UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ALAIR DIAS JÚNIOR

ANÁLISE DE COBERTURA E GERAÇÃO DE VETORES DE TESTE PARA MÓDULOS DESCRITOS EM SYSTEMC

Belo Horizonte
2008

ALAIR DIAS JÚNIOR

ANÁLISE DE COBERTURA E GERAÇÃO DE VETORES DE TESTE PARA MÓDULOS DESCRITOS EM SYSTEMC

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Engenharia de Computação e Telecomunicações

ORIENTADOR: Prof. Dr. Diógenes Cecílio da Silva Júnior

Belo Horizonte
2008

ALAIR DIAS JÚNIOR

ANÁLISE DE COBERTURA E GERAÇÃO DE VETORES DE TESTE PARA MÓDULOS DESCRITOS EM SYSTEMC

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Diógenes Cecílio da Silva Júnior, UFMG
PhD. em Computer Engineering pela University of Southern California – Califórnia, EUA

Banca Examinadora:

Prof. Dr. Wang Jiang Chau, USP
Doutor pela Syracuse University – Nova York, EUA

Prof. Dr. Antônio Otávio Fernandes, UFMG
Doutor pela Institute National Polytechnique de Grenoble – Grenoble, França

Coordenador do PPGEE: _____
Prof. Dr. Hani Camille Yehia

Belo Horizonte, Fevereiro de 2008.

DEDICATÓRIA

Dedico este trabalho aos meus pais, Alair e Iêta.

AGRADECIMENTOS

Agradeço aos meus pais, por me darem as condições de chegar até este momento de minha vida, por vezes, sacrificando suas próprias vontades; ao meu irmão, Marcelo, por ter assumido minhas obrigações familiares durante a reta final do desenvolvimento deste trabalho; e a todos os meus familiares, em especial à madrinha Delaine, por indicar direções em momentos onde não encontrava as respostas certas sozinho.

Agradeço também aos meus colegas de laboratório pelo suporte nos momentos necessários. Ao Adriano por sempre demonstrar confiança sobre as direções de minha pesquisa e ao Carlos por sacrificar seu tempo quando estava às voltas com os testes do *Controlador de LCD*. Agradeço aos meus amigos, principalmente ao Rafael e ao Joel, que se dispuseram a ler trechos deste trabalho.

Ao meu orientador, Diógenes, por estar sempre disposto a ouvir e por acreditar nas direções que tomei no desenvolvimento desta pesquisa.

Ao CNPq, por financiar a pesquisa.

Por último, mas especialmente, agradeço à minha namorada, Amália, por ser meu porto seguro em momentos difíceis.

RESUMO

O crescimento da complexidade dos sistemas VLSI e a necessidade cada vez maior de reduzir o tempo de desenvolvimento pressiona o ciclo de projeto de SoCs até seus limites. Garantir o funcionamento correto desses sistemas complexos também é um desafio, uma vez que o número de estados a serem verificados aumenta exponencialmente.

Hoje, a verificação é a etapa mais dispendiosa do desenvolvimento, consumindo até 70% dos recursos disponíveis. É clara a necessidade de ferramentas e metodologias que aumentem a sua eficiência. O aumento do nível de abstração, utilizado inicialmente para reduzir o *gap* de produtividade dos projetos, contribui para a verificação, permitindo que esta comece mais cedo no ciclo de projeto. Porém, a utilização de linguagens de alto nível obriga a indústria de microeletrônica a se reorganizar. Um caminho natural é a utilização de técnicas e ferramentas empregadas na indústria de *software*, onde as linguagens de alto nível de abstração são utilizadas há várias décadas, estando bem consolidadas.

Este trabalho possui três objetivos. O primeiro é desenvolver uma ferramenta capaz de analisar a cobertura estrutural atingida pelos vetores de teste aplicados a um determinado sistema descrito em SystemC. A ferramenta deve permitir analisar a qualidade dos vetores de teste, disponibilizando informações que podem ser utilizadas pelo engenheiro de verificação para melhorar o ambiente de verificação.

O segundo objetivo é apresentar um método para a geração de vetores de teste para sistemas combinatórios descritos em SystemC. Com esse método espera-se aumentar a controlabilidade do sistema, permitindo que o engenheiro de testes crie vetores para estimular uma determinada porção do código fonte.

O terceiro objetivo do trabalho é apresentar uma metodologia híbrida para testes de sistemas descritos em SystemC que combina teste funcional com teste estrutural visando a aumentar a eficiência dos testes. A partir das informações recolhidas pela ferramenta de análise de cobertura, pode-se determinar partes do código não exercitadas, ou exercitadas de modo insatisfatório, durante os testes funcionais. Uma vez identificadas essas partes, um método para geração de vetores pode ser empregado para criar os estímulos que exercitem essas porções não cobertas.

Palavras-chave: Verificação, cobertura de código, SoC, SystemC.

ABSTRACT

The growing complexity of VLSI systems and the necessity to reduce the development time push the design cycle to its limits. Assuring the correct behavior of these designs is a major problem as the number of states to verify explodes.

Today, verification is the most consuming step of SoC development, representing 70% of the total development effort. It is clear the need for new tools and methodologies to increase verification efficiency. High level design, which was first meant to reduce the productivity gap, contributes to verification, allowing it to begin earlier in the design cycle. Although, using high-level languages to describe hardware requires a reformulation on the microelectronics industry. The natural way is to use tools and techniques developed in the software industry, where these languages have been used for decades.

This work has three goals. The first is to create a tool capable of analyzing the structural coverage achieved by the test vectors applied to a system described using SystemC. The tool should allow verifying the quality of the test vectors, providing information that could be used by the test engineer to improve the verification environment.

The second goal is to present a method for the generation of test vectors for combinational systems described using SystemC. This method is expected to increase the controllability of the system, allowing the test engineer to create test vectors to exercise a given fragment of the source-code.

The third goal is to describe a hibrid methodology for the test of designs described using SystemC that combines functional testing with structural testing in order to improve test performance. Using the coverage information extracted by the structural coverage tool, it is possible to determine which portions of the source-code were not exercised properly by the funcional test. A method for generating test vectors could be applied to create the vectors necessary to exercise these uncovered portions.

Keywords: high-Level Verification, SoC, SystemC, code coverage.

LISTA DE ILUSTRAÇÕES

Figura 1: Comparação de Linguagens 18
Figura 2: Arquitetura do SystemC 19
Figura 3: Fluxo de Compilação Simplificado do Código SystemC 20
Figura 4: Exemplo de um Grafo Dirigido 24
Figura 5: Programa com Enunciados de Três Endereços 26
Figura 6: Exemplo de Grafo de Fluxo 26
Figura 7: Ciclo de Projeto de SoC 30
Figura 8: Níveis de abstração e velocidade de simulação 31
Figura 9: Custo da correção do erro X Momento da detecção 34
Figura 10: Aumento da Eficiência na Verificação 35
Figura 11: Ciclo de Verificação 36
Figura 12: Forma típica de um *Testbench* 41
Figura 13: Relação entre as Coberturas de Código 48
Figura 14: Contra Exemplo 1 50
Figura 15: Contra Exemplo 2 50
Figura 16: Contra Exemplo 3 52
Figura 17: Relação entre as Coberturas de Fluxo de Dados 55
Figura 18: Execução simbólica para o caminho 0, 1, 2, 3, 5, 6 e 7. 55
Figura 19: Execução simbólica para o caminho 0, 1, 2, 3, 4 e 7. 56

Figura 20: Fluxograma da Metodologia Proposta 63
Figura 21: Detalhe da Verificação no Ciclo de Projeto de SoC 64

Figura 22: Funções de Penalidade para o Método Numérico 71
Figura 23: Grafo de Fluxo para Exemplo de Aplicação do Método Numérico . . 74

Figura 24: Fluxo de Compilação Detalhado do Código SystemC 78
Figura 25: Fluxo de Compilação do Código SystemC usando a Ferramenta ScCov 79
Figura 26: Diagrama de Blocos da Ferramenta ScCov 80
Figura 27: Regras em Cascata no Analisador Léxico 83
Figura 28: Regras e Extratores de Símbolos do Editor de Código 84
Figura 29: Relatório de Cobertura de Funções 102

