UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARGRIT RENI KRUG

Aumento da Testabilidade do Hardware com Auxilio de Técnicas de Teste de Software

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Dr. Marcelo Soares Lubaszewski
Orientador

Porto Alegre, maio de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Krug, Margrit Reni

Aumento da Testabilidade do Hardware com Auxilio de Técnicas de Teste de Software/ Margrit Reni Krug – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

102.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientador: Marcelo Soares Lubaszewski.

1.Teste de hardware 2. Teste de software 3. ATPG 4. Scan parcial. I. Lubaszewski, Marcelo Soares. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ATPG	Automatic Test Pattern Generator
ASIC	Application Specific Integrated Circuits
ATE	Automatic Test Equipament
BIST	Built-In Self-Test
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CF	Cobertura de Falhas
CFG	Control Flow Graph
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processor Unit
CR	Constant Replacememt
DD	Decision Diagram
DFT	Design for Testability
EDIF	Electronic *De*sign Interchange Format
FDG	Flow Dependence Graph
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HTG	Hierarchical Test Generation
IEEE	Institute of Electrical and Electronics Engineers
ITC	International Test Conference
LFSR	Linear Feedback Shift-Register
LOR	Logic Operator Replacement
RMHC	Random Mutation Hill Climber
RTL	Register Transfer Level
SCOAP	Sandia Controlability Observability Analysis Program
SDL	Structural Description Language
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration
VR	Variable Replacement

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O projeto, seja ele de software ou hardware, envolve uma série de atividades que, apesar das técnicas, ferramentas e métodos empregados, não estão livres de erros que podem levar ao mau funcionamento do produto final. Estes erros podem ocorrer durante a especificação do projeto, como também em estágios finais do desenvolvimento ou no processo de manufatura. A fim de minimizar prejuízos é necessário garantir a qualidade do sistema a partir da verificação do projeto, da validação de protótipo e do teste de fabricação.

Por muito tempo o teste de hardware e o teste de software foram estudados como disciplinas completamente independentes. Porém, similaridades entre o desenvolvimento de software e o projeto de hardware já foram exploradas com sucesso em adaptações de técnicas originalmente desenvolvidas para um sendo utilizadas por outro. Um exemplo é a cobertura de código, que foi inicialmente desenvolvida para o teste de software, e agora é comumente utilizada na verificação de hardware.

Visto que dispositivos são descritos em linguagem de descrição de hardware, e estas possuem características semelhantes às linguagens de programação, parece uma boa alternativa valer-se desta semelhança para utilizar os métodos propostos pela engenharia de software para garantir a qualidade do hardware desenvolvido. Utilizar tais métodos para gerar padrões de teste para dispositivos de hardware descritos em HDL (*Hardware Description Language*) e identificar nestas descrições características que, alteradas, aumentem a testabilidade dos mesmos, são os principais objetivos desta tese.

Palavras-Chave: Teste de hardware, teste de software, ATPG, parcial scan.

Hardware Testyability Increase with Software Testing Techniques

ABSTRACT

Both software and hardware designs require several tasks to increase reliability and ensure high quality of the final system. Although different techniques, tools and methods can be applied, error free products are difficult to be achieved. Errors may occur on design specification, on development stages and also during manufacturing process. To increase system quality and minimize costs it is mandatory to perform design verification, prototype validation and manufacturing test.

For a long time hardware and software tests were studied as disciplines completely apart. However, similarities between software development and hardware design have already been explored successfully by adapting techniques originally developed for one of them, and applying to the other. For instance, code coverage concept and methods were firstly developed for software testing, but nowadays are commonly used in hardware verification.

Due to the high similarity observed between software programming languages and hardware description languages (HDL), it seems to be a valuable approach applying software engineering techniques to help ensuring a high quality hardware device. Therefore, the main purpose of this thesis is to use such techniques to extract test patterns from HDL descriptions of hardware devices and to identify at these descriptions means to increase hardware testability.

Keywords: Hardware testing, software testing, ATPG, partial scan.

1 INTRODUÇÃO

Com o avanço da engenharia e da informática, os sistemas computacionais passaram a executar tarefas cada vez mais complexas, e passou-se a exigir cada vez mais qualidade dos produtos que chegam ao mercado. Para assegurar esta qualidade, a etapa de teste torna-se indispensável no projeto e implementação de um sistema [LEE 96], que possui elementos de software e de hardware. No que diz respeito ao software, este necessita ser bem planejado, bem projetado, codificado e testado [SHA 2001]. Fundamentalmente, as técnicas de teste de software têm o objetivo de detectar erros no código do programa [MYE 79]. Já considerando o hardware, este deve executar corretamente sua função, apresentar desempenho compatível com o especificado e ser confiável. Para assegurar estas características, deve-se garantir que, na etapa de projeto, não sejam embutidos erros e que, na etapa de fabricação, não haja a manifestação de defeitos que venham a comprometer a funcionalidade do dispositivo. Os métodos de teste de hardware objetivam detectar defeitos físicos (modelados como falhas lógicas) em dispositivos manufaturados. O teste é, portanto, uma atividade fundamental dentro do processo de desenvolvimento de um sistema e de extrema importância para a garantia de qualidade do projeto (software e hardware) desenvolvido. Segundo Pressman [PRE 2000], o teste é mesmo um elemento crítico para a garantia da qualidade de sistemas.

Até então, quando se pensava em teste de dispositivos de hardware, o que vinha em mente era a aplicação de técnicas específicas para a geração e a aplicação de teste. O crescimento da complexidade de circuitos digitais terminou impondo o uso de geradores automáticos de padrões de teste (ATPG – *Automatic Test Pattern Generator*), cujo objetivo principal é o de encontrar valores de entrada para os circuitos que possibilitem a detecção de falhas físicas advindas de sua fabricação. Em termos de aplicação do teste, este paradigma acarretava, por exemplo, em embutir código no sistema computacional para exercitar dispositivos de hardware ou desenvolver, de maneira *ad-hoc*, outros componentes de hardware específicos para realizar o teste.

Com a automatização do projeto de hardware, as ferramentas computacionais passaram a permitir a descrição destes sistemas como programas de software, utilizando-se linguagens de descrição de alto nível, como VHDL [ASH 98] [PER 91] e Verilog [PAL 2003]. Conseqüentemente, uma falha de projeto, a qual afeta a especificação do sistema, pode ser considerada como uma falha de software, por isso a necessidade de se pensar em verificar, validar e planejar o teste do hardware o mais cedo possível.

Considerando a similaridade entre as descrições HDL (*Hardware Description Language*) e as linguagens de programação tradicionais, vários trabalhos surgiram com o intuito de adotar no projeto do hardware, técnicas utilizadas para o teste de software, principalmente para a etapa de verificação de projeto. A tarefa de verificar se o projeto do hardware está correto pode ser feita, dentre outras maneiras, através de simulação, sendo realizada a partir de uma seqüência de teste e comparando respostas de simulação com respostas corretas (resultados esperados). Vários métodos de teste existem e são usualmente utilizados para verificar, em software, se os casos de teste ativam ou não uma dada instrução e/ou trocam o valor de uma variável de saída do programa [MYE 79], daí a imediata aplicação ao caso da verificação de hardware baseada em simulação.

Outro aspecto importante é que, tanto em teste de software como de hardware, características de controlabilidade e observabilidade são essenciais para determinar o quão difícil será gerar um conjunto de teste para obter alta cobertura de falhas. A este conjunto de teste chama-se casos de testes, no teste de software, e vetores de teste, no teste de hardware. Torna-se importante, então, a identificação de pontos de baixa observabilidade e controlabilidade no sentido de alterar o projeto (hardware ou software) para torná-lo mais facilmente testável.

Neste contexto, o objetivo deste trabalho consiste em conduzir um estudo para avaliar em que medida as técnicas apresentadas pela engenharia de software podem contribuir com o processo de teste de hardware, com o intuito de auxiliar os desenvolvedores de circuitos eletrônicos no processo de geração de vetores de teste e na identificação de pontos do circuito com baixa testabilidade. A partir de descrições VHDL, em um nível de abstração mais elevado do que o atualmente utilizado (nível de portas lógicas), quer-se antecipar tarefas importantes do processo de teste antes da implementação física do circuito, desvinculando a etapa de preparação do teste da de síntese do circuito.

Mais especificamente, os objetivos desta tese são:

- Acelerar o processo de preparação do teste, utilizando os vetores gerados através da aplicação de técnicas de teste de software como um conjunto de dados inicial para a ferramenta de ATPG, a qual pode partir destes vetores para gerar os padrões de teste necessários para detectar as falhas não detectadas pelo primeiro conjunto;
- Utilizar a controlabilidade favorecida pela aplicação das técnicas de teste de software para aumentar a testabilidade dos dispositivos descritos em VHDL, e
- Identificar partes do código que possuem influência na observabilidade do circuito, e a partir das informações obtidas realizar alterações no circuito que venham a aumentar sua testabilidade (inserção de cadeias *scan*).

Sendo objeto deste trabalho valer-se das facilidades e vantagens da engenharia de software para auxiliar no processo de teste de hardware, serão expostos no capítulo 2 características e conceitos relativos ao projeto de hardware utilizando linguagens HDL, ao mesmo tempo em que relaciona as etapas de desenvolvimento de um software às etapas de projeto de um dispositivo de hardware. No capítulo 3, conceitos de teste de software serão apresentados. Fundamentos e ferramentas utilizadas para testar um dispositivo de hardware serão apresentados no capítulo 4.

A aplicação de técnicas de teste de software diretamente sobre descrições de hardware para o teste de fabricação, sem que o mesmo sofra alterações, será apresentada

no capítulo 5. No capítulo 6 será apresentada uma abordagem que aplica o teste de software para identificar elementos seqüenciais que possuem pouca observabilidade, com o objetivo de modificar o circuito para inserir *scan* parcial a partir dos *flip-flops* identificados. No capítulo 7 serão apresentadas as conclusões obtidas durante o desenvolvimento desta tese.

2 PROJETO DE HARDWARE UTILIZANDO HDLS

2.1 Desenvolvimento de Software

Pode-se definir o software como sendo um conjunto de instruções que, quando executadas, produzem a função e o desempenho desejados, através de estruturas de dados que permitem que as informações relativas ao problema a ser solucionado sejam manipuladas adequadamente. A Figura 2.1 mostra um esquemático das etapas de desenvolvimento de um software.

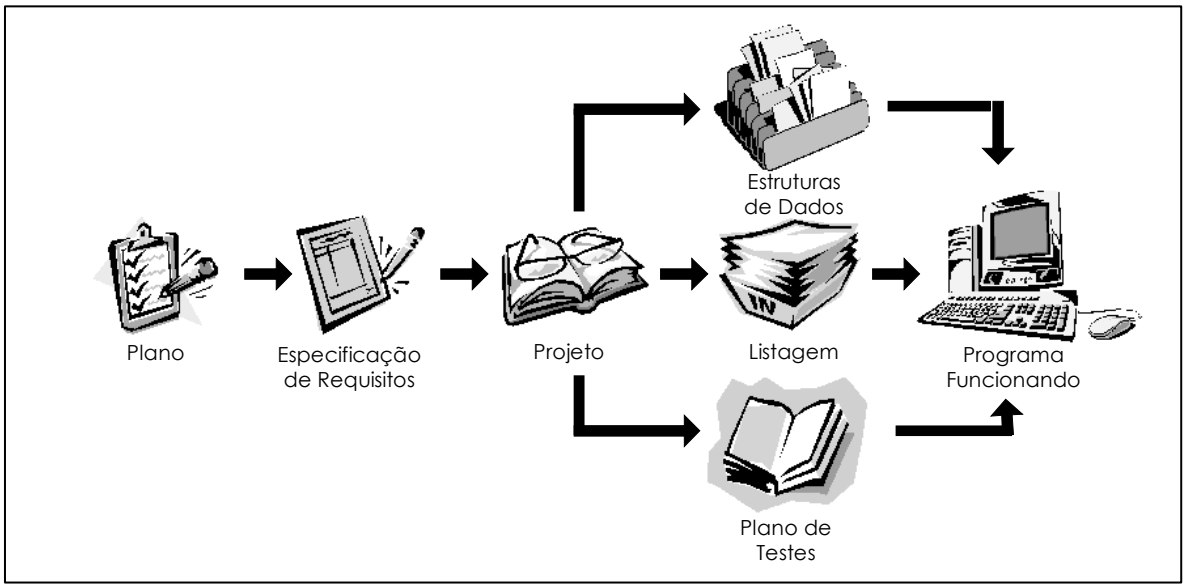

Figura 2.1: Etapas de desenvolvimento de um software

2.1.1 Modelos de Desenvolvimento de Software

Um modelo de desenvolvimento corresponde a uma representação abstrata do processo que geralmente define como as etapas relativas ao desenvolvimento do software serão conduzidas e inter-relacionadas para atingir o objetivo, ou seja, um produto de software de alta qualidade a um custo relativamente baixo. De um modo geral, pode-se organizar o processo de desenvolvimento de um software a partir de três grandes fases: a fase de definição, a fase de desenvolvimento e a fase de manutenção.

2.1.1.1 Fase de Definição

A fase de definição está associada à determinação do que vai ser feito. Nesta fase, o desenvolvedor deve identificar as informações que deverão ser manipuladas, as funções que serão processadas, qual o nível de desempenho desejado, quais as interfaces que devem ser oferecidas, as restrições do projeto e os critérios de validação. Isto terá de ser feito não importando o modelo de desenvolvimento adotado para o software e independente da técnica utilizada pra fazê-lo. Esta fase é caracterizada pela realização de três etapas específicas:

- Análise do sistema, a qual vai permitir determinar o papel de cada elemento (hardware, software, equipamentos e pessoas), cujo objetivo é determinar, como resultado principal, as funções atribuídas ao software;
- Planejamento do projeto de software, no qual, a partir da definição do escopo, será feita uma análise de riscos e a definição dos recursos, custos e a programação do processo de desenvolvimento, e;
- Análise de requisitos, que vai permitir determinar o conjunto das funções a serem realizadas, assim como as principais estruturas de informação a serem processadas.

2.1.1.2Fase de Desenvolvimento

Esta fase é responsável por determinar como realizar as funções do software. Aspectos como: a arquitetura do software, as estruturas de dados, os procedimentos que serão implementados, a forma como o projeto será transformado em linguagem de programação, a geração de código e os procedimentos de teste, devem ser encaminhados. Normalmente, esta fase é também organizada em três principais etapas:

- Projeto de software, através de representações gráficas, tabulares ou textuais, os requisitos do software definidos na fase anterior são representados, permitindo definir, com um alto grau de abstração, aspectos do software como a arquitetura, os dados, algoritmos e características da interface;
- Codificação, mapeamento em uma ou em várias linguagens de programação os passos realizados na etapa anterior, e;
- Testes de software, onde o programa obtido será submetido a uma bateria de testes para verificar (e corrigir) defeitos relativos às funções, lógica de execução, interfaces, entre outras.

2.1.1.3Fase de Manutenção

A fase de manutenção, que se inicia a partir da entrega do software, é caracterizada pela realização de alterações de naturezas mais diversas, seja para corrigir erros residuais da fase anterior, para incluir novas funções exigidas pelo cliente, ou para adaptar o software a novas configurações de hardware. Sendo assim, pode-se caracterizar esta fase pelas seguintes atividades:

- Manutenção corretiva, a qual consiste da atividade de correção de erros observados durante a operação do sistema;

- Manutenção adaptativa, a qual realiza alterações no software para que ele possa ser executado sobre um novo ambiente (CPU, arquitetura, novos dispositivos de hardware, novo sistema operacional, etc.);
- Melhoramento funcional, onde são realizadas alterações para melhorar alguns aspectos do software, como por exemplo, o seu desempenho, a sua interface, a introdução de novas funções, etc.

2.2 Desenvolvimento de Hardware

O processo de desenvolvimento de hardware não é muito diferente do desenvolvimento de sistemas de software, como já mencionado anteriormente. Uma estrutura de hardware pode ser escrita em uma linguagem de alto nível. Descrever um circuito como um esquemático digital é também possível, mas isso é bem menos popular e mais complexo que utilizar ferramentas baseadas em linguagens.

A principal diferença entre o projeto de hardware e de software encontra-se na forma como o desenvolvedor pensa para resolver um problema. Desenvolvedores de software tendem a pensar seqüencialmente, mesmo quando estão desenvolvendo aplicações multitarefa. As linhas de código sempre são escritas para serem executadas em uma ordem, pelo menos dentro de uma tarefa em particular. Mesmo havendo um sistema operacional usado para criar a aparência de paralelismo, ainda há um núcleo de execução para o controle. Durante o projeto de hardware os projetistas precisam pensar, e programar, em paralelo. Todos os sinais são processados em paralelo, pois trafegam através de um caminho de execução próprio até o sinal de saída. Dessa forma, a descrição do hardware cria estruturas que podem ser "executadas" todas ao mesmo tempo.

Além disso, existem outras diferenças elementares entre um software e um hardware [PRE 2000]:

- O software é desenvolvido, ou passa por um processo de engenharia, não é manufaturado no sentido clássico, e;
- O software não sofre desgaste, isto é, o software não é suscetível a defeitos físicos assim como é o hardware. O hardware, por sua vez, pode apresentar defeitos de fabricação e durante sua vida devido aos efeitos cumulativos de poeira, vibração, temperaturas extremas, etc. Entretanto, quando o hardware se desgasta este pode ter um de seus componentes substituído por um componente sobressalente. Para o software, onde não existem componentes sobressalentes, este processo envolve a manutenção do software que é consideravelmente mais complexa que a manutenção do hardware.

Existem várias formas de modelar um dispositivo de hardware, através da interconexão dos seus componentes, o que é chamado de modelo estrutural, ou definindo suas entradas e saídas através de procedimentos, o que é chamado de modelo funcional. O uso de descrições funcionais em linguagens de descrição de hardware possibilita o uso de construções de alto nível, assim como sentenças *if-then-else* e *case* para descrever condições e sentenças *for* e *while* para descrever laços.

Tipicamente, a etapa de início do projeto é seguida ou compartilhada com períodos de simulação funcional. Este é o momento onde um simulador é utilizado para a execução do projeto para confirmar se estão sendo produzidas saídas corretas para um

conjunto de entradas. Embora problemas como tamanho ou sincronismo possam ainda alterar isto mais tarde, o projetista pode, pelo menos, certificar-se que a sua lógica de funcionamento está correta antes de passar para o próximo estágio de desenvolvimento.

A etapa de compilação do projeto só ocorre depois que a representação funcional do hardware estiver correta. Esta compilação é realizada em duas etapas. Primeiro uma representação intermediária do projeto do hardware é produzida. Este passo é chamado de síntese (*synthesis*) e o resultado é uma representação chamada de *netlist*. O *netlist* é armazenado geralmente em um formato padrão, conhecido como EDIF (*Electronic Design Interchange Format*).

A segunda etapa do processo de tradução envolve traçar as estruturas lógicas descritas na *netlist* em macrocélulas, interconexões e pinos de entrada e saída.

2.2.1 Fluxo de Projeto

Segundo Nicolici [NIC 2000] e Norwood [NOR 97], o fluxo de um projeto VLSI (Very Large Scale Integration) é dividido em três passos: 1) especificação; 2) implementação, e; 3) fabricação. Observando a Figura 2.2, nota-se que o teste é considerado em todas as etapas do fluxo de projeto.

O teste é um método de avaliação do sistema digital, realizado através de um confronto entre a especificação do sistema e a sua implementação. A facilidade de realização do mesmo é expressa através do parâmetro testabilidade.

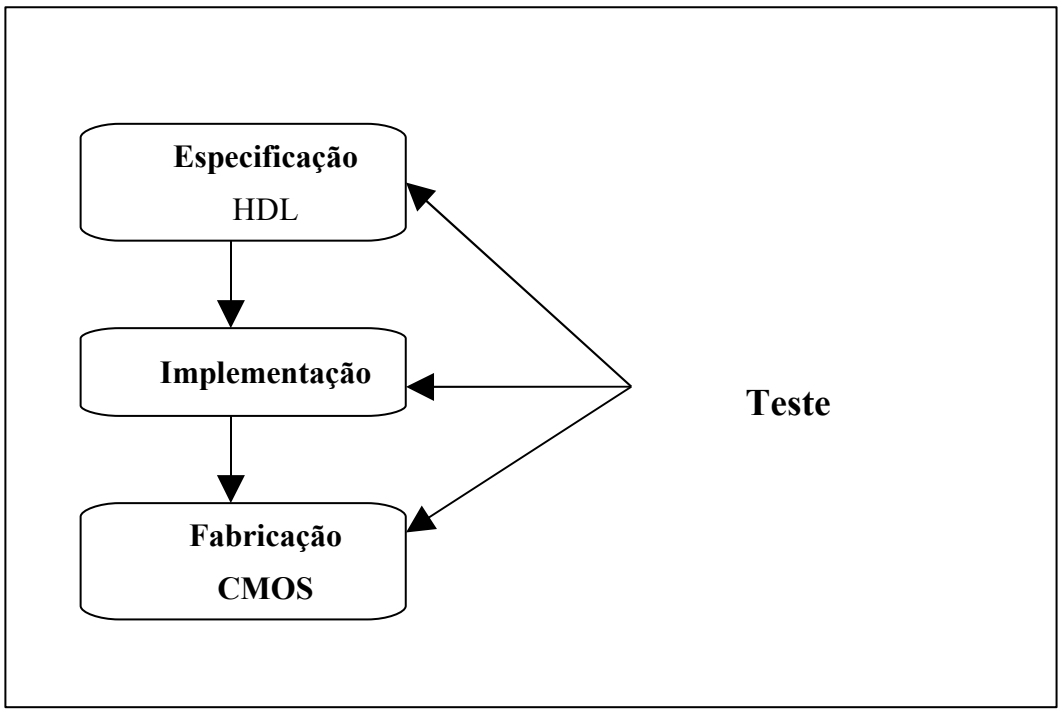

Figura 2.2: Fluxo de projeto VLSI

2.2.1.1Especificação

A etapa de especificação é responsável por descrever a funcionalidade do circuito. Nesta etapa se utiliza linguagens de descrição de hardware como: VHDL [ASH 98] [NAV 97] [PER 91], Verilog [THO 98] [PAL 2003] ou SystemC [GRÖ 2002], podendo ser descritas em dois domínios de projeto, o estrutural e o comportamental, e em diferentes níveis de abstração [NIC 2000].

No nível de abstração chamado lógico as representações são realizadas através de expressões booleanas no caso do domínio comportamental, e através de interconexões de portas lógicas, no domínio estrutural.

Aumentando-se o nível de abstração chega-se ao nível de transferência de registradores (RTL – *Register Transfer Level*). Neste nível o circuito é visto na forma de uma seqüência lógica, possuindo registradores e unidades funcionais.

O mais alto nível de abstração é o algorítmico, onde a especificação descreve abstratamente a funcionalidade do circuito.

2.2.1.2Implementação

Na etapa de implementação é realizada a geração de uma rede de componentes (*netlist*) que realiza as funções descritas na especificação. Pode-se utilizar duas metodologias de implementação: *full-custom* ou *semi-custom* [REI 2002]. O uso da metodologia *full-custom* requer um grande esforço de projeto, pois requer que cada característica do circuito seja detalhada. A metodologia *semi-custom* permite o uso de ferramentas CAD (*Computer Aided Design*) que capturam a especificação inicial e as transformam em uma implementação estrutural.

2.2.1.3Fabricação

Na etapa de fabricação, o *netlist* gerado na implementação é convertido para uma descrição que considera requisitos tecnológicos, tais como: área, energia consumida e freqüência, para então ser gerado o leiaute do circuito, o qual é uma representação geométrica do mesmo. Após a criação do leiaute, o projeto é enviado para uma fábrica para a construção física [SHE 98].

A fabricação é uma das etapas finais do fluxo de projeto de um circuito, resultando na conexão de transistores conforme a implementação e utilizando uma tecnologia de fabricação. A tecnologia de fabricação refere-se ao processo utilizado para a produção do circuito integrado, a qual pode possuir características como: tipo de semicondutor; tipo de transistor e detalhes da tecnologia do transistor.

2.2.2 Alto Nível de Abstração de Hardware

A complexidade do processo de projeto de hardware implica na decomposição hierárquica deste em um conjunto de passos de projeto. Um mecanismo fundamental para conduzir a decomposição de um projeto é o uso de abstração de informações. Abstração é o nome que se dá ao processo de representar um modelo através de um conjunto de informações limitadas aos aspectos relevantes para um dado tipo de manipulação.

Durante o projeto, um grande número de descrições com diferentes níveis de detalhamento são manipulados. O número de níveis de abstração manipulados durante o

projeto de sistemas complexos tem aumentado à medida que aumenta a complexidade dos mesmos. Estas características têm levado os projetistas a utilizarem ferramentas e métodos que facilitem esta tarefa. Com isso, linguagens de descrição de hardware, as quais permitem a descrição de um dispositivo em diferentes níveis de abstração (em nível algorítmico, RTL e em nível de porta lógica) vem sendo largamente utilizadas.

2.3 Linguagens de Descrição de Hardware

Estas linguagens são utilizadas pelos fabricantes de ferramentas para o projeto de sistemas eletrônicos (CAE – *Computer Aided Engineering*), como padrão para as entradas e saídas de suas ferramentas de simulação, síntese, leiaute, teste, etc. Utilizando-se tais ferramentas de descrição pode-se implementar o sistema de duas formas: circuito dedicado (ASIC – *Application Specific Integrated Circuits*) ou através do mapeamento em lógica programável.

As primeiras linguagens de descrição de hardware surgiram no final dos anos 60, desenvolvidas especialmente para o Departamento de Defesa dos Estados Unidos. Esta entidade comprava circuitos de aplicação específica e quando necessitavam atualização, muitas vezes a empresa desenvolvedora do circuito não existia mais. Para evitar isso pensou-se, então, em criar uma forma de adquirir ao invés de placas e componentes eletrônicos prontos, a descrição HDL dos mesmos, ou seja, passava-se a adquirir não mais o dispositivo físico, mas sim a descrição daquele hardware.

Até a década de 70 existiam diversas linguagens de descrição de hardware, porém não havia uma padronização entre elas. Isto dificultava a portabilidade das descrições já existentes, além de criar uma dependência muito forte com os desenvolvedores dos dispositivos, os quais, na maioria dos casos, utilizavam linguagens próprias. Em 1983 iniciou-se um programa chamado VHSIC (*Very High Speed Integrated Circuits*), do qual originou-se um grupo para a padronização IEEE [IEE 88] da linguagem de descrição de hardware VHDL (VHSIC *Hardware Description Language*).

Embora existam diversas linguagens para a descrição de hardware, como: Verilog, VHDL, SystemC, AHDL, etc., neste trabalho apenas a linguagem VHDL será estudada e analisada.

2.3.1 VHDL

VHDL é uma linguagem de descrição de hardware complexa que permite descrever circuitos em níveis comportamental e estrutural, ou uma combinação de ambos. A linguagem VHDL possui a vantagem, sobre as outras HDLs, de ser um padrão IEEE difundido no mundo inteiro.

A linguagem VHDL permite três níveis de descrição. O mais baixo nível de descrição é o nível estrutural. Neste nível, somente a arquitetura ou interconexão de cada módulo é conhecida. Este se torna similar ao projeto esquemático, exceto pelo módulo de interconexão que é descrito textualmente em vez de desenhado.

O segundo nível de descrição é o *dataflow* (comportamental). A descrição do circuito é feita em nível RTL. O circuito é descrito por seus registradores, barramentos, unidades funcionais e os movimentos completos dos dados. Sentenças executadas em nível *dataflow* são executadas de forma concorrente. As descrições VHDL utilizadas como estudo de caso nesta tese utilizam o nível RTL (*dataflow*), cujos detalhes de implementação serão vistos mais adiante.

O VHDL designa-se a suprir um grande número de necessidades do processo de desenvolvimento de hardware. Primeiro, permite a descrição da estrutura de um sistema, isto é, sua decomposição em subsistemas e como são realizadas as interconexões entre eles. Segundo, permite a especificação da função do sistema utilizando construções familiares de linguagens de programação. Terceiro, permite que o sistema seja simulado antes de sua fabricação (antes que o dispositivo físico seja confeccionado), possibilitando a identificação prévia de problemas. Quarto, permite detalhar a estrutura de um projeto a ser sintetizado a partir de uma especificação, possibilitando ao projetista se concentrar mais nas estratégias de decisão de projeto e reduzindo o tempo de desenvolvimento [ASH 98].

O uso da linguagem VHDL traz várias vantagens na especificação de sistemas digitais [LIP 89]:

- Portabilidade: facilita que diferentes grupos consigam utilizar descrições já existentes;
- O projeto torna-se independente da tecnologia, instanciada pelas ferramentas de síntese;
- Proporciona compatibilidade entre ferramentas de CAD;
- Facilita a manutenção, pois é muito mais fácil de compreender uma descrição VHDL, ao invés do esquemático de um dispositivo;
- Utiliza um nível maior de abstração e,
- Viabiliza simular o sistema implementado.

2.3.1.1Modelamento em VHDL

O modelo de um circuito em VHDL é hierárquico, sendo representado por uma entidade (*entity*). Esta pode conter entidades de nível hierárquico mais baixo que o seu e assim sucessivamente, sendo que o conjunto descreve a funcionalidade do circuito.

Uma entidade possui uma interface, a qual define os canais de comunicação desta com o meio externo. Uma das informações visíveis são as *ports*, que definem os sinais externos da entidade. A entidade descreve um componente como uma caixa preta, onde apenas as portas de entrada e saída são visíveis. As entradas, as saídas e o comportamento são conhecidos, mas não como este está estruturado internamente.

As entidades são os blocos básicos da linguagem VHDL para compor um projeto eletrônico. Uma entidade é qualquer componente VHDL que tenha comunicação com outras entidades, onde uma entidade com maior nível hierárquico corresponde a todo o projeto. Como exemplo de entidades, tem-se: portas lógicas, somadores ou mesmo um microprocessador.

A descrição operacional de uma entidade é feita através da arquitetura. A arquitetura é a parte da entidade que descreve o seu comportamento ou sua estrutura. Em outras palavras, é um conjunto de primitivas.

Após o processo de compilação do modelo, o mesmo pode ter seu comportamento simulado. Todas as declarações em VHDL são executadas em paralelo, assim como nos componentes de um circuito real, e a execução explícita para o paralelismo é feita pelo bloco *process*.

Em VHDL cada operação é definida como um processo (*process*) e os caminhos pelos quais os valores são passados ao sistema identificados por sinais (*signals*). Em VHDL, entende-se por sinal a ligação física entre as partes do circuito, como um fio; portanto um sinal pode assumir valores.

Um processo é uma abstração de um hardware que está sempre atuando [CAR 2001]. É basicamente o modelo de um componente físico, que possui uma lista de sinais dos quais ele depende, sendo sensível a sinais de entrada. Tem seu comportamento modificado conforme a descrição contida em sua estrutura.

