomia apresentada em [GUN00], que classifica os algoritmos baseado na proposta de métodos para computar o atraso, classifica-se o DETA como um algoritmo baseado em ATPG com sensibilização concorrente de caminhos.

O DETA propõe uma solução para acelerar o processo de busca baseado em enumeração de atrasos. A fim de enumerar os atrasos, o algoritmo explora técnicas de aceleração baseadas em algoritmos de ATPG incorporando ao procedimento o fator temporal. A premissa básica do algoritmo consiste em justificar atrasos em nodos internos do circuito. De acordo com os experimentos relatados é possível afirmar que pode-se incorporar ao algoritmo DETA técnicas de aceleração dos algoritmos de ATPG. Neste caso, por exemplo, pode-se utilizar um procedimento que implementa um múltiplo *backtrace* ou então fazer uso de *headlines* durante a tomada de decisões. Entende-se que ao aplicar determinadas técnicas de aceleração os resultados obtidos pelo algoritmo pode ser refinados.

REFERÊNCIAS

[ABR 90] ABRAMOVICI, M.; BREUER, M.; FRIEDMAN, A. **Digital Systems Testing and Testable Design**. Piscataway, NJ: IEEE Press, 1990. 652p.

[BEN 84] BENNETS, R. G. **Design of Testable Logic Circuitos**. Reading, MA: Addison –Wesley, 1984.

[BEN 90] BENKOSKI, J. et al. Timing Verification Using Statically Sensitizable Paths. **IEEE Transanctions on CAD of Integrated Circuits and Systems**, Los Alamitos, CA, v. 9, n. 10, p. 1073-1084, Oct. 1990.

[BRA 88] BRAND, D.; IYENGAR, V. Timing Analysis Using Functional Analysis **IEEE Transactions on Computers**, New York, v.37 n.10 p.1309-1314, October 1988.

[BRG 84] BRGLEZ, F. et al. Applications of Testability analisys: From ATPG to critical path tracing. **IEEE Int. Test Conf**., **Proceedings...** [S.l.:s.n.], 1984. p. 705-712.

[CAD99] CADABRA. **CLASSIC-SC Automated Transistor Layout Tool**. Available at: <http://www.cadabradesign.com>. Visited on September 24, 1999.

[COR 90] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. **Introduction to Algorithms**. [S.l.]:McGraw-Hill, 1990.

[CHA 89] CHANDRA, S. J.; PATEL, J. H. Experimental Evaluation of Testability Measures for Test Generation. **IEEE Trans. On Computer-Aided Design**, Los Alamitos, California, v.8, n. 1, p. 93-98, Jan. 1989.

[CHA93] CHANG, Hoon; ABRAHAM, Jacob A. VIPER: An Efficient Vigorously Sensitizable Path Extractor. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 30., 1993, Dallas, Texas. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1993. p.112-117.

[CHE 91] CHEN, H.-C.; DU, D. Path Sensitization in Critical Path Problem. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p. 208-211.

[CHE 93] CHEN, H.-C.; DU, D. Path Sensitization in Critical Path Problem **IEEE Transanctions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.2,, p.196-207, February 1993.

[DET87] DETJENS, E.; GANNOT, G.; RUDELL, R. L. Technology Mapping in MIS. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1987. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1987. p.116-119.

[DEV 91] DEVADAS, S.; KEUTZER, K.; MALIK, S. Delay Computation in Combinational Logic Circuits: Theory and Algorithms. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p. 176-179.

[DEV 93] DEVADAS, S.; KEUTZER, K.; MALIK, S. Computation of Floating Mode Delay in Combinational Circuits: Theory and Algorithms. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.12, p.1913-1923, Dec. 1993.

[DEV 93a] DEVADAS, S. et al. Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.12, p.1924-1936, Dec. 1993.

[DEV 94] DEVADAS, S.; GHOSH, A.; KEUTZER, K. **Logic Synthesis**. New York: McGraw-Hill, 1994. 404p.

[DU 89] DU, David H. C.; YEN, Steve H. C.; GHANTA, S. On the General False Path Problem in Timing Analysis In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p. 555-560.

[FUJ 83] FUJIWARA, H.; SHIMONO, T. On the Acceleration of Test Generation Algorithms. **IEEE Transactions on Computers**, Los Alamitos, California, v.C-32, n.12, p. 1137-1144, December. 1983.

[GOL 79] GOLDSTEIN, L. H. Controllability/Observability Analysis of Digital Circuits. **IEEE Trans. On Circuits and Systems**, [S.l.], v. CAS-26, n. 9, p. 685-693, Sept. 1979.

[GOE 81] GOEL, Prabhakar. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. **IEEE Transactions on Computers**, Los Alamitos, California, v.C-30, n.3, p. 215-222, Mar. 1981.

[GÜN 98] GÜNTZEL, José L.; PINTO, Ana Cristina M.; REIS, Ricardo A. L. Improving Path Enumeration Accuracy by Considering Different Fall and Rise Gate Delays. In: WORKSHOP IBERCHIP, 4.,1998, Mar del Plata (Argentina). **Memorias...** Buenos Aires: IBERCHIP/Universidad Nacional de La Plata, 1998. p. 91-100.

[GÜN 98a] GÜNTZEL, José L.; PINTO, Ana Cristina M.; MORAES, Fernando; REIS, Ricardo A. L. An Improved Path Enumeration Method Considering Different Fall and Rise Gate Delays. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT DESIGN, SBCCI, 11., I998, Búzios. **Proceedings...** Los Alamitos (California): IEEE Computer Society, 1998. p. 208-211.