Figura 30: Unidade Aritmética Simples 108
Figura 31: Impacto no Desempenho da Unidade Aritmética 110
Figura 32: CPU RISC da Synopsys 111
Figura 33: Impacto no Desempenho da CPU RISC da Synopsys 113
Figura 34: Controlador de LCD 114

Figura 35: Imagem Reconstruída pelo Controlador de LCD 115

LISTA DE TABELAS

Tabela 1: Resultado das Iterações do exemplo 75

Tabela 2: Conjunto de Delimitadores do extrator de símbolos 82
Tabela 3: Conjunto de Delimitadores Atômicos do Extrator de Símbolos 82
Tabela 4: Exemplo Seqüência de Valores para os Termos do Enunciado *if* . . . 96

Tabela 5: Enumeráveis da UA 108
Tabela 6: Vetores de Teste Aplicados no Experimento 2 109
Tabela 7: Enumeráveis da CPU RISC da Synopsys 111
Tabela 8: Enumeráveis do Controlador de LCD 114
Tabela 9: Impacto no Desempenho da Simulação do Controlador de LCD . . . 115

LISTA DE ABREVIATURAS

ASIC Application Specific Integrated Circuit
ASM Abstract State Machine
CAD Computer Aided Design
CPU Central Processing Unit
FPGA Field Programmable Gate Array
HDL Hardware Description Language
IP Intellectual Property
IPC Interprocess Communication
LHS Left Hand Side
NRE Non-Recurring Expenditures / Expenses
RHS Right Hand Side
RISC Reduced Instruction Set Computer
RTL Register Transfer Level
SDL System-level Description Language
SoC System-on-a-Chip
UML Unified Modeling Language
VHDL VHSIC Hardware Description Language
VHSIC Very High Speed Integrated Circuits
VLSI Very Large Scale Integrated
XML Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Motivação 15
1.2 Objetivos 15
1.3 Sumário 16

2 REVISÃO BIBLIOGRÁFICA 17
2.1 SystemC 17
2.1.1 Descrição de Sistemas usando SystemC 20
2.2 Grafos Dirigidos 23
2.2.1 Grafos de Fluxo de Programa 24
2.2.2 Grafos de Definição e Uso 26
2.3 Projeto de SoCs 29
2.4 Verificação em Projetos de SoCs 33
2.4.1 Ciclo de Verificação 35
2.4.2 Níveis de Observação 38
2.4.3 Tipos de Verificação 39
2.4.4 Níveis de Verificação 42
2.4.5 Análise de Cobertura 44
2.5 Geração de Vetores de Teste baseada em Execução Simbólica 54
2.6 Trabalhos Relacionados 56

3 METODOLOGIA HíBRIDA PARA VERIFICAÇÃO DE SOCS 60
3.1 Considerações Sobre o Emprego de Cobertura Estrutural na Verificação de SoCs 60
3.2 Metodologia Híbrida para Verificação 62

4 MÉTODO NUMÉRICO PARA GERAÇÃO DE VETORES DE TESTE PARA MÓDULOS DESCRITOS EM SYSTEMC 67
4.1 Descrição do Método Numérico 67
4.2 Resultados da Aplicação do Método Numérico 72
4.3 Análise do Método 75

5 SCCOV: FERRAMENTA PARA ANÁLISE DE COBERTURA DE TESTE 77
5.1 Estrutura da Ferramenta ScCov 79
5.1.1 Extrator de Símbolos 81
5.1.2 Analisador Léxico 82
5.1.3 Editor de Código 83
5.1.4 Servidor de Dados e Interface de Usuário 99

5.1.5 Relatórios de Cobertura 100

6 RESULTADOS DA FERRAMENTA SCCOV 104
6.1 Metodologia 104
6.1.1 Descrição do Experimento 1 105
6.1.2 Descrição do Experimento 2 106
6.1.3 Descrição do Experimento 3 107
6.1.4 Descrição do Experimento 4 107
6.2 Unidade Aritmética Simples 108
6.2.1 Resultados do Experimento 1 108
6.2.2 Resultados do Experimento 2 109
6.2.3 Resultados do Experimento 3 109
6.2.4 Resultados do Experimento 4 110
6.3 CPU RISC da Synopsys 111
6.3.1 Resultados do Experimento 1 111
6.3.2 Resultados do Experimento 2 111
6.3.3 Resultados do Experimento 3 112
6.3.4 Resultados do Experimento 4 112
6.4 Controlador de LCD 113
6.4.1 Resultados do Experimento 1 114
6.4.2 Resultados do Experimento 2 114
6.4.3 Resultados do Experimento 3 114
6.4.4 Resultados do Experimento 4 115
6.5 Análise dos Resultados dos Experimentos 116

7 CONCLUSÕES 118

REFERÊNCIAS 122

APÊNDICE A RELATÓRIOS DE COBERTURA 126
A.1 Relatório de Cobertura de Ramos 126
A.2 Relatório de Cobertura de Termos 127
A.3 Relatório de Cobertura Exaustiva de Termos 127
A.4 Relatório de Cobertura Relacional 128
A.5 Relatório de Cobertura de Laços 129

1 INTRODUÇÃO

O expressivo crescimento da utilização de sistemas computacionais, impulsionado principalmente pelo emprego de tecnologias móveis, pressiona a evolução do mercado de eletrônicos. Hoje, *Time-to-Market* é a expressão decisiva na bilionária indústria de microeletrônica onde lançar um produto no mercado antes do concorrente, e com preço acessível, pode significar a diferença entre o sucesso e o fracasso total.

Capacidade de memória e velocidade de processamento não são mais problemas imediatos. Os sistemas computacionais atuais, principalmente no que se refere aos computadores pessoais, oferecem desempenho satisfatório para a grande maioria dos usuários a preços relativamente baixos (Tredennick e Shimamoto, 2004). Reduzir o consumo energético dos dispositivos eletrônicos passa a ser um dos novos desafios da indústria de microeletrônica, juntamente com o aumento de produtividade no projeto de sistemas VLSI (*Very Large Scale Integrated*).

A evolução dos processos de fabricação de circuitos integrados reduziu o tamanho do canal do transistor de dezenas de milionésimos do metro, na década de sessenta, a dezenas de bilionésimos do metro hoje em dia. Essa redução, aliada ao aumento do tamanho da pastilha de silício (*die*) e ao aumento do rendimento do processo (*Yield*), levou a um crescimento no número de elementos ativos disponíveis, chegando a sistemas com centenas de milhões de transistores. Os projetistas tornaram-se incapazes de desenvolver sistemas na mesma velocidade em que são disponibilizados elementos ativos, criando o problema do *gap* de produtividade (Chang et al., 1999).

Assim como ocorrido na indústria de *software* há algumas décadas, o aumento do ní-

vel de abstração dos projetos de VLSI foi uma das soluções empregadas para aumentar a produtividade. Linguagens de descrição de *hardware*, como VHDL e Verilog, foram criadas para permitir a descrição e simulação de SoCs (*System-on-a-Chip*)(Chang et al., 1999) em níveis mais altos de abstração, onde o desenvolvimento do sistema é independente da tecnologia de fabricação do circuito integrado. Com isso, a mesma descrição pode ser utilizada tanto para mapeamento em FPGAs (*Field Programmable Gate Arrays*) quanto em ASICs (*Application Specific Integrated Circuits*), tornando possível o desenvolvimento baseado em plataformas (Claasen, 2003; Bergamaschi e Cohn, 2002; Gajski et al., 2000; Bricaud, 1999; Chang et al., 1999) e o reúso de componentes por meio da utilização de blocos de propriedade intelectual (IP).

Apesar de garantirem um aumento de produtividade, as primeiras linguagens de descrição de *hardware* (HDLs) estavam limitadas aos níveis RTL (*Register Transfer Level*) e comportamental, além de não permitirem a co-simulação entre *hardware* e *software*. Por este motivo, a tendência é a utilização de linguagens de descrição de sistemas, que podem ser utilizadas desde os primeiros estágios do desenvolvimento até níveis mais baixos de abstração, acelerando a curva de aprendizado dos projetistas. A co-simulação de módulos descritos em diferentes níveis de abstração, e até mesmo *software*, torna-se possível, fazendo com que a exploração de arquiteturas e a verificação possam ser iniciadas cada vez mais cedo no ciclo do projeto.