Pode-se utilizar em uma descrição VHDL funções previamente definidas, isto se faz através do uso de bibliotecas(*library*). As vantagens do uso de bibliotecas de funções são as mesmas das demais linguagens de programação que utilizam este recurso, ou seja: permite a divisão de tarefas entre projetistas; facilita o aproveitamento de uma mesma função em mais de um projeto, entre outras. Além disso, através de bibliotecas pode-se adiar a decisão de qual versão de um componente a utilizar até o momento final de mapeamento [CAR 2001].

2.3.1.2Descrição Estrutural

Nas descrições estruturais são feitas declarações e instanciações de componentes. As declarações de componentes definem as interfaces dos sub-componentes utilizados na entidade. As instanciações de componentes permitem a criação de uma ou mais instâncias de um componente declarado e o estabelecimento das interconexões com os demais sub-componentes instanciados, para isso, utiliza sinais globais. Desta forma, pode-se descrever uma estrutura de forma hierárquica. Um exemplo de código estrutural, descrito na linguagem VHDL, pode ser visto na Figura 2.3.

```
-- Comparador de 4 bits
Entity comp4 is
Port(a, b       : in bit_vector(3 downto 0);
equals: out bit);
End comp4;
Use work.gates.all;
Arquitecture Estrut of comp4 is
Signal s bit_vector (0 to 3);
Begin
U0: xnor2 port map(a(0), b(0), s(0));
U1: xnor2 port map(a(1), b(1), s(1));
U2: xnor2 port map(a(2), b(2), s(2));
U3: xnor2 port map(a(3), b(3), s(3));
U4: and4 port map(s(0), s(1),s(2), s(3), equals)
End Estrut
```

Figura 2.3: Exemplo de uma descrição estrutural em VHDL

2.3.1.3Descrição Comportamental

O modelo comportamental pode ser definido como a interpretação funcional de um sistema. Esta forma de arquitetura descreve o comportamento lógico do circuito modelado. Em alguns sistemas automatizados ou em algumas linguagens, o comportamento não é acessível ao desenvolvedor exceto através de bibliotecas.

Em descrições comportamentais são declaradas estruturas de dados e algoritmos que operam sobre estas, determinando os valores de saída, em função da entrada das *ports*. As estruturas de dados são especificadas através da declaração de variáveis e sinais, e os algoritmos são descritos usando-se comandos de atribuição, seleção e repetição.

A Figura 2.4 mostra um exemplo de descrição comportamental VHDL. O circuito é modelado somente pelas características de entrada e saída, e os detalhes internos são ignorados. VHDL comportamental faz uso do bloco *process*, onde as execuções seqüenciais das instruções são definidas. Registradores são substituídos por variáveis; unidades funcionais, assim como subtratores e comparadores, são substituídas por operadores primitivos e subrotinas; e operadores de controle de programa, assim como *if* e *while*, são usados. Neste tipo de descrição também são permitidas entradas e saídas para arquivo e tela. Um código VHDL comportamental pode ser facilmente portado para uma linguagem de programação convencional, assim como Pascal ou C. Isto ocorre devido ao fato da linguagem VHDL basear-se fortemente na linguagem ADA [IEE 88]. Esta semelhança com outras linguagens de programação é que justifica a exploração de técnicas de teste desenvolvidas para o teste de software na geração do teste de dispositivos de hardware.

```
-- Comparador de 4 bits
Entity comp4 is
     Port(a, b       : in bit_vector(3 downto 0);
          equals: out bit);
End comp4;

Arquitecture comport of comp4 is
begin
Comp: process (a, b)
begin
     if a = b then
               equals < = ´1´;
     else
               equals <=´0´;
     end if;
end process Comp;
end
```

Figura 2.4: Descrição comportamental de um circuito em VHDL

2.4 A Linguagem VHDL e Outras Linguagens de Programação Tradicionais

Um processo pode ser visto como um programa, sendo construído por sentenças procedimentais e podendo chamar sub-rotinas, assim como é feito em linguagens de programação para software tradicionais, como Pascal e C.

Na linguagem VHDL é possível a criação de dois tipos de sub-rotinas: funções e procedimentos. A primeira difere da segunda pelo fato de retornar um valor. Conseqüentemente, as chamadas de uma e de outra também diferem entre si, pois devido ao fato da função retornar um valor, na sua chamada deverá se atribuir a uma variável o seu retorno. Fora esta distinção, os dois tipos possuem características similares. Já para a linguagem C só existe um tipo de sub-rotina, que é a função, com as mesmas características que na linguagem VHDL. Já no caso da linguagem Pascal a diferença entre procedimento e função é explicita. A linguagem C vai mais além, todas as estruturas que coloquialmente chamamos de comandos na verdade são chamadas de funções. Portanto, os programas são montados através da chamada de diversas funções.

Em VHDL, todos os processos em um modelo existem para ser executados concorrentemente. Por isso, um modelo VHDL é uma coleção de programas independentes rodando em paralelo. Talvez esta seja a maior diferença existente entre um programa descrito em linguagem de programação para software e uma descrição VHDL de um dispositivo de hardware

As linguagens VHDL, C e C++ são fortemente tipadas, pois realizam consistência de tipos. Por exemplo, não se pode atribuir o valor de uma variável do tipo *string* a uma variável do tipo *integer*.

A linguagem VHDL, assim como ADA, é concorrente, enquanto C, Pascal e C++ não. A concorrência permite a execução com mais de uma *thread* de controle.

A grande maioria de comandos existentes na linguagem de programação para software encontra-se em VHDL. Ambas possuem comandos de seleção (**if** e **case**), com construções bastante semelhantes, porém com propósitos diferentes após a compilação. Os comandos de iteração (laço) aparecem com maior freqüência e variedade em códigos escritos em linguagens de programação tradicionais.

Analisando as diferenças e semelhanças de ambas as linguagens, pode-se concluir que possuem estruturas bastante parecidas, ressaltando, é claro, a diferença do propósito de cada uma.

Com o surgimento das linguagens de descrição de hardware pode-se dizer que pessoas que modelavam hardware (projetistas), agora, fazem o papel de programadores.

Entretanto um erro comum ao escrever um circuito em VHDL é o de utilizar muitas variáveis e laços como em linguagens de programação de software. As ferramentas de síntese podem ser incapazes de gerar o circuito com estas características, ou se o circuito conseguir ser gerado pode estar incorreto ou ineficiente.

2.5 Considerações Finais

Atualmente, o projeto de dispositivos de hardware deve ser um processo: maduro, sistematizado, automatizado e viável economicamente. O crescimento da complexidade dos dispositivos e o aumento na demanda por aplicações computacionais críticas têm

gerado grande interesse no desenvolvimento de técnicas para melhorar a confiabilidade e qualidade de circuitos digitais resultantes deste processo.

A concepção de um sistema de hardware é normalmente dividida em etapas devido à complexidade desta tarefa. Cada uma destas etapas de síntese pode ser realizada através de ferramentas específicas. A síntese pode ser vista como uma seqüência de transformações incrementais aplicadas a uma descrição inicial da funcionalidade do circuito [REI 2002]. Pesquisas propõem e adotaram várias abordagens para o projeto de sistemas digitais, todas visando facilitar o processo de desenvolvimento.

Neste capítulo teve-se a oportunidade de verificar as diversas etapas de divisão do processo de projeto de dispositivos de hardware, confrontando-o com as etapas de desenvolvimento de um software, entre as quais as etapas de teste de hardware. Sobretudo, como o objetivo de motivar o estudo de técnicas da engenharia de software com o intuito de serem aplicadas sobre este nível do projeto de hardware, visando com isso: a independência da implementação física, o aumento da velocidade do processo de teste (especialmente na geração de vetores de teste), e; a possibilidade de fornecer subsídios para a localização de partes do circuito com baixa testabilidade através de métodos de software.

3 TESTE DE SOFTWARE

3.1 Introdução

Com a evolução da tecnologia e o crescente uso do computador nos últimos anos, surgiu a necessidade de produzir sistemas mais sofisticados e mais complexos, que exigem que o processo de desenvolvimento de software seja sistemático e rigoroso. Porém, o desenvolvimento de software, por mais cuidadoso que seja, está sujeito a erros. Para eliminar os erros que surgem na construção de um sistema aplica-se o teste de software [MYE 79] [PRE 2000].

Este capítulo tem por finalidade apresentar o estado da arte do teste de software, onde serão abordadas características de técnicas que atualmente são também utilizadas na validação de hardware, mas com maior detalhamento nos métodos adotados nesta tese, onde o foco principal encontra-se em facilitar e melhorar o processo de teste de dispositivos de hardware descritos em HDL.

3.2 Teste, Validação e Verificação no Contexto de Software

O processo de desenvolvimento de um sistema envolve uma série de atividades, e mesmo com o uso de métodos, técnicas e ferramentas, ainda podem permanecer erros no produto [PRE 2000]. A fim de minimizar esses aspectos são necessárias atividades que garantam a qualidade, entre elas: a verificação, a validação e o teste, visando reduzir a ocorrência de erros e os riscos associados.

Juntamente com o aumento da complexidade dos sistemas de software, os procedimentos de verificação e validação têm evoluído de forma a abranger o ciclo de vida completo de desenvolvimento do software. Os objetivos a que esses procedimentos se propõem são os de garantir que o projeto do software tenha sido realizado de acordo com os requisitos do usuário, a implementação esteja de acordo com o projeto, e que sejam atingidos os níveis de operacionalidade e confiabilidade esperada.

A validação do projeto de sistemas complexos é uma tarefa difícil e cara. Esta tem o objetivo de revelar falhas de especificação e codificação [NGU 2001], isto é, a validação assegura que o produto final corresponde ao que foi solicitado para o software. Então, validação refere-se ao processo de avaliar a qualidade de um software, observando se o mesmo está desempenhando o que lhe foi requerido, no sentido de atender às reais necessidades do usuário.

A especificação refere-se à funcionalidade de um projeto em particular, porém, algumas vezes, a solução proposta pode não atingir os objetivos esperados. Além disso, especificações são escritas por pessoas, e por essa razão, podem apresentar erros. Portanto, um sistema que alcança seus objetivos é útil, enquanto um sistema que é consistente com suas especificações é seguro. O objetivo do processo de verificação é avaliar a consistência de uma implementação com uma especificação, assegurando que o produto será completo e correto.

A validação refere-se a toda especificação do sistema e ao código final. Com respeito à especificação de todo sistema, a validação busca por discrepâncias entre necessidades atuais e a especificação do sistema como pretendido pelos projetistas, para assegurar que se construirá um programa que alcançará seus objetivos.

Entre as atividades de verificação e validação de software, a de teste continua a ter grande importância, pois dentre as atividades utilizadas para garantir a qualidade de um produto, o teste é uma das mais utilizadas, sendo de grande importância para a detecção e eliminação de erros ou defeitos [MAL 92].

Enquanto o termo teste é muitas vezes usado informalmente em ambas medidas (validação e verificação), as atividades diferem em objetivos e abordagens. O foco do teste está em primeiro lugar na verificação e depois na validação. O teste examina o comportamento do produto através de sua execução.

O teste envolve, basicamente, quatro etapas: 1) planejamento, 2) projeto de casos de teste, 3) execução e 4) avaliação dos resultados [BEI 90] [MYE 79] [PRE 2000].

A atividade de teste deve ser planejada e organizada a fim de revelar o maior número de erros com baixo custo. No entanto, o teste é uma atividade cara, chegando a atingir 50% dos custos do desenvolvimento de sistemas [HAR 2000]. Na tentativa de reduzir custos e melhorar a qualidade dos produtos desenvolvidos, técnicas e critérios que auxiliam na condução e avaliação do teste têm sido propostas. As técnicas de teste são, em geral, classificadas em funcional, estrutural e baseadas em falhas.

A diferença entre as técnicas de teste refere-se à origem da informação utilizada para avaliar ou definir o conjunto de casos de teste, sendo que cada técnica possui uma variedade de critérios para esse fim. Os critérios podem ser utilizados tanto para a geração de casos de teste, quanto para a avaliação destes.

Completar a tarefa de teste é, muitas vezes, impraticável ou impossível, por isso várias técnicas têm sido pesquisadas, visando minimizar o esforço enquanto maximiza a detecção de falhas e a confiança que o programa está correto. Dentre essas abordagens, selecionou-se para esta tese aquelas que são mais apropriadas ao teste de hardware e à geração automática de teste, permitindo com isso a integração entre essas duas áreas de conhecimento.

Outro termo que muitas vezes é utilizado como teste é a depuração, a qual refere-se a uma consistência não previsível do teste, isto é, refere-se a um exame do programa que executa passo a passo para verificação de cada parte do código e dos valores intermediários de variáveis. Quando um defeito é revelado este deve ser encontrado e corrigido. O teste é, portanto, o processo que, a partir de um conjunto de valores de entrada, estimula as variáveis do programa que, por sua vez, executa suas funcionalidades e por fim gera resultados, os quais são comparados com os resultados esperados para aquela aplicação.

Em suma, verificação e validação são duas atividades importantes para assegurar a confiança e qualidade do software. O teste de software e suas técnicas oferecem elementos de auxílio à verificação e validação.

3.3 Objetivos do Teste de Software

O principal objetivo da atividade de teste é aumentar a confiabilidade e garantir a qualidade dos produtos de software produzidos. Tem-se como principal aspecto do teste de software o de encontrar erros [MYE 79]. Erroneamente, pensa-se em teste como a forma de provar que o software está correto. Pensando desta forma o testador tende a produzir vetores de teste pobres, com poucas chances de encontrar erros. Caso contrário, tem-se um bom potencial de produzir vetores de teste para descobrir erros não detectados.

Dizer que o objetivo do teste de software é encontrar erros parece uma abordagem pessimista. Porém, provar que um programa está correto é considerado um problema não computável [MAL 92]. Então, como objetivos do teste de software tem-se:

- Encontrar erros não detectados;
- Mostrar que o programa executa as funções para as quais ele foi projetado, bem como verificar se o programa faz o que ele não deveria fazer e,
- Verificar se o usuário tem facilidade em utilizar o software (usabilidade).

Entre os desenvolvedores de aplicativos é comum dizer que os erros só aparecem quando o usuário vai utilizar o sistema, isto é, quando o mesmo já está em produção (utilização). Isto acontece porque o desenvolvedor ao realizar os testes o faz de forma viciosa, fazendo-os em cima das consistências existentes em seu programa, enquanto que o usuário, sob as mais variadas condições, irá utilizar o sistema para diversas situações e combinações, que em muitos casos não foram simuladas na fase de desenvolvimento do software. Portanto, deve-se destacar que o software deve estar preparado para qualquer situação, que muitas vezes o desenvolvedor acredita que pode ser impossível de ocorrer.

3.4 Testabilidade do Software

O conceito de testabilidade de um componente de software foi introduzido por Freedman [FRE 91] como uma combinação de controlabilidade e observabilidade, onde controlabilidade refere-se à efetiva cobertura do domínio de saída declarado pelo domínio de entrada, enquanto a observabilidade relata os efeitos que devem ser evitados (efeitos da não declaração de variáveis como módulos de entrada). Um componente é testável por domínio se este é controlável e observável. Conseqüentemente, um componente pode ser modificado para ser testável por um domínio pela adição de todas as variáveis usadas no componente em parâmetros de entrada e reduzindo as saídas declaradas para uso de domínios reais. Entende-se por domínio os valores possíveis que uma variável pode assumir de acordo com sua declaração (tipo de dado e tamanho).

A testabilidade do software possui algumas características que auxiliam ou influenciam no teste do software, entre elas:

- Operabilidade: refere-se ao fato do programa operar de maneira clara, ser de fácil operação;

- Observabilidade: refere-se à facilidade de visualizar os resultados, de identificar saídas incorretas e à precisão com que saídas são geradas para cada entrada;
- Controlabilidade: refere-se à capacidade do processo poder ser controlado e de os testes poderem ser automatizados e reproduzidos;
- Capacidade de decomposição: indica se módulos do software podem ser testados independentemente;
- Estabilidade: refere-se ao fato de poucas mudanças serem requeridas durante o teste, e;
- Compreensão: refere-se ao fato do programa ser facilmente entendido.

3.5 Erro, Defeito e Falha no Contexto do Teste de Software

Conforme a definição dos padrões da IEEE [MAL 2004] erros são definidos como uma ação não apropriada cometida pelo programador ou projetista. As falhas ou *bugs* são a manifestação e resultados dos erros durante a codificação de um programa. Um defeito ocorre quando erros ou falhas causam um resultado inesperado quando o programa está executando com uma certa entrada.

Um erro de software é uma discrepância entre a implementação e a especificação, isto é, ocorre quando as saídas do sistema não estão de acordo com o planejado. Os erros podem ser classificados em categorias, sendo elas:

- Funcionalidade: o programa não realiza o que se espera dele, refere-se diretamente à sua função;
- Comunicação: refere-se à interface do sistema, à forma como é feita a entrada de dados e à chamada dos módulos;
- Estrutura dos comandos: um determinado objeto realiza uma função que não é a sua;
- Desempenho: enquanto algum processo está sendo realizado, o cliente não possui uma interface amigável que lhe diga o que está ocorrendo;
- Erro no tratamento de erros: ocorre quando o programa não possui dispositivos que avisem quando alcançou seu valor limite; o sistema sinaliza um erro de maneira antecipada, sem a certeza que o mesmo venha a ocorrer; rotinas de tratamento de erros com defeito, por exemplo, levam à não emissão de mensagem de erro em situações onde os mesmos ocorrem;
- Erros de limite: relacionam-se a tamanhos de variáveis, temporização de alarmes, entre outros;
- Erros de cálculo: erros de lógica ou utilização de comandos incorretos da linguagem de programação;
- Erros iniciais: ocorrem quando o programa roda a primeira vez por falta de inicialização de variáveis, por exemplo;
- Erros finais: ocorre quando em uma rotina os valores são sempre inicializados, nunca sendo atualizados;

- Controle de fluxo: relaciona-se diretamente com um erro de lógica de programação, onde após um desvio é executado um comando que não deveria, ou então ocorre uma execução infinita de um determinado módulo do programa(*loop*);
- Erros de interpretação de dados: ocorrem devido a problemas de formato de dados(tipo do dado), manifestando-se em cálculos, comunicação com arquivos ou outros módulos do programa;
- Erros na relação com o hardware: ocorrem quando os programas ignoram sinais de erro emitidos pelo hardware, assim como: *timeouts*, sinais de *busy*, etc.;
- Erros de teste: ocorrem quando comete-se erros na execução dos testes, e;
- Versões: ocorrem quando se disponibiliza documentações e módulos que não conferem com outros módulos do sistema.

Já as falhas, manifestam-se de diferentes maneiras. Elas podem ser resultado de erros de codificação, de especificação ou de omissão. A Figura 3.1 [JOH 95] mostra um trecho de código com uma especificação correta. As Figuras 3.2, 3.3 e 3.4 exemplificam como cada falha se manifesta, respectivamente: erro de codificação, especificação e omissão, a partir da especificação vista na Figura 3.1.

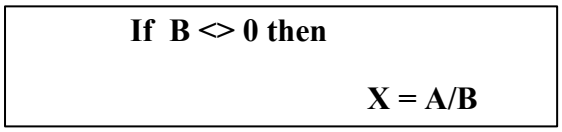

Figura 3.1: Código correto

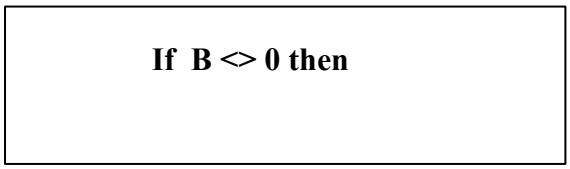

Figura 3.2: Código com erro de codificação

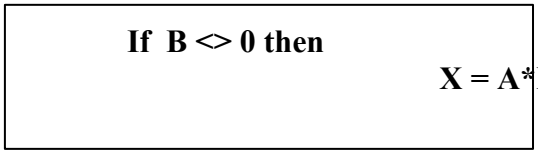

Figura 3.3: Código com erro de especificação

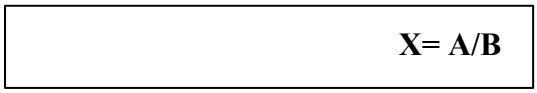

Figura 3.4: Código com erro de omissão

Na Figura 3.2 pode-se ver que um erro de codificação faz a divisão de *B* por *A* ao invés de *A* por *B*, sendo que a especificação era de executar a divisão de *A* por *B*. Na Figura 3.3 vê-se que a especificação apresenta-se incorreta, pois foi realizada uma multiplicação de *A* por *B* ao invés de uma divisão de *A* por *B*. Finalmente, na Figura 3.4 vê-se que existe um erro de omissão, onde não foi tomado o cuidado com a divisão por zero.

O erro de codificação pode ser facilmente encontrado por qualquer conjunto de dados onde $A <> B$. O erro de especificação é detectado por uma pessoa que possui bom conhecimento da funcionalidade do módulo. O erro de omissão pode ser detectado somente se $B = 0$. Portanto, devido à natureza diversa das falhas, o problema do teste de software é dificultado.

O defeito refere-se a uma deficiência algorítmica que se ativada pode levar a uma falha, geralmente de origem humana. Os defeitos são gerados na comunicação e na transformação de informações. A maioria encontra-se em partes do código raramente executadas. Quanto antes o defeito for revelado menor será o custo da correção e maior a probabilidade de corrigi-lo.

3.6 Casos de Teste e Cobertura de Falhas do Software

O objetivo dos casos de teste é descobrir erros, utilizando-se de um critério com um mínimo esforço e tempo. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não detectado. As descrições devem identificar: estados do sistema antes da execução do teste; funções a serem testadas; valores de parâmetros para o teste e resultados esperados do teste.

Um dos aspectos mais importantes no teste de software é a identificação dos casos de teste ou vetores de teste. Devido ao fato de ser impossível provar que um programa não possui erros, a melhor estratégia de teste é identificar um conjunto de dados que possua uma alta probabilidade de descobrir o maior número de erros em um programa com a quantidade mínima de tempo e esforço [MYE 79].

O ideal, para ter-se a certeza que um programa está livre de erros, seria testar o sistema de maneira exaustiva. No entanto, o teste sendo realizado sob todas as condições de entrada resultaria em um número muito grande de possibilidades e em um custo bastante elevado. O teste exaustivo torna-se impraticável, devido às seguintes razões [WHI 97]:

- O software permite entradas variadas: existem vários tipos de dados para as entradas (inteiro, caracteres, real, etc.), criando um grande conjunto de valores possíveis que podem ser aplicados;
- As entradas do software podem ser seqüenciais: a ordem em que as entradas são aplicadas podem gerar um número muito elevado de combinações.

Para viabilizar o processo de teste utiliza-se a geração de dados de teste significativos e em número reduzido. Deve-se, porém, observar alguns detalhes na elaboração de vetores de teste:

- Os dados gerados devem atender aos requisitos do usuário, ou seja, eles devem espelhar o uso real do sistema pelo usuário;
- Deve-se evitar redundância nos testes, o que viria a tornar o teste mais caro e demorado, além de não contribuir com o processo;
- Deve-se ter consciência que é impossível testar tudo;
- As pessoas responsáveis pelo processo de teste de um programa (sistema) não devem participar de seu desenvolvimento, e vice versa. Além disso, estes devem ser treinados para esta função e na utilização de ferramentas apropriadas para esta tarefa e,

- O teste deve começar assim que os requisitos do usuário sejam definidos e concentrar-se nas menores unidades do programa (teste de unidade).

Um aspecto fundamental de um caso de teste refere-se à definição dos resultados esperados, estes também devem ser planejados para auxiliar na identificação de entradas inválidas e inesperadas (assinatura do sistema). Um caso de teste sempre corresponde à execução completa de um programa. A forma da geração dos vetores de teste depende do tipo de teste escolhido.

Em torno de 30% da tarefa do teste pode ser gasta com a composição dos casos de teste [CHI 99], se a mesma for realizada de maneira manual. Portanto, a automatização deste processo é de extrema importância. Isto se dá através de ferramentas computacionais, que utilizam alguma técnica de teste para a geração de vetores.

Diversas técnicas têm como objetivo a criação de casos de teste que permitem a avaliação do sistema sob teste e a identificação de possíveis erros. Neste contexto, casos de teste são compostos por: um conjunto de ações ou valores de entrada e por um conjunto de ações ou resultados esperados.

A cobertura é medida pela utilização de um programa para determinar como um conjunto de testes exercita o programa sob teste. Existem duas classes de cobertura: coberturas baseadas em caminho, as quais requerem a execução de componentes em particular de um programa, assim como sentenças, desvios, ou caminhos completos, e a cobertura baseada em falha, que requer que o conjunto de teste exercite o programa em um caminho que revele falhas prováveis.

3.7 Tipos de Teste de Software

Pode-se dividir em três tipos o teste de software: funcional, estrutural e baseado em falhas. A diferença entre eles está na origem da informação utilizada na criação dos dados de teste [HET 87].

Os tipos de teste não devem ser aplicados isoladamente. Os três tipos podem e devem ser utilizados para se complementarem.

3.7.1 Teste Funcional

Para o teste funcional, também chamado de teste de caixa preta (*black box testing*), a obtenção de vetores de teste ocorre a partir de uma análise realizada sobre a funcionalidade do programa, sem levar em conta a estrutura interna do mesmo [MYE 79] [PRE 2000]. Neste tipo de teste um conjunto de dados é selecionado a fim de verificar as funções do sistema [PRI 91].

Neste tipo de teste os vetores são injetados através das entradas do sistema e a identificação dos erros é feita a partir da comparação das saídas com valores previamente esperados. Portanto, caso exista a necessidade de algum dispositivo para auxiliar no processo, este será externo ao sistema.

Este tipo de teste baseia-se praticamente em identificar as funções para as quais o sistema foi criado e gerar vetores de teste para avaliar estas funções [TOM 97]. O teste pode ser aleatório ou pode usar a especificação para testar a funcionalidade do programa. Vetores aleatórios são criados antes do processo de teste e, por isso, rapidamente um grande número de vetores pode ser criado. Porém, um grande número de vetores de testes não garante bons resultados.

O teste baseado na especificação depende da pessoa que projetou o sistema. A automatização no nível de especificação não é prática, porque requer especificações muito formais. Este tipo de formalismo pode fazer a especificação assemelhar-se ao código do programa, o qual é considerado como sendo o produto final, não o início do processo de projeto. Escrever bons testes em nível funcional requer que o testador tenha uma boa experiência em linguagens de programação, devendo ter conhecimento sobre como os programas são codificados e os possíveis problemas que podem ocorrer durante esta tarefa.

Erros de interface, funções incorretas ou ausentes, erros de inicialização ou término, erros nas estruturas de dados ou em acessos externos são as categorias de erros mais comuns detectadas por este tipo de teste [PRE 2000].

O teste funcional de software se assemelha muito ao teste funcional de hardware, pois se preocupa apenas com a função, descuidando da estrutura interna e detendo-se no conhecimento das entradas que estimulam determinadas saídas. Caso algum elemento de saída difira do esperado, diz-se que um erro foi detectado.

Tanto para o teste funcional de software como para o de hardware, a fim de avaliar a cobertura existe a necessidade de um elemento externo que auxilie na tarefa de injeção de falhas e na de comparação de resultados obtidos com os esperados, o que eleva o custo de preparação do teste do sistema.

Outra característica deste tipo de teste é que ele é aplicado quando o sistema está praticamente pronto, o que pode dificultar a correção dos erros, pois quanto mais tarde é identificado um erro no ciclo de desenvolvimento de um software, mais caro fica para consertá-lo.

Por fim, o teste funcional deve ser aplicado juntamente com outras técnicas de teste, pois este se limita a identificar se o sistema desempenha adequadamente suas funções.

3.7.2 Teste Estrutural

O teste funcional não é completo na medida em que é praticamente impossível abranger todas as possibilidades de execução de um programa. O teste funcional não consegue detectar algumas situações, pois este se preocupa em verificar se as saídas estão de acordo com o esperado devido às entradas fornecidas. Portanto, se internamente existe um erro que não venha a interferir nas saídas do sistema, este não será identificado pelo teste funcional.

O teste estrutural, também chamado de teste de caixa branca (*white box testing*), baseia-se na estrutura interna do programa, depende totalmente da implementação do mesmo, e analisa aspectos relacionados aos fluxos de controle e de dados dos módulos do programa. Seu objetivo é detectar partes do código que não foram testadas pelos dados de teste [TOM 97]. A Figura 2.5 mostra um esquemático do teste estrutural em um sistema.

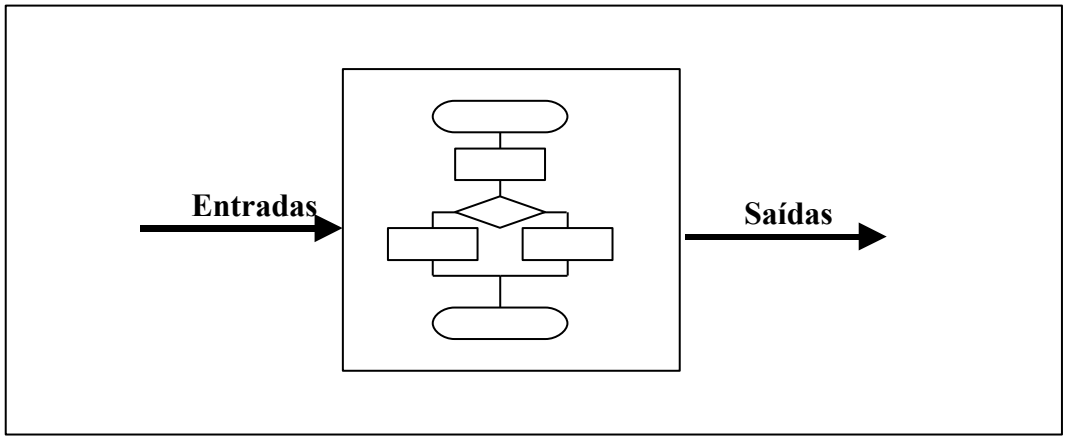

Figura 3.5: Esquemático do teste estrutural

No teste estrutural os testes são derivados do próprio código. Uma técnica usada é a leitura do código. Nesta o código é lido por completo cuidadosamente a procura de erros. Esta técnica produz bons resultados, mas tem custo elevado. Neste caso, o que se utiliza é a geração de teste em nível de especificação. No entanto, necessita-se que o pessoal envolvido no processo de teste tenha grande experiência técnica.

O segundo caso é o teste de software estrutural propriamente dito. Neste a estrutura de controle do programa é usada como base para desenvolver ou avaliar os testes. Desta maneira estratégias de teste incluem sentenças, desvios (*branch*) e caminhos de teste. O teste de sentença requer que cada sentença no programa seja executada pelo menos uma vez durante o teste. Para o teste de desvio cada controle de decisão deve ser executado de forma verdadeira e falsa pelo menos uma vez durante o teste. O teste de caminho requer que todos os caminhos do programa sejam executados.

A maior vantagem do teste estrutural é que este se presta para a automatização, ou seja, os dados de teste podem ser gerados automaticamente.

Os dados, para o teste estrutural, devem ser gerados para verificar: caminhos independentes, sendo que estes devem ser testados pelo menos uma vez; repetições com valores que ficam dentro dos limites e nas suas extremidades; decisões lógicas com situações falsas ou verdadeiras e estruturas de dados internas para garantir a sua validade [PRE 2000].