[GÜN 99] GÜNTZEL, José L.; PINTO, Ana Cristina M.; FRAGOSO, João L.; DALL PIZZOL, Guilherme; REIS, Ricardo A. L. Path Enumeration Algorithms for Timing Analysis of Digital Circuits. In: WORKSHOP IBERCHIP,5.,1999,Lima(Peru).Memorias... Lima: IBERCHIP/Pontificia Universidad Católica del Perú, 1999. p. 334-341.

[GÜN 2000] GÜNTZEL, José Luís. **Functional Timing Analysis of VLSI Circuits Containing Complex Gates**. 2000. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HSU 98] HSU, Y.-C.; CHEN, H.-C.; SUN, S.; DU, D. Timing Analysis of Combinational Circuits Containing Complex Gates. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 1998, Austin, Texas. **Proceedings...** Los Alamitos, California: IEEE, 1998. p.407-412

[KEU91] KEUTZER, K.; MALIK, S.; SALDANHA, A. Is Redundancy Necessary to Reduce Delay? **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.10, n.4, p.427-435, April 1991.

[KUK 97] KUKIMOTO, Y.; BRAYTON, R. Exact Required Time Analysis via False Path Analysis. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 34., 1997, San Jose, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1997.

[KUK 98] KUKIMOTO, Y.; BRAYTON, R. Hierarchical Functional Timing Analysis. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 35., 1998, San Francisco, California. **Proceedings...** Los Alamitos, California: IEEE Computer Press, 1998.

[LAM 94] LAM, W.; BRAYTON, R. **Timed Boolean Functions:** A Unified Formalism for Exact Timing Analysis. Norwell, MA: Kluwer Academic Publishers, 1994. 273p.

[MCG 89] MCGEER, P.; BRAYTON, R. Efficient Algorithms for Computing the Longest Viable Path in a Combinational Circuit In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Press, 1989. p.561-567.

[MCG 91] MCGEER, P.; SALDANHA, A.; STEPHAN, P.; BRAYTON, R.; SANGIOVANNI-VICENTELLI, A. Timing Analysis and Path Delay-

Fault Test Generation using Path-Recursive Functions. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1995. p.180-183.

[MCG91a] MCGEER, P. et al. Timing Analysis and Path Delay-Fault Test Generation using Path-Recursive Functions. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p.180-183.

[MCG 93] MCGEER, P. et al. Delay Models and Exact Timing Analysis. In: SASAO, T. (Ed.). **Logic Synthesis and Optimization**. Norwell, MA: Kluwer Academic Publishers, 1993. p. 167-189.

[MOR97] MORAES, F.; REIS, R.; LIMA F. An Efficient Layout Style for Three-Metal CMOS Macro-Cells. In: REIS, R.; CLAESEN, L. (Ed.). **VLSI**: Integrated Systems on Silicon. London: Chapman & Hall, 1997. p.415-426.

[NAG 75] NAGEL, W. **SPICE2, A Computer Program to Simulate Semiconductor Circuits**. Berkeley, California: University of California, Department of Electrical Engineering and Computer Sciences, 1975. 63p. (UCB/ERL M75/520).

[OUS85] OUSTERHOUT, John K. A Switch-Level Timing Verifier for Digital MOS VLSI, **IEEE Transactions on Computer-Aided Design**, Los Alamitos, California, v. CAD-4, n. 3, p.336-349, July 1985.

[PER 89] PERREMANS, S.; CLAESEN, L.; DE MAN, H. Static Timing Analysis of Dynamically Sensitizable Paths. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p.568-573.

[PES94] PESET LLOPIS, R. Exact Path Sensitization in Timing Analysis In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1994, Grenoble. **Proceedings...** Los Alamitos: IEEE, 1994. p.380-385.

[REI 97] REIS, André I. et al. Library Free Technology Mapping. In: REIS, R.; CLAESEN, L. (Ed.). **VLSI:** Integrated Systems on Silicon. Chapman & Hall, 1997. p. 303-314.

[ROT 66] ROTH, J. P. Diagnosis of Automata Failures: A Calculus and a Method. **IBM Journal of Research and Development**, [S.l.], v.10, n. 4, p. 278-291, July 1966.

[ROT 67] ROTH, J. P.; BOURICIUS, W. G.; SCHNEIDER, P. R. Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits. **IEEE Trans. on Eletronic Computers**, [S.l.], v.EC-16, n. 10, p. 567-579, Oct. 1967.

[SCH 67] SCHENEIDER, R. R. On the Necessity to Examine D-Chains in Diagnostic Test Generation. **IBM Journal of Research and Development**, [S.l.], v. 11, n. 1, p. 114, Jan. 1967.

[SIL 94] SILVA, João P.M.; SAKALLAH, Karem. Dynamic Search-Space Pruning Techniques in Path Sensitization. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 31., 1994, San Diego, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1994.

[SIL94a] SILVA, João P.M.; SAKALLAH, Karem. Dynamic Search-Space Pruning Techniques in Path Sensitization. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 31., 1994, San Diego, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1994.

[SIL 99] SILVA, Luís Jorge B. M. G. e. **Models and Algorithms for Timing Analysis of Combinational Circuits**. 1999. Tese (Mestrado) - Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisboa.

[YEN88] YEN, S.; GHANTA, S.; DU, D. A Path Selection Algorithm for Timing Analysis In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 25., 1988. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1988. p.720-723.

[YEN 89] YEN, S.; DU, D.; GHANTA, S. Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings…** Los Alamitos, California: IEEE Computer Society Press, 1989. p.649-652.

[YEN91] YEN, S.; DU, D.; GHANTA, S. Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis. **International Journal of Computer Aided VLSI Design**,[S.l.],v.3, n.2, p.193-215, 1991.

[WIS84] WISTON, Patrick H. **Altificial Intelligence**. 2nd ed. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984. 524p.