Outros fatores, além do *gap* de produtividade, determinam a necessidade de uma melhoria nas metodologias de projeto. Os SoCs atuais integram uma grande variedade de funcionalidades, como radiofreqüência, funções analógicas e de núcleos baseados em *software*, e requerem uma equipe de projeto preparada para integrar todas essas funcionalidades, além de ferramentas de apoio a esse desenvolvimento. Muitas questões ainda estão em aberto no projeto de SoCs em nível de sistema, muitas delas relacionadas à síntese de sistemas descritos acima do nível RTL. Porém, garantir o funcionamento correto desses sistemas cada vez mais complexos é o grande desafio da indústria de microeletrônica atual.

1.1 Motivação

Hoje, o papel da verificação no ciclo de projeto é maior do que apenas garantir o funcionamento correto do sistema. A verificação atua de forma positiva em três grandezas importantes do projeto: o custo, o tempo de desenvolvimento e a qualidade do produto final. A principal característica deste trinômio é que, otimizando-se duas destas grandezas, a terceira será afetada de forma negativa. Como a verificação atua de forma positiva nos três itens deste trinômio, aumentando a qualidade do produto final e reduzindo os custos e o tempo gastos com retrabalho durante o desenvolvimento, ela se torna uma etapa crítica, consumindo em torno de 70% dos recursos do projeto (Bergeron, 2003). Assim, técnicas mais eficientes são necessárias para compensar a crescente complexidade requerida para uma verificação efetiva dos sistemas.

Hoje, nota-se uma carência de ferramentas que ofereçam suporte eficiente à verificação de SoCs, principalmente para a geração de vetores de testes complexos requeridos por grandes sistemas (Claasen, 2003). Porém, tanto quanto os testes em si, ferramentas, critérios e métodos para medir a qualidade dos testes aplicados em um sistema, ou a sua cobertura, são importantes para definir quando um sistema foi suficientemente testado. Na indústria de *software*, a análise de cobertura é largamente utilizada, principalmente a cobertura estrutural que possui critérios e métodos bem definidos. No entanto, apenas recentemente, essas ferramentas começaram a ser desenvolvidas para sistemas descritos em SystemC (OSCI, 2007).

1.2 Objetivos

Este trabalho possui três objetivos. O primeiro, e principal, é desenvolver uma ferramenta capaz de analisar a cobertura estrutural atingida pelos vetores de teste aplicados a um determinado sistema descrito em SystemC. A ferramenta deve permitir analisar a qualidade dos vetores de teste, disponibilizando informações que podem ser utilizadas pelo engenheiro de verificação para melhorar o *testbench* (uma bancada de testes do dispositivo em desenvolvimento descrita na própria linguagem SystemC) e aumentar a observabilidade da verificação.

O segundo objetivo do trabalho é apresentar um método para a geração de vetores de teste para sistemas combinatórios descritos em SystemC. Com esse método, espera-se aumentar a controlabilidade do sistema, permitindo que o engenheiro de teste crie vetores para estimular uma determinada porção do código fonte.

O terceiro objetivo é apresentar uma metodologia híbrida para testes de sistemas descritos em SystemC que visa aumentar a eficiência dos testes. A partir das informações recolhidas pela ferramenta de análise de cobertura, pode-se determinar partes do código não exercitadas (ou exercitadas de modo insatisfatório) durante os testes funcionais (*black-box*). Uma vez identificadas essas partes, um método para a geração de vetores pode ser empregado para criar os estímulos que exercitem essas porções não cobertas.

1.3 Sumário

Tópicos importantes para o desenvolvimento do trabalho serão apresentados no capítulo 2, bem como uma revisão dos trabalhos já publicados na área. O capítulo 3 apresenta uma metodologia híbrida para verificação de SoCs, além de considerações sobre a aplicação de cobertura estrutural em projetos de SoCs. Um método para geração de vetores de teste para sistemas combinatórios é apresentado no capítulo 4. O capítulo 5 apresenta uma ferramenta para a análise de cobertura de sistemas descritos em SystemC, que oferece suporte à metodologia apresentada no capítulo 3. Resultados da aplicação da ferramenta são apresentados no capítulo 6. O trabalho termina com conclusões e direções para trabalhos futuros no capítulo 7.

2 REVISÃO BIBLIOGRÁFICA

Nesse capítulo, será feita uma abordagem de vários tópicos importantes para o desenvolvimento do trabalho. Inicialmente, a biblioteca SystemC será apresentada, pois ela é a base das ferramentas desenvolvidas e dos experimentos realizados. Grafos dirigidos, grafos de fluxo e grafos de definição e uso serão abordados na seção 2.2. Em seguida, será apresentado o ciclo de projeto de SoCs, definindo alguns níveis de abstração possíveis na descrição de sistemas em alto nível. Uma introdução à verificação de sistemas será dada na seção 2.4, enfatizando-se a verificação estrutural baseada em simulação, que será largamente utilizada nesse trabalho. Noções de geração de vetores de teste baseadas em execução simbólica de código serão apresentadas. Por fim, será feita uma revisão sobre os trabalhos afins já publicados.

2.1 SystemC

A modelagem de sistemas por meio de linguagens de abstração de *hardware* não é novidade na indústria de microeletrônica. As linguagens de abstração de *hardware*, ou HDLs, foram utilizadas, por muito tempo, como uma maneira de agilizar o processo de simulação, abstraindo-se das características físicas dos semi-condutores e concentrando-se no funcionamento lógico dos sistemas. A necessidade do aumento de produtividade pressionou o desenvolvimento de ferramentas de apoio ao projeto (CAD) que permitissem a síntese de *hardware* a partir das HDLs e uma nova indústria de microeletrônica surgiu. Hoje, diversas linguagens são utilizadas para modelar tanto o *hardware* quanto o *software* dos sistemas computacionais e cada uma delas possui seus pontos fortes e fracos.

A figura 1 foi traduzida de (Black e Donovan, 2005) e compara diversas linguagens utilizadas na descrição de SoCs, indicando os domínios nos quais essas linguagens são geralmente empregadas. Pode ser visto que o padrão SystemC (OSCI, 2007) apresenta uma grande abrangência de domínios, permitindo que a mesma linguagem seja utilizada durante quase todo o ciclo de projeto de SoCs. A abrangência do padrão e sua flexibilidade foram as principais motivações para a escolha do SystemC como linguagem base para o desenvolvimento deste trabalho.

Figura 1: Comparação de Linguagens

A arquitetura interna do SystemC é apresentada na figura 2, que foi traduzida e adaptada de (OSCI, 2007). Nela, pode-se verificar que o SystemC, formalmente, não é uma linguagem de programação, mas uma extensão do padrão C++ direcionada para a descrição e simulação de sistemas digitais. Um núcleo de simulação baseado em eventos foi adicionado de forma a permitir a execução de processos concorrentes, necessários para simular o paralelismo de um circuito digital.

Logo acima do núcleo de simulação, está o núcleo da linguagem, que define o comportamento básico para os *Modules*[1], *Ports*, *Interfaces* e *Channels*. *Modules* são os blocos básicos do SystemC, permitindo dividir um sistema complexo em unidades mais simples.

[1]Para não ocorrer ambigüidades, optou-se por não se traduzir os nomes dos componentes do SystemC.

Cada *Module* pode possuir mais de um processo, indicando comportamentos paralelos internos e, inclusive, pode possuir outros *Modules*, formando uma hierarquia. Os *Modules* se comunicam com o ambiente externo por meio das *Ports*. As *Ports* são as instâncias, ou objetos reais, de *Channels*. Por sua vez, os *Channels* são as definições, ou implementações, das *Interfaces*. De uma forma simples, as *Interfaces* definem quais operações, como *read()* ou *write()*, podem ser efetuadas sobre as *Ports* e os *Channels* definem como essas operações devem ser efetuadas.