Os principais problemas da abordagem estrutural são: programas com repetições que possuem um número de caminhos infinito, já que a análise da estrutura interna do programa é feita estaticamente e, a existência de caminhos não executáveis no programa, que causam desperdício de tempo e recursos na tentativa de gerar vetores de teste que possam executar estes caminhos [VER 97].

A Figura 3.6 [JOH 95] mostra um exemplo onde **A-F** representam as sentenças, e **a-d** representam os predicados do programa. Um til (~), ou a falta de, antes da variável predicado é usado para representar a direção falsa ou verdadeira, completando uma condição. Os caminhos são descritos por predicados e são avaliados enquanto atravessam o caminho. Por exemplo, para atravessar o caminho que executa as sentenças **A**, **B**, **E** e **F**, os predicados são **a~bd**.

Os critérios que utilizam caminhos são poderosos, mas impraticáveis em algumas situações. No teste de caminho, já que cada caminho tem que ser atravessado, o número de caminhos pode, rapidamente, tornar-se enorme; para cada desvio condicional não

aninhado, o número de casos de teste requeridos é no mínimo o dobro. Na presença de *loops*, o número de caminhos pode ser infinito.

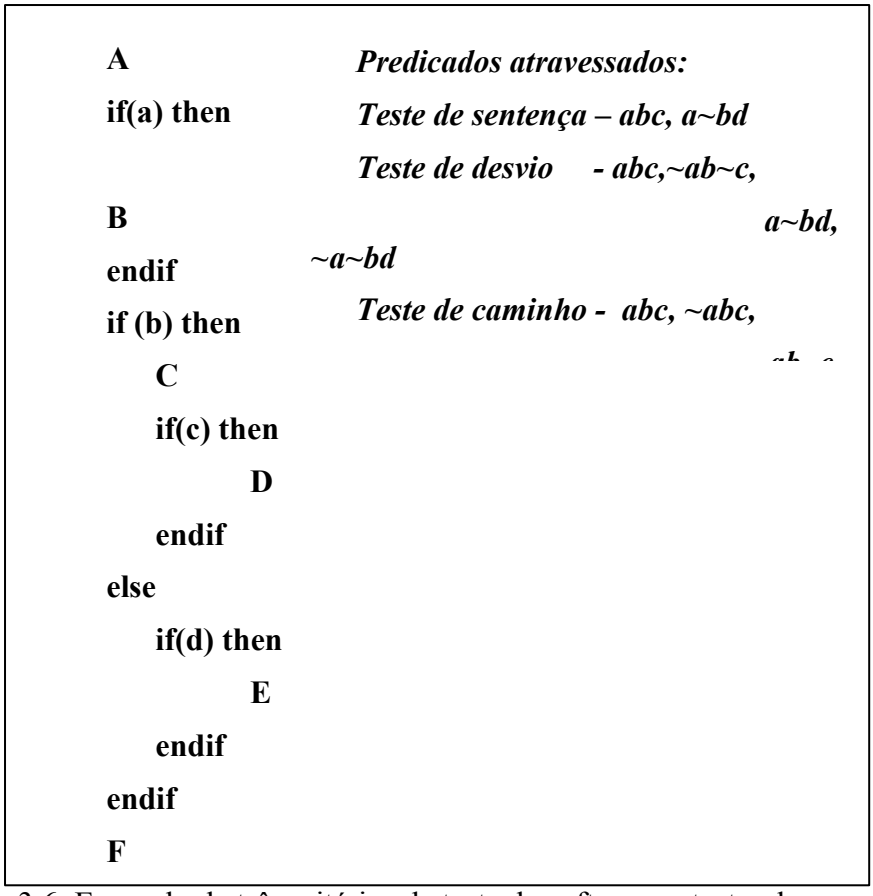

Figura 3.6: Exemplo de três critérios de teste de software estrutural

3.7.2.1Grafo de Programa

Como visto na seção anterior, para realizar o teste estrutural é necessária a seleção de caminhos que tem como objetivo cobrir o código ou sua representação gráfica, compondo os critérios de cobertura (ou critérios de seleção). Um critério é válido se a execução de pelo menos um dos vetores de teste detecta erros no programa.

Existem diversos critérios de cobertura, os quais se dividem em três categorias: teste de fluxo de controle, teste de fluxo de dados e *slicing*. Tanto o critério de fluxo de controle quanto o de fluxo de dados baseiam-se no conceito de grafo (ou grafo de programa).

O grafo de programa, também chamado de grafo de fluxo de controle, é um grafo onde os nodos representam comandos executáveis e as arestas representam o fluxo de controle. Uma aresta só existe entre dois nodos se um comando sucessor pode ser executado na seqüência de seu predecessor.

Os nodos do grafo são formados pelos comandos executáveis. Isto exclui declarações de variáveis, comandos que indicam início e fim de bloco.

A Figura 3.7 mostra as principais representações gráficas das estruturas de controle existentes nas linguagens de programação, utilizadas para a geração de grafos de

programa, são elas: a) comando seqüencial; b) estrutura *if-then*; c) estrutura *if-then-else*; d) comando *while-do*; e) comando *repeat-until*, e; f) sentença *case*.

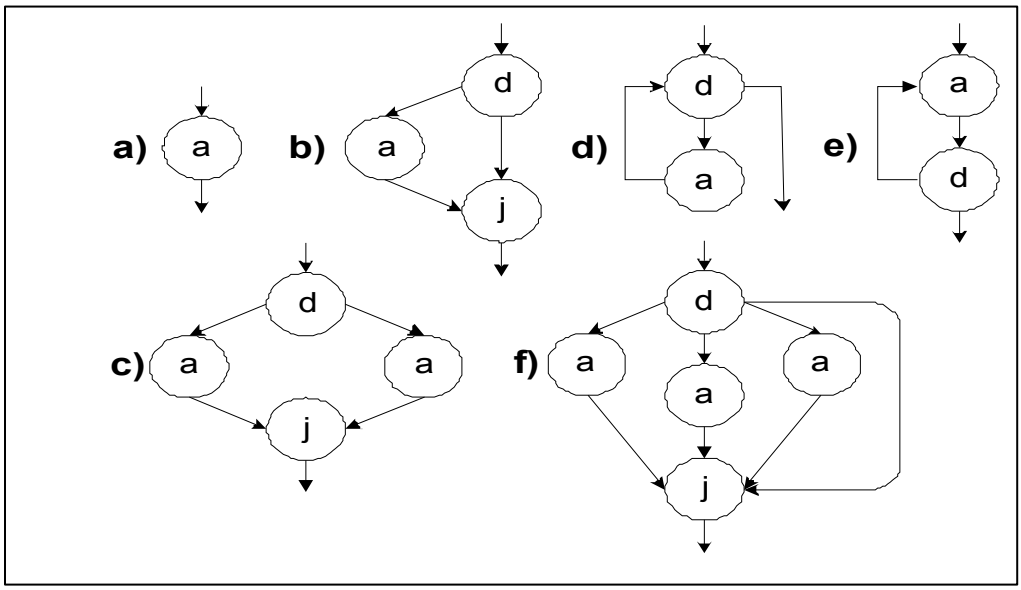

Figura 3.7: Principais estruturas de um grafo de programa

Como exemplo desta representação, na Figura 3.8 é montado um grafo a partir do programa escrito na linguagem de programação C, o qual identifica o tipo do triângulo, conforme os valores informados para cada um dos lados.

Os nodos de número 4 e 13 são, respectivamente, chamados de nodo inicial e nodo final do grafo. Quando um programa é estruturado deve possuir apenas um nodo inicial e um final.

Com o grafo do programa gerado definem-se com melhor precisão os vetores de teste que irão exercitar cada parte do programa.

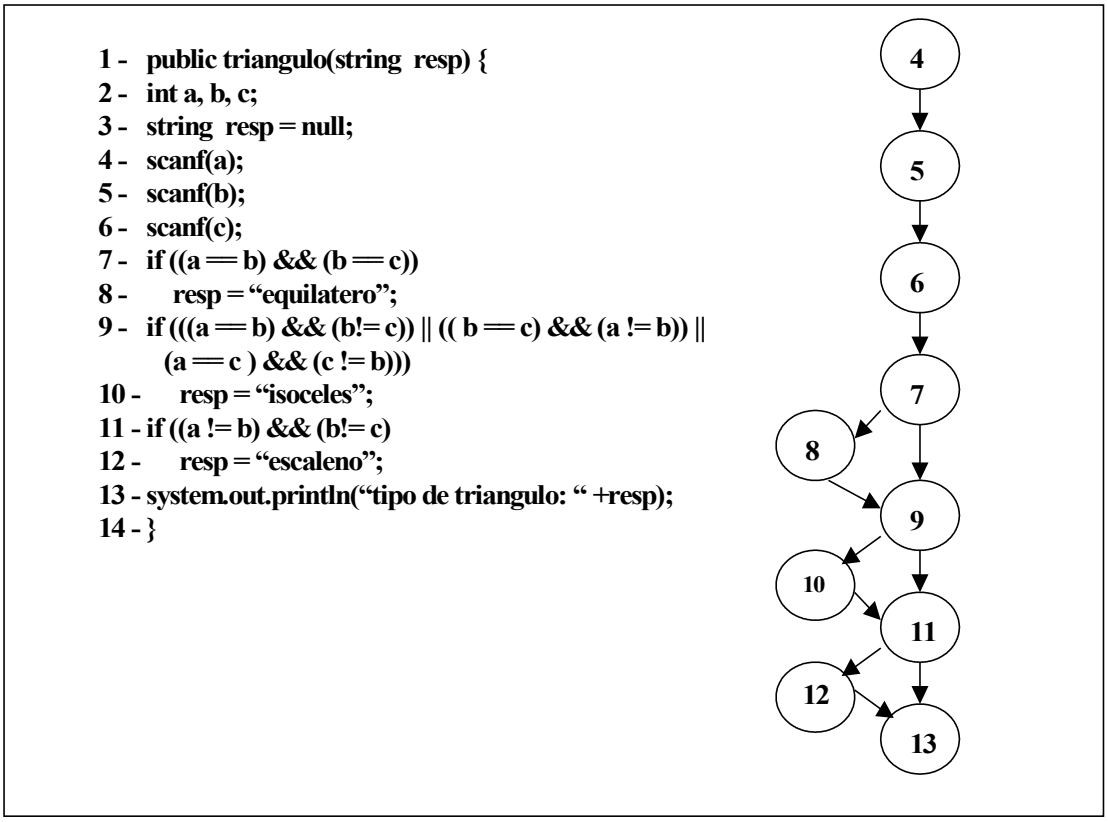

Figura 3.8: Grafo de programa

3.7.2.2Teste de Fluxo de Controle

Esta técnica, também chamada de teste de caminho, foi inicialmente proposta por McCabe [MCC 76]. O conjunto de caminhos a executar é derivado de uma medida da complexidade lógica de um projeto procedural. Estes vetores de teste têm a garantia de executar cada instrução do programa pelo menos uma vez durante a atividade do teste.

Para representar o critério de teste baseado no fluxo de controle, o código deve ser decomposto em blocos, de maneira que a execução do primeiro comando de um bloco desencadeie a execução de todos os outros comandos, em ordem, naquele mesmo bloco (grafo de programa). Todos os comandos possuem um único predecessor, com exceção do primeiro, e apenas um sucessor exceto o último.

Nas técnicas que utilizam o teste de fluxo de controle, os vetores de teste são gerados a partir do grafo de programa. Um caso de teste corresponde a um caminho completo no grafo de programa, ou seja, do nodo inicial ao nodo final.

Os vetores para as técnicas de teste de fluxo de controle são definidos de maneira que cada caso corresponda a um caminho diferente do grafo. De forma resumida, pode-se definir o teste de fluxo de controle através das seguintes etapas:

- Construção do grafo de programa;
- Determinação dos caminhos possíveis;
- Seleção dos caminhos para o teste, e
- Identificação dos resultados esperados a partir dos dados de entrada para os vetores de teste.

O teste de fluxo de controle, ou teste de caminho, agrupa um conjunto de estratégias que possuem características afins. A diferença entre elas encontra-se no tipo e nível de cobertura, e isto é feito através de critérios de cobertura.

Um critério de cobertura é uma medida de quanto um determinado conjunto de vetores de teste exercita um dado caminho. Existem diversos critérios de cobertura de caminho, cada uma dá origem a uma estratégia de teste de fluxo de controle, sendo elas: cobertura de comando, cobertura de condição, cobertura de condição múltipla, cobertura de repetição e complexidade ciclomática.

Na estratégia de cobertura de comando os vetores de teste são escolhidos de maneira que cada nodo do grafo de programa seja testado, isto é, cada comando do programa é executado pelo menos uma vez. A métrica de cobertura de comando conta os nodos cobertos.

A estratégia denominada cobertura de condição é um dos critérios mais utilizados para o teste de fluxo de controle. Nesta estratégia os vetores de teste são gerados com o intuito de cobrir cada saída possível de cada nodo (comando) onde ocorre uma decisão. Isto equivale a cobrir cada aresta do grafo de programa ao invés de cada nodo. A métrica utilizada para esta cobertura conta as arestas cobertas.

A cobertura de condição implica em uma cobertura de comando. No entanto, a cobertura de comando não necessariamente abrange a cobertura de condição.

O critério que utiliza a cobertura de condição múltipla realiza a geração dos vetores de teste para cada combinação possível de valores das sub-condições (verdadeiras e falsas). Então vetores de teste para cada linha da tabela verdade de cada nodo que possui uma decisão são gerados.

A cobertura de repetições (*loop*) tem como objetivo testar as repetições e recursividades existentes em um programa. Estes dois tipos de estrutura aumentam a complexidade dos programas, devido ao fato da possibilidade de situações de execuções infinitas (*loops*). Cada comando de repetição (*while*, *do while*, *repeat*, etc.) introduz um ciclo no grafo de programa, como pode ser visto na Figura 3.9. Visto que um grafo cíclico possui infinitos caminhos, torna-se inviável testar todos os caminhos gerados.

O critério de **complexidade ciclomática** [MCC 76] é utilizado para comparar a complexidade de programas através do seu grafo. Para o teste estrutural a complexidade ciclomática fornece um número de caminhos independentes do grafo.

Para calcular a complexidade ciclomática de um grafo de programa, necessita-se transformar o grafo em um grafo fortemente conectado; isto é feito acrescentando uma aresta ligando cada nodo com o nodo final (Figura 3.10).

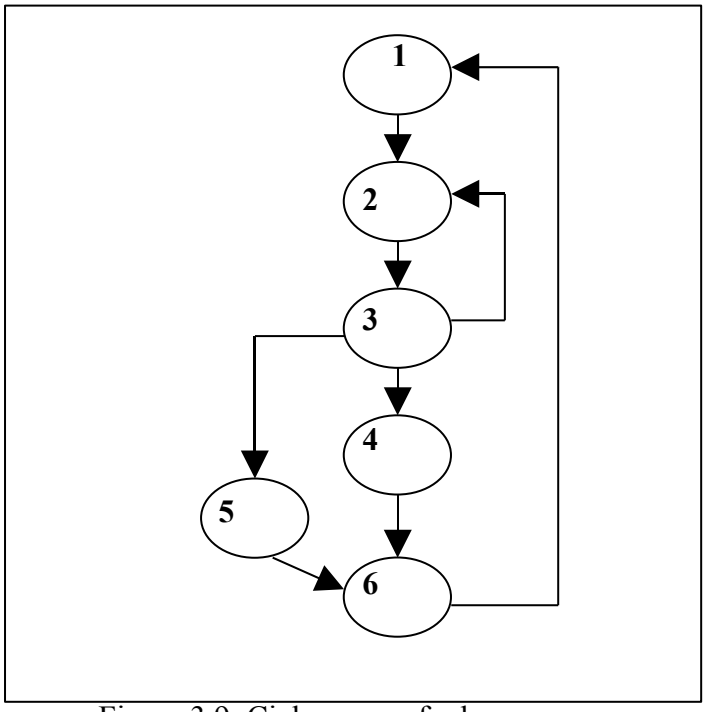

Figura 3.9: Ciclos no grafo de programa

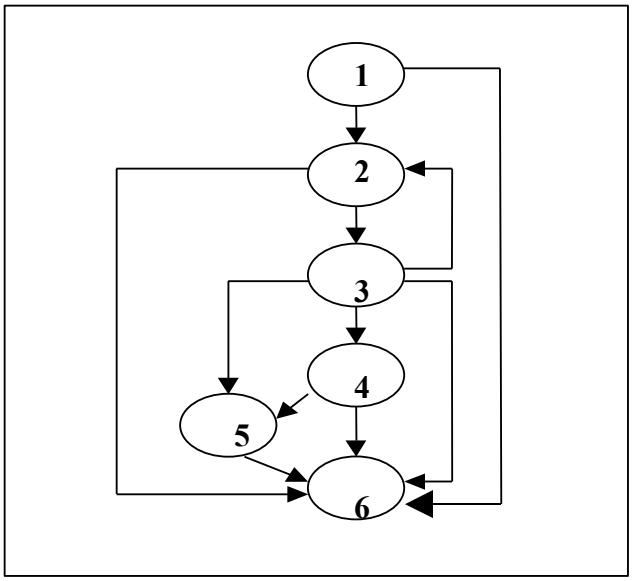

Figura 3.10: Grafo fortemente conectado

O número ciclomático é obtido através da fórmula **V = E – N + P**, onde **E** corresponde ao número de arestas do grafo, **N** ao número de nodos e **P** ao número de componentes (**P = 1** quando o programa é estruturado). O número ciclomático correspondente ao grafo de programa da Figura 3.10 é 6.

3.7.2.3Teste de Fluxo de Dados

Este tipo de teste gera vetores analisando o comportamento das variáveis do programa e através do grafo de programa procura cobrir os caminhos do grafo que correspondem a diferentes combinações de estados e usos das variáveis.

A análise do fluxo de dados preocupa-se com as relações entre definição e uso de variáveis no programa. A análise do fluxo de dados foi inicialmente utilizada na otimização do código gerado a partir de compiladores, com a finalidade de aprimorar a alocação de registradores pelas variáveis do programa. Aplicadas ao teste de software, as informações coletadas sobre as variáveis servem de base para selecionar caminhos para testar o programa.

Para realizar este tipo de teste são necessários alguns conceitos, tais como: definição e uso de variáveis. Comandos de atribuição e de entrada são nodos definição, enquanto comandos de saída e comandos condicionais são nodos uso. Em alguns casos estes dois nodos podem se confundir, assim como na situação **cont = cont + 1**, onde o nodo correspondente é de definição e de uso em relação à variável **cont**. Isto porque o valor atual de **cont** é utilizado, e o próximo valor é definido no comando.

Nodos uso podem ser classificados pelo tipo de uso: uso-p (uso em predicado) ou uso-c (uso em computação). O primeiro tipo ocorre quando o comando é uma condição (o nodo é origem de mais de uma aresta do grafo de programa), enquanto que o segundo corresponde a um comando que não é do tipo condicional, proveniente de uma única aresta.

Para o teste de fluxo de dados, em comparação ao teste de fluxo de controle, é exigida uma cobertura de caminho bem maior, pois para cobrir combinações de definições e usos de variáveis é necessário observar outros tipos de caminhos.

No teste de fluxo de dados a observação de sub-caminhos é importante, pois estes ligam um nodo definição a um nodo uso de uma variável. Os caminhos testados neste tipo de teste são identificados por:

- Caminhos-d-u (*definition-use*) corresponde a um caminho de um grafo, tal que o nodo inicial do caminho é um nodo definição da variável e o nodo final é o nodo uso da mesma variável.
- Caminho-d-c (*definition-clear*) corresponde a um caminho-d-u, tal que não existe outro nodo definição da variável no caminho, exceto pelo nodo inicial.

Pode-se observar que caminhos-d-u que não são caminhos-d-c são caminhos onde a variável em questão é definida mais de uma vez antes de ser utilizada. O que pretende-se com este método é identificar, antes do inicio do processo de teste, variáveis que venham a ser utilizadas sem que a elas sejam atribuídos valores; a situação onde uma variável é definida duas vezes, sendo que o valor atribuído inicialmente a ela não é utilizado; ou onde valores são atribuídos a variáveis e nunca são usados.

Com a definição de caminho-d-u e caminho-d-c, pode-se definir alguns critérios de cobertura para o teste de fluxo de dados, onde o fluxo é representado pelas definições e usos das variáveis do programa, sendo eles [RAP 85]:

- All-defs (cobertura de todas as definições**)**: para cada nodo definição de cada variável o conjunto de testes deve cobrir pelo menos um caminho-d-c para algum nodo uso;
- All-uses (cobertura de todos os usos): esta estratégia é satisfeita quando para cada variável o conjunto de testes cobre pelo menos um caminho-d-c de cada nodo definição da variável para cada nodo uso da mesma;

- All-P-uses (cobertura de todos os usos-P**)**: um conjunto de testes satisfaz este critério quando para cada variável o conjunto cobre pelo menos um caminho-d-c de cada nodo definição da variável para cada nodo uso-P da mesma variável, e cada nodo sucessor deste nodo uso;
- All-C-uses (cobertura de todos os usos-C**)**: um conjunto de testes satisfaz este critério quando para cada variável o conjunto cobre pelo menos um caminho-d-c de cada nodo definição da variável para cada nodo uso-C da mesma variável, e cada nodo sucessor deste nodo uso;
- All-P-uses/some-C-uses (cobertura de todos os usos-P e alguns usos-C): este critério tem sua cobertura aceita se o conjunto de testes cobre pelo menos um caminho-d-c de cada nodo definição da variável para cada nodo uso-P da mesma, ou para pelo menos um uso-C, caso a variável não tenha nenhum uso-P;
- All-C-uses/some-P-uses (cobertura de todos os usos-C e alguns usos-P): este critério tem sua cobertura aceita se o conjunto de testes cobre pelo menos um caminho-d-c de cada nodo definição da variável para cada nodo uso-C da mesma, ou para pelo menos um uso-P, caso a variável não tenha nenhum uso-C, e
- All-DU-paths (cobertura de todos os caminhos-d-u): um conjunto de testes satisfaz este critério se, para cada variável, o conjunto cobre todos os caminhos-d-c acíclicos ou que percorre cada ciclo no máximo uma vez, de cada nodo definição da variável para cada nodo uso da mesma variável e cada nodo sucessor deste nodo uso.

A cobertura de fluxo de dados tradicional analisa as variáveis de maneira independente. Porém, é comum que as variáveis interfiram umas nas outras. Para cobrir esta situação utiliza-se o critério de cobertura de interação, o qual realiza uma análise entre variáveis, duas a duas, três a três, etc.

Outro critério de cobertura, chamado de cobertura de contexto, ocorre quando existe um conjunto de definições de variáveis para as quais existe um caminho-d-c ligando as variáveis a um determinado nodo.

3.7.2.4Slicing (fatiamento)

Esta técnica [WEI 84] procura analisar as influências entre diferentes pontos do programa. Combina conceitos das técnicas de fluxo de controle e fluxo de dados. Tem aplicação em diversas tarefas que necessitem da análise do código fonte do programa, tais como: depuração, otimização, etc.

Um *slice* de um programa em relação a um determinado comando e suas variáveis é um conjunto de comandos que podem influenciar o valor das variáveis do programa no comando em questão.

As técnicas de *slicing* podem ser agrupadas em duas dimensões:

- Regressivo (*backward*) x progressivo (*forward*), ou
- Estático x dinâmico

No *slicing* regressivo a fatia é um conjunto de comandos que afetam o valor das variáveis no nodo, enquanto no progressivo, é um conjunto de comandos que podem ser afetados pela execução do nodo.

No *slicing* estático, a análise é realizada sem suposição sobre as entradas do programa, enquanto no dinâmico, considera-se uma entrada em particular, isto é, uma execução específica do programa.

Para o *slicing* estático dois conceitos são fundamentais: dependência de controle e dependência de dados. No primeiro caso um nodo **j** é dependente de controle de um nodo **i** (no grafo de programa), se todo caminho de **i** até o nodo final passa por **j** e se existe um caminho de **i** até **j**, tal que, todos os nodos, exceto **i** e **j,** tenham todos os caminhos até eles e o nodo final passando por **j**. Em outras palavras, **j** é dependente de controle de **i,** se **i** puder evitar a execução de **j**. Isto é comum acontecer em comandos de seleção ou repetição.

A dependência de dados ocorre quando um nodo **j** depende de dados de um nodo **i** no grafo de programa e se existe uma variável **v**, onde **v** é definida por **i** e utilizada por **j** e que exista um caminho de **i** até **j**.

3.7.3 Teste Baseado em Falhas

O teste baseado em falhas é também conhecido como análise de mutantes [DEM 78] [WON 94]. A análise de mutantes parte do princípio que o programa sob teste está correto (ou quase correto). A partir daí gera-se um conjunto de programas semelhantes a ele, os quais são chamados de mutantes. Os programas mutantes possuem algumas mudanças sintáticas entre eles e o programa que está sendo testado, pois assume-se que programadores experientes não cometem erros de codificação. Desta forma, verifica-se que grande parte dos erros dos programas são introduzidos por variações sintáticas, que não configuram erros sintáticos, mas sim semânticos. Após a criação dos mutantes deve-se gerar vetores de teste que sejam capazes de provocar diferenças de comportamento entre o programa e seus mutantes.

No teste de mutação, um dado programa é mutado em inúmeras cópias, cada uma é alterada em um caminho para representar um provável erro de programação. O programa na Figura 3.11 representa uma mutação do programa dado na Figura 3.1. Neste, o predicado **B<>0** foi trocado por **B = 0**, representando um possível erro de programação. Testes são criados para distinguir os programas mutantes do programa original na esperança que, durante este processo de mutante distinto, o programa seja exercitado em um caminho que manifeste as falhas no programa original.

```
If  B = 0 then
                    X = A/B
```

Figura 3.11: Programa mutante

Este tipo de teste ocorre em quatro etapas: 1) geração de mutantes, 2) execução do programa a partir de um conjunto de vetores de teste, 3) execução dos mutantes com o mesmo conjunto de vetores de teste e 4) análise dos mutantes.

Os vetores de teste são adequados para um programa se eles podem distinguir pequenas mudanças sintáticas. Um conjunto de vetores de teste é apresentado ao programa. Quando um deles apresenta uma saída incorreta então assume-se que um

mutante foi encontrado. Os elementos requeridos por este critério são baseados em falhas comuns cometidas por programadores durante o desenvolvimento do software [VIN 97].

Este critério surgiu na década de 70, sendo baseado em um método clássico de detecção de erros lógicos em circuitos digitais, o modelo de teste de falha única [DEM 78] [VIN 97].

Pode-se obter o número de mutantes que um programa pode ter, através da expressão matemática: **M = K + E + N,** onde **M** é o número total de mutantes do programa, **K** é o número de mutantes mortos (*killed*), **E** é o número de mutantes equivalentes e **N** o número de mutantes não detectados.

Um mutante é dito equivalente quando após ser sensibilizado não propaga erro. Um mutante é do tipo não detectado quando não foi sensibilizado por nenhum vetor de teste. Os mutantes mortos ou detectados são aqueles que quando sensibilizados propagam o erro para a saída.

Esta técnica elege um conjunto de vetores de teste adequado quando a cobertura das falhas alcança 100%, isto é, quando todos mutantes tornam-se mortos.

O processo realizado pela análise de mutantes ocorre da seguinte forma: se um mutante apresenta resultados diferentes do programa original, diz-se que este mutante está morto, pois o conjunto de dados de teste identificou o erro. Porém, quando os dados de teste apresentam os mesmos resultados, no mutante do programa original, diz-se que ele está vivo. O que pode ocorrer, se os programas são equivalentes (desempenham a mesma função), é que erros não possam ser detectados, para isso devem ser gerados novos vetores de teste.

Assim, o processo de teste baseado em falhas divide-se nas seguintes etapas:

- Identificação do programa a ser testado (chamado de programa original) e um conjunto de testes, inicialmente vazio;
- Geração de um conjunto de mutantes do programa original, através de operadores de mutação, alterações sintaticamente corretas, no programa original;
- Geração de vetores de teste e verificação se estes matam todos os mutantes;
- Se todos os mutantes foram mortos, o conjunto de dados de teste é dito adequado, então pode-se terminar o processo. Caso contrário, acrescenta-se um teste para cada mutante sobrevivente até encontrar o conjunto de teste adequado.

Os operadores de mutação variam conforme a linguagem de programação utilizada e os tipos de erros considerados. Alguns exemplos são:

- Substituição de uma variável por outra;
- Substituição de operador relacional, ou;
- Substituição de operador aritmético.

O critério de teste de mutação baseia-se nos erros que podem ser cometidos durante o processo de desenvolvimento do software. Para sua aplicação é necessária a caracterização de um conjunto de operadores de mutação, o qual modela os tipos de erros que se deseja evidenciar no projeto a ser testado.

O custo de aplicação deste critério de teste é elevado, devido ao grande número de mutantes que podem ser gerados e também devido à inviabilidade de sua utilização sem o apoio de uma ferramenta de teste. No entanto, a sua eficácia em detecção de erros tem levado à definição de estratégias que visam minimizar o custo de sua aplicação.

O interesse pela utilização do teste de mutação vem aumentando e, com isso, vários estudos vêm sendo conduzidos para a avaliação deste critério de teste sendo aplicado em diferentes contextos.

4 FUNDAMENTOS E FERRAMENTAS PARA O TESTE DE HARDWARE

4.1 Introdução

A tecnologia atual de fabricação de circuitos integrados permite a implantação de sistemas complexos e que possuem cada vez mais um número maior de transistores, conseqüentemente, aumentando a dificuldade do teste dos sistemas eletrônicos.

Neste capítulo serão apresentados conceitos relacionados ao teste de dispositivos de hardware. As técnicas utilizadas e as dificuldades de aplicação serão abordadas, para que possamos avaliar, posteriormente, a metodologia proposta nesta tese.

Este capítulo tem também como finalidade apresentar as ferramentas utilizadas nas diferentes implementações realizadas durante o desenvolvimento desta tese.

4.2 Teste e Verificação no Contexto de Hardware

Em cada uma das etapas do fluxo de projeto de um circuito podem ser realizadas etapas de teste que, dependendo da etapa, recebem diferentes nomenclaturas: verificação e teste.

Pradhan [PRA 96] distingue duas fases para avaliar a qualidade de circuitos digitais: 1) **verificação** é a fase inicial na qual o primeiro protótipo do circuito é validado para assegurar que atende as especificações funcionais, isto é, para verificar se o projeto está correto, e; 2) **teste** é a fase onde deve-se assegurar que somente circuitos livres de defeitos sejam comercializados, neste são detectadas falhas de fabricação.

O teste de um circuito antes de sua implementação é conhecido formalmente como verificação. Seu objetivo é verificar se o projeto está de acordo com a especificação. Atualmente, simulação é a ferramenta de verificação mais utilizada.

No momento em que o projeto está implementado em silício, ele pode ser verificado. Esta verificação, responsável por achar erros de fabricação do circuito impresso, é chamada formalmente de teste.

4.3 Defeito, Falha e Erro no Contexto do Teste de Hardware

Na área de teste de sistemas de hardware, conceitua-se o que leva ao mau funcionamento de um circuito como segue [BUS 2000]:

- **Defeito físico**: refere-se à manifestação de um desgaste do material utilizado na construção de um dispositivo, podendo ocorrer devido ao uso do dispositivo ou à fabricação do próprio;
- **Defeito de projeto**: refere-se à manifestação de um erro de projeto;
- **Falha**: refere-se à manifestação de um defeito de forma interna ao sistema, podendo ser permanente ou transitória, e
- **Erro**: refere-se à manifestação externa de uma falha. Portanto um erro refere-se a um valor incorreto gerado por um sistema defeituoso.

Com isso pode-se dizer que um procedimento de teste serve para a verificação da presença ou ausência de defeitos no circuito, cujas falhas são detectadas pela manifestação de erros.

Muitos defeitos de um circuito podem ser inerentes ao substrato de silício utilizado na fabricação dos dispositivos. Outros podem ser resultado de impurezas encontradas no material utilizado na produção dos *wafers.* Problemas que ocorrem durante várias etapas do processo de fabricação são, por exemplo: alteração de resistividade de contatos; presença de partículas na sala limpa ou nos materiais; desalinhamento das máscaras podendo resultar em alteração nas dimensões de transistores, etc. Todos estes defeitos conduzem, em geral, a falhas que afetam simultaneamente vários dispositivos, as quais são chamadas de falhas múltiplas.

Os defeitos que ocorrem quando o sistema já está em operação são devido ao transporte (problemas mecânicos), a fenômenos eletromagnéticos (interferências) ou fatores térmicos (diferenças importantes de temperatura) [ABR 90] [BUS 2000]. Em geral estes defeitos produzem falhas simples.

Defeitos da fabricação ou uso do circuito podem resultar em falhas permanentes, como interconexões abertas e em curto, portas que não chaveiam, entre outras. Falhas transitórias ocorrem devido a fenômenos intermitentes, tais como interferências eletromagnéticas ou radiações [ABR 90] [BUS 2000].

As falhas são modeladas em diversos níveis de abstração e conforme o efeito no sistema é o modelo que representa o efeito no sistema do defeito que se quer testar. Alguns modelos, tais como: falhas do tipo colagem (*stuck-at*), falhas de ponte (*bridging faults*) e falhas de atraso (*delay*) serão detalhados na seção 4.5.

4.4 Métodos e Tipos de Teste de Hardware

Como já mencionado, desde o projeto do protótipo de um circuito, este é submetido a uma série de etapas de verificação, segundo as diversas fases de sua vida.

A escolha do método de teste depende de vários fatores, tais como: custo; qualidade procurada; tempo para aplicação; nível de testabilidade desejado, e possibilidade de interrupção das funções do sistema.

Os métodos de teste são classificados conforme alguns critérios [ABR 90]:

- **Momento da aplicação do teste**: pode ocorrer de forma concorrente à aplicação (teste *on-line* ou teste concorrente), ou como uma atividade independente (teste *off-line*);

- **Estímulos de teste**: podendo ser gerado no próprio sistema (auto-teste) ou através de um dispositivo externo(testador). Estes estímulos podem ser recuperados de uma memória (teste com padrões pré-calculados) ou gerados durante o processo de teste(teste algorítmico ou teste por comparação). A aplicação destes estímulos pode ocorrer em uma ordem fixa, pré-determinada ou dependente de resultados obtidos até o momento (teste adaptativo). A aplicação dos estímulos pode ocorrer em uma velocidade menor que a velocidade de operação normal do sistema (teste DC ou estático) ou na velocidade normal de operação do sistema (teste AC ou teste *at-speed*);
- **Alvo do teste**: depende da etapa de projeto no qual é aplicado, o qual tenta identificar erros de projeto (verificação de projeto), erros e defeitos de fabricação (teste de aceitação e *burn-in*), falhas físicas imediatas (teste de qualidade) e falhas físicas que ocorrem durante o uso do sistema (teste de campo ou teste de manutenção);
- **Análise dos resultados obtidos**: pode-se observar todas as saídas ou apenas algumas funções (teste compacto);
- **Verificador de resultados**: refere-se a quem verifica os resultados, podendo ser o próprio sistema, através do auto-teste ou auto-verificação, ou através de um dispositivo testador (teste externo), e
- **Objeto do teste**: o objeto de teste pode ser um circuito integrado (teste de componente), uma placa (teste de placa) ou um sistema com vários níveis de complexidade (teste de sistema).

Para verificar o comportamento de um circuito pode-se utilizar tanto o teste funcional quanto o teste estrutural [NAD 99] [BUS 2000], aplicando-se valores na entrada do mesmo e comparando a saída com o valor esperado. Ao valor aplicado chama-se vetor de teste e define-se como cobertura de falhas o percentual de falhas detectadas pelos vetores de teste aplicados durante o teste, em relação ao total de falhas possíveis do circuito.

O teste funcional ignora a estrutura interna e identifica como um circuito sem falhas aquele que executa as funções a ele especificadas, por isso, é conhecido, também, como teste de caixa preta. Apresenta a possibilidade de ser executado de forma implícita, concorrente ou "*on-line*", também chamado de *checking*, onde realiza testes durante a operação do sistema para a detecção de erros.

O objetivo do teste estrutural é verificar se a estrutura implementada fisicamente confere com a estrutura especificada no projeto. Chama-se estrutural, pois depende da estrutura do circuito (portas lógicas, transistores, etc.) e verifica estes elementos dentro do circuito, por isso pode apresentar coberturas de falhas elevadas, pois os vetores de teste gerados são gerados para detectar falhas específicas dentro do circuito [NAD 99] [ABR 90].

4.5 Modelos e Simulação de Falhas

Os modelos de falhas são utilizados para representar defeitos físicos em um nível de abstração maior, geralmente em nível de porta lógica. Portanto, um modelo de falhas é definido como a descrição abstrata dos efeitos de alguns defeitos ou de combinações de

defeitos em um circuito. A principal função da abstração é reduzir a complexidade da análise de comportamento do circuito na presença de defeitos.

O uso de modelos de falhas possui algumas vantagens e desvantagens. As vantagens no uso de modelos de falhas no processo de teste referem-se à independência da tecnologia em uso e à cobertura do teste. Como desvantagens no uso de modelos de falhas no teste de hardware tem-se: a imprecisão na identificação de certos defeitos (por exemplo, portas abertas em circuitos CMOS) e a detecção de falhas. Os testes só podem ser eficientes se falhas realistas forem modeladas a partir de mecanismos físicos e leiautes reais.

Na tentativa de representar defeitos físicos [WAD 78], [GAL 80], [COU 81], [RAJ 92] em dispositivos de hardware, surgiram vários modelos de falhas, entre eles: *stuck-at*, referindo-se às entradas e saídas de portas lógicas coladas em 1 ou 0, *stuck-on/open*, representando transistores que sempre ou nunca conduzem; *slow-to-rise* e *slow-to-fall*, referindo-se a portas com atraso de subida e descida, entre outros. Curtos entre fios (*bridging*), fios desconectados (*opens*) e atrasos cumulativos de portas e interconexões no caminho crítico (*path delay*) são exemplos de modelos de falhas de interconexão. Geralmente, os modelos de falhas são complementares, pois nenhum pode representar todos os defeitos físicos possíveis.

O modelo de falhas mais utilizado é o *stuck-at*. Neste modelo de falhas pressupõem-se que em um circuito os possíveis defeitos têm como efeito lógico colocar permanentemente um dado nó em valor 1 (*stuck-at 1*) ou zero (*stuck-at 0*). O modelo de falhas *stuck-at* é adequado para modelar a maioria das falhas reais (ou defeitos) observadas nos circuitos.

Um processo comumente utilizado na preparação do teste de dispositivos de hardware é a simulação de falhas. Em um processo de simulação de falhas o circuito é acrescido de falhas do modelo, para que possa ser realizada a comparação dos resultados obtidos com os valores de uma simulação do circuito sem falhas. O processo de simulação de falhas tem como objetivo verificar se as falhas injetadas no circuito são detectadas através da aplicação de determinados estímulos de entrada. A Figura 4.2 mostra como é realizada a simulação de falhas.

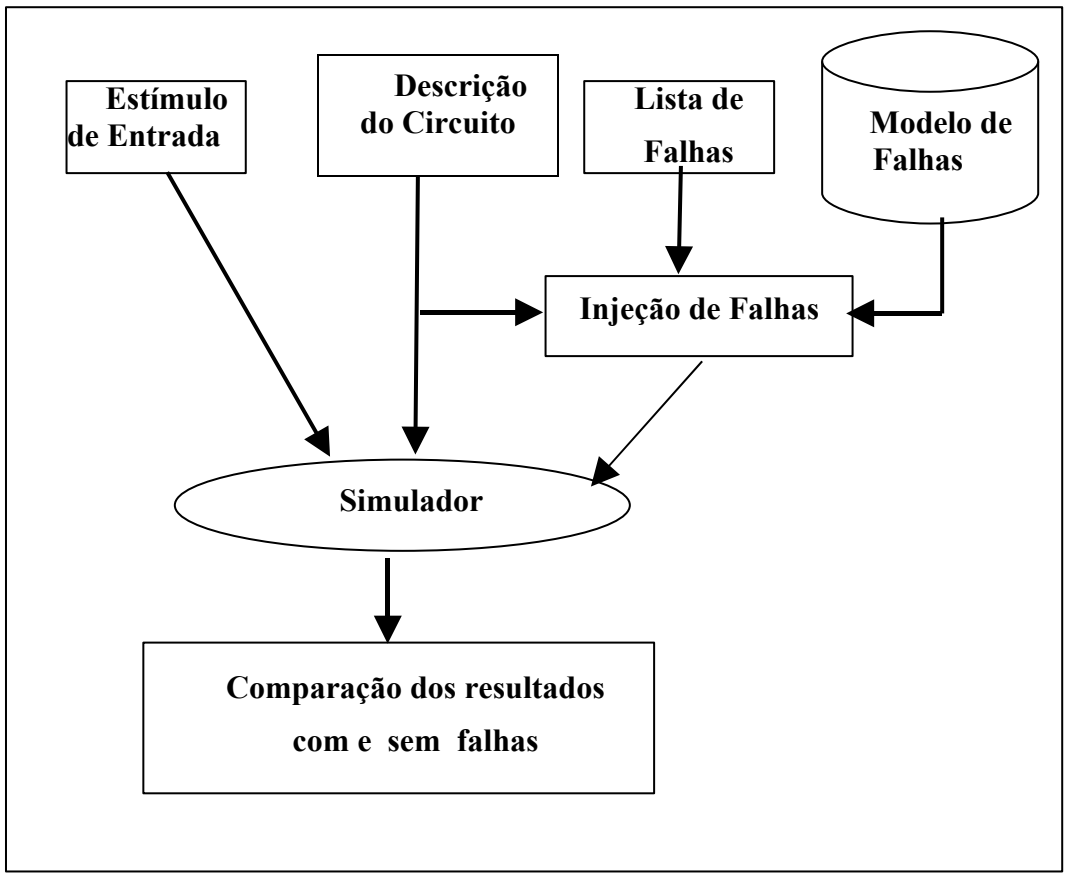

Figura 4.2: Processo genérico de simulação de falhas

4.6 Geração de Padrões Teste

A partir da escolha do modelo de falhas e da utilização de ferramentas de simulação de falhas, a geração de vetores de teste pode ser realizada de maneira automática (Figura 4.3). Existem ferramentas de geração de vetores de teste que geram os vetores de entrada utilizados pelos dispositivos testadores.

Existem quatro classificações possíveis para a geração de teste: exaustiva, pseudo-exaustiva, pseudo-aleatória e determinística. O teste exaustivo utiliza todas as combinações de entradas como padrões de teste. Este tipo de teste, apesar de garantir uma alta cobertura de falhas, é inviável para circuitos com muitas entradas ou circuitos seqüenciais que possuem muitos estados. O teste pseudo-exaustivo divide o circuito em sub-circuitos fracamente conectados testando cada partição exaustivamente. O teste pseudo-aleatório (randômico) [PAT 2002] [CUM 2003] utiliza uma estrutura como, por exemplo, um LFSR (*Linear Feedback Shift-Register*) para gerar padrões de maneira pseudo-aleatória. O teste determinístico ou orientado a falhas gera padrões específicos a uma falha [PAT 2002] [CUM 2003]. Geralmente estes padrões são calculados por ferramentas de geração automática de padrões de teste (ATPG).

A geração pseudo-aleatória é obtida através de geradores especiais (geradores de números aleatórios) que utilizam sementes pré-definidas, as quais representam diferentes vetores de inicialização, e diferentes tamanhos para as seqüências de teste. Este método garante, geralmente, uma boa cobertura de falhas. O processo de geração

dos padrões de teste é independente da estrutura interna do circuito. Entretanto para alcançar uma cobertura de falhas adequada é necessário um número elevado de vetores de teste.

Para a geração determinística existem dois tipos de métodos: métodos algébricos e topológicos. Nos métodos algébricos, são computados vetores de teste a partir de expressões booleanas que descrevem o funcionamento do circuito. Em métodos topológicos, os vetores de teste são derivados a partir da estrutura do circuito. Um dos processos de geração topológica mais conhecido é o algoritmo D [ROT 67].

O algoritmo D consiste na utilização de quatro valores lógicos 0, 1, D e D'. O valor D é utilizado para distinguir o comportamento do circuito sem falhas do comportamento do circuito falho. Quando o comportamento estiver correto o valor de D é igual a 1 e seu complemento 0. A mesma analogia pode ser feita para o complemento de D. O algoritmo D começa seu processo propagando uma falha única do tipo *stuck-at* para uma saída primária do circuito. Em seguida, justifica as entradas do circuito de forma a produzir no nodo falho um comportamento oposto ao previsto quando na ausência da falha. Este algoritmo foi desenvolvido, inicialmente, para circuitos combinacionais, e posteriormente, estendido à geração de vetores para circuitos seqüenciais. PODEM (*Path-Oriented Decision Making*) [GOE 81], FAN (*FANout-oriented*) [FUJ 83] e TOPS (*Topological Search*) [KIR 87], todas baseadas no algoritmo D, são exemplos de ferramentas amplamente utilizadas para a geração de vetores de teste.

Mesmo com o auxilio dos ATPGs, falhas difíceis de detectar acabam denegrindo com a relação entre a cobertura de falhas e o tempo de aplicação do teste. Técnicas de projeto visando a testabilidade, por preocuparem-se com o teste já na fase de projeto do circuito, alteram a sua construção e incorporam ao circuito algumas estruturas especiais que melhoram a acessibilidade aos elementos difíceis de testar, esta técnica será apresentada na próxima seção.

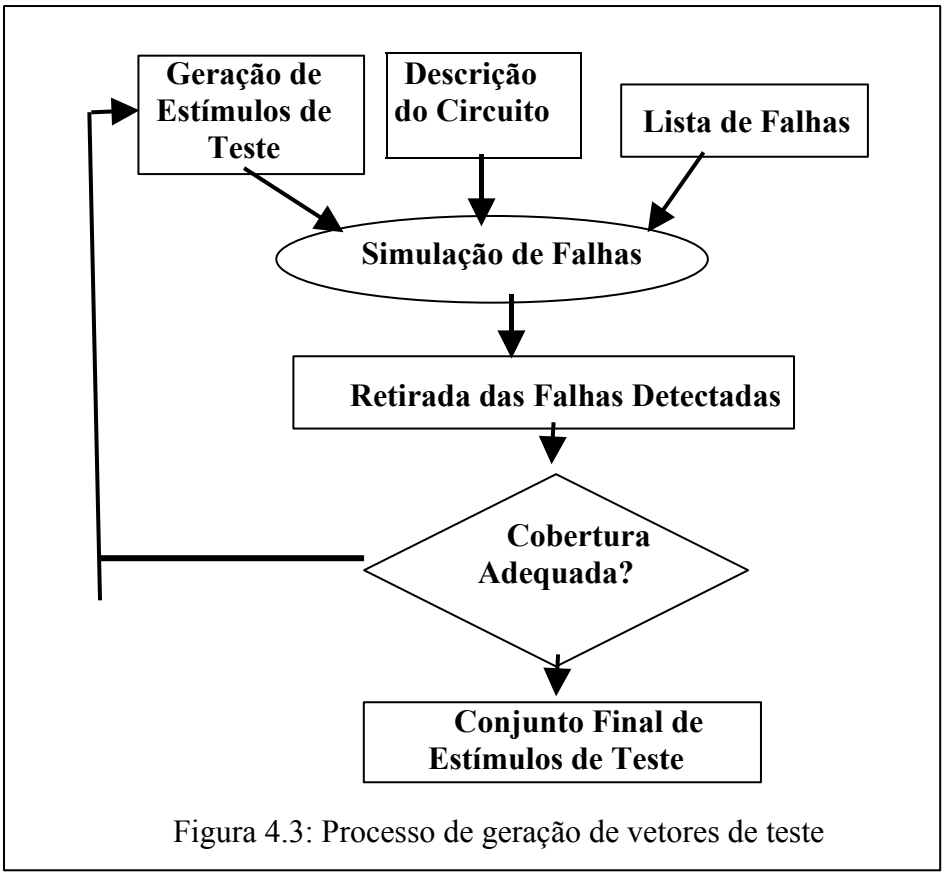

Figura 4.3: Processo de geração de vetores de teste

4.7 Testabilidade do Hardware

A geração e aplicação do teste podem ser mais eficientes se a testabilidade do circuito for considerada e melhorada durante a fase de projeto do circuito. O objetivo é melhorar a controlabilidade e observabilidade do circuito com o mínimo possível de incremento de área e sem degradar o desempenho. Se possível, este processo deve resultar em uma alta cobertura de falhas e em um baixo tempo de aplicação do teste, além de facilitar a geração dos padrões de teste.

A controlabilidade e observabilidade são os fatores mais importantes na determinação da complexidade da geração de padrões de teste. A controlabilidade refere-se à facilidade de controlar um determinado valor lógico em um determinado ponto do circuito, através da modificação dos valores das entradas do mesmo. Por outro lado, a observabilidade avalia a facilidade de observar o valor lógico de um determinado ponto do circuito nas saídas primárias deste [ABR 90] [NOR 97].

A análise de testabilidade procura mensurar a controlabilidade e observabilidade dos nós internos do circuito. Goldstein [BUS 2000] propôs um algoritmo para determinar a dificuldade para controlar e observar sinais em um circuito, tendo desenvolvido o algoritmo chamado SCOAP (*Sandia Controlability Observability Analysis Program*) [BUS 2000]. Este algoritmo possibilita a obtenção de seis medidas para cada nó do circuito. Três medidas combinacionais, as quais referem-se ao número de nós e portas lógicas que devem ser atravessados para controlar ou observar o sinal em um

determinado nó do circuito e três seqüenciais que relacionam-se ao número de ciclos de relógio necessários para controlar ou observar o sinal. Estas medidas são utilizadas, por exemplo, como indicadores de que partes do circuito devem ser re-projetadas para garantir uma testabilidade adequada.

As técnicas de projeto visando a testabilidade (DfT) normalmente são utilizadas por projetistas que trabalham no projeto lógico do circuito, podendo ser classificado em: técnicas *ad-hoc* e estruturadas.

As técnicas de DfT *ad-hoc*, ou caso a caso, são medidas tomadas para melhorar a testabilidade durante a fase de projeto de forma não estruturada e não sistematizada [ABR 90] [IEE 94]. Estas levam em conta a experiência adquirida pelos projetistas. Alguns métodos de DfT *ad-hoc* são [BUS 2000] [JER 2002]:

- Não permitir realimentação assíncrona: as realimentações em lógica combinacional podem ocasionar oscilações face à alguns valores nas entradas;
- Introdução de sinais de *reset*: a utilização de *flip-flops* com sinais de *reset* e *clear* facilitam a controlabilidade do circuito;
- Não permitir portas lógicas com *fan-in* elevado: *fan-in* elevado nas portas de entrada dificultam a controlabilidade da saída da porta lógica e a observabilidade das entradas, e;
- Controle de teste para sinais de difícil controlabilidade: inserir pinos de controle nos circuitos melhorando a controlabilidade dos mesmos.

A utilização de técnicas *ad-hoc* em circuitos muito grandes nem sempre é possível, além do fato de existir uma forte dependência com as ferramentas de síntese, pois é difícil de encontrar em uma descrição HDL problemas ou áreas difíceis de testar sem conhecer o comportamento da ferramenta de síntese utilizada.

As **técnicas estruturadas** de DfT representam um enfoque sistemático para o aspecto da testabilidade, superando certas limitações das técnicas *ad-hoc* e viabilizando a automatização através de ferramentas de CAD.

As técnicas de DfT estruturadas utilizam lógica extra e sinais são inseridos nos circuitos para permitir o teste com alguns procedimentos pré-definidos. Os circuitos possuem, além de seu estado de funcionamento normal, um ou mais modos de teste. Os métodos estruturados mais utilizados são o teste de varredura (*scan*) e o auto-teste (*Built-In Self-Test*) [BUS 2000].

4.7.1 Teste de Varredura

A técnica mais tradicional de DfT é o *scan path*. Nesta técnica, os registradores (*flip-flops*) do sistema são modificados para que passem a possuir dois modos de operação: modo normal e de teste. No modo normal os registradores têm a função de carga paralela de dados. No modo de teste, estes passam a ter a função de registradores de deslocamento, com isso aumentando o número de nós internos controláveis e observáveis [BRG 85] [IEE 94] a partir de pinos de entrada (*scan-in*) e pinos de saída (*scan-out*).

A utilização desta técnica apresenta algumas vantagens, tais como: aumentar a controlabilidade e observabilidade, melhorando a cobertura de falhas e possibilitar a

implementação automatizada através de ferramentas comerciais. Por outro lado, o uso do *scan path* acarreta em custos, tais como [JER 2002] [NOR 97] [BUS 2000]:

- Aumento da área gasta pelo circuito, devido à modificação dos *flip-flops* que compõem os registradores de deslocamento;
- Necessidade de pinos extra no circuito para a inserção, retirada e controle da aplicação dos vetores de teste;
- Diminuição da freqüência máxima de operação do circuito;
- Aumento no consumo de energia durante o processo de teste, e;
- Possível aumento do tempo de teste, devido à aplicação serial de vetores de teste.

Existem duas abordagens diferentes para o teste de varredura: varredura parcial (*partial scan*) e varredura total (*full scan*).

O *partial scan* utiliza apenas um subgrupo dos *flip-flops* do circuito para compor as cadeias de deslocamento. Por outro lado, o *full scan* utiliza todos os *flip-flops* do circuito para compor a cadeia.

A principal vantagem do uso do *partial scan* é a redução dos custos de área e o aumento da velocidade do teste, se comparado à abordagem *full scan*. Esta por sua vez, na maior parte dos circuitos apresenta melhor cobertura de falhas.

4.7.2 Auto Teste

Uma alternativa às técnicas de DfT, onde os vetores de teste são gerados por um testador externo (ATE – *Automatic Test Equipament*), é o auto teste integrado ou BIST. BIST é uma técnica que utiliza partes do circuito para testar, internamente, o restante do circuito. Os vetores de teste são gerados internamente, utilizando blocos integrados no circuito. Para isso, são incorporados módulos de hardware responsáveis pela geração de vetores de teste, comparação da resposta (assinatura) e controle do fluxo de teste [ABR 90] [AGR 93a] [AGR 93b] [GUP 94] [IEE 94].

A adoção de BIST apresenta algumas vantagens, tais como: [ALY 2004]:

- Menor custo de implementação se comparado ao uso de teste externo utilizando ATE, e;
- Possibilita a aplicação do teste na velocidade normal de aplicação do circuito (*at-speed*).

Entretanto, o uso de BIST (*Built-In Self-Test*) apresenta algumas desvantagens, assim como aumento da área do circuito, degradação de desempenho e em alguns casos, para obter alta cobertura de falhas, podem ser necessários muitos vetores de teste, tornando o teste proibitivo [ALY 2004].

4.8 Ferramentas de Projeto e Teste

A concepção de um circuito envolve o uso de uma série de etapas, como já mencionado anteriormente. Cada uma destas etapas pode ser feita através do uso de ferramentas CAD é o que chamamos de projeto auxiliado por computador. Nesta tese utilizamos algumas destas ferramentas, tais como:

- Editores de linguagens de descrição de hardware, que permitem descrever um circuito utilizando HDL;
- Ferramentas de síntese que permitem a passagem automática de um nível de descrição para um outro nível inferior;
- Ferramentas de simulação, estas permitem calcular a resposta de um circuito a partir de um conjunto de estímulos de entrada, e;
- Ferramentas de auxilio ao teste, utilizadas quando o projeto está chegando ao fim, pois cada dispositivo fabricado deverá ser testado para detectar erros de fabricação [ABR 90] [BRE 76] [FUJ 85] e de projeto (para estes pode-se utilizar, também, ferramentas de verificação [HUA 98]).

Para os testes e experimentos realizados nesta tese foram utilizadas as ferramentas do sistema Mentor Graphics™ [MEN 99]. Este sistema é responsável pela síntese, validação e teste de circuitos integrados. Essas ferramentas são: compilador de descrições de alto nível de circuitos digitas, simuladores lógicos e elétricos, mapeador tecnológico, posicionador, roteador, extrator elétrico, compactador de leiaute, entre outras.

Neste capítulo serão detalhadas apenas as ferramentas utilizadas para o desenvolvimento desta tese, são elas: Leonardo Spectrum™, Flextest™, Fastscan™ e DFTAdvisor™.

4.8.1 LeonardoSpectrum™

É uma das ferramentas de síntese do sistema Mentor™. É o software responsável pela compilação do VHDL. Ele faz um mapeamento para biblioteca utilizada e cria um arquivo contendo o projeto sintetizado em nível de portas lógicas (*netlist)*.

Nesta ferramenta, para realizar a síntese, escolhe-se uma tecnologia alvo. No caso desta tese, os resultados de síntese foram obtidos sintetizando as descrições para ASIC utilizando a tecnologia 0.35μm da biblioteca ADK, configurando a ferramenta Leonardo Spectrum™ para manter a hierarquia dos blocos da descrição e tendo como objetivo obter uma menor área do circuito.

Esta ferramenta é também utilizada para a obtenção de valores relativos à área do circuito e freqüência de operação, para a realização de um comparativo destes valores sobre o circuito original e o circuito após as diferentes modificações implementadas nesta tese.

4.8.2 DFTAdvisor™

O DFTAdvisor™ é a ferramenta da Mentor™ que provê análise de testabilidade e insere estruturas internas de teste no projeto. Esta ferramenta utiliza o *netlist* gerada pela ferramenta Leonardo Spectrum™ para gerar um novo *netlist* com a cadeia *scan*, relatórios relacionados ao processo de inserção do *scan* e arquivos de procedimentos de teste *scan* que serão utilizados pela ferramenta de ATPG.

Esta ferramenta pode identificar e inserir uma variedade de estruturas de teste, incluindo diferentes arquiteturas de *scan* e pontos de teste. O DFTAdvisor™ trabalha com quatro diferentes estruturas de teste:

- O *full scan* é um estilo de estrutura de teste que identifica e converte todos os elementos seqüenciais(que são possíveis de verificar) para compor a cadeia *scan*;
- Na estrutura de teste *partial scan* um conjunto de elementos seqüenciais é identificado e selecionado para compor a cadeia *scan*;
- A estrutura de teste denominada *partition scan* é um estilo que identifica e converte certos elementos seqüenciais dentro de partições para as cadeias *scan*, e;
- O *test points* é um método que identifica e insere pontos de controle e observação no projeto para aumentar a testabilidade do mesmo.

Nesta tese utilizamos as estruturas de teste *full scan* e *partial sca*n. A ferramenta DFTAdvisor™ possibilita que o número de *flip-flops* que irão compor a cadeia do *scan* parcial seja determinado. Nesta tese, utiliza-se esta característica para comparar os métodos de inserção de *scan* com a abordagem proposta.

4.8.3 Fastscan™ e Flextest™

A ferramenta Fastscan™ é utilizada como ATPG para circuitos combinacionais e com *full scan*. Caracteriza-se por realizar suas tarefas rapidamente(processamento rápido) e utilizar pouca memória.

Nesta tese a ferramenta Fastscan™ é utilizada para obter resultados de cobertura de falhas relacionadas à inserção de *full scan*, e para verificar a quantidade de *flip-flops* necessários para a implementação.

A ferramenta Flextest™ destina-se à simulação de falhas e geração automática de padrões de teste para circuitos seqüenciais (ATPG) com poucas ou nenhuma estrutura de teste inserida. É utilizada, portanto, para circuitos seqüenciais sem *scan* ou com *scan* parcial. A ferramenta pode ser utilizada para o teste funcional e/ou como um ATPG seqüencial.

Flextest™ suporta diferentes modelos de falhas. Pode ser utilizada em conjunto com a ferramenta Fastscan™ para aumentar a cobertura de áreas seqüenciais do projeto. Por outro lado, em comparação com a ferramenta Fastscan™, apresenta processamento lento, geração de teste pode levar dezenas de horas, e necessita de muita memória para ser executada.

Uma outra funcionalidade da ferramenta Flextest™ é a possibilidade de executar o processo de teste com vetores de teste internos ou externos. A geração de padrões interna utilizada para todos os experimentos foi do tipo determinístico. Os padrões de teste externos foram criados baseados na abordagem do teste de software estrutural, que será vista no capítulo 5. A ferramenta Flextest™ permite, também, utilizar a funcionalidade de complementar um conjunto de padrões de teste inicial, gerados externamente, com padrões internos, os quais são gerados baseados em uma lista de falhas não detectadas pelo primeiro conjunto de vetores de teste. Esta segunda funcionalidade foi utilizada nesta tese e será detalhada também no capítulo 6. A única restrição do uso dos padrões de teste externos é que o formato do arquivo de entrada deve estar de acordo com os padrões que a ferramenta impõe. Diversos dados estatísticos são fornecidos por esta ferramenta, os quais podem ser habilitados ou não na própria ferramenta, seja através de *scripts* ou pela *interface* gráfica da mesma. Nesta tese utilizamos os seguintes dados: cobertura de falhas, quantidade de vetores de teste

gerados, total de falhas do circuito, falhas detectadas pelos padrões de teste e o tempo de teste.

5 APLICAÇÃO DE TÉCNICAS DE TESTE DE SOFTWARE AO TESTE DE HARDWARE

5.1 Introdução

No atual fluxo de projeto de hardware a geração de padrões de teste normalmente está inclusa em uma etapa que é realizada somente após a síntese lógica do circuito. Entretanto, a geração dos padrões de teste o mais cedo possível pode prover alta qualidade do teste e reduzir o esforço do ATPG.

Neste capítulo, será detalhada a aplicação de técnicas originárias da engenharia de software, baseadas no fluxo de controle de programas para o teste de caminho, para a extração de vetores de teste a partir de descrições HDL comportamentais. Resultados experimentais mostram que combinando a geração de vetores de teste no alto nível com os vetores gerados por ATPGs em nível de portas lógicas pode-se melhorar a qualidade do teste, aumentando a cobertura de falhas e/ou reduzindo o tamanho do conjunto de teste. A Figura 5.1 mostra um esquemático da ferramenta utilizada para a geração dos vetores de teste utilizando a abordagem proposta.