Figura 2: Arquitetura do SystemC

O SystemC define também tipos básicos de dados em adição aos tipos do padrão C++, incluindo bits, tipos de dados com quatro valores (0, 1, X, Z) para simular o comportamento de fios, inteiros de precisão arbitrária e números com representação em ponto fixo. Por utilizar o C++, tipos de dados definidos pelo usuário também pode ser criados. Alguns canais básicos são definidos pelo padrão SystemC. Esses canais são utilizados para implementar a comunicação entre módulos. É possível para o projetista definir outros canais mais complexos e bibliotecas específicas, conforme necessário.

A figura 3 mostra um fluxo de compilação simplificado de um código SystemC. Por se tratar, basicamente, de um código C++, pode-se utilizar o compilador GCC (GNU,

2007a) para gerar o modelo executável do sistema. Na figura, as etapas internas do compilador GCC foram omitidas por simplicidade. O código fonte SystemC, desenvolvido pelo projetista, é compilado e ligado às bibliotecas do C++ e do SystemC para gerar um modelo executável do sistema com núcleo de simulação embutido.

Figura 3: Fluxo de Compilação Simplificado do Código SystemC

Em seguida, a descrição de sistemas utilizando SystemC é detalhada.

2.1.1 Descrição de Sistemas usando SystemC

A descrição de sistemas em SystemC é baseada na extensão das classes definidas no padrão, onde o projetista adiciona o código fonte que define o comportamento de cada uma das classes derivadas. Por esse motivo, SystemC pode ser considerado um *framework* para aplicações de simulação baseada em eventos. O código do usuário é compilado e ligado às bibliotecas básicas do *framework*, criando um código executável que simula o sistema.

Considere um exemplo de um somador com duas entradas, a e b, e uma saída s, todas do tipo inteiro. O sistema soma o valor das duas entradas e apresenta o resultado na saída. Pode-se criar um *Module* somador usando SystemC conforme apresentado a seguir:

```
1  SC_MODULE(somador)
2  {
3    // ports, processos e estruturas de dados
4    SC_CTOR(somador)
5    {
6      // corpo do construtor
7      // declaração de processos, lista de sensibilidade
8    }
9  };
```

Nesse trecho de código, a macro SC_MODULE define a classe somador como uma classe derivada de *::sc_core::sc_module*. SC_CTOR cria um construtor para a classe *somador* e também algumas definições necessárias para a declaração de processos e listas de sensibilidade. Ambas as macros são empregadas apenas para facilitar a digitação e leitura do código, não sendo obrigatória a sua utilização. No arquivo *sc_module.h*, distribuído com o SystemC, SC_MODULE é definida como:

```
1  #define SC_MODULE(user_module_name)
2      struct user_module_name : ::sc_core::sc_module
```

Para que o *Module somador* se comunique com o ambiente externo, é necessário que *Ports* sejam adicionadas a ele. *Ports* são classes *template*, ou seja, são classes que necessitam que um ou mais parâmetros sejam fornecidos no momento da sua definição. A definição desses parâmetros se dá entre os sinais "<" e ">". *templates* para as classes *sc_in* e *sc_out* são fornecidos com o padrão do SystemC e correspondem a *Ports* de entrada e saída, respectivamente. Elas podem ser adicionadas ao *Module* definido anteriormente da seguinte forma:

```
1  SC_MODULE(somador)
2  {
3    sc_in<int> a;
4    sc_in<int> b;
5    sc_out<int> s;
6    SC_CTOR(somador)
7    {
8      // corpo do construtor
9      // declaração de processos, lista de sensibilidade
10   }
11 };
```

As *Ports a* e *b* são as entradas do *Module somador*. Como parâmetro para o *Template*,

foi passado o tipo *int*, que determina que essas *Ports* podem receber valores inteiros. A *Port s* é a saída do resultado e também fornece um valor inteiro.

O comportamento do sistema é determinado dentro de processos. Existem vários tipos de processos em SystemC, sendo os mais comuns *SC_METHOD* e *SC_THREAD* (Black e Donovan, 2005). Uma das diferenças entre estes dois tipos de processo é que um processo do tipo *SC_METHOD*, uma vez disparado, sempre executa todo o código definido em seu interior, do início ao fim sem interrupções. Uma *SC_THREAD*, por sua vez, pode ter sua execução suspensa por uma chamada *wait()*. Quando suspensa, a *thread* armazena o ponto onde foi interrompida e o valor de todas as variáveis locais. Quando disparada novamente, a *SC_THREAD* executa a partir do ponto onde foi suspensa.

O trecho de código a seguir ilustra a utilização de processos do tipo *SC_METHOD*. O método *calcula* executa a soma de *a* e *b* e apresenta o resultado na saída *s*. A execução desse método é feita pelo núcleo de simulação do SystemC. A macro SC_METHOD define um processo que executa o método *calcula* sempre que um dos eventos em sua lista de sensibilidade é sinalizado. Para incluir um evento na lista de sensibilidade de um processo, emprega-se a chamada *sensitive*.

```
1  SC_MODULE(somador)
2  {
3    sc_in<int> a;
4    sc_in<int> b;
5    sc_out<int> s;
6
7    void calcula()
8    {
9      s = a + b;
10   }
11
12   SC_CTOR(somador)
13   {
14     SC_METHOD(calcula);
15     sensitive << a << b;
16   }
17 };
```

Para uma descrição mais completa das funcionalidades do SystemC, pode-se consultar (Grötker et al., 2002), (Black e Donovan, 2005) e (OSCI, 2007)

2.2 Grafos Dirigidos

Inúmeras aplicações existem para grafos dirigidos. As mais comuns em computação são as representações de máquinas de estado e de grafos de fluxo. Neste trabalho, os grafos dirigidos, na forma de grafos de fluxo de programa e grafos de definição e uso, são empregados para definir critérios de análise de cobertura estrutural.

> **Definição 1.** Um **grafo dirigido**, $G(V_G, A_g)$, é uma estrutura que consiste em um conjunto de vértices $V_G = \{v_1, v2, ...\}$ e um conjunto de arcos $A_G = \{a_1, a_2, ...\}$. Uma função de incidência Γ_G associa cada arco, a, a um par ordenado de vértices $\{u, v\}$, não necessariamente distinto.
> onde $u, v \in V$.

Seja um grafo dirigido $G(V_G, A_G)$, dois vértices $u, v \in V_G$ e um arco $a \in A_G$. Se a é incidente a $\{u, v\}$, diz-se que a conecta u a v, sendo o vértice u a *cauda* de a e v a *cabeça* de a. Nesse caso, define-se que u é o *predecessor* de v e v é o *sucessor* de u. Um arco cuja *cabeça* e *cauda* são iguais é chamado de *laço*, enquanto um arco cuja *cabeça* e *cauda* são vértices distintos é chamado de *elo*. Um grafo é dito *finito* quando seu conjunto de vértices, V_G, e seu conjunto de arcos, A_G, são ambos finitos e é *simples* quando não possui *laços* e não existem arcos com a mesma *cauda* e *cabeça*.

> **Definição 2.** Um **passeio dirigido** w em um grafo $G(V_G, A_G)$ é uma seqüência finita e não nula, $w = (v_0, a_1, v_1, ..., a_k, v_k)$, cujos termos são alternadamente vértices e arcos, tais que, para $i = 1, 2, ..., k$, a *cabeça* do arco a_i é v_i e sua *cauda* é v_{i-1}.
> onde $v_0, v_1, ..., v_k \in V_G$ e $a_0, a_1, ..., a_k \in A_G$.

> **Definição 3.** Uma **trilha**, t, é um *passeio dirigido* cujos arcos são todos distintos.

> **Definição 4.** Um **Caminho**, c, é uma *trilha* cujos vértices são todos distintos.

Em um grafo simples, um *passeio dirigido* pode ser representado por uma seqüência de vértices, omitindo-se seus arcos.