Nas próximas seções será detalhado cada um dos itens da ferramenta apresentada na Figura 5.1.

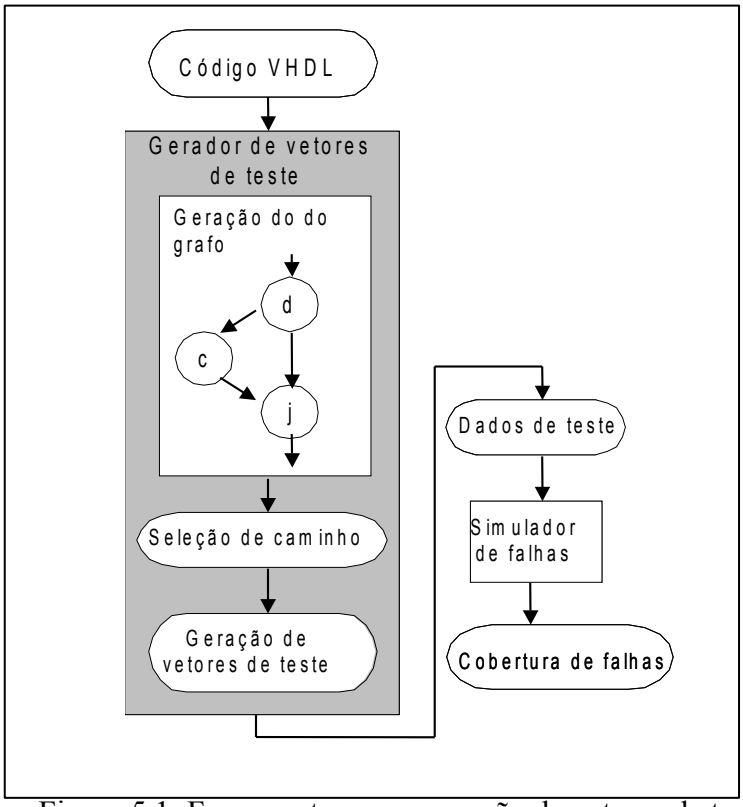

Figura 5.1: Ferramenta para a geração de vetores de teste

5.2 Descrições de Hardware Utilizadas

Para implementar os diferentes métodos da engenharia de software estudos utilizou-se um sub-conjunto de *benchmarks* do ITC'99, desenvolvidos pelos pesquisadores da Politécnico de Torino e cinco diferentes implementações de uma raiz quadrada [ANG 2005].

O sub-conjunto de *benchmarks* do ITC'99 utilizado consiste de 14 circuitos seqüenciais RTL, os quais apresentam estruturas diferenciadas com relação ao número de portas lógicas e *flip-flops* que os compõem. A Tabela 5.1 mostra as características deste sub-conjunto utilizado nos experimentos desta tese.

O outro grupo de descrições utilizado baseia-se na descrição de uma raiz quadrada, para a qual foram feitas várias modificações no estilo de descrição afim de analisar o efeito das mudanças sobre o comportamento da testabilidade do circuito após a síntese [ANG 2005]. As características do segundo conjunto de *benchmarks* podem ser visualizados na Tabela 5.2.

Cada algoritmo da Tabela 5.2 apresenta os sinais *r*, *d* e *s* em suas descrições VHDL, como variáveis que apresentam dependência de dados entre si, as quais são resolvidas através de máquinas de estados. As variáveis são implementas em registradores, cada uma com seu próprio operador. As diferenças entre cada versão referem-se a: parte de controle e parte operativa estarem implementadas juntas ou separadas; máquinas de estados descritas com construção IF-THEN-ELSE ou CASE; comunicação entre a parte de controle e operativa realizada através do estado da máquina ou por código microprogramado, e; número de estados da máquina de estados varia de uma descrição para outra [ANG 2005].

Tabela 5.1: Características dos *benchmarks* do ITC99 utilizados

Bench	Descrição	Linhas	Processos	Portas Lógicas	FFs	Pinos Entrada	Pinos Saída
b01	FSM que compara fluxos seriais	110	1	49	5	2	2
b02	FSM que reconhece números BCD	70	1	28	4	1	1
b03	Recurso de árbritro	141	1	160	30	4	4
b04	contabiliza min e max	102	1	737	80	8	11
b05	Elabora os conteúdos de memória	332	3	998	66	1	36
b06	Manipulador de interrupção	128	1	56	9	6	9
b07	Conta pontos em uma linha reta	92	1	441	41	1	8
b08	Procura inclusões seqüenciais de números	89	1	183	21	9	4
b09	Convertedor serial para serial	103	1	170	28	1	1
b10	Sistema de votação	167	1	206	21	11	6
b11	Altera *string* com variável cifrada	118	1	770	59	7	6
b12	Adivinha seqüências(jogo de um jogador)	569	4	1076	144	5	6
b13	Interface para sensores meteorológicos	296	5	362	59	10	10
b14	Processador Viper (*subset*)	509	1	10098	290	32	54

Tabela 5.2: Características das descrições da raiz quadrada utilizadas

Bench	Descrição	Linhas	Processos	Portas Lógicas	FFs	Pinos Entrada	Pinos Saída
sr1	Raiz quadrada, FSM de 7 estados e PC e PO separadas	117	2	317	35	11	5
sr2	Raiz quadrada, FSM de 6 estados e PC e PO juntas	102	1	334	26	11	5
sr3	Raiz quadrada, FSM de 4 estados e PC e PO separadas	118	2	319	30	11	5
sr4	Raiz quadrada, FSM de 7 estados e PC e PO juntas	112	1	297	24	11	5
sr5	Raiz quadrada, FSM de 3 estados e PC e PO juntas	88	1	317	30	11	5

5.3 Trabalhos Relacionados

A aplicação das técnicas de teste de software para auxiliar no processo de teste de hardware não é uma abordagem completamente nova. A diversidade existente nos critérios de teste de software tem motivado pesquisadores a conduzir estudos com o objetivo de avaliar custo e eficiência dos diferentes critérios quando aplicados em projetos digitais. Vários trabalhos já utilizaram uma ou mais destas técnicas em teste de hardware.

Os trabalhos encontrados na literatura, na maioria, são baseados em métodos de teste estrutural de software para a geração de vetores de teste em alto nível de abstração utilizados, principalmente, para a verificação do projeto.

Em [JOH 95] foi apresentado um experimento que gera um conjunto de dados de teste a partir de um **critério de teste de software estrutural para uma descrição VHDL estrutural** e utiliza a ferramenta denominada PROOFS [NIE 90] para avaliar se os vetores em nível lógico são funcionalmente equivalentes. A ferramenta PROOFS é um simulador de falhas que classifica o circuito pelo número de falhas *stuck-at* detectadas. A cobertura de falhas obtida é então comparada com os vetores de teste

gerados pela ferramenta HITEC [NIE 91], um gerador de teste determinístico em nível de porta lógica. Com isto, pode-se observar a correlação dos testes gerados para a descrição em nível de porta e os testes gerados para a descrição comportamental do circuito.

Neste estudo foram utilizados seis *benchmarks* para a realização das comparações entre a cobertura de falhas obtida pelos vetores de teste gerados a partir de técnicas de teste de software e os gerados pelo HITEC, sendo eles:

- s344, s444 e s641, um conjunto de *benckmarks* do ISCAS89 [BRG 89], e;
- TLC, GCD e am2901 são as descrições VHDL de alto nível correspondentes em nível de porta para o HLSynth92 [IEE 88].

Os resultados obtidos na cobertura de falhas em [JOH 95] mostram que o **teste de fluxo de dados**, sozinho, não apresenta um bom resultado na geração de vetores de teste se comparado com os vetores de teste gerados pelo HITEC. Adicionalmente, este trabalho possui a limitação relacionada ao tipo de *benchmarks* utilizados para testar cada uma das técnicas de teste estrutural de software implementadas.

O objetivo do trabalho apresentado por [JOH 95] assemelha-se ao desta tese, pois os dois geram vetores de teste para o teste de fabricação. Porém, a diferença está no nível de abstração utilizado nos dois trabalhos, onde o primeiro utiliza descrições estruturais e nesta tese o estilo de descrição utilizado é a comportamental (RTL). Outra diferença refere-se ao tipo de técnica de teste de software utilizada: em [JOH 95] utiliza-se teste de fluxo de dados e nesta tese, teste de fluxo de controle.

Uma proposta que combina técnicas baseadas em teste de software (**cobertura de sentença**) para a geração de seqüências de teste em alto nível, enriquecida com seqüências de teste utilizando técnicas aplicadas em nível de porta lógica, foi apresentada em [RUD 98]. Várias seqüências, geradas manualmente, foram criadas para assegurar 100% de cobertura de todas as sentenças em alto nível de descrições VHDL, ou para maximizar a cobertura de caminhos. Os experimentos (falhas detectadas, tempo e número de vetores gerados) foram comparados com os valores fornecidos pelas ferramentas HITEC e GATEST [RUD 94]. Os resultados da combinação entre alto nível e nível de porta mostraram-se bons com relação ao número de vetores gerados e ao número de falhas detectadas. Quanto ao tempo de geração de padrões de teste, este se mostrou muito variável dependendo do *benchmark* analisado. As Tabelas 5.3 e 5.4 apresentam os resultados apresentados no artigo de [RUD 98]. A primeira tabela (Tabela 5.3) refere-se à comparação entre o método de alto nível de abstração combinado com o nível de porta lógica comparado com os valores obtidos pelas ferramentas HITEC e GATEST. A outra tabela (Tabela 5.4) relata os valores obtidos pela aplicação da cobertura de caminho e cobertura de sentença diferenciando em cada uma a forma de geração dos padrões de teste (seqüencial ou aleatória).

Tabela 5.3: Comparação das coberturas de falhas de [RUD 98]

Bench	*Alto nível + nível de porta*					*HITEC*			*GATEST*		
	falhas	*Det*	*Vet*	*T*	*Estr.*	*Det*	*Vet*	*T*	*Det*	*Vet*	*T*
B01	135	133	44	6.02s	Seq	133	110	0.50s	133	80	12.0s
B02	72	69	27	5.80s	Seq	69	54	0.29s	70	69	10.1s
B03	452	334	105	46.5s	Rand	333	214	1.25h	334	169	1.12m
B04	1396	1204	113	1.17m	Rand	1177	303	1.42h	1217	220	4.60m
B05	1884	905	85	2.36m	Rand	913	396	11.9h	902	129	1.41m
B06	206	190	31	17.7s	Seq	190	89	0.89s	190	62	19.6s
B07	1271	888	100	6.80m	Seq	878	206	4.87h	871	88	1.39m
B08	489	311	54	1.37m	Seq	461	563	1.16m	261	84	46.3s
Barcode	1091	580	77	1.68m	Rand	689	1816	28.7h	552	161	4.52m
GCD	2199	1988	356	17.8m	Rand	1638	206	13.7h	1377	227	10.6m
DHRC	9468	8861	317	48.9m	Seq	8864	1094	15.2h	8860	820	1.55h
DIFFEQ	18,216	17,881	335	1.80h	Rand	17,730	803	23.6h	18,009	662	7.71h

Det – falhas detectadas Vet – quantidade de vetores utilizados T – tempo de teste
Estr – tipo de estratégia utilizada(seqüencial ou randômica)

Tabela 5.4: Comparação entre cobertura de caminho e sentença [RUD 98]

Bench	*Cobertura de Caminho*						*Cobertura de Sentença*					
	Seqüencial			*Aleatória*			*Seqüencial*			*Aleatória*		
	Det	*Vet*	*T*	*Det*	*Vet*	*T*	*Det*	*Vet*	*T*	*Det*	*Vet*	*T*
b01	133	44	6.02s	133	55	6.42s	128	38	4.76s	132	39	5.08s
b02	69	27	5.80s	69	30	6.37s	67	20	5.37s	69	29	5.44s
b03	334	108	36.4s	334	105	46.3s	334	108	36.3s	334	105	46.2s
b04	1189	95	1.16m	1204	113	1.17m	1199	91	1.26m	1200	104	1.28m
b05	153	31	2.55m	905	85	2.36m	232	71	2.48m	232	71	2.48m
b06	190	31	17.7s	190	41	17.8s	190	37	33.1s	190	30	33.1s
b07	888	100	6.80m	887	97	6.81m	877	67	6.45m	877	67	6.46m
b08	311	54	1.37m	301	43	1.50m	311	54	1.34m	302	40	1.47m
Barcode	573	91	1.93m	580	77	1.68m	575	110	3.36m	575	110	3.35m
GCD	1914	302	18.6m	1988	356	17.8m	1662	304	16.8m	1769	283	13.6m
DHRC	8861	317	48.9m	8843	312	51.6m	8860	404	1.29h	8860	404	1.24h
DIFFEQ	17,881	335	1.79h	17,881	335	1.80h	17,881	335	1.79h	17,881	335	1.80h

Det – falhas detectadas Vet – quantidade de vetores utilizados T – tempo de teste
Estr – tipo de estratégia utilizada(seqüencial ou randômica)

Na Tabela 5.4 as colunas denominadas seqüencial e aleatória representam a forma como foram gerados os vetores utilizados para os testes gerados, respectivamente, seqüencialmente e aleatoriamente.

O trabalho apresentado em [RUD 98] é o que na bibliografia estudada apresenta maior semelhança ao proposto e implementado nesta tese, principalmente no que se refere à geração de vetores combinando o alto nível de abstração aos vetores gerados em nível de portas lógicas. Cabe ressaltar que o estilo de descrição é o mesmo, apresentando até mesmo alguns *benchmarks* utilizados nesta tese (b01, b02, b03, b04, b06, b07 e b08), portanto ambos trabalhos utilizam descrições comportamentais VHDL. Entretanto, as ferramentas utilizadas para a validação e comparação dos dados obtidos são diferentes. Nesta tese utiliza-se a ferramenta Flextest da Mentor e em [RUD 98] utiliza-se HITEC e GATEST. Ao final deste capítulo será apresentada uma tabela comparativa dos resultados apresentados em [RUD 98] e os dados obtidos na técnica adotada nesta tese, referindo-se aos mesmos *benchmarks* (Tabela 5.14).

Em [FER 99] é apresentada uma abordagem para a geração de padrões de teste, considerando um modelo de falhas baseado em VHDL. Este modelo permite a cobertura de um grande número de erros de projeto, maior que a cobertura baseada em sentença. Neste trabalho a geração de teste é feita em cada processo em isolado através da injeção de falhas comportamentais no código VHDL. Como resultado desta abordagem tem-se: o conjunto de padrões de teste que cobre todos os bits e assim todas as sentenças de cada processo; a realização da análise da controlabilidade da interconexão entre os processos e; detecção de redundâncias no código VHDL, se existirem.

O critério de **cobertura de desvios** foi aplicado em [MAY 2000] como um critério para verificação de projetos de hardware descritos em VHDL. A metodologia é baseada na técnica de teste de software adaptada às necessidades impostas pela descrição VHDL, utilizada para analisar a sua aplicabilidade. Considera-se o modelo VHDL como uma rotina de software com alguma informação específica de hardware. O objetivo deste trabalho é apresentar um novo método para aumentar as medidas de cobertura de um conjunto de teste selecionado. O método escolhido foi a cobertura de desvio e os experimentos mostram que esta é uma boa estratégia para o teste do código VHDL, ou seja, a cobertura do código.

Realizando um confronto entre a abordagem utilizada nesta tese e a metodologia utilizada em [MAY 2000], pode-se destacar que esta utiliza-se do teste de software para verificação de projeto, além do fato de utilizar o método de cobertura de desvio, enquanto, nesta tese, utiliza-se cobertura de caminho para o teste de fabricação.

Uma métrica de cobertura de falhas baseada na **análise de fluxo de dados** foi proposta em [ZHA 2000]. Neste trabalho conclui-se que trata-se de um método eficiente para revelar falhas de projeto. A métrica proposta avalia a cobertura do fluxo de dados alcançada pelo uso de uma determinada seqüência de padrões de teste. A abordagem proposta é dividida em três passos: 1) geração do grafo de fluxo de uma descrição HDL comportamental; 2) seleção de um subgrupo de caminhos que serão executados, ou seja, a identificação de todos os *du pairs* [CHE 99] (este método é aplicado para monitorar o comportamento de cada caminho durante o processo de teste), e; 3) simulação da descrição HDL com os padrões de teste para determinar que fração dos caminhos será executada. Neste trabalho foi realizada uma comparação entre a cobertura obtida pelo método de fluxo de dados baseado na métrica de todos os usos de uma variável e a cobertura de sentença (Tabela 5.5).

Tabela 5.5: Comparação entre cobertura de sentença e *du pairs* [ZHA 2000]

Bench	*Sentenças*	*Du pairs*	*Du pairs cobertos*	*Sentenças cobertas*
ARMS_COUNTER	32	69	0.87	1
BARCODE	44	68	0.69	0.95
TLC	38	47	1	1
BUS_ARBITER	24	45	0.78	1
FIFO	59	92	0.86	1

Para comparar a cobertura de sentença com a cobertura de todos os *du pairs* foram selecionados padrões de teste pseudo-aleatórios suficientes para fazer a cobertura de sentenças chegar próxima de 100%. Os resultados apresentados mostram que a cobertura de sentença é melhor do que a de fluxo de dados utilizada. Isto significa que alguns casos de teste não são excitados pelos padrões de teste. Portanto, se as falhas estiverem nestes fluxos não exercitados não podem ser detectadas.

O trabalho apresentado em [ZHA 2000] difere da abordagem proposta nesta tese em diversos aspectos: 1) propõe-se à verificação do projeto, não ao teste; 2) foca no teste de fluxo de dados (*all du pairs*), e; 3) utiliza um conjunto de *benchmarks* diferente do utilizado nesta tese.

Em Jervan et al [JER 2002] inicialmente é realizada uma análise de vários modelos de falhas em alto nível de abstração (*statement coverage*, *bit coverage* [FER 2001] [LAJ 2000] e *condition coverage*) para selecionar o que melhor se adapta à estimativa de testabilidade de circuitos, a partir de suas descrições comportamentais, e para auxiliar no processo de geração do teste em nível comportamental. Para isso, a geração do teste em alto nível de abstração é feita utilizando um algoritmo de ATPG. O gerador de teste utilizado leva em consideração informações estruturais de níveis mais baixos de abstração, enquanto gera seqüências de teste em nível comportamental. Neste trabalho criou-se um ambiente denominado HTG (*Hierarchical Test Generation*) que aceita como entrada um código VHDL comportamental que é traduzido para o modelo DD (*Decision Diagram*). Este modelo é utilizado como uma plataforma matemática para geração de teste, para posteriormente ser transformado em um modelo Prolog, utilizado para resolver regras. Para resolver regras se utilizada a ferramenta comercial SICStus [SIC 2003].

O algoritmo HTG é utilizado para gerar os vetores de teste finais. Os resultados experimentais mostram que a cobertura de falhas é superior à obtida pelo ATPG de alto nível com menor número de vetores, sendo porém inferior à obtida pelo ATPG em nível de porta lógica (testgen),que requer um tempo de CPU muito superior. Esta característica pode ser vista nesta tese, também, pois uma das motivações para o uso do alto nível de abstração refere-se à velocidade de geração dos padrões de teste através da técnica de teste de software escolhida e o número reduzido de vetores gerados, com isso reduzindo o tempo de CPU necessário para a execução do processo de teste.

Na comparação entre os modelos de falhas em [JER 2002] conclui-se que quando os circuitos são *data-dominated* a cobertura de falhas baseada em sentença é baixa se comparada à cobertura de falhas em nível de porta lógica. Já a cobertura que utiliza o método de *bit coverage* combinado com a cobertura de condição apresenta uma alta correlação. Para circuitos *control-dominated* observa-se uma boa relação entre as duas coberturas: sentença e *bit* combinada com cobertura de condição. A Tabela 5.6 mostra

os valores retirados de [JER 2002]. Com estes dados conclui-se que a cobertura baseada em *bit* combinada com a de condição pode ser utilizada para avaliar a efetividade do conjunto de teste em alto nível de abstração.

Tabela 5.6: Comparação das coberturas obtidas em [JER 2002]

Tipo de circuito	***Cobertura de sentença***	***Cobertura bit + condição***
Data dominated	0.67	0.97
Control dominated	0.83	0.80

Em outro experimento realizado em [JER 2002] utilizou-se o modelo de falhas de alto nível de abstração e a cobertura de *bit* combinada com cobertura de condição. O algoritmo RMHC (*Random Mutation Hill Climber*) foi utilizado para gerar os vetores de teste. Os dados obtidos foram comparados com os obtidos através do ATPG em nível de porta lógica testgen, a Tabela 5.7 mostra os valores obtidos neste teste e a comparação realizada.

Tabela 5.7: Comparação da cobertura de falhas entre RMHC e testgen [JER 2002]

Bench	**ATPG em Alto Nível**			**testgen**		
	FC(%)	**Vetores**	**Tempo(s)**	**FC(%)**	**Vetores**	**Tempo(s)**
Biquad1	68.27	287	2,139	37.06	154	10,817
Biquad2	86.94	287	2,139	70.75	245	11,060
Fir1	91.27	2,413	1,157	94.71	8,742	12,211
Fir2	89.77	2,413	1,157	91.38	4,621	25,038
TLC	80.88	110	225	84.09	500	1,579

Um terceiro tipo de experimento foi, realizado no trabalho de [JER 2002] o qual foi denominado de geração de teste hierarquico (HTG – *Hierarchical Test Generation*). Esta técnica utiliza informações de diferentes níveis de abstração. Neste trabalho esta abordagem foi utilizada para explorar o nível comportamental. O algoritmo HTG gera dois tipos de teste. Um para testar o comportamento do sistema e outro para a implementação final. O conjunto de teste é gerado a partir de informações puramente comportamentais baseado em uma métrica de cobertura de código [JER 99]. Os dados referentes ao terceiro experimento podem ser vistos na Tabela 5.8.

Tabela 5.8: Comparação das diferentes coberturas de falhas [JER 2002]

	ATPG em Alto Nível			***ATPG Hierarquico em Alto Nível***			***ATPG em Nível de Porta Testgen***		
Bench	***FC(%)***	***Vetores***	***Tempo(s)***	***FC(%)***	***Vetores***	***Tempo(s)***	***FC(%)***	***Vetores***	***Tempo(s)***
DIF 1	97.25	553	954	98.05	199	468	99.62	1,177	4,792
DIF 2	94.57	553	954	96.46	199	468	96.75	923	4,475

Em [PAO 2002] a geração de vetores de teste, que é formada de estímulos (transições de valores de entrada) e de respostas esperadas, simula ambas as descrições: RTL e nível algorítmico. Seus resultados são comparados para verificar se a descrição RTL preserva o mesmo comportamento. Este trabalho foca em um gerador de dados de teste orientado a caminhos no qual apresenta uma ferramenta que aceita como entrada um programa escrito em VHDL e um **critério de teste de caminho**. Esta ferramenta gera automaticamente dados de teste. O teste baseado em caminhos é utilizado por ser mais rigoroso e eficiente, valendo-se da complexidade ciclomática como um índice do número de caminhos a serem testados em um programa.

A tabela 5.9 retirada de [PAO 2002] apresenta os resultados obtidos com o uso do software desenvolvido sobre um sub-conjunto de *benchmarks* do ITC99: número de nodos do grafo (#n); número de arcos do grafo (#e); o número da complexidade ciclomática (#v(G)); o tempo para a construção do grafo (#t1), e; o tempo de geração do conjunto de caminhos (#t2).

Tabela 5.9: Tempo para a geração de caminhos de teste de [PAO 2002]

Bench	***#n***	***#e***	***#v(G)***	***#t1***	***#t2***
b01	50	67	19	0.16	0.06
b02	29	40	13	0.05	0.00
b03	73	90	19	0.22	0.00
b04	47	57	12	0.11	0.05
b05	153	203	52	0.77	0.06
b06	64	79	17	0.17	0.05
b07	45	58	15	0.11	0.00
b08	33	42	11	0.05	0.00
b09	45	54	11	0.11	0.00
b10	96	123	29	0.28	0.05
b11	56	77	23	0.22	0.06
b12	254	331	79	0.88	0.05
b13	138	192	56	0.60	0.05
b14	366	514	164	1.38	0.11
b15	381	485	111	1.38	0.10
b20	736	1035	329	3.52	0.39
b22	1100	1548	492	5.99	0.61

A diferença do trabalho apresentado por [PAO 2002] e esta tese se encontra no objetivo dos vetores gerados. Naquele caso utiliza os dados gerados para a verificação do projeto. Além disso, não apresenta resultados relacionados à cobertura de falhas obtida, o que nesta tese é um dos dados mais relevantes.

A **análise de mutantes** vem sendo adaptada para o uso no teste de circuitos complexos de hardware em diversos trabalhos [HAY 96] [AKT 98] [ALH 99] [BEN 99] [VAD 2000] [SHO 2003] [SHO 2005] e segundo [AKT 98] o uso de mutação é um sucesso no teste de dispositivos de hardware. O conjunto de teste resultante é aplicado

na estrutura em nível de porta do sistema, para medir sua capacidade de cobrir falhas de hardware assim como falhas *stuck-at*.

Em [HAY 96] utiliza-se a ferramenta Mothra [KIN 91] para a geração dos programas mutantes, a qual cria automaticamente todos os **mutantes** de um programa. Esta ferramenta, porém, só aceita como entrada um programa escrito na linguagem de programação Fortran, por isso existe a necessidade de uma interface que traduza a descrição em VHDL para Fortran.

Em [AKT 98] é proposta uma abordagem para adaptar a **análise de mutantes** para gerar dados de teste para descrições VHDL comportamentais. A idéia básica é definir uma estratégia unificada que teste ambas as especificações comportamentais, vistas como um programa de software junto ao modelo de falhas de fabricação do dispositivo de hardware. Para gerar testes adequados à implementação em hardware, características de hardware devem ser consideradas. Características como comprimento em bits de sinais e variáveis na descrição do hardware são orientações típicas no processo.

Com a **análise de mutantes** é possível gerar vetores de teste com maior possibilidade de detectar falhas. Em [BEN 99] concluiu-se que este método de teste gera vetores eficientes no caso de dispositivos programáveis, apresentando cobertura de falhas em torno de 90%. Neste trabalho é apresentada a aplicação da análise de mutantes em FPGAs (*Field-Programmable Gate Arrays*) para a geração de vetores de teste. Uma vez obtidos os vetores, eles são manipulados por um processo intenso. Este processo permite que propriedades do hardware, assim como tamanhos de dados, que não são considerados no teste de software, sejam levados em conta. O conjunto de vetores de entrada é aplicado ao hardware sintetizado. Neste trabalho, a síntese em FPGAs é considerada condição para o teste funcional. A Figura 5.2 mostra estes passos.

Segundo [ALH 99], descrições funcionais VHDL são escritas, verificadas e corrigidas como programas escritos em linguagens de programação tradicionais, assim um erro de uma descrição VHDL tem as mesmas causas e efeitos que em um software, por isso o uso de **análise de mutantes**. No teste de mutação o modelo de falhas é representado pela mutação de operadores. Este estudo definiu o modelo de falhas para descrições funcionais VHDL pela seleção de subconjuntos de operadores usados no teste de software. Este subconjunto consiste de 9 operadores de mutação vistos na Tabela 5.10.

Figura 5.2: Utilização da análise de mutantes na geração de teste

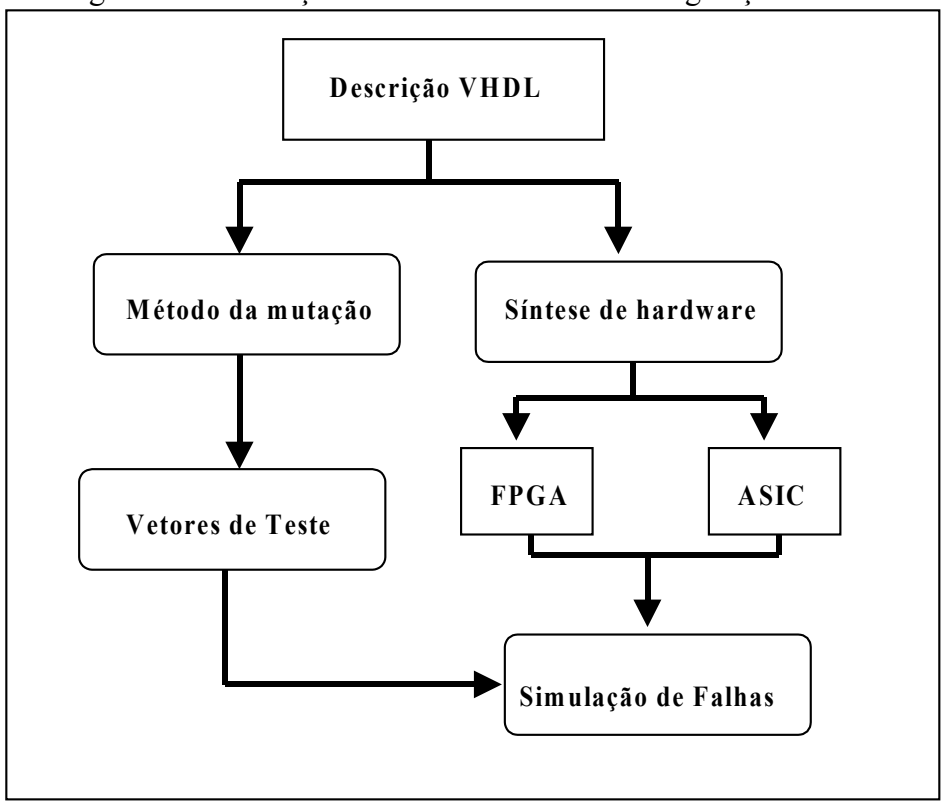

Tabela 5.10: Operadores de mutação para descrições funcionais VHDL

Tipo	Descrição
AOR	Substituição do operador aritmético
ABS	Inserção de valor absoluto
CR	Substituição de constante
CVR	Substituição de constante por variável
ROR	Substituição de operador relacional
LOR	Substituição de operador lógico
VCR	Substituição de variável por constante
VR	Substituição de variável
UOR	Inserção de operador unário

Para validar a proposta de aplicação do teste de mutação a descrições VHDL, Al-Hayek [ALH 99] propôs o processo visto na Figura 5.3. Neste, uma ferramenta é utilizada para gerar a estrutura em nível de portas para descrições VHDL. Para medir a qualidade do conjunto de dados gerados pelo teste de mutação é necessário o uso de um simulador de falhas em nível de portas, o qual calcula a cobertura das falhas de mutação de um hardware falho assegurada por um conjunto de dados de teste. A ferramenta de síntese é utilizada para gerar uma estrutura em nível de portas a partir de uma descrição VHDL.

Uma das dificuldades para aplicar o teste de mutação refere-se à geração de dados de teste. Em [ALH 99] foram propostas duas abordagens. A primeira refere-se à geração de dados de teste aleatórios a partir de uma descrição VHDL. A segunda abordagem refere-se à geração determinística. Esta abordagem foi proposta em [DEM 91] e é baseada em regras para a geração de dados de teste que faz a diferença no

comportamento do mutante. Uma ferramenta, chamada Godzilla foi implementada para gerar dados de teste para programas Fortran 77. Em [AHL 99] esta ferramenta foi utilizada para a geração de dados de teste para descrições VHDL.

Em [TER 99] foi realizado um estudo utilizando a **análise de mutantes** para tornar uma descrição de um dispositivo de hardware confiável. Neste trabalho foi realizada uma comparação da cobertura de falhas obtida através de métodos tradicionais de teste de hardware utilizando-se o modelo de falhas *stuck-at* e a aplicação da técnica de análise de mutantes. Utilizando-se um mesmo número de vetores de teste conclui-se que a cobertura obtida através da análise de mutantes é igual ou ligeiramente menor que a obtida pelo método de teste de hardware utilizando o modelo de falhas *stuck-at*, mostrando que a análise de mutantes proporciona uma medida conservadora da cobertura de falhas.

Vado et al [VAD 2000] estudaram a validação de hardware baseada em **mutação**, através de uma ferramenta escrita em C. Um conjunto de programas mutantes foi aplicado em vários estilos de códigos VHDL (comportamental, funcional e misto). A ferramenta, inicialmente, analisa segmentos da descrição para determinar programas mutantes que sejam relevantes, pois criando apenas mutantes relevantes o número de mutantes equivalentes é reduzido. Em um segundo passo, os programas de mutação são aplicados em descrições VHDL para produzir mutantes destas descrições. Os vetores funcionais gerados pelo projetista são aplicados na descrição original e nos mutantes. Os resultados da simulação produzida pela descrição original são armazenados e comparados com os resultados da simulação com as descrições dos mutantes. Se uma diferença for detectada, o mutante é definido como morto. Se der diferença, o mutante é colocado de lado para um processamento futuro, e um novo mutante é gerado, simulado e comparado com a assinatura da descrição original. Ao final, após a geração de todos os mutantes, quando todo o processo foi realizado e os mutantes sobreviventes colocados em uma lista, a cobertura de falhas obtida através da validação pode ser aumentada e o processo reiniciado para os mutantes sobreviventes. Para contabilizar a eficiência este método foi comparado com a validação de hardware baseada na ferramenta Mothra, onde a metodologia criada apresentou um número reduzido de mutantes utilizados, bem como o número de mutantes equivalentes.

Sholivé [SHO 2003] motivado pelo fato dos ATPGs consumirem muito tempo e memória, além do fato das estratégias de DFT tenderem para as técnicas aleatórias permitindo baixa otimização do teste, utilizou a **análise de mutantes** aplicados com três diferentes operadores de mutação, utilizando um gerador de mutantes: 1) alteração em operadores relacionais (LOR – *logic operator replacement*); 2) alteração em constantes (CR – *constant replacement*), e; 3) alteração em variável (VR – *variable replacement*). Os experimentos foram realizados sobre dois *benchmarks* ITC99, o b01 e o b02, e as coberturas foram comparadas com as obtidas pela ferramenta Flextest, mostrando que o número de vetores gerados pela ferramenta baseada na análise de mutantes é menor e a cobertura de falhas semelhante, como pode ser visto na Tabela 5.11.

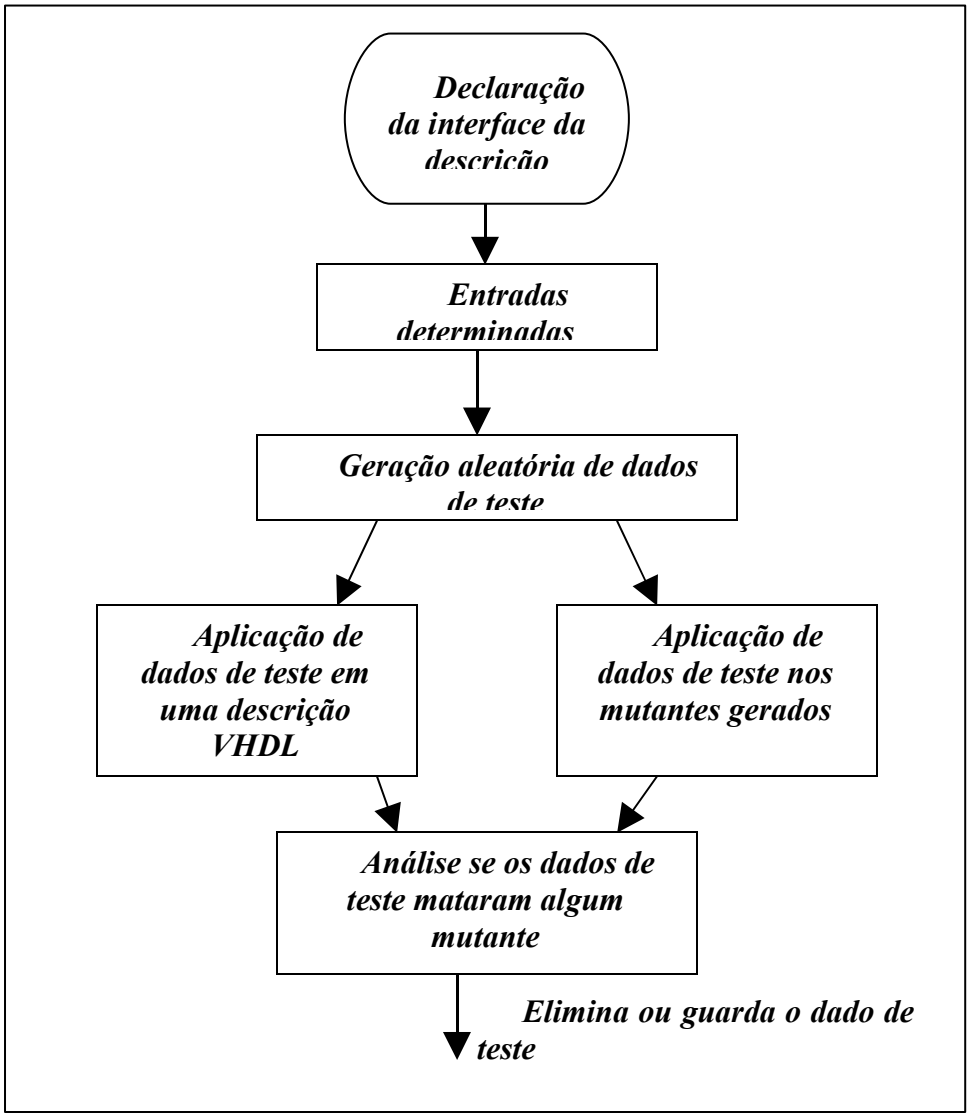

Figura 5.3: Geração de dados de teste aleatório para a análise de mutantes

Tabela 5.11: Comparação entre o uso de mutação e Flextest [SHO 2003]

	Mutação		Flextest	
bench	Vetores	Falhas(%)	Vetores	Falhas(%)
B01	43	98,6	80	99,6
B02	17	94,8	41	99,1

Em [SHO 2005] foi realizada uma comparação entre o conjunto de teste gerado pela técnica de **mutação** com a de teste aleatório de algumas descrições seqüenciais. Deste foram tiradas algumas informações: 1) o tamanho do conjunto e teste obtido com a abordagem da validação baseada na análise de mutantes é sempre menor que a seqüência de teste pseudo-aleatória; 2) a cobertura de falhas da análise de mutantes é superior à obtida pelos dados de teste aleatório, mostrando que estes dados de validação detectam um número superior de falhas difíceis de testar. O uso de teste de mutação

para circuitos complexos é uma estratégia que reduz o tempo de simulação preservando a eficiência da validação e do teste.

Recentemente os critérios de teste estrutural e de **mutação** vêm sendo investigados para a validação de especificações SDL (*Structural Description Language*) [SUG 2005]. Neste artigo é avaliado o teste de mutação, dois testes baseados no fluxo de controle (bloco e decisão) e três baseados no fluxo de dados (*c-use*, *p-use* e *all-uses*). Com os resultados obtidos, verificou-se que: o conjunto de dados de teste gerado para o teste de mutação é também adequado para o critério baseado no fluxo de controle e de dados, mas o contrário não é verdadeiro.

5.4 Geração de Teste Baseado em Caminho

O teste baseado em caminho, também chamado de teste baseado em fluxo de controle, foi proposto, inicialmente, por McCabe [MCC 76]. Nesta técnica, vetores de teste devem assegurar a execução de cada instrução do programa pelo menos uma vez. Para isso, a geração de padrões de teste deve partir de um grafo de programa, onde os nodos representam os comandos executáveis e os arcos representam o fluxo de controle entre os comandos.

No grafo de programa, arcos somente existem entre dois nodos se um comando sucessor pode ser executado imediatamente após o seu predecessor. Os nodos do grafo possuem blocos de comandos executáveis do programa. Conseqüentemente, eles não incluem declarações de variáveis. Em cada bloco, a execução do primeiro comando deve ter somente um predecessor, exceto o primeiro, e tem somente um sucessor, exceto o último.

A ferramenta proposta nesta tese (Figura 5.1), inicialmente identifica todos os comandos executáveis e, posteriormente, gera os caminhos para percorrer cada aresta do CFG, necessitando para isso identificar quais são as entradas primárias (sinais de entrada) e as variáveis internas que possuem influência sobre os caminhos.

Para a realização da geração de caminhos basta percorrer toda a descrição VHDL e numerar cada comando executável. No caso de comandos condicionais são gerados caminhos diferentes para a condição falsa e para a verdadeira. A geração dos vetores de teste baseia-se nos caminhos gerados e nos sinais de entrada. Finalmente, para cada caminho gera-se uma seqüência de vetores que irá cobri-lo. Estes vetores são escritos em um arquivo texto com os padrões utilizados pela ferramenta Flextest.

5.4.1 Construção do Grafo de Fluxo de Controle

O primeiro passo do processo de teste de caminho é a construção do grafo de programa que representa o fluxo de controle da descrição comportamental do circuito. Já que a técnica de teste baseada em fluxo de controle foi originalmente desenvolvida para o teste de software em programas seqüenciais, inicialmente não é considerado o paralelismo proveniente da existência de mais de um processo. Com isso, elimina-se os problemas que podem ocorrer quando existe concorrência no programa (que é o caso do hardware). Posteriormente, será apresentado como se procede a construção do grafo de programa quando existe mais de um processo na descrição do circuito.

O grafo de programa é construído somente para comandos existentes dentro do processo. Conseqüentemente, os passos descritos aqui aplicam-se somente para o bloco da descrição VHDL. Inicialmente comandos que estão dentro do processo são

numerados para identificar todas as sentenças que irão compor o grafo. A seguir cada linha do código é lida para verificar se esta possui uma sentença condicional (if ou case). Estas sentenças irão causar bifurcações no grafo. Para facilitar a geração do grafo, sentenças case são convertidas em árvores binárias. Comandos entre sentenças condicionais irão compor um bloco seqüencial, que é, um nodo do grafo. A Figura 5.4 apresenta o algoritmo que lê uma descrição VHDL do circuito e gera seu grafo de fluxo de controle.

```
Lines_Numeration //Numera linhas do código para gerar grafo
  cont = 0
  loop Parser  //para cada linha
    if (line != "process") && (line != end_file)
      cont = cont + 1
      Numerate()  //associa um número para a sentença
    end if
  end loop

Graph_Generation
  loop Parser     //para cada linha
    if line == ”if” or “elseif”
//inclui sentenças no lado direito até encontrar um ELSE ou
//ENDIF
      Graph[right_link] = line_number
    else
      if line == “else”
//inclui sentenças no lado esquerdo até encontrar ENDIF
        Graph[left_link] = line_number
      else
        if line == “case”
          loop When
//inclui no lado direito até encontrar o segundo WHEN
            Graph[right_link] = line_number
          end loop
        else
          if line == “when”
            Create_Node_Brother()
          else
            Graph[right_link] = line_number
          end if
        end if
      end if
    end if
  end loop
```

Figura 5.4: Algoritmo para a construção do grafo de fluxo de controle

A Figura 5.5 apresenta um exemplo de uma descrição VHDL incompleta. Neste código as sentenças já estão numeradas. O grafo de fluxo de controle gerado pela aplicação do algoritmo da Figura 5.4 no código VHDL da Figura 5.5 é ilustrado na Figura 5.6. Neste exemplo, sentenças executadas seqüencialmente foram agrupadas em nodos simples. A Figura 5.7 representa o grafo resultante da transformação do grafo da

Figura 5.6 em uma árvore binária, ou seja, nota-se que a sentença case não origina múltiplos filhos, mas gera uma sub-árvore binária.

Árvore [WIR 89] em computação é uma das mais importantes classes de estruturas de dados. É uma estrutura que se caracteriza por uma relação de hierarquia que utiliza algoritmos relativamente simples, recursivos eficientes.

Segundo a teoria dos grafos [WIL 97], uma árvore binária é definida como um grafo acíclico, conexo, dirigido, em que cada nó não possui grau maior que 3. Assim sendo, só existe um caminho entre dois nós distintos.

Para facilitar a manipulação deste tipo de estrutura de dados existe uma regra para transformar uma árvore qualquer (grafo) em uma árvore binária, facilitando assim a manipulação desta estrutura. A transformação segue os seguintes passos:

1) A raiz da árvore será a raiz da árvore binária;
2) O nodo filho mais à esquerda da árvore (sub-árvore) será o filho mais a esquerda da raiz da árvore (sub-árvore) binária. Cada nodo irmão da esquerda para a direita será o nodo filho à direita do nodo irmão à esquerda, até que todos os nodos filhos da raiz da árvore (sub-árvore) já tenham sido incluídos na árvore binária em construção.

Em alguns experimentos descritos mais adiante, alguns circuitos possuem dois blocos de processo. Conseqüentemente, é necessário adaptar o processo de construção do grafo orig reve o caminho de controle do c ro descreve o *data path*. A cons ntrole do circuito é trivial e segu sso que executa o *data path*, g os com comandos executados e nodos de estado no grafo que rep pectivos grafos dos *data paths*. E nde cada estado da máquina de e tado.

```
process (clock, reset, line1, line2)
1   if reset = '1' then
2     state := a;
3     output <= '0';
4     overflow <= '0';
5   elsif clock'event and clock = '1' then
6     case state is
7       when a =>
8         if line1 = '1' and line2 = '1' then
9           state := f;
10        else
11          state := b;
12        end if;
13        output <= line1 xor line2;
14        overflow <= '0';
15      when e =>
16        if line1 = '1' and line2 = '1' then
17          state := f;
18        else
19          state := b;
20        end if;
21        output <= line1 xor line2;
22        overflow <= '1';
23      when b =>
.         ...
.     end case;
.   end if;
     end process;
```

Figura 5.5: Exemplo de um código VHDL

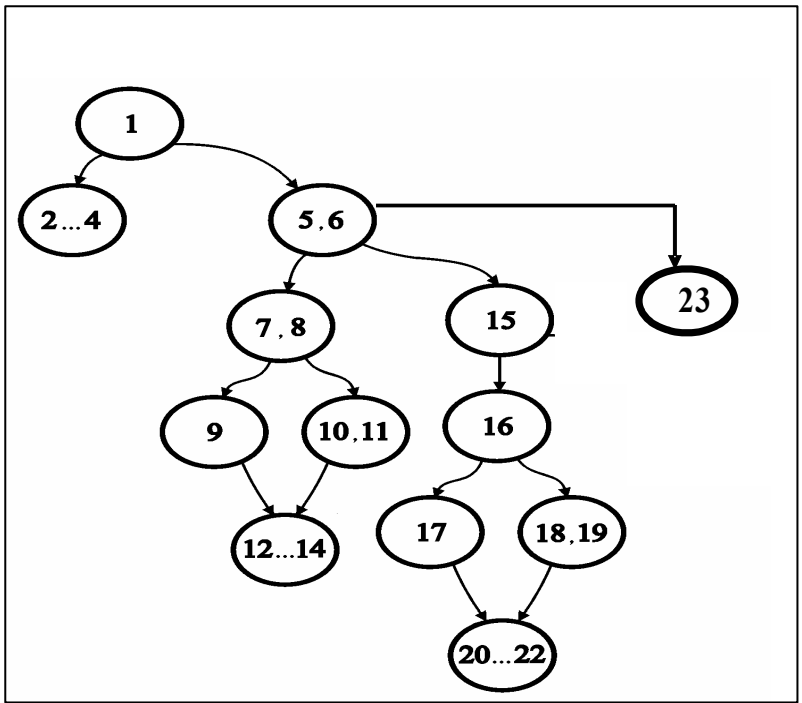

Figura 5.6: Grafo de fluxo de controle do código VHDL (árvore)

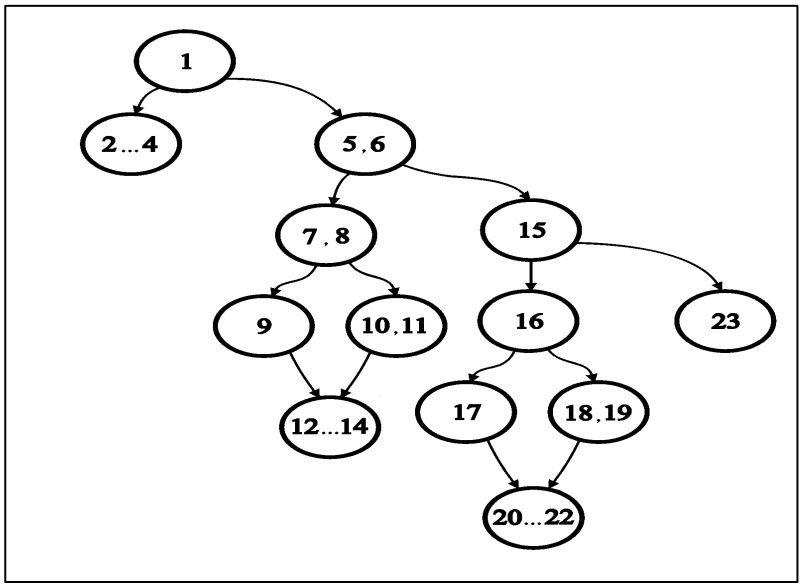

Figura 5.7: Exemplo da árvore binária representando o grafo de fluxo de controle

5.4.2 Geração dos Vetores de Teste

Uma vez construído o grafo de fluxo de controle deve-se identificar todos os possíveis caminhos e selecionar todos ou alguns destes para a geração do teste. Para o grafo ilustrado na Figura 5.7, os caminhos possíveis são: a) 1, 2, 3,4; b) 1, 5, 6, 7, 8, 9 12, 13, 14; c) 1, 5, 6, 7, 8, 10, 11, 12, 13, 14; d) 1, 5, 6, 15, 16, 17, 20, 21, 22; e) 1, 5, 6, 15, 16, 18, 19, 20, 21, 22, e; f) 1, 5, 6, 15, 23.

Baseado nos caminhos identificados no grafo, vetores de teste podem ser gerados. Considera-se que todos os caminhos no grafo devam ser cobertos. Entretanto, deve-se encontrar valores para as entradas do circuito que forcem a execução de todos os comandos em cada caminho. Nota-se que para exercitar alguns caminhos, muitas vezes é necessária a execução de outros caminhos mais de uma vez, de modo que alguns sinais ou variáveis internas cheguem ao seu valor apropriado. Este procedimento é

exaustivo e consome muito tempo de processamento para ser realizado manualmente, principalmente para descrições HDL complexas, portanto requer automatização.

A Figura 5.8 apresenta o algoritmo que realiza a geração das seqüências de teste que causam a execução dos caminhos do grafo de fluxo de controle do programa.

```
   Paths_Identification
     Generate_Paths_List
   //visita todos os nodos em pré-ordem para o caminho
//selecionado


   Vectors_Generation
     loop Visit_Paths_List
       Read_Code(Path)
       if line == "if" or "case"
         if variable == INPUT_LIST
           Create_Vector (variable IN)
         else
   //verifica dependências em variáveis ou sinais internos
           Verify_Dependencies()
           Create_Vector(variable IN)
         end if
       else
         Create_Vector(variable IN)
       end if
     end loop
```

Figura 5.8: Algoritmo para gerar padrões de teste

Se a variável que participa de um comando condicional (if ou case) é um sinal de entrada, para isso utiliza-se a lista INPUT_LIST, e insere na lista de valores de entrada um valor apropriado para a mesma. Entretanto, se a variável não é um sinal de entrada a rotina verifica as dependências da variável utilizada no comando condicional até encontrar um sinal que seja de entrada, adicionando-se assim um valor apropriado no vetor.

A Tabela 5.12 mostra uma possível seqüência de teste para executar os caminhos *a*, *b*, *c* e *f* do exemplo apresentado na Figura 5.5 e Figura 5.7, onde clock, reset, line1 e line2 são os pinos de entrada do circuito. Nota-se que para executar os caminhos *b* e *c* o caminho *a* deve ser executado duas vezes.

Tabela 5.12: Exemplo de geração de padrões de teste

Caminho	***Pinos de Entrada***			
	clock	**reset**	**line1**	**line2**
a	0	1	0	0
b	0	0	1	1
	1	0	1	1

a	0	1	0	0
c	0	0	0	0
	1	0	0	0
f	0	0	0	0
	1	0	0	0

5.4.3 Resultados Experimentais

Para avaliar a técnica de teste de software baseada no fluxo de controle aplicada ao teste de hardware, aplicou-se esta abordagem nos circuitos descritos nas Tabelas 5.1 e 5.2.

Os resultados obtidos a partir dos vetores de teste gerados no alto nível de abstração foram comparados com os resultados oriundos do ATPG em nível de portas lógicas. Em ambos os casos a ferramenta Flextest foi utilizada para a obtenção dos valores relacionados a cobertura de falhas. O Flextest foi também utilizado para a geração dos vetores de teste em nível de porta lógica. Também foram realizados experimentos onde a geração de vetores de teste foi complementada com os vetores de teste gerados pelo ATPG em nível de portas lógicas. O objetivo foi o de verificar se esta abordagem pode obter melhores coberturas de falhas que aquela utilizando apenas o ATPG em nível de portas lógicas, com menos padrões de teste.

A Tabela 5.13 apresenta os resultados, em termos de cobertura de falhas do tipo *stuck-at* (CF) e número de vetores de teste utilizados [KRU 2006a]. Nesta tabela, a coluna mais a esquerda identifica os circuitos utilizados. As segunda e terceira colunas apresentam os resultados obtidos através do ATPG em nível de porta lógica, utilizando a ferramenta Flextest. As próximas duas colunas apresentam resultados da abordagem baseada no alto nível de abstração, utilizando a técnica de teste de software baseada no fluxo de controle da descrição. Finalmente, as duas colunas mais a direita apresentam resultados que utilizam padrões de teste gerados em alto nível de abstração (os mesmos utilizados nas duas colunas anteriores) como um conjunto inicial para o ATPG em nível de porta lógica (Flextest) complementá-lo com outros vetores afim de detectar falhas não detectadas com o conjunto inicial.

Tabela 5.13: Comparação entre coberturas do Flextest e da abordagem proposta

Bench	***Gate Lavel ATPG***			***High Level ATPG***			***Gate Level + High Level TPG***		
	CF(%)	***Vet.***	***T(s)***	***CF(%)***	***Vet.***	***T(s)***	***CF(%)***	***Vet.***	***T(s)***
b01	97.50	47	1.0	92.50	17	0.6	97.08	**31**	**0.9**

b02	**96.09**	24	0.9	86.72	14	0.5	92.19	26	1.0
b03	71.74	135	72.7	67.01	23	0.6	71.51	**117**	**71.8**
b04	49.86	41	26.5	38,56	12	1.3	**85.92**	198	31.4
b06	99.59	46	1.0	95.53	32	0.6	**99.60**	**44**	**0.7**
b07	6.37	43	175.3	**53.08**	87	1.7	**53.08**	87	**126.9**
b08	86.81	221	23.5	57.91	37	0.8	**90.44**	244	30.9
b09	81.54	418	22.7	67.85	51	0.7	**85.09**	**331**	26.0
b10	23.08	44	18.0	**49.55**	**24**	0.6	**77.11**	98	23.5
b11	22.90	31	172.3	**65.51**	107	3.5	**65.63**	108	**3.5**
b14	51.58	542	4979.8	34.17	146	47.9	**60.83**	984	**4088.3**
sr1	78.21	277	445.1	78.11	100	1.7	**78.32**	**113**	**10.2**
sr2	**89.65**	403	491.5	81.28	100	0.8	88.33	259	**28.6**
sr3	**88.83**	588	894.8	80.73	98	1.0	83.40	193	**24.4**
sr4	85.84	555	834.7	58.95	94	1.0	85.51	**439**	1770.8
sr5	71.89	917	745.7	65.24	104	1.3	**79.51**	**278**	**24.8**

CF(%) – percentual da cobertura de falhas obtida na simulação de falhas
Vet . – quantidade de vetores de teste utilizados na simulação das falhas
T(s) – tempo de teste, medido em segundos

Observando a Tabela 5.13, a abordagem que utiliza o alto nível de abstração gera, na maioria dos casos, um menor número de vetores de teste se comparado ao uso do ATPG em nível de porta lógica. Nestes casos, a cobertura de falhas obtida é menor. Entretanto, circuitos que possuem um maior número de elementos seqüenciais, como é o caso dos *benchmarks* b07, b10 e b11, faz a tarefa do ATPG em nível de portas lógicas ser mais difícil. Para estes circuitos, a abordagem baseada no alto nível de abstração gerou mais padrões de teste e, conseqüentemente, obteve maior cobertura de falhas. Nota-se, no entanto que, o circuito b10 obteve maior cobertura de falhas usando um menor número de vetores de teste.

Outra informação apresentada na Tabela refere-se ao tempo de execução do procedimento de preparação do teste (T) de cada circuito, onde inclui-se o tempo de geração dos padrões de teste e o tempo de simulação para a obtenção das cobertiras de falhas (CF). Nota-se que o tempo para a preparação do teste, na maioria dos casos, é inferior ao tempo utilizado pela ferramenta Flextest.

Quando o teste de alto nível de abstração é complementado com os padrões gerados pelo ATPG em nível de porta lógica, obtêm-se na maior parte dos casos: b04, b06, b07, b08, b09, b10, b11, b12, sr1 e sr5, maiores coberturas de falhas que aqueles utilizando somente o ATPG em nível de porta lógica. Em alguns destes casos: b06, b09, sr1 e sr5, a abordagem do complemento apresenta maior cobertura de falhas utilizando um menor conjunto de teste. Em casos onde a cobertura de falhas é menor do que aquela obtida utilizando-se apenas os padrões gerados pelo ATPG em nível de porta lógica, mas muito próxima desta: b01, b03 e sr4, o conjunto de vetores de teste é menor. Somente em três casos dos 16 circuitos avaliados (b02, sr2 e sr3) o uso da técnica de teste de software baseada no fluxo de controle não trouxe nenhuma vantagem.

5.5 Considerações Finais

A abordagem da análise de mutantes, ou teste baseado em falhas, assemelha-se em muito à simulação de falhas, pois ambas consistem basicamente em simular um sistema na presença de falhas e comparar os resultados com aqueles obtidos com o sistema sem falhas. Esta abordagem já foi aplicada em diversos trabalhos relacionados ao teste de descrições de hardware, como foi descrito anteriormente, onde pode-se observar que os

vetores gerados obtiveram uma boa cobertura de falhas. Porém, este tipo de estratégia consome um tempo de computação considerável, principalmente para a geração dos mutantes.

A abordagem baseada na métrica de teste de software de fluxo de dados, quando utilizada sozinha, não apresenta bons resultados quanto à cobertura de falhas, como foi apresentado em [JOH 95] e em [ZHA 2000].

Analisando os diversos trabalhos realizados nota-se que a motivação está em encontrar um método que garanta boa cobertura de falhas utilizando, se possível, um reduzido tempo de computação e uma quantidade mínima de vetores de teste.

Os diversos trabalhos que utilizam os métodos de fluxo de controle, principalmente os baseados em caminhos, para a validação de hardware, apresentam bons resultados, pois estas abordagens proporcionam controlabilidade do código. Além do fato que os métodos de teste de software utilizados para validar uma descrição HDL adotam a abordagem de cobertura de código, onde se verifica todas as sentenças de um código e se os vetores gerados forçam a passagem por cada uma das sentenças.

A Tabela 5.14 faz uma comparação em termos de cobertura de falhas e quantidade de vetores utilizados entre o trabalho apresentado na seção de trabalhos relacionados [RUD 98] e os dados obtidos durante a fase de experimentos desta tese. Comparou-se estes dados por dois motivos: 1) semelhança na abordagem proposta,e; 2) uma razoável quantidade de *benchmarks* em comum, no caso b01, b02, b03, b04, b06, b07 e b08 do ITC99.

Tabela 5.14: Comparação dos dados coletados em RUD98 e os adquiridos na tese

	RUD98						***Tese***			
	Alto Nível + Porta		***HITEC***		***GATEST***		***Original Flextest***		***Alto Nível + Porta***	
Benc	***CF(%)***	***Vet***	***CF(%)***	***Vet***	***CF(%)***	***Vet***	***CF(%)***	***Vet***	***CF(%)***	***Vet***

h										
b01	**98.52**	**44**	98.52	110	98.52	80	97.50	47	**97.08**	**31**
b02	**95.83**	**27**	95.83	54	97.22	69	96.09	24	92.19	26
b03	**73.89**	**105**	73.67	214	73.89	169	71.74	135	**71.51**	**117**
b04	**86.25**	113	84.31	303	87.18	220	49.86	41	**85.92**	198
b06	92.23	31	92.23	89	92.23	62	99.59	46	**99.60**	**44**
b07	**69.87**	**100**	69.08	206	68.53	88	6.37	43	**53.08**	87
b08	**63.60**	**54**	64.27	563	53.37	84	86.81	221	**90.44**	244

Os resultados experimentais da abordagem proposta para a geração de padrões de teste a partir de descrições comportamentais mostram que utilizar apenas o alto nível de abstração não é vantajoso. Entretanto, combinando os vetores de teste gerados a partir das descrições comportamentais com os vetores gerados pelo ATPG em nível de porta pode, na maioria dos casos, aumentar a cobertura de falhas e reduzir o esforço do ATPG. Em outros casos, pode-se reduzir o tamanho do conjunto de teste e mesmo assim obter aumento na qualidade do teste.

Nota-se que em ambos os trabalhos a combinação entre o alto nível de abstração e o nível de portas lógicas apresentou na maioria dos *benchmarks* cobertura de falhas maior com um número menor de vetores de teste gerados, se comparado com os ATPGs comerciais, também utilizados na comparação.