A equação 1 apresenta um exemplo de um grafo.

$$G = (V_G, A_G) \tag{1}$$

Onde:

$$V_G = \{v_0, v_1, v_2, v_3\} \tag{2}$$

$$A_G = \{a_1, a_2, a_3, a_4, a_5\} \tag{3}$$

E Γ_G é definido como:

$$\begin{aligned} &\Gamma_G(a_1) = (v_0, v_1),\ \Gamma_G(a_2) = (v_0, v_0),\ \Gamma_G(a_3) = (v_1, v_2), \\ &\Gamma_G(a_4) = (v_2, v_3),\ \Gamma_G(a_5) = (v_1, v_3) \end{aligned} \tag{4}$$

Grafos dirigidos podem ser representados graficamente de inúmeras formas. Na mais comum, um ponto ou um círculo representa um vértice, enquanto uma seta que conecta dois vértices representa um arco. A ponta da seta aponta para a *cabeça* do arco, e o outro lado da seta aponta para a sua *cauda*. Tanto os vértices quanto os arcos são identificados por rótulos. A figura 4 mostra o grafo da equação 1.

Figura 4: Exemplo de um Grafo Dirigido

2.2.1 Grafos de Fluxo de Programa

Um programa pode ser decomposto em um conjunto de blocos básicos e o fluxo de controle entre esses blocos pode ser modelado por meio dos grafos de fluxo de programa (Aho et al., 1986). O primeiro passo para a construção do grafo de fluxo de programa, ou simplesmente grafos de fluxo, é separar o programa em blocos básicos. Um bloco básico

é definido como um conjunto de enunciados que são executados de maneira consecutiva, sem uma parada ou possibilidade de ramificação.

> **Definição 5.** Um **bloco básico** é um conjunto de enunciados ordenados $b = \{s_1, ..., s_n\}$ | se $n > 1, \forall\, i = 2, ..., n$, s_i é o único sucessor de s_{i-1} e s_{i-1} é o único predecessor de s_i.

Cada bloco básico identificado em um programa se torna um nó do grafo de fluxo. Um arco conecta um bloco b_1 a um bloco b_2 sempre que o bloco b_2 pode se seguir imediatamente após um bloco b_1, o que acontece quando:

- Existir um enunciado de desvio condicional ou incondicional, do último enunciado de b_1 para o primeiro enunciado de b_2;
- b_2 seguir imediatamente a b_1 na ordem do programa e b_1 não terminar por um desvio condicional.

Assim, pode-se definir um grafo de fluxo de programa.

> **Definição 6.** Seja B o conjunto de blocos básicos de um programa P. Um **grafo de fluxo de programa** $G_P(V_P, A_P)$ é definido como um grafo dirigido, $G(V_G, A_G)$, no qual uma função bijetora Υ associa cada um dos elementos $v \in V_G$, a um elemento, $b \in B$. A função de incidência Γ associa um arco $a \in A_G$ a um par ordenado de vértices $\{u, v\}$, sempre que o bloco básico associado a v puder se seguir imediatamente após o bloco básico associado a u.
> onde $u, v \in V_G$ e $V = V_G$, $A = A_G$ e $\Gamma = \Gamma_G$.

A figura 5 mostra um exemplo de um programa construído utilizando uma seqüência de enunciados de três endereços. Nessa figura, pode-se dividir os enunciados em três blocos básicos: *A*, *B* e *C*. O bloco *A* é o bloco inicial do programa de exemplo. Desse bloco, o fluxo de execução pode fluir para o bloco *B*, caso o teste *b = 0* seja verdadeiro, ou para o bloco *C* caso contrário. A figura 6 mostra o grafo de fluxo para o programa da figura 5.

Grafos de fluxo são extensamente utilizados para a otimização de código, identificação de códigos inalcançáveis (*dead-code*) e laços infinitos.

```
0: (A) a = x + 10;
1: (A) b = y - 10;
2: (A) if (b = 0) goto (4)
3: (B)   a = a / b;
4: (C) s = a;
```

Figura 5: Programa com Enunciados de Três Endereços

Figura 6: Exemplo de Grafo de Fluxo

2.2.2 Grafos de Definição e Uso

Cada ocorrência de uma variável em um programa pode ser classificada como uma definição, um uso-computação ou um uso-prefixo.

> **Definição 7.** Uma **definição** de uma variável ocorre quando ela se encontra do lado esquerdo (LHS) de um enunciado de atribuição.

> **Definição 8.** Um **uso-computação** de uma variável ocorre quando ela se encontra do lado direto (RHS) de um enunciado de atribuição.

> **Definição 9.** Um **uso-prefixo** de uma variável ocorre quando seu valor é testado no prefixo de um enunciado de controle de fluxo.

Seja $f(x_1, ..., x_n)$ uma função das variáveis $x_1, ..., x_n$. Uma atribuição da forma $y = f(x_1, ..., x_n)$ contém *usos-computação* de $x_1, ..., x_n$ e uma *definição* de y. Da mesma maneira, um teste de condição da forma *if* $f(x_1, ..., x_n)$ *then* contém *usos-prefixo* de $x_1, ..., x_n$.

Um vértice de um grafo de fluxo contém um *uso-computação* ou uma *definição* de

uma variável se existe um enunciado no bloco básico associado contendo, respectivamente, um *uso-computação* ou uma *definição* daquela variável. Assim, o bloco ***B*** da figura 5 contém *usos-computação* das variáveis a e b e uma *definição* da variável a.

De forma diferente dos *usos-computação* e das *definições*, os *usos-prefixo* não se associam aos vértices dos grafos, mas sim aos arcos. Sempre que o bloco básico associado ao vértice u termina em um desvio condicional *if* $(x_1, ..., x_n)$ *goto (...)* e sendo os dois sucessores de u os vértices v_1 e v_2, diz-se que os arcos (u, v_1) e (u, v_2) contém *usos-prefixo* das variáveis $x_1, ..., x_n$. Sendo assim, os arcos que saem do bloco *A* da figura 6 contém *usos-prefixo* da variável b.

Utilizando-se os conceitos de *definição*, *uso-computação* e *uso-prefixo*, pode-se definir os conjuntos *def*(v), *uso-c*(v) e *uso-p*(a).

> **Definição 10.** Seja B_P o conjunto dos blocos básicos do programa P. Seja $G_P(V_P, A_P)$ o grafo de fluxo que representa P e X o conjunto das variáveis de P.
> O conjunto ***def***(v) contém todos os elementos de X para os quais existe uma *definição* no bloco básico, b, associado ao vértice v.
> onde $v \in V_P$ e $b \in B_P$.

> **Definição 11.** Seja B_P o conjunto dos blocos básicos do programa P. Seja $G_P(V_P, A_P)$ o grafo de fluxo que representa P e X o conjunto das variáveis de P.
> O conjunto ***uso-c***(v) contém todos os elementos de X para os quais existe um *uso-computação* no bloco básico, b, associado ao vértice v.
> onde $v \in V_P$ e $b \in B_P$.

> **Definição 12.** Seja P um programa representado pelo grafo de fluxo $G_P(V_P, A_P)$ e X o conjunto de variáveis de P.
> O conjunto ***uso-p***(a) contém todos os elementos de X para os quais existe um *uso-prefixo* no arco a.
> onde $a \in A_P$.

Enquanto os grafos de fluxo são utilizados para determinar o fluxo de controle através de um programa, um grafo de definição e uso (Rapps e Weyuker, 1985) pode ser usado para determinar o fluxo de dados. É possível construir um grafo de definição e uso a partir de um grafo de fluxo, associando-se os conjuntos *def*, *uso-c* e *uso-p*.

Definição 13. Um **grafo de definição e uso**, $G_D(V_D, A_D)$, que representa o programa P, é definido como um grafo de fluxo de programa, $G_P(V_P, A_P)$, ao qual são associados os conjuntos *def*(v), *uso-c*(v) e *uso-p*(a).
onde $v \in V_P$, $a \in A_P$, $V_D = V_P$ e $A_D = A_P$.

As definições e usos-computação de uma variável em um programa podem ser classificadas como locais ou globais. Para se formalizar os conceitos de *definições* e *usos-computação* globais, é necessário explorar o conceito de *caminho livre de definições*.