Analisando-se a Tabela 5.14, verifica-se que a abordagem desta tese leva a melhores coberturas de falhas nos circuitos b06 e b08, em ambos os casos necessitando um número maior de vetores. [RUD 98] atinge melhores resultados nos circuitos b01, b02, b03, b04 e b07, com um maior número de vetores em b01, b02 e b07, e com menos vetores em b03 e b04. Detendo-se naqueles circuitos para os quais a abordagem proposta perde em termos de cobertura de falhas, verifica-se que em 4 dos 5 *benchmarks* (b01, b03, b04 e b07), o ATPG em nível de porta lógica utilizado nesta tese (Flextest) sempre levou a uma cobertura de falhas inferior aos ATPGs utilizados em [RUD 98] (HITEC e GATEST). No caso dos circuitos b04 e b07, por exemplo, a cobertura de falhas dos vetores gerados pelo Flextest ficou muito aquém das obtidas pelo HITEC e GATEST. Nos outros dois casos (b01 e b03) a diferença percentual de cobertura entre os ATPGs puros é comparável à diferença entre a abordagem “alto nível + porta lógica” desta tese e de [RUD 98]. Uma comparação mais justa só será possível se os mesmos ATPGs tivessem sido usados nos dois trabalhos e para um número maior de *benchmarks*.

6 TESTE DE SOFTWARE UTILIZADO NA IDENTIFICAÇÃO DE CADEIAS PARA SCAN PARCIAL

6.1 Introdução

As ferramentas de síntese geralmente otimizam a geração de um circuito visando ocupar menor área e atingir maior freqüência de operação. Porém, tais ferramentas nem sempre produzem circuitos que sejam otimizados para a testabilidade, exigindo um esforço computacional muitas vezes proibitivo para a geração dos vetores de teste de circuitos seqüenciais. Várias técnicas de DfT tem sido desenvolvidas para facilitar o processo de geração do teste [ABR 90]. Uma abordagem comum consiste em transformar o circuito em um circuito *full scan*. Considerando todos os *flip-flops* internos como entradas e saídas controláveis e observáveis, o problema da geração do teste para um circuito seqüencial torna-se o da geração para um circuito combinacional. No entanto, a aplicação de DfT requer lógica extra na descrição, além de possuir algumas restrições: 1) alta dependência das ferramentas de projeto e das bibliotecas utilizadas; 2) para projetos complexos, este requer um alto tempo de computação, e; 3) o acréscimo de lógica não traz vantagens na otimização total. Além disso, a área e desempenho de um circuito são afetados, o que muitas vezes é inaceitável. Dentro deste contexto, o *scan* parcial com *full scan* é uma técnica comumente utilizada para aumentar a testabilidade do circuito enquanto mantém características relativas à área e desempenho do circuito em níveis aceitáveis [AGR 88].

Neste capítulo, será apresentado um método para selecionar elementos seqüenciais (*flip-flops*) para compor uma cadeia *scan* parcial. Utilizou-se para esta implementação uma técnica da engenharia de software para identificação de variáveis e sinais internos à descrição comportamental do circuito que possuem baixa observabilidade. Experimentos mostram que a abordagem proposta leva a alta cobertura de falhas incluindo-se poucos *flip-flops* na cadeia scan.

6.2 Trabalhos Relacionados

A seleção de *flip-flops* para compor a cadeia *scan* tem sido objeto de vários trabalhos já há alguns anos [CHE 90] [KIM 90] [PAR 95], visto que a redução do número de elementos seqüenciais da cadeia *scan* significa redução da área do circuito e do tempo de teste. Adicionalmente, uma seleção cuidadosa dos registradores pode

aumentar significativamente a capacidade do ATPG. Métodos tradicionais são usualmente classificados em quatro categorias:

1) *structure-based* [FLO 97] [KIM 95] [XIA 96]: escolhe o circuito *scan* utilizando técnicas de seleção de *scan* baseadas na estrutura do circuito. Estas técnicas incluem *loop breaking*, *self-loop breaking* e limite da profundidade seqüencial do projeto;
2) *testability-analysis based* [AGR 88] [PAR 95]: escolhe o circuito *scan* baseado no aumento da controlabilidade e observabilidade determinada por medidas de testabilidade;
3) *ATPG-based* [FLO 97] [XIA 96]: escolhe o circuito *scan* baseado no algoritmo de ATPG para uma determinada lista de falhas, e;
4) *Mixed approaches ou automatic-based* [FLO 97] [XIA 96]: escolhe outros circuitos *scan* baseado na necessidade de alcançar alta cobertura de falhas, combinando as técnicas de seleção anteriores.

Nestes métodos a seleção das cadeias *scan* parcial são normalmente feitas em nível de porta lógica, depois das estruturas lógicas já terem sido sintetizadas [XIA 96]. Os principais obstáculos referem-se ao fato das soluções serem dependentes da implementação física e, que uma decisão tardia no fluxo de projeto, em geral, torna impossível a ativação das otimizações globais. Duas diferentes abordagens tem sido exploradas para resolver estes problemas: mudar o processo de síntese para competir com regras de testabilidade ou escolher *flip-flops* em alto nível, incluindo eventuais mudanças no estilo de projeto.

Existem propostas que a partir de descrições HDL visam melhorar as características de testabilidade do circuito após sua síntese [FLO 95] [VIS 93]. Alguns trabalhos têm como objetivo reduzir os laços internos ao circuito, aumentar a controlabilidade e observabilidade dos registradores ou gerar circuitos levando em consideração medidas de testabilidade obtidas durante a alocação dos registradores [SAF 2003]. Estas técnicas necessitam ter acesso aos processos e algoritmos de síntese, o que nem sempre é possível.

A identificação de *flip-flops* que irão compor a cadeia do *scan* parcial em níveis mais altos de abstração, se comparada às técnicas convencionais que selecionavam *flip-flops* em nível de porta lógica, não possui o intuito de melhorar a qualidade dos *flip-flops* selecionados, mas sim de desvincular a identificação de qualquer implementação física. Alguns trabalhos prévios que obtiveram aumento da qualidade dos elementos do *scan* utilizando níveis mais altos de abstração foram propostos em [DEY 93], [GU 94], [CHI 94], [FER 96], [POT 95], [HSU 98], [HSI 98] e [SES 2002].

Em [DEY 93] os autores apresentam uma técnica que gera um circuito com *scan* parcial mínimo. O algoritmo implementado primeiro quebra todos os laços formados no grafo de controle e de dados do circuito com um número mínimo de *flip-flops* na cadeia *scan*; então não se introduz nenhum laço no *datapath*. Com isso verificou-se que a testabilidade do circuito pode ser aumentada pela seleção dos mesmos registradores da cadeia *scan*.

A abordagem proposta por Gu [GU 94] utiliza uma descrição VHDL transformando-a em uma representação de redes de *petri* estendida. As medidas de controlabilidade referem-se ao custo da aplicação de um valor em uma linha que conecta duas unidades funcionais. As medidas de observabilidade referem-se à dificuldade de observar estes

valores nas saídas. Os custos são relativos ao tempo gasto para encontrar os vetores para detectar falhas em uma linha ou para observar um valor na saída. Após obter as medidas de testabilidade, duas técnicas para melhorar a testabilidade do circuito são aplicadas, tendo como base os valores contabilizados. Primeiro, se existem registradores responsáveis por áreas difíceis de testar, estes são transformados para fazer parte de uma cadeia de varredura. A segunda estratégia insere células de teste nas linhas que conectam as unidades funcionais que apresentaram problemas de testabilidade.

A análise de testabilidade em códigos VHDL foi também explorada em [CHI 94] para identificação de áreas de difícil testabilidade. Neste trabalho são analisados os ciclos de relógio necessários para controlar e observar nós da descrição. Após esta análise, podem ser inseridos caminhos de varredura parciais, visando melhorar a testabilidade do circuito, sem comprometer a área do mesmo.

Um algoritmo de alocação foi proposto em [FER 96]. Este algoritmo primeiro analisa o projeto em nível RTL para identificar laços. Então o algoritmo de alocação é aplicado para produzir uma estrutura acíclica visando diminuir a área do circuito com o *scan* parcial.

Em [POT 95] os autores propuseram transformações para minimizar a área do circuito após a síntese com a inclusão do *scan* parcial. Neste são retiradas dependências de dados em laços, atribuições em laços e laços seqüênciais falsos. Compartilhando registradores de *scan* entre várias variáveis o custo total do *scan* parcial pode ser minimizado.

A seleção de variáveis para o *scan* parcial foi proposta em [HSU 98] com o propósito de aumentar a testabilidade do cicuito, focando no fluxo de controle das descrições VHDL. Para isso, realiza alterações na descrição, o que permite que o projetista tenha uma visão preliminar da testabilidade antes da síntese do circuito. Os autores propõem alterar o fluxo de controle da execução inserindo pontos de controle em laços e desvios. Diz-se que um determinado nodo é controlável se pode ser controlado direta ou indiretamente por uma entrada primária. Se o valor da controlabilidade é elevado, então este nodo torna-se candidato a receber um dos três tipos de alteração em laços proposta. A controlabilidade é obtida inserindo-se pinos de teste com funções lógicas AND, OR ou XOR para forçar os nodos de decisão à execução de seu tratamento verdadeiro ou falso. Concluiu-se que com a inserção de poucos pinos de teste foi alcançada melhora na cobertura de falhas das descrições utilizadas.

Uma estratégia que seleciona *flip-flops* através de uma medida de observabilidade e de pesquisa dinâmica é apresentada em [HSI 98]. Neste trabalho uma ferramenta denominada IDROPS foi desenvolvida para selecionar o melhor e o menor conjunto de *flip-flops* para o *scan* que resultará em uma maior cobertura de falhas. Para selecionar os *flip-flops* cria-se um índice que identifica as falhas abortadas com potencialidade de serem detectadas, o qual irá modificar o circuito para a inclusão do *scan* parcial e posteriormente gerar o teste através do gerador STRATEGATE para que seja possível a medida da testabilidade alcançada. Este processo é feito antes da inclusão do *scan* parcial, tornando-se cíclico. Assim, o primeiro ciclo é feito sobre o circuito original para que seja feita a primeira medida de testabilidade e identificação dos primeiros *flip-flops*.

O trabalho de Seshadri [SES 2002] utiliza a análise de testabilidade para identificar pontos do circuito difíceis de testar. Para isso utiliza-se o fluxo de dados e de controle da descrição VHDL para determinar a controlabilidade e observabilidade de todos os

operandos. Após esta análise são inseridos pinos de teste nos pontos do circuito que possuem baixa observabilidade e se insere cadeias de *scan* parcial nos registradores identificados como de difícil testabilidade. Uma vantagem desta abordagem é o aumento da cobertura de falhas, sem aumentar significativamente a área do circuito.

6.3 Seleção de Variáveis Baseada na Dependência de Dados

6.3.1 Alterações nas Descrições de Hardware

Em [ANG 2005] é apresentada uma relação entre o estilo da descrição VHDL e a testabilidade resultante do circuito, identificando formas de descrição que geram circuitos mais facilmente testáveis. Como estudo de caso, diferentes descrições VHDL de um mesmo algoritmo foram utilizadas. Os resultados mostram que a utilização de diferentes descrições VHDL tem grande impacto nas medidas de testabilidade do circuito final e que características de algumas descrições podem ser utilizadas para modificar outras descrições e com isso aumentar a testabilidade do circuito resultante.

Regras de projeto também são apresentadas em [HAM 98], o qual aplica uma abordagem que visa melhorar a testabilidade no processo de síntese comportamental, utilizando transformações no código VHDL baseadas em regras que podem facilitar a exploração de técnicas de teste, como BIST ou *scan* parcial. Os resultados apresentados, realizados sobre descrições de filtros, mostram que a cobertura, para testes determinísticos aumentou em todos os circuitos experimentados, com um mínimo aumento de área.

Para aumentar a testabilidade, principalmente a observabilidade de um circuito, utilizou-se a técnica apresentada em [COS 2000] que faz uma análise da cobertura das sentenças (ligada à validação, isto é, cobertura de código) para contabilizar a observabilidade destas em um programa de software (programa escrito em linguagem C). Adaptada esta abordagem, no caso para descrições VHDL, e realizadas as devidas modificações em atribuições e comandos condicionais possibilitou-se observar o impacto desta técnica na testabilidade do hardware.

O trabalho de Costa et al [COS 2000] baseou-se numa métrica para calcular observabilidade do hardware apresentada em [FAL 98]. Neste trabalho altera-se descrições HDL para a inclusão de novas variáveis e para a retirada de sentenças de dentro dos comandos condicionais, a fim de aumentar a observabilidade do circuito, auxiliando na correção de problemas de funcionamento do hardware (criação de melhores testes funcionais).

No trabalho apresentado em [COS 2000] para analisar a observabilidade do software são feitas algumas alterações no código fonte, o que é realizado em duas etapas. Na primeira, transforma-se o código fonte do programa pela adição de uma chamada de função para cada sentença. Esta função processa informações sobre a sentença, criando uma lista de dependências partindo das sentenças que apresentam atribuição a variáveis. Na segunda etapa, o programa transformado é compilado e executado com diferentes estímulos de entrada para estimar a observabilidade. Este processo é feito contabilizando quantas vezes cada sentença é executada pelo conjunto de vetores de teste gerados. Este processo permite identificar quais sentenças não são relevantes para as saídas do programa, com isso facilitando a identificação de melhores vetores.

A abordagem baseia-se na criação de listas de dependências para cada variável do programa com o intuito de contabilizar a observabilidade do software, armazernando as localizações de todas as sentenças que apresentam atribuições a variáveis. Com isso, para cada variável do programa existe uma lista de dependências que é a união da lista de dependências de variáveis que estão do lado direito de sentenças de atribuição, ou seja, variáveis que possuem seu valor alterado no código do programa com variáveis que fazem parte de comandos condicionais.

Nesta tese as alterações nas descrições VHDL foram feitas em etapas. Inicialmente foram eliminadas atribuições compostas, tais como a = a + b + 1, dissolvendo as mesmas em quantas atribuições simples fossem necessárias, como por exemplo: temp = a + b e a = temp + 1. Os valores obtidos com estas alterações nos *benchmarks* que apresentam este tipo de atribuição podem ser vistos na Tabela 6.1.

Outro experimento, também motivado pelo trabalho de Costa et al [COS 2000] refere-se à realização de um comparativo entre diferentes formas de codificação do comando if. O que pode ser chamado de ifs encadeados e ifs compostos. Um if encadeado possui a seguinte estrutura:

```
IF (a = b) THEN
    IF (c = b) THEN
```

Enquanto um if composto para a mesma situação seria:

```
IF (a = b) and (b = c) THEN
```

Nos experimentos realizados, para o teste de hardware verificou-se que o uso de ifs encadeados é melhor, em termos de cobertura de falhas. Quanto à quantidade de vetores e número de falhas tivemos pequenas alterações. Com relação à área do circuito alterado esta diminuiu ou manteve-se inalterada. A cobertura de falhas, quantidade de vetores e número de falhas dos experimentos realizados são visualizados na tabela 6.1. Esta abordagem foi aplicada sobre as descrições b01, b10 e b14, por serem estas descrições do sub-conjunto de *benchmarks* utilizados que apresentaram as características desejadas. Cabe ressaltar que a estrutura do comando condicional if utilizado em todas as descrições, originalmente é composta, portanto as alterações foram no sentido de transformá-las em ifs encadeados.

Tabela 6.1: Comparação entre as descrições originais e as descrições alteradas

	Original		***Atribuições***		***Alteração no uso do If***	
Bench	***CF(%)***	***Vetores***	***CF(%)***	***Vetores***	***CF(%)***	***Vetores***
b01	97.50	47	95.90	47	**97.62**	44
b04	49.86	41	**96.06**	1465	-	-
b07	6.37	43	**52.24**	86	-	-
b09	81.54	418	79.46	468	-	-
b10	23.08	44	20.42	46	**43.91**	49
b12	20.65	65	20.18	65	-	-
b14	51.58	542	**64.66**	495	51.58	542
b15	10.65	49	4.23	36	-	-
sr1	78.21	277	75.40	80	-	-
sr4	85.84	555	80.15	253	-	-

Conforme a Tabela 6.1, na implementação que desmembra atribuições compostas em várias atribuições simples. No caso de b04, b07 e b14 obteve-se melhora na cobertura de falhas, mas utilizou-se um número superior de vetores de teste em contra-partida, enquanto para os outros *benchmarks* observou-se uma queda no percentual de cobertura de falhas. As alterações realizadas sobre os *benchmarks* que apresentavam uso de estruturas condicionais independentes após a alteração para ifs aninhados apresentaram melhora, no caso de b01 e b10, ou constância no valor original como é o caso do b14.

Nota-se que a aplicação de uma técnica que para o teste de software facilita o processo de teste (especialmente na geração de dados de teste e na cobertura de falhas) não necessariamente apresenta a mesma característica no caso do teste de hardware, pois o desmembramento de atribuições compostas em mais atribuições aumenta o número de fios e conseqüentemente o número de falhas. O que pode ser observado nos valores apresentados para estes experimentos é que dados obtidos não são conclusivos devido ao fato de não apresentarem um comportamento constante para todos os *benchmarks* e de não poderem ser aplicados a muitos dos *benchmarks* no caso do uso de ifs compostos.

Embora as alterações nas descrições de hardware propostas em [COS 2000] não tenham conduzido a resultados dos mais favoráveis, o conceito de lista de dependência lá utilizado para avaliar a observabilidade de variáveis será aqui adaptado para a escolha de *flip-flops* para compor cadeias de *scan* parcial.

6.3.2 Aplicação das Listas de Dependências

No trabalho apresentado por Costa et al. [COS 2000], para cada variável do programa existe uma lista de dependências, que é a união da lista de dependências de variáveis que estão do lado direito de sentenças de atribuição (variáveis que possuem seu valor alterado no código do programa) com as variáveis declaradas e utilizadas em comandos condicionais. Resolveu-se adaptar este conceito para as descrições VHDL, pois aumentar a observabilidade de um software refere-se à facilidade com que se pode visualizar o que se está testando (de maneira interna) em tempo de execução, enquanto observabilidade para hardware refere-se à facilidade de observar valores internos através das saídas do circuito.

Para descrições de hardware a aplicação desta técnica é bem simples, pois basta varrer a descrição para identificar quais as sentenças que influenciam na atribuição de valores aos sinais de saída, criando assim uma lista de dependências. No caso de programas escritos em linguagens de programação tradicionais a identificação de variáveis de saída só é possível no momento em que elas participam de algum comando de saída (tela, arquivo ou relatório), pois não existe diferença na declaração da variável. No caso do hardware, no entanto, os sinais são declarados como de entrada ou saída.

A aplicação desta técnica possibilita a identificação de pontos internos na descrição que possuem influência na testabilidade do hardware, não só com relação à controlabilidade, mas também considerando a observabilidade. A partir da lista de dependências criada, identifica-se as sentenças que possuem influência direta sobre a atribuições aos sinais de saída do circuito, tais como sentenças if e case, desde que estas não estejam envolvidas em condições relativas a sinais de entrada do circuito.

O algoritmo que identifica quais variáveis ou sinais internos influenciam na testabilidade do dispositivo é decomposto em: a) criação de uma estrutura com todos os sinais de saída (OUT) e uma com os sinais de entrada (IN); b) utilização de um *parser* que faz a leitura de toda a descrição VHDL para identificar os comandos, variáveis ou sinais; c) criação de uma lista com as variáveis ou sinais que compõem um comando condicional; d) criação de uma lista de variáveis que compõem um comando de condição antes de um comando de atribuição a um sinal de saída. A Figura 6.1 mostra um trecho do código que implementa esta aplicação.

Basicamente o algoritmo realiza uma leitura da descrição VHDL e gera as listas de dependências. O algoritmo verifica as atribuições que envolvem todos os sinais de saída que estão localizados dentro de comandos condicionais. Outras atribuições não interressam porque elas têm alta observabilidade, já que elas sempre são executadas independentemente de outras variáveis. Sinais de entrada são descartados, já que estes são naturalmente observáveis.

De acordo com a Figura 6.1, o primeiro passo do algoritmo consiste na geração das listas denominadas LIST_SIGNALS_IN e LIST_SIGNALS_OUT que possuem todos os sinais de entrada (IN) e de saída (OUT), respectivamente identificados na definição da entidade no código VHDL. O próximo passo é auxiliado por um *parser* para a localização das sentenças condicionais. Nesta implementação apenas comandos if, else, elseif e case foram considerados, já que em descrições VHDL comandos for e while não são tão comuns. Após a identificação das instruções condicionais o *parser* atravessa todas as sentenças para verificar se dentro destas existe atribuição a algum sinal de saída. Uma vez um sinal de saída seja encontrado, a sentença é incluida na lista denominada LIST_STATEMENTS e todas as variáveis e sinais envolvidos neste comando condicional são incluídos na lista LIST_VARIABLES. Porém, variáveis que estão presentes na lista LIST_SIGNALS_IN não são adicionadas nesta lista. No final da execução do algoritmo a lista chamada LIST_VARIABLES possui variáveis e sinais que devem ser observados.

```
LIST_SIGNALS_IN array of String
LIST_SIGNALS_OUT array of String
LIST_STATEMENTS array of Integer
LIST_VARIABLES array of String

begin
  Generate (LIST_SIGNALS_IN)
  Generate (LIST_SIGNALS_OUT)

  Search_Conditional_Statements()
    cont = 0
    loop Parser
      if statement="if" or "elseif" or "case"
        if Verify_Out (assigned_variable)
          cont = cont + 1
          Insert_List_Statements (cont)
          Insert_List_Variables (cont)
        endIf
      endIf
    endLoop
end
  Verify_Out(variable)
    loop Index
      if LIST_SIGNALS_OUT[Index] = variable
        return true
      endIf
    endLoop

  Verify_In(variable)
    loop Index
      if LIST_SIGNALS_IN[Index] = variable
        return true
      endIf
    endLoop

Insert_List_Statements(cont)
    LIST_STATEMENTS[cont] = statement_number

  Insert_List_Variables(cont)
    forEach conditional_variable in cont
      if not Verify_In (variable)
        LIST_VARIABLES[last+1] = variable
      endIf
    endForEach
```

Figura 6.1: Algoritmo para geração das listas de dependência

A Figura 6.2 possui um exemplo de uma descrição comportamental VHDL. Neste código, as sentenças condicionais já estão numeradas. As listas (LIST_SIGNALS_IN, LIST_SIGNALS_OUT, LIST_STATEMENTS e LIST_VARIABLES) geradas pela aplicação do algoritmo da Figura 6.1 são apresentadas na Tabela 6.2.

Tabela 6.2: Listas geradas para o exemplo VHDL

LISTA	CONTEÚDO
LIST_SIGNALS_IN	clock, reset, line1, line2
LIST_SIGNALS_OUT	output, overflow
LIST_STATEMENTS	1,2,3
LIST_VARIABLES	state

No exemplo apresentado (Figura 6.2 e Tabela 6.2), somente uma variável é válida, ou seja, estará na LIST_VARIABLES, pois os outros sinais que se encontravam na lista eram sinais de entrada. Portanto, apenas a variável denominada state é selecionada como variável que deve ter sua observabilidade e controlabilidade aumentadas.

```
entity b01 is
port(
  line1, line2 ,reset, clock : in bit;
  output : out bit;
  overflow : out bit);
end b01;
   ...
   process (clock, reset, line1, line2)
 1   if reset = '1' then
       state := a;
       output <= '0';
       overflow <= '0';
 2   elsif clock'event and clock = '1' then
 3     case state is
         when a =>
         if line1 = '1' and line2 = '1' then
             state := f;
           else
             state := b;
           end if;
           output <= line1 xor line2;
           overflow <= '0';
         when e =>
          if line1 = '1' and line2 = '1' then
             state := f;
           else
             state := b;
```

Figura 6.2: Exemplo de um código VHDL

6.4 Dados Experimentais

Utilizando-se a abordagem e o algoritmo descritos anteriormente foram realizadas duas implementações diferentes:

1) para obter melhores taxas de observabilidade criou-se para cada uma das variáveis ou sinais envolvidos na sentença condicional (LIST_VARIABLES) um sinal de saída e para esta implementação foram obtidos os valores apresentados na Tabela 6.3, e
2) para obter melhores taxas de observabilidade criou-se para cada uma das variáveis ou sinais envolvidos na sentença condicional (LIST_VARIABLES) um sinal de

saída e para esta implementação foram obtidos os valores apresentados na Tabela 6.3, e

Tabela 6.3: Comparação entre inclusão de sinais de saída com a descrição original

Benchmark	*Original*				*Alterado (pinos externos)*			
	CF(%)	*Vetores*	*Pinos*		*CF(%)*	*Vetores*	*Pinos*	
			IN	*OUT*			*IN*	*OUT*
b01	97.50	47	4	2	**97.54**	56	4	5
b02	96.09	24	3	1	**96.72**	25	3	4
b03	71.74	135	6	4	**76.62**	149	6	8
b04	49.86	41	13	8	**80.55**	77	13	18
b06	99.59	46	4	6	**99.60**	55	4	9
b07	6.37	43	3	8	**52.77**	130	3	11
b08	86.81	221	11	4	**89.78**	313	11	9
b09	81.54	418	3	1	**86.32**	540	3	12
b10	23.08	44	13	6	**79.93**	74	13	14
b11	22.90	31	9	6	**41.74**	47	9	19
b12	**20.65**	65	7	6	20.24	55	7	18
b13	31.99	78	12	10	**46.61**	134	12	31
b14	51.58	542	34	54	**58.39**	944	34	65
b15	10.65	49	38	70	**12.18**	39	38	94
sr1	78.21	277	11	5	**81.59**	286	11	17
sr2	**89.65**	403	11	5	74.98	74	11	17
sr3	**88.83**	588	11	5	79.75	135	11	22
sr4	85.84	555	11	5	**86.63**	304	11	13
sr5	71.89	917	11	5	**82.99**	304	11	20

Nota-se que as alterações para aumentar a observabilidade, na maioria das descrições, resultou em um aumento na cobertura de falhas, exceto para os circuitos b12, sr2 e sr3. Porém, o problema desta abordagem está no aumento do número de pinos de saída do circuito, o que em alguns *benchmarks*, principalmente descrições pequenas, torna-se um fator proibitivo.

Tabela 6.4: Comparativo entre circuito original e o circuito com *scan* parcial

Benchmark	*FFs*	*Original*				*Abordagem Proposta*					
		CF(%)	*Vetores*	*Pinos IN*	*Pinos OUT*	*CF(%)*	*Vetores*	*Pinos IN*	*Pinos OUT*	*Scan FFs*	*FFs (%)*
b01	5	97.50	47	4	2	**100.00**	60	6	3	3	60.00
b02	4	96.09	24	3	1	**100.00**	42	5	2	3	75.00
b03	30	71.74	135	6	4	**77.72**	324			2	6.87
b04	80	49.86	41	13	8	**98.32**	873	15	9	32	40.00
b06	9	99.59	46	4	6	**100.00**	49	6	7	4	44.44
b07	41	6.37	43	3	8	**61.37**	722	5	9	3	7.32
b08	21	**86.81**	221	11	4	82.69	506	13	5	2	9.52
b09	28	81.54	418	3	1	**88.97**	665	5	2	11	39.29
b10	21	23.08	44	13	6	**99.55**	273	15	7	12	57.14
b11	59	22.90	31	9	6	**94.85**	881	11	7	34	57.63
b12	144	20.65	65	7	6	**53.14**	991	9	7	25	17.36
b13	59	31.99	78	12	10	**98.69**	719	14	11	36	61.02
b14	290	51.58	542	34	54	**94.31**	2987	36	55	107	36.90
sr1	35	78.21	277	11	5	**96.88**	204	14	7	20	57.14
sr2	26	89.65	403	11	5	**95.78**	714	13	6	2	7.69
sr3	30	**88.83**	588	11	5	88.15	637	13	6	6	20.00
sr4	24	**85.84**	555	11	5	78.83	400	13	6	7	23.33
sr5	30	71.89	917	11	5	**92.32**	1500	13	6	2	8.33

Os resultados da Tabela 6.4 mostram, como era esperado, que para a maior parte dos circuitos foram obtidas coberturas de falhas superiores se comparados os resultados do circuito original com o circuito acrescido com o DfT. Porém, necessita-se de um maior número de vetores de teste, aumenta-se a área do circuito (DFFs) e acrescenta-se pinos (3 na maioria dos casos) para a implementação do *scan*.

Ao confrontar os dados apresentados nas Tabelas 6.3 e 6.4 observa-se que se obtém maior cobertura de falhas quando os registradores selecionados são utilizados para a composição do *scan* parcial ao invés de utilizá-los para compor os sinais de saída do circuito. Isto se deve ao fato que aumentando o número de pinos de um circuito, além de aumentar seu tamanho, aumenta-se o número de conexões internas, conseqüentemente aumentando o número de falhas possíveis.

A coluna da Tabela 6.4 que mostra o percentual de *flip-flops* selecionados para a implementação do *scan* parcial possui dois propósitos. O primeiro é para comparar com uma implementação de *full scan*, para a qual 100% dos *flip-flops* são selecionados (Tabela 6.5). O segundo propósito é o de estipular o número de registradores a serem selecionados por outras técnicas de *scan* parcial a fim de comparar a abordagem proposta com outras abordagem disponíveis na ferramenta utilizada. A tabela 6.6 mostra a relação dos outros métodos aplicando-se o mesmo percentual de *flip-flops* utilizados pela abordagem proposta para cada um dos *benchmarks* de teste.

Tabela 6.5: Comparativo entre parcial *scan* e *full scan*

Benchmark	*Abordagem Proposta*				*Full Scan*			
	CF(%)	*FFs*	*Pinos*		*CF(%)*	*FFs*	*Pinos*	
			IN	*OUT*			*IN*	*OUT*
b01	**100.00**	**3**	6	3	**100.00**	5	6	2
b02	**100.00**	**3**	2	3	**100.00**	4	5	1
b03	77.72	2	8	5	99.21	30	8	4
b04	98.32	**32**	15	9	98.58	80	15	8
b06	**100.00**	**4**	7	4	**100.00**	9	6	6
b07	61.37	3	9	3	99.56	41	5	8
b08	82.69	2	13	5	99.10	21	13	4
b09	88.97	11	5	2	99.26	28	5	1
b10	**99.55**	**12**	15	7	99.50	21	15	6
b11	94.85	34	11	7	98.96	59	11	6
b12	53.14	25	9	7	98.94	144	9	6
b13	98.69	36	14	11	99.00	59	14	10
b14	94.31	107	36	55	98.56	290	36	54
sr1	96.88	20	14	7	99.43	35	14	6
sr2	95.78	2	13	6	99.77	26	14	6
sr3	88.15	6	13	6	99.92	30	13	5
sr4	78.83	7	13	6	99.36	24	13	5
sr5	92.32	2	13	6	97.88	30	13	5

Na Tabela 6.5 pode-se notar que na maioria dos casos (exceto em b03, b07, b08, b09, b12, sr3 e sr4) com a abordagem que seleciona *flip-flops* para compor a cadeia *scan* parcial obteve-se coberturas de falhas próximas as obtidas pela implementação do *full scan* com a vantagem de um número menor de *flip-flops* selecionados.