Definição 14. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P e x uma variável de P.
O caminho $c = \{v_0, v_1, ..., v_m\}$ é um **caminho livre de definições** da variável x de v_0 a v_m, ou *cld(x, v_0, v_m)*, se $x \notin$ *def*$_D(v_i)$ $\forall$ $1 \leq i \leq m-1$.
onde $v_0, v_1, ..., v_m \in V_D$ e $m > 1$.

Um *uso-computação* de uma variável x é global quando ele ocorre dentro de um bloco básico para o qual não existam *definições* de x que o precedam. Uma *definição* de uma variável x é global se ela é a última *definição* de x em um bloco associado ao vértice v e existe um *caminho livre de definições* da variável x de v até um vértice contendo um *uso-computação* global ou até um arco contendo um *uso-prefixo* de x.

Um conceito importante que pode ser extraído de um grafo de fluxo é o conceito de *caminho definição e uso*, ou *caminho-du*, que corresponde a um caminho entre uma definição e um uso de uma variável. Para a definição deste conceito, é preciso de se definir o conceito dos conjuntos *def-global(v)* e *uso-c-global(v)*.

Definição 15. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P e X o conjunto de variáveis de P.
Define-se ***def-global(v)*** como o conjunto de variáveis X_v para as quais ocorre uma *definição global* no bloco básico b associado a v.
onde $v \in V_D$, $b \in B_D$ e $X_v \subset X$.

Definição 16. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P e X o conjunto de variáveis de P.
Define-se ***uso-c-global(v)*** como o conjunto de variáveis X_v para as quais ocorre um *uso-computação global* no bloco básico b associado a v.
onde $v \in V_D$, $b \in B_D$ e $X_v \subset X$.

Definição 17. O caminho $c = (v_1, v_2, ..., v_m)$ é um **caminho-du** da variável x se $x \in$ *def-global*(v_1) e:

$$\begin{cases} x \in \textit{uso-c-global}(v_m); \text{ ou} \\ x \in \textit{uso-p}(a). \end{cases}$$

onde a é o arco cuja cauda é v_{m-1} e a cabeça é v_m.

Além desses três conjuntos básicos, mais dois serão definidos. O conjunto *def-uso-c*(x, v) e o conjunto *def-uso-p*(x, a).

Definição 18. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P e X o conjunto de variáveis de P.
Define-se ***def-uso-c(***x, u***)*** como o conjunto de vértices $V_C \mid v \in V_C$ se $x \in$ *uso-c-global*(v) e existe *cld(*x*, u, v)*.
onde $v \in V_D$ e $x \in X$.

Definição 19. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P e X o conjunto de variáveis de P.
Define-se ***def-uso-p(***x, u***)*** o conjunto de arcos $A_P \mid a \in A_P$ se $x \in$ *uso-p*(a) e existe *cld(*x*, u, v)*.
onde $u, v \in V_D$, $x \in X$, $a \in A_D$ e a cabeça do arco a é v

Os conjuntos e definições tratados nesta seção serão utilizados para a análise de cobertura de fluxo de dados, apresentada no item 2.4.5.2.

2.3 Projeto de SoCs

Como foi visto anteriormente, o aumento do nível de abstração na descrição de sistemas digitais foi uma das soluções empregadas para diminuir o problema do *gap* de produtividade (Chang et al., 1999). Essa abordagem possui muitas vantagens em relação ao projeto diretamente no nível de circuito, pois quanto maior o nível de abstração, menor o detalhamento da solução e, conseqüentemente, mais rápida é a simulação do seu modelo. O aumento na velocidade da simulação permite a exploração de arquiteturas e também, que a verificação comece mais cedo no ciclo de projeto.

Uma conseqüência do aumento do nível de abstração é a possibilidade do emprego da metodologia de refinamentos sucessivos. Essa metodologia de projeto permite que

o projetista se concentre em detalhes pertinentes a cada nível de abstração, deixando as demais características para serem definidas posteriormente, em outro nível. Quando um modelo está suficientemente validado, o sistema é refinado para um nível mais baixo, onde mais detalhes são incluídos. O processo de refinamento prossegue até que o sistema esteja em um nível adequado para a criação do circuito físico.

Um ciclo de projeto de SoCs baseado na metodologia de refinamentos sucessivos é apresentado na figura 7, que é uma versão modificada da apresentada em (Wile et al., 2005). A análise de requisitos dos clientes é o início do ciclo de projeto, onde são determinadas as características requeridas e desejáveis no produto final. Depois de analisados, os requisitos são traduzidos para a especificação e arquitetura do sistema, que guiarão o desenvolvimento durante todo o ciclo de projeto. A partir da especificação, constrói-se um modelo em alto nível de abstração que reflete toda a funcionalidade do sistema sem se preocupar, no entanto, com a implementação final em silício. Esse modelo é verificado e refinado gradativamente, até que se chegue ao nível do circuito físico. Depois de verificado, caso não sejam detectados defeitos durante o teste, o circuito físico será produzido em larga escala e entregue para os clientes.

Figura 7: Ciclo de Projeto de SoC

Para se utilizar a metodologia de refinamentos sucessivos, é preciso que sejam definidos níveis de abstração utilizados no projeto do sistema. Cada nível de abstração determina quais características devem estar presentes no modelo do sistema durante uma etapa do projeto. Não existe um consenso em relação aos níveis de abstração possíveis em um ciclo de projeto de SoC, sendo que cada autor define os níveis e as características pertinentes a cada um deles. A figura 8 apresenta uma divisão entre os níveis de abstração proposta por (Dias, 2007). Nela, existem 4 níveis de abstração, dois onde a divisão entre *hardware* e *software* não existe e dois outros níveis onde essa divisão está bem caracterizada. O nível RTL é o último tratado no trabalho de (Dias, 2007), pois este é o último nível suportado pelo SystemC, linguagem utilizada no trabalho.

Figura 8: Níveis de abstração e velocidade de simulação

A *Especificação Executável* é o mais alto nível proposto por (Dias, 2007), e traduz diretamente os requisitos do consumidor para um modelo unicamente funcional. Nesse modelo, não existe noção de temporização nem preocupação com a implementação final. Além de ser uma prova de conceito do produto, a especificação executável também é utilizada como referência para o comportamento que deve ser apresentado pelos demais níveis de abstração.

Abaixo da *Especificação Executável*, está o nível de *Data-flow Temporizado*. Nesse nível, inicia-se a modelagem dos atrasos para refletir as restrições de tempo da especificação, tempos de processamento e latências da implementação alvo. A partição entre *hardware* e *software* começa a ser analisada e arquiteturas são testadas para verificar o desempenho do sistema. Estruturas de dados e protocolos de comunicação ainda não são

definidos nesse estágio.

Uma vez que a divisão entre *hardware* e *software* esteja definida, começa-se a refinar o modelo de *hardware*. No nível *comportamental* já existe precisão de pinos e ciclos, mas as estruturas internas ainda não são as definitivas. As interações entre os diferentes módulos do sistema já podem ser verificadas e a velocidade das simulações ainda é maior do que no nível RTL. Algumas ferramentas começam a ser utilizadas para síntese do circuito físico a partir desse nível.

O último nível da figura 8 é o RTL. Neste nível, o funcionamento do sistema é descrito por meio de máquinas de estado finito e caminhos de dados. A descrição já possui precisão de pinos e *clock* e a estrutura do sistema reflete precisamente os registradores e lógicas combinatórias da implementação alvo. Apesar disso, esse nível ainda é independente da tecnologia de fabricação de circuitos integrados. Ferramentas para síntese a partir desse nível já estão bem consolidadas. Estas ferramentas transformam a descrição de um sistema em um determinado nível de abstração para um nível de abstração mais baixo, geralmente gerando o circuito físico a partir do nível RTL. Depois de obtido o modelo RTL, o circuito físico pode ser criado a partir de ferramentas de síntese ou de *layout* personalizado.

A eficiência do ciclo de projeto de SoCs pode ser medida por meio de três parâmetros: o tempo para atingir o mercado, também chamado de *Time-to-Market*; o custo total, desde o início do projeto até a fabricação do produto final; e a qualidade obtida no produto.