A diversidade de técnicas para a seleção de *flip-flops* para o *scan* parcial motivou a aplicação do percentual de registradores necessários para implementar a cadeia *scan* da abordagem proposta e avaliar o custo e a eficiência desta abordagem quando comparada com os métodos disponibilizados pela ferramenta DFTAdvisor. Com a finalidade de comparação utilizou-se as quatro abordagens de seleção de flip-flops mencionadas no início deste capítulo sendo elas:

- *sequential ATPG-based*: este método escolhe o circuito *scan* baseado no algoritmo utilizado pelo ATPG Flextest™;
- *automatic-based*: o circuito *scan* é baseado na necessidade de uma alta cobertura de falhas. Este método combina várias técnicas de seleção de *scan*, mesmo quando utiliza-se limitação do número de células;
- *SCOAP-based*: escolhe o circuito *scan* baseado na melhoria de controlabilidade e observabilidade determinada pela abordagem SCOAP (*Sandia Controllability Observability Analisis Program*). O DFTAdvisor™ contabiliza os números SCOAP de cada elemento de memória e escolhe os elementos com maior valor

para compor a cadeia do *scan* parcial. Este método apresenta rapidez na seleção das células, e;

- *structured-based*: escolhe o circuito *scan* utilizando técnicas de seleção conhecidas por *structured*. Estas técnicas incluem *loop breaking*, *self-loop breaking* e limite da profundidade do projeto seqüencial.

A Tabela 6.6 apresenta a comparação entre a abordagem proposta e os métodos de seleção da própria ferramenta DFTAdvisor ™. A melhor cobertura de falhas aparece em negrito para cada *benchmark*.

Considerando todo o conjunto de experimentos, a abordagem proposta apresenta melhor cobertura de falhas em 7 dos 18 circuitos testados, contra 4 da abordagem de seleção *ATPG-based*, 6 da abordagem *automatic-based*, 8 da abordagem *SCOAP-based* e 7 da abordagem de seleção *structure-based*. Em geral, para circuitos com baixa complexidade (b01, b02 e b06) todas as abordagens apresentam os mesmos resultados. A abordagem proposta nesta tese caracteriza-se por apresentar coberturas de falhas sempre superiores a pelo menos uma das outras abordagens em 3 casos (b04, b07 e b14). Em 4 casos (b01, b02, b06 e b10) se iguala a pelo menos uma das outras 4 abordagens. As principais conclusões que podem ser obtidas destes resultados são:

- cada abordagem apresenta uma contribuição para a seleção do *scan* parcial que não pode ser coberta por outras, neste sentido elas se complementam;
- a abordagem adotada nesta tese apresenta uma importante contribuição já que esta cobre as outras em vários casos.

Tabela 6.6: Comparação entre as abordagens proposta e a do DFTAdvisor ™

Benchmark	***Abordagem Proposta***		***ATPG-Based***		***Automatic-based***		***SCOAP-Based***		***Structure-based***	
	CF(%)	***Vet.***	***CF(%)***	***Vet.***	***CF(%)***	***Vet.***	***CF(%)***	***Vet.***	***CF(%)***	***Vet.***
b01	**100.00**	45	**100.00**	45	**100.00**	45	**100.00**	45	**100.00**	45
b02	**100.00**	42	**100.00**	42	**100.00**	42	**100.00**	42	**100.00**	42
b03	77.72	324	85.71	180	80.65	288	76.09	292	**86.39**	276
b04	**98.32**	873	93.66	1311	96.40	815	93.08	930	87.34	388
b06	**100.00**	49	**100.00**	49	**100.00**	49	**100.00**	49	**100.00**	49
b07	**61.37**	722	26.62	38	57.38	461	57.58	494	26.62	38
b08	82.69	506	89.89	497	85.47	299	**91.67**	319	82.69	506
b09	88.97	665	91.43	673	**92.22**	441	88.97	665	85.73	539
b10	**99.55**	273	90.68	370	**99.55**	273	**99.55**	273	91.33	410
b11	94.85	881	85.45	605	94.20	884	**95.80**	940	93.92	1350
b12	53.14	991	42.86	699	76.96	3765	75.17	2880	**77.32**	4355
b13	98.69	719	96.84	491	94.71	867	**98.79**	725	81.00	595
b14	**94.31**	2987	73.62	703	90.51	2564	89.81	3009	82.75	1181
sr1	96.88	204	94.70	180	93.75	254	**97.58**	198	75.56	241
sr2	95.78	714	88.98	437	**96.29**	1735	89.34	322	93.77	1765
sr3	88.15	637	60.18	129	89.07	296	88.50	465	**93.28**	516
sr4	78.83	400	42.86	148	88.39	479	88.81	285	**96.74**	831
sr5	92.32	1500	**92.81**	878	91.10	1022	91.10	1022	91.10	376

Em [KRU 2006b] apresentou-se a abordagem proposta neste capítulo, porém a cobertura de falhas difere dos valores apresentados na Tabela 6.6 devido à forma como utilizou-se a ferramenta Flextest. Para obter os resultados apresentados em [KRU

2006b] utilizou-se o *edif* gerado pela ferramenta Leonardo na ferramenta DFTAvisor selecionando-se os registradores manualmente (aplicando a técnica de seleção apresentada neste capítulo) ou utilizando-se uma das quatro outras técnicas disponíveis na ferramenta DFTAdvisor. Porém, quando utilizou-se o Flextest para a geração dos vetores de teste e obtenção da cobertura de falhas utilizou-se a ferramenta sem especificar a cadeia *scan* inserida anteriormente.

Apesar de pequenas diferenças encontradas na comparação feita aqui e em [KRU 2006b], as conclusões se mantêm.

6.5 Considerações Finais

Entre as várias técnicas de DfT desenvolvidas para facilitar a tarefa de geração de padrões de teste para circuitos sequenciais, a abordagem do *scan* parcial tem se mostrado a mais popular. De modo diferente do *full scan*, onde todos os *flip-flops* do circuito tornam-se observáveis e controláveis, a técnica do *scan* parcial seleciona um sub-conjunto de *flip-flops* para compor a cadeia *scan*. Nesta tese um dos objetivos foi o de implementar um projeto de *scan* parcial, selecionando o menor número de *flip-flops* possível e buscando coberturas de falhas superiores, sem conduzir a aumentos importantes na área do circuito e nem degradação do seu desempenho.

A abordagem proposta nesta tese aplica uma técnica da engenharia de software que permite gerenciar uma lista de dependências de varáveis para cada saída do circuito. Baseado nas variáveis e sinais presentes nesta lista pode-se extrair os *flip-flops* para compor a cadeia *scan*. Os resultados apresentados mostram que o método proposto apresenta alta cobertura de falhas utilizando poucos *flip-flops* no *scan* parcial. Comparando os resultados obtidos com outros métodos clássicos de identificação, a metodologia proposta mostrou-se competitiva.

Encontrou-se dificuldade em realizar um comparativo entre a abordagem de seleção de registradores para compor uma cadeia *scan* parcial proposta com outras abordagens implementadas em outros trabalhos encontrados na literatura, pois o objeto experimentado não é o mesmo. A abordagem desta tese trabalha com descrições VHDL comportamentais no estilo RTL, enquanto os trabalhos relacionados trabalham com descrições estruturais, principalmente, sobre o conjunto de *benchmarks* do ISCAS89.

7 CONCLUSÕES

Esta tese procurou aplicar técnicas originárias da engenharia de software ao processo de teste de hardware. Em um primeiro momento geraram-se vetores de teste a partir de descrições HDL e, a seguir, buscou-se aumentar a testabilidade do circuito sob teste aumentando a observabilidade de algumas variáveis destas descrições.

Na primeira implementação, utilizou-se uma técnica de engenharia de software, mais especificamente do teste de software, chamada de teste de fluxo de controle baseado em caminho, para gerar padrões de teste a partir de descrições comportamentais VHDL de circuitos digitais. Posteriormente, utilizou-se o conjunto de vetores de teste gerados pela técnica de teste de software como vetores iniciais para que a ferramenta de ATPG os complemente com os vetores necessários para detectar as falhas não detectadas pelo primeiro conjunto. Resultados experimentais mostraram que utilizando somente padrões de teste gerados a partir de informações obtidas no alto nível de abstração não traz benefícios do ponto de vista da cobertura de falhas. Entretanto, combinando os vetores de teste gerados a partir da descrição comportamental com vetores gerados pelo ATPG em nível de porta lógica na maioria dos casos aumenta a cobertura de falhas e reduz o esforço do ATPG. Em alguns casos, reduz-se o tamanho do conjunto de teste ao mesmo tempo em que se aumenta a qualidade do teste. Além do fato de diminuir o tempo de exposição ao teste, para a maior parte dos circuitos testados, devido ao fato do maior tempo gasto em todo o processo de teste estar, justamente associado à geração de vetores de teste.

A outra abordagem proposta e implementada constituiu na alteração do circuito para a inserção da técnica de DfT denominada *scan* parcial. A técnica de *scan* tem sido largamente utilizada para facilitar o processo de teste de circuitos. Apesar da abordagem denominada *full scan* apresentar resultados com alta cobertura de falhas reduzindo o esforço do ATPG, esta introduz área adicional ao circuito e reduz seu desempenho, o que muitas vezes não é aceitável. Por isso, a técnica de *parcial scan* é comumente utilizada para aumentar a testabilidade de circuitos seqüenciais, enquanto atende restrições de projeto em termos de área e desempenho. Propôs-se um método para selecionar elementos seqüenciais (*flip-flops*) para compor uma cadeia de *scan* parcial baseado em técnicas da engenharia de software para identificar variáveis e sinais internos em descrições comportamentais de circuitos que possuem baixa observabilidade. Experimentos demonstraram que a abordagem conduz a altas coberturas de falhas incluindo um número reduzido de *flip-flops* na cadeia *scan*.

De uma maneira geral, pode-se afirmar que a utilização de técnicas de engenharia de software pode auxiliar o processo de teste de hardware. No entanto, tão somente aplicar de maneira direta as técnicas de teste de software não garante boa cobertura de falhas, visto que, no nível onde são aplicadas estas técnicas não existem detalhes da implementação física do dispositivo de hardware. Esta tese demonstrou que ao adaptar e/ou combinar estas técnicas com ATPG e DfT no nível de porta lógica, é possível aumentar a cobertura de falhas, reduzindo o tempo de preparação do teste.

Como desenvolvimento futuro, pode-se mencionar a necessidade de se aprimorar os algoritmos para possibilitar a geração de padrões de teste que possam ser aplicados em dispositivos de hardware que possuam processos em execução concorrente, e a conseqüente necessidade de se dispor de *benchmarks* mais complexos que permitam validar tais algoritmos. Outra melhoria importante passaria por alterar a ferramenta para torná-la mais amigável ao uso, proporcionando de maneira gráfica a visualização dos caminhos da descrição a serem percorridos por cada padrão de teste gerado e implementando uma funcionalidade que permita estipular um número de vetores mínimo e máximo a ser gerado, possibilitando assim um melhor controle sobre a qualidade e quantidade dos vetores gerados.

REFERÊNCIAS

ABRAMOVICI, M.; BREUER, M.; FRIEDMAN, A. **Digital Systems Testing and Testable Design**. Washington: IEEE Press, 1990. 652p.

AGRAWAL, V. D; CHENG, K-T. JOHNSON, D.D. Designing Circuits with partial scan. **IEEE Design & Test of Computers**, [S.l.], v. 5, n. 2, p. 8-15, 1988.

AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. A Tutorial on Build-In Self-Test I. **IEEE Design and Test of Computers**, [S.l.], v. 10-1, p. 73-82, 1993.

AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. A Tutorial on Build-In Self-Test II. **IEEE Design and Test of Computers**, [S.l.], v. 10-2, p. 69-77, 1993.

AL-HAYEK, G. **Vers une Approche Unifiée Pour la Validationet le Test de Circuits Intégrés Spécifiés en VHDL.** 1999. Thesis, Institut National Polytechnique de Grenoble, Grenoble.

AL-YAMANI, A. **Deterministic Built-In Self Test for Digital Circuits**. 2004. 136f. Dissertation (Degree of Doctor of Philosophy) – Standford University, San Francisco, USA.

ANGELO, R.; COTA, E., CARRO, L.; LUBASZEWSKI, M. Implications of the High-Level Design Style in the Testability: A Case Study.In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador. **Digest of papers**. [S.l.:s.n.], 2005. p. 291-295.

ASHENDEN, P. J. **The Student's Guide to VHDL.** San Francisco: Morgan Kaufmann, 1998.

BEIZER, B. **Software Testing Techniques**. 2nd ed. [S.l.]: Van Nostrand Reinhold, 1990.

BENKAHLA, O. et al. Periodic Testing of FPGAs Using a Software Testing Method, proceed, In: INTERNATIONAL ON-LINE TESTING WORKSHOP, 1999, Rhodes, Greece . **Proceedings...** New York: IEEE, 1999.

BREUER, M. A.; FRIEDMAN, A. D. **Diagnosis and Reliable Design of Digital Systems**. Woodland Hills: Computer Science Press, 1976.

BRGLEZ, F.; BRYAN, D.; KOZMINSKI, K. Combinational Profiles vs Sequential Benchmark Circuits. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1989. **Proceedings...** New York: IEEE, 1989. p. 1929-1934.

BRGLEZ, F.; FUJIWARA, H. A Neutral Netlist of 10 Combinatorial Benchmark Circuits. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1985. **Proceedings...** New York: IEEE, 1985. p. 695-698.

BUSHNELL, M.; AGRAWAL, V. **Essentials of Eletronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits**. Boston: Kluwer Academic Publishers, 2000.

CARRO, L. **Projeto e Prototipação de Sistemas Digitais**. Porto Alegre: Ed. da UFRGS, 2001. 176p.

CHENG, K.T.; AGRAWAL, V.D. A Partial Scan Method for Sequential Circuits with Feedback. **IEEE Transactions on Computer**, [S.l.], v. 39, n. 4, p. 544-548, Apr. 1990.

CHICKERMANE, V.; LEE, J.; PATEL, J.H. Addressing Design for Testability at the Architectural Level. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.13, n.7, p.920-934, July 1994.

CHILLARGE, R. **Software Testing Best Practices**. [S.l.:s.n.], 1999. IBM Technical Report.

COSTA, J.C.; DEVADAS, S.; MONTEIRO, J.C. Observability Analysis of Embeded Software for Coverage-Directed Validation. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2000. **Proceedings...** New York: IEEE/ACM, 2000. p. 27-32.

COURTOIS, B. Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits. In: VLSI CONFERENCE, 1981. **Proceedings...** Maryland: Computer Science Press, 1981.

CUMANI, G. **High-Level Test of Electronic Systems**. 2003. 88f. Tese (Doutorado em Engenharia de Computação e Sistemas) – Universidade de Torino, Torino, Itália.

DEMILLO, R. A. et al. Hints on Test Data Selection: Help for the Practising Programmer. **Computer**, New York, v. 11, n. 4, p. 31-41, 1978.

DEMILLO, R. A.; OFFUTT, J. A. Constraint-Based Automatic Test Data Generation. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 9, p. 900-910, 1991.

DEY, S.; POTKONJAK, M.; ROY, R.K. Exploiting Hardware Sharing in High-Level Synthesis for Partial Scan Optimization. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTED-AIDED DESIGN, 1993. **Proceedings...** New York: ACM, 1993. p. 20-25.

FALLAH, F.; DEVADAS, S.; KEUTZER, K. OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Simulation. In: DESIGN AUTOMATION CONFERENCE, DAC, 35., 1998. **Proceedings…** New York: ACM, 1998. p. 152-157.

FERNANDEZ, V.; SANCHEZ, P. Partial Scan High-Level Synthesis. In: IEEE EUROPEAN DESIGN AND TEST CONFERENCE, 1996. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p.481-485.

FERRADI, F. et al. Functional Test Generation for Behaviorally Sequential Models. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2001. **Proceedings...** [S.l.:s.n.], 2001. p. 403-410.

FERRADI, F. et al. Symbolic Functional Vector Generation for VHDL Specifications. In: DATE, 1999. **Proceedings…** Los Alamitos: IEEE, 1999. p.442-446.

FLOTTES, M.L. et al. High Level Synthesis for Partial Scan. In: EUROPEAN DESIGN AND TEST CONFERENCE, 1997. **Proceedings...** [S.l.:s.n.], 1997.

FLOTTES, M.L.; HAMMAD, D.; ROUZEYRE, B. High-Level Synthesis for Easy Testability. In: EUROPE DESIGN & TEST CONFERENCE, ED&T, 1995. **Proceedings...** Paris, France: [s.n.], 1995. p.198-206.

FREEDMAN, R.S. Testability of Softwere Components. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 6, p. 553-564, 1991.

FUJIWARA, H**. Logic Testing and Design for Testability**. Cambridge: The MIT Press, 1985.

FUJIWARA, H.; SHIMONO, T. On the Acceleration of Test Generation Algorithms. **IEEE Transactions on Computers**, Los Alamitos, v. 32, p. 1137-1144, 1983.

GALAY, J. A.; CROUZET, Y.; VERNIAULT, M. Physical Versus Logical Fault Models MOS-LSI Circuits: Impact of Their Testability. **IEEE Transactions on Computers**, New York, v. 29, n. 6, p. 527-531, 1980.

GOEL, P. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. **IEEE Transactions on Computers**, Los Alamitos, v. 30, p. 215-222, 1981.

GRÖTKER, T. et al. **System Desigh With SystemC**. [S.l.]:Springer, 2002. 240p.

GU, X. et al. A Controller Testability Analisys and Enhancement Technique. In: EUROPE DESIGN AND TEST CONFERENCE, 1997. **Proceedings...** [S.l.:s.n.], 1997. p. 153-157.

GUPTA, S.; RAJSKI, J.; TYSZER, J. Test Pattern Generation Based on Arithmetic Operations. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1994. **Proceedings...** Los Alamitos: IEEE, 1994. p. 117-124.

HAMILTON, A. N.; GONZALEZ, T.; ORAILOGLU, A. Design Rule Driven Behavioral Synthesis for Test. Signals. In: ASIA CONFERENCE ON SYSTEMS AND COMPUTERS, 32., 1998. **Proceedings...** [S.l.:s.n.], 1998. v. 2, p. 1033-1037.

HAYEK, G. A.; ROBACH, C. From Specification to Hardware Testing: A Unified Methods. In: INTERNATIONAL TEST CONFERENCE, ITC, 1996, Washington D.C.(USA). **Proceedings...** [S.l.:s.n.], 1996. p. 855-893.

HETZEL, W. **The Complete Guide to Software Testing**, Wellesley, MA: QED Information Science, 1987.

HSIAO, M.S. et al. Partial Scan Selection Based on Dynamic Reachability and Observability Information. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 11., 1998. **Proceedings...** [S.l.:s.n.], 1998. p.174-180.

HSU, F.; PATEL, J. High-Level Variable Selection for Partial-Scan Implementation. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1998. **Proceedings...** New York: ACM SIGDA, 1998. p.79-84.

HUANG, S.-Y; CHENG, K-T. **Formal Equivalence Checking and Verification Algorithms**. [S.l.]: Kluwer Academic Publishers, 1998.

IEEE. **IEEE Standard 1149-1b:** IEEE Standard Test Access Port and Boundary-Scan Architecture. [S.l.], 1994.

INSTITUTE OF ELECTRICAL AND ELECTRONICS. **IEEE Standard VHDL Language Reference Manual**. New York, 1988.

JERVAN, G. **High-Level Test Generation and Built-In Techniques for Digital Systems**. 2002. 112f. PhD Thesis (degree of Licentiate of Engineering) - Department of Computer and information Science Limköpings Universitet, Suécia.

JOHNSON, M. W. **High Level Test Generation Usign Software Testing Metrics**.1995. Master Thesis of University of Illinois, Center for Reliable and High Performance Computing, Urbana, Illinois.

KIM, K.S.; KIME, C.R. Partial Scan by Use of Empirical Testability. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1990. **Proceedings...** [S.l.:s.n.], 1990. p.314-317.

KING, K. N.; OFFUTT, J. A Fortran Language System for Mutation-Based Software Testing. **Software Practice and Experience**, London, v. 21, n. 7, p.686-718, July 1991.

KIRKLANDT, T.; MERCER, M. A Topological Search Algorithm for ATPG. In: IEEE DESIGN AUTOMATION CONFERENCE, DAC, 24., 1987, Miami Beach. **Proceedings...** New York: ACM, 1987. p.502-508.

KRUG, M. R.; MORAES, M. S.; LUBASZESKI, M. S. Using a Software Testing Technique to Identify Registers for Partial Scan Implementation. In: SBCCI, 2006. **Proceedings…** New York: ACM, 2006.

KRUG, M. R.; MORAES, M. S.; LUBASZESKI, M. S. Improving ATPG Gate-Level Fault Coverage by using Test Vectors generated from Behavioral HDL Descriptions, In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 14., 2006. **Proceedings…** Grenoble: Tima, 2006.

LAJOLO, M. et al. Behavioral-level Test Vector Generation for System-on-chip Designs. In: HIGH-LEVEL DESIGN VALIDATION AND TEST WORKSHOP, 2000. **Proceedings...** [S.l.:s.n.], 2000. p. 21-26.

LEE, D.; YANNAKAKIS, M. Principles and Methods of Testing Finite State Machines - a Survey. **Proceedings of the IEEE,** Los Alamitos, v. 84, n. 8, p. 1090-1123, 1996.

MALDONADO, J.C. **Critérios Potenciais Usos:** Uma Contribuição ao Teste Estrutural de Software. 1992. 260p. Tese (Doutorado) - DCA/FEE/UNICAMP, Campinas.

MALDONADO, J.C. et al. **Introdução ao Teste de Software**. Notas Didáticas do ICMC, Instituto de Ciências Matemáticas e de Computação – ICMC/USP São Carlos, 2004.

MAYHAUSER, A. et al. On choosing test criteria for behavioral level hardware design verification, In: IEEE INTERNATIONAL HIGH-LEVEL VALIDATION AND TEST WORKSHOP, HLDVT, 2000, California. **Proceedings**… [S.l.]: IEEE, 2000. p.124.

MCCABE, T. A Software Complexity Measure. **IEEE Transactions on Software Engineering,** [S.l.], v. 2, n. 6, Dec. 1976.

MENTOR GRAPHICS CORPORATION. **Scan and ATPG Process Guide**: Manual Software Version V8.6_4. [S.l.], 1999.

MYERS, G. J. **The Art of Software Testing**. New York: John Wiley & Sons, 1979.

170 p.

NADEAU-DOSTIE, B. **Design for at-Speed Test, Diagnosis and Measurement**. Boston: Kluwer Academic Publishers, 1999. 264p.

NAVABI, Z. **VHDL:** Analysis and Modeling of Digital Systems. 2nd ed. [S.l.]: McGraw-Hill Professional, 1997. 656p.

NGUYEN, T.B.; ROBACH, C. Mutation Testing Applied to Hardware: the Mutant Generation. In: IFIP INTERNATIONAL CONFERENCE ON VERY SCALE INTEGRATION, 11., 2001. **Proceedings...** Montpellier: LIRMM, 2001.

NICOLICI, N. **Power Minimisation Techniques for Testing Low Power VLSI Circuits**. 2000. 268f. PhD Thesis(degree of Doctor of Philosophy) – University of Southampton, England.

NIERMANN, T. **Techniques for Sequential Circuits Automatic Test Generation**. 1991. Ph.D. dessertation, University of Illinois, Urbana, Il.

NIERMANN, T.; CHENG, W.; PATEL, J. PROOFS: A Fast Memory Efficient Fault Simulator for Sequential Circuits. In: DESIGN AUTOMATION CONFERENCE, DAC, 27., 1990, Orlando, USA. **Proceedings**... New York: IEEE, 1990.

NORWOOD, R.B. **Synthesis-for-Scan:** Reducing Scan Overhead with High Level Synthesis. 1997. 67f. Dissertation (Partial Fulfillment of the Requirements for the Degree of Doctor of Phiilosophy), Stanford University, San Francisco, USA.

PALNITKAR, S. **Verilog HDL.** 2nd ed.[S.l.]: Prentice Hall, 2003.

PAOLI, C. et al. Path_Oriented Test Data Generation of Behavioral VHDL Description. In: IEEE INTERNATIONAL WORKSHOP ON ELETRONIC DESIGN, TEST AND APPLICATIONS, 2002. **Proceedings…**[S.l.]: IEEE, 2002.

PARK, I.; HA, D.S.; SIM, G. A New Method for Partial Scan Design Based on Propagation and Justification Requirements of Faults. In: INTERNATIONAL TEST CONFERENCE, ITC, 1995. **Proceedings...** [S.l.:s.n.], 1995.

PATNAIK, L.M. et al. The State of VLSI Testing. **IEEE Potentials**, [S.l.], v.21, n.3, 2002. p.12-16.

PERRY, D. **VHDL**. San Ramon, California: MacGraw-Hil, 1991.

POTKONJAK, M.; DEY, S.; ROY, K.R. Considering Testability at Behavioral Level: Use of Transformations for Partial Scan Cost Minimization Under Timing and Area Constraints. **IEEE Transactions on Computer-Aided Design**, New York, v. 14, n.5, p. 531-546, May 1995.

PRADHAN, D.K. **Fault-Tolerant Computer System Design**. Upper Saddle River: Prentice Hall, 1996. 550p.

PRESSMAN, R. S. **Software Engineering**. New York: MacGraw-Hill, 2000.

PRICE, A. M. A. Teste em Engenharia de Software. In: WORKSHOP EM PROGRAMAÇÃO CONCORRENTE, SISTEMAS DISTRIBUÍDOS E ENGENHARIA DE SOFTWARE, 1991. **Anais...** São Carlos: ICMSC – USP, 1991. p. 93-104.

RAJSUMAN, R. **Digital Hardware Testing:** Transistor-Level Fault Modelling and

Testing. [S.l.]:Artech House, 1992.

RAPPS, S.; WEYUKER, E. Selecting Software Test Data Using Data Flow Information. **IEEE Transactions on Software Engineering**, New York, v.SE-11, n. 4, Apr. 1985.

REIS, R. et al. **Concepção de Circuitos Integrados**. 2 ed. Porto Alegre: Sagra Luzzatto, 2002. 272p.

ROTH, J.P.; BOURICIUS, W.G.; SCHNEIDER, P.R. Programmed Algorithms to Compute Test to Detect and Distinguish between Failures in Logic Circuits. **IEEE Transactions on Electronic Computers,** New York, v. EC-16, n. 5, p. 567-580, Oct. 1967.

RUDNICK, E. M. et al. Sequential Circuit Test Generation in Genetic Algorithm Framework. In.: DESIGN AUTOMATION CONFERENCE, DATE, 1994. **Proceedings...** [S.l.:s.n.], 1994. p. 698-704.

RUDNICK, E.M. et al. Fast sequential circuit test generation using high-level and gate-level techniques. In: DESIGN AUTOMATION AMD TEST IN EUROPE, DATE, 1998. **Proceedings…** New York: ACM, 1998. p. 570-576.

SAFARI, S.; ESMAEILZADEH, H.;JAHANGIR, A. A Novel Improvement Technique for High-Level Test Synthesis. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2003. **Proceedings...** [S.l.:s.n.], 2003. p.609-612.

SESHADRI, S.; HSIAO, M. Behavioral-Level DFT via Formal Operator Testability Measures. **Journal of Electronic Testing: Teory and Applications**, [S.l.], v. 18, n.6, p.595-611, 2002.

SHAFER, J. **Improving Software Testing**. Disponível em: <http://www.data-dimensions.com/TestersNetwork/testability.html>. Acesso em: 05 jun. 2001.

SHERWANI, N. **Algorithms for VLSI Physical Design Automation**. 3rd ed. USA: Kluwer Academic, 1998. 608p.

SHOLIVÉ, M. et al. Mutation Sampling Technique for the Generation of Structural Test Data. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2005. **Proceedings...** Piscataway: IEEE Computer Society, 2005. v. 2, p. 1022-1025.

SHOLIVÉ, M. et al. Software-Based Testing of Sequential VHDL Descriptions. In: IEEE EUROPEAN TEST WORKSHOP, ETW, 8., 2003. **Proceedings...** [S.l.:s.n.], 2003. p. 199-200.

SICSTRUS. **SICStrus Prolog User´s Manual**. [S.l.]: Swedish Institute of Computer Science, 2003.

SUGETA, T.; MALDONADO, J.C.; WONG, W.E. Structural and Mutation Testing for SDL Specifications: A Case Study. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador. **Digest of papers** [S.l.:s.n.], 2005. p.296-301.

TERROSO, A.R. **Projeto de Arquiteturas Tolerantes a Falhas Através da Linguagem de Descrição de Hardware VHDL**. 1999. Dissertação (Mestrado em Ciências da Computação) PUCRS, Porto Alegre.

THOMAS, D.; MOORBY, P. **The Verilog Hardware Description Language**. 4th ed.

USA: Kluwer Academic, 1998. 376p.

TOMAZELA, M. G. J. M.; MALDONADO, J. C. Avaliação do Custo de Aplicação dos Critérios Potenciais Usos no Teste de Programas COBOL. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997. **Anais...** Águas de Lindóia: ICMSC/USP, 1997. p.147-159.

VADO, P. et al. Methodology for Validating Digital Circuits with Mutation Testing. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2000, Geneva, Switzerland. **Proceedings…** Piscataway: IEEE, 2000. p.343-346.

VERGÍLIO, S.R.; MALDONADO, J.C.; JINO, M. Aumentando a Eficácia dos Critérios Estruturais Através da Utilização de Critérios Restritos. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997. **Anais...** Águas de Lindóia: ICMSC/USP, 1997. p. 137-148.

VINCENZI, A. M. R. et al. Critério Análise de Mutantes: Estado Atual e Perspectivas. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997. **Anais...** Águas de Lindóia: ICMSC/USP, 1997. p. 123-134.

VISHAKANTAIAH, P.; ABRAHAM, J. High-Level Testability Analysis Usign VHDL Descriptions. In: EUROPEAN CONFERENCE ON DESIGN AUTOMATION, 1993. **Proceedings..**. Los Alamitos: IEEE Computer Society, 1993. p. 170-174.

WADSACK, R. L. Fault Modelling and Logic Simulation of CMOS and MOS Integrated Circuits. **The Bell System Technical Journal**, New York, 1978.

WEISER, M. Program slicing. **IEEE Transactions on Software Engineering,** New York, v. SE-10, n. 4, p. 352-357, July 1984.

WHITTAKER, J. A. Stochastic Software Testing. **Annals of Software Engineering**, Amsterdam, v. 4, n. 1, p. 115-131, 1997.

WILSON, R. J. **Introduction to Graph Theory.** 4th ed. [S.l.]: Addison-Wesley, 1997.

WIRTH, N. **Algoritmos e Estruturas de Dados**. Rio de Janeiro : Prentice, 1989.

WONG, M. et al. Constrained Mutation in C Programs. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 8.,1994, Curitiba. **Anais...** Curitiba: Centro Internacional de Tecnologia de Software, 1994.

XIANG, D. et al. Partial Scan Design Based on Circuit State Information. In: DESIGN AUTOMATION CONFERENCE, DAC, 1996. **Proceedings...** New York: ACM, 1996. p. 807-812.

ZHANG, Q.; HARRIS, I.G. A data flow fault coverage metric for validation of behavioral HDL descriptions. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, 2000. **Proceedings...** [S.l.:s.n.], 2000. p. 369-373.