Mantendo-se os custos de desenvolvimento e produção os menores possíveis, maior será o lucro obtido com as vendas. Um custo menor também significa um preço de venda menor, o que pode garantir uma vantagem em relação aos concorrentes. Além do custo, o *Time-to-Market* é uma característica fundamental na indústria de microeletrônica. O sucesso de um produto de base tecnológica depende deste atingir o mercado no tempo certo, que geralmente significa o quanto antes. Quanto menor for o tempo entre a identificação da demanda de um determinado produto até o momento em que este está disponível para o consumidor, maior será a chance de uma empresa garantir uma fatia do mercado.

Somente atender à demanda em um tempo curto não significa que o produto de uma

empresa obtenha sucesso no mercado. A satisfação dos consumidores também ditará o futuro das vendas. A qualidade é um aspecto que define não somente se um produto atendeu a determinados padrões estabelecidos por organismos de padronização, mas também se ele cumpre, de maneira eficiente, o que ele se propõe a fazer. Um produto de qualidade duvidosa no mercado pode criar uma imagem negativa da empresa que, dificilmente, irá se recuperar do problema.

A dificuldade do trinômio Custo-Tempo-Qualidade está no fato de que a otimização de duas dessas grandezas geralmente afeta a terceira de forma negativa. Aumentando-se a qualidade e diminuindo-se o custo, aumenta-se o tempo de projeto. Aumentando-se a qualidade e diminuindo-se o tempo, o produto final terá um custo maior. Diminuindo-se o custo e o tempo de projeto, o produto final obtido terá uma qualidade menor. A verificação é a principal maneira de atingir significativamente todos estes três aspectos (Wile et al., 2005). Na indústria de microeletrônica, onde os custos não reembolsáveis (*NRE*) necessários para a produção dos produtos são muito altos, a verificação se torna uma etapa muito importante.

2.4 Verificação em Projetos de SoCs

Hoje, a verificação consome 70% dos recursos do projeto de SoCs (Bergeron, 2003), tornando-se uma parte crítica desse ciclo. Na seção 2.3, foi afirmado que a verificação afeta de forma positiva todos os itens do trinômio Custo-Tempo-Qualidade, sendo uma etapa fundamental no desenvolvimento de um SoC. A forma como a qualidade é promovida pela verificação é clara. Quando a verificação de um sistema é feita de forma precisa, as chances de que um requisito do consumidor não seja atendido, ou seja atendido de forma errada, são menores. No entanto, o modo como as outras duas grandezas do trinômio são atingidas é menos direto.

A figura 9, traduzida de (Wile et al., 2005), mostra o crescimento qualitativo do custo em relação ao momento onde uma falha é encontrada. Uma falha encontrada durante a verificação tem um custo pequeno para ser reparada. Se essa mesma falha for encontrada durante o teste do sistema após a fabricação do *chip*, esse custo aumenta, sendo neces-

sário, além de corrigir o problema, refabricar o *chip*. Refabricar o *chip* pode envolver o alto custo da criação de novas máscaras para o processo de fundição, aumentando o NRE do projeto. Se essa falha não for detectada durante o teste do sistema, atingindo o consumidor, o custo se torna mais alto ainda, podendo ser necessário recolher os produtos que já foram vendidos. Portanto, quanto mais cedo os erros de implementação forem identificados, menor o custo para sua reparação.

Figura 9: Custo da correção do erro X Momento da detecção

A verificação antecipada também ajuda na redução do tempo de projeto, pois falhas simples de serem corrigidas em níveis mais altos de abstração podem ser propagadas para níveis mais baixos, onde sua correção se torna mais complexa. Com uma verificação eficiente, o retrabalho no ciclo de projeto será reduzido, o que acarretará em um tempo de projeto menor. A figura 10 mostra como o aumento da eficiência na verificação leva a uma redução no tempo de projeto. Uma verificação ineficiente faz com que erros de implementação continuem a ser detectados durante o teste do sistema. Uma equipe de verificação com alta eficiência consegue detectar uma maior quantidade de erros durante as fases iniciais do projeto.

Além do momento onde os erros são detectados, a eficiência da equipe de verificação também pode ser medida em função da qualidade dos erros encontrados. Erros fáceis de serem detectados devem ser encontrados cedo no ciclo de verificação. Encontrar erros simples tardiamente significa uma verificação ineficiente. Detectar os erros simples an-

Figura 10: Aumento da Eficiência na Verificação

tecipadamente, permite que os esforços sejam direcionados para criar conjuntos de testes mais elaborados, capazes de detectar os erros mais complexos.

Para uma verificação eficiente, é necessária a utilização de um ciclo de verificação consistente e um ambiente de verificação completo e robusto. Esses dois itens são abordados a seguir. Primeiro, será apresentado o ciclo de verificação geralmente empregado no projeto de SoCs. Em seguida, os níveis de observação possíveis em sistemas sob verificação serão apresentados. Os tipos de verificação serão abordados na seção 2.4.3. Na seção 2.4.4, serão descritos os diferentes níveis de verificação possíveis em um projeto. Finalmente, alguns critérios para controle da qualidade da verificação serão apresentados.

2.4.1 Ciclo de Verificação

Para que a verificação do sistema seja efetuada de maneira consistente, é preciso que seja utilizado um ciclo de verificação bem estruturado. A figura 11 foi adaptada de (Wile et al., 2005) e mostra as etapas do ciclo de verificação. O ciclo é percorrido no sentido horário, começando da *Especificação Funcional*.

A *Especificação Funcional* descreve as funcionalidades desejadas no sistema a ser criado e, por esse motivo, é a base para a criação do plano de verificação. Na figura 11, pode-se verificar que existe uma redundância entre a implementação HDL do sistema durante o *Ciclo de Projeto* e a criação do *Plano de Verificação*, pois os dois itens utilizam

Figura 11: Ciclo de Verificação

a *Especificação Funcional* como insumo. Essa redundância é importante para reduzir a possibilidade de interpretações erradas da *Especificação Funcional*.

O *Plano de Verificação* é o documento que apresenta uma descrição detalhada do processo de verificação, indicando o que deve ser verificado e como deve ser verificado. Durante a criação do plano, os projetistas e arquitetos do sistema são consultados para que um ambiente de verificação robusto seja elaborado. Os níveis de observação e os tipos e níveis de verificação a serem empregados no ambiente também são determinados. Um *Plano de Verificação* geralmente:

- Descreve um cronograma das atividades a serem realizadas;
- Especifica os métodos e os testes que serão utilizados;
- Indica uma lista de ferramentas e de recursos que serão empregados durante a verificação, como *software*, pessoas, etc.;
- Determina um critério de cobertura, que indica quando os testes aplicados foram satisfatórios.

Uma vez que o *Plano de Verificação* esteja pronto, começa-se a desenvolver o *Am-*

biente de Verificação, que consiste em um conjunto de ferramentas que permite a sua implementação. Esta é a maior etapa do ciclo de verificação, pois engloba a criação de ambientes de verificação formal, dos *testbenches* e a geração dos estímulos e modelos de referência para verificação baseada em simulação.

A próxima etapa é integrar o código HDL desenvolvido pela equipe de projeto com o *Ambiente de Verificação* para que o sistema seja depurado. Os testes determinados no *Plano de Verificação* são executados e os resultados obtidos são comparados aos esperados para se encontrar evidências de falhas no projeto ou no próprio *Ambiente de Verificação*. O erro então deve ser corrigido, tomando-se cuidado para que novos erros não sejam inseridos em partes já verificadas do sistema.

Sempre que ocorrer alguma alteração no código HDL ou no *Ambiente de Verificação*, devem ser realizados *Testes de Regressão*. O intuito desses testes é verificar se a alteração afetou outra parte do sistema que já havia sido verificada anteriormente. Além disso, uma vez que o *Ambiente de Verificação* geralmente utiliza aleatoriedade na geração dos casos de teste, sempre que um teste é executado, o sistema pode ser exercitado de um modo diferente, o que aumenta a cobertura dos possíveis erros.

Quando o critério de cobertura for atingido, o sistema pode ser liberado para a fabricação do circuito integrado. Uma vez fabricado, pode-se iniciar a *Depuração do Hardware*. Caso algum erro seja encontrado no sistema fabricado, é preciso identificar se esse erro é devido ao processo de fabricação ou é um erro de projeto. Idealmente, nenhum erro de projeto deve ser encontrado durante essa etapa pois qualquer correção do projeto de *hardware* irá envolver os altos custos da refabricação das máscaras.

Se algum erro de projeto for encontrado durante a *Depuração do Hardware Fabricado*, é preciso realizar uma análise do *Plano de Verificação* para que seja encontrado o motivo que levou a não identificação desse erro durante as etapas de verificação, antes da fabricação do *hardware*. Quando possível, o erro deve ser simulado no *Ambiente de Verificação* para que se garanta que ele foi devidamente entendido para evitar que erros semelhantes ocorram em verificações futuras.

2.4.2 Níveis de Observação

Quanto ao nível de observação interna do sistema, a verificação pode ser classificada em *caixa-preta* (*Black-Box*), *caixa-branca* (*White-Box* ou *Glass-Box*), ou *caixa-cinza* (*Grey-Box*) (Wile et al., 2005).

Grande parte dos ambientes de simulação de SoCs inicia-se numa abordagem do tipo *caixa-preta*. Nesse tipo de abordagem, a estrutura interna do sistema é ignorada e o teste é realizado baseando-se na especificação de requisitos. Um conjunto de vetores de teste é escolhido entre todos os possíveis valores de entrada do sistema. Os vetores são aplicados e a resposta do sistema é comparada ao valor esperado. Esse tipo de verificação, portanto, necessita de um oráculo, ou seja, um método que pode determinar se a saída produzida é correta ou não (Rapps e Weyuker, 1985).

Existem duas principais vantagens de se utilizar observação do tipo *caixa-preta*. A primeira é que os casos de teste não serão dirigidos pela estrutura interna do sistema, permanecendo o mais independentes possível da implementação. Isso cria uma redundância no ciclo de projeto, evidenciando possíveis falhas de especificação ou interpretação dos requisitos do sistema. A segunda vantagem é que, com uma verificação independente da estrutura interna, mudanças na implementação resultam em pouco impacto no ambiente de verificação, que pode ser reutilizado durante quase todo o projeto.

A principal desvantagem na utilização de observação *caixa-preta* é que os erros de implementação só serão identificados por meio dos sintomas apresentados na saída do sistema. Localizar esses erros na estrutura interna torna-se uma tarefa a parte, dificultada pela falta de observabilidade. Além disso, sem considerar a implementação durante a realização dos testes, algumas partes do sistema podem esconder erros por não terem sido exercitadas, ou apenas exercitadas de um modo ineficiente.

O oposto da abordagem *caixa-preta* é a *caixa-branca*, onde todas as informações da estrutura interna do sistema estão disponíveis durante a verificação. À medida que os vetores de teste são aplicados, o fluxo de execução é seguido e os resultados de cada parte do sistema podem ser verificados individualmente. A utilização dessa abordagem garante que informações sobre partes do sistema não exercitadas estarão disponíveis e

a alta observabilidade permite que os erros sejam identificados em sua fonte. Mais do que isso, durante a verificação pode-se criar vetores de teste que exercitem partes do sistema de um modo mais eficiente. A abordagem *caixa-branca* pode empregar modelos de estruturas internas para verificar o funcionamento correto das mesmas, além da utilizar asserções para monitorar casos inválidos, como o estouro de uma pilha.

Uma desvantagem da utilização da abordagem *caixa-branca* é que o ambiente de verificação fica dependente da implementação, requerendo que seja modificado sempre que a implementação seja alterada. A principal desvantagem, no entanto, é que os testes podem se tornar polarizados pela estrutura interna do sistema. A ocorrência disso pode levar ao extremo onde os casos de testes são criados apenas para aumentar a cobertura da verificação, escondendo erros possíveis de projeto e quebrando a redundância entre a implementação e a verificação.

A maioria dos ambientes de verificação termina como uma *caixa-cinza*. Trata-se de uma combinação entre as abordagens anteriores, onde apenas alguns sinais específicos são observados enquanto o resto permanece oculto. A vantagem da utilização dessa abordagem é que o ambiente de verificação mantém certa independência da implementação, enquanto partes críticas podem ser verificadas de um modo mais profundo.

2.4.3 Tipos de Verificação

Nesse ponto é necessário definir a diferença entre verificação e teste. O teste de um sistema é um experimento no qual o sistema é exercitado e sua resposta é analisada para verificar se este comportou-se corretamente (Abramovici et al., 1990). Por outro lado, a verificação de um sistema é o processo cujo propósito é analisar, por meio de evidências objetivas, que as saídas de um sistema satisfazem aos requisitos impostos por suas entradas (Stanley, 2001) e, portanto, engloba o conceito de teste. Existem três tipos básicos de verificação presentes no projeto de SoCs: a verificação baseada em simulação, a verificação formal e o teste do sistema.

O objetivo da verificação formal é provar que algumas propriedades, como regras de protocolo ou projeto, se mantém verdadeiras em todos os casos possíveis. O grande problema com a verificação formal é que o tamanho dos sistemas nos quais ela pode ser

aplicada é limitado. Como todos os estados possíveis são verificados, a verificação formal pode requerer grandes recursos computacionais, mesmo para sistemas relativamente pequenos. Por esse motivo, a verificação formal geralmente é aplicada em partes críticas do sistema e não no sistema como um todo, sendo um grande complemento para a verificação baseada em simulação (Wile et al., 2005). A verificação formal não será tratada nesse trabalho.

Por teste do sistema, entende-se o teste realizado após a fabricação do circuito integrado. O principal objetivo da verificação é evitar que erros sejam encontrados depois que o *hardware* tenha sido fabricado, uma vez que a correção de erros nessa fase é muito cara. Depurar o *hardware* diretamente é muito mais difícil do que durante a utilização do ambiente de verificação, pois a observabilidade do *hardware* é menor do que a do ambiente de verificação. Além disso, caso algum erro seja encontrado no sistema fabricado, é preciso identificar se esse erro é devido ao processo de fabricação ou é um erro de projeto. Se for encontrado um erro de projeto, deve-se corrigi-lo, por meio de modificação do micro-código do sistema (quando for o caso) ou por meio da refabricação do sistema. Os testes do sistema também não serão tratados nesse trabalho.

O foco desse trabalho é a verificação baseada em simulação, que é o tipo mais utilizado na indústria de SoC. Nela, *estímulos* são aplicados a um modelo do projeto, geralmente descrito em uma linguagem de descrição de *hardware*, e um *verificador* confere se a resposta obtida foi correta. A utilização desses modelos com nível mais elevado de abstração agiliza o ciclo de projeto, pois quanto maior o nível de abstração, menor o detalhamento da solução e, conseqüentemente, mais rápida é a simulação do modelo. O aumento na velocidade da simulação permite a exploração de arquiteturas e também, que a verificação comece mais cedo no ciclo de projeto. Para a utilização da verificação baseada em simulação, é necessário que seja construído, ou obtido de alguma maneira, um oráculo. O oráculo é um método que determina se uma saída produzida pelo sistema está correta ou não (Rapps e Weyuker, 1985) e, geralmente, é chamado de *modelo de referência*.

2.4.3.1 *Verificação Baseada em Simulação*

Uma forma típica de emprego da verificação baseada em simulação é o *testbench*, onde o gerador de *estímulos*, o *verificador*, o *modelo de referência* e o *sistema sob verificação* são agrupados como mostrado na figura 12.

Figura 12: Forma típica de um *Testbench*

Um estímulo gerado é entregue tanto para o *modelo de referência* quanto para o *sistema sob verificação*. A respostas desses dois modelos são, então, comparadas pelo *verificador*, determinando-se se o *sistema sob verificação* comportou-se conforme esperado.

Além dos componentes já descritos, um *testbench* como o da figura 12, também possui um *driver* e um *monitor*. Esses dois componentes são responsáveis por manter a implementação do *testbench* o mais independente possível do nível de abstração empregado na descrição do *sistema sob verificação*. À medida que o ciclo de projeto evolui, o *sistema sob verificação* é refinado para níveis de abstração mais baixos, onde sinais de controle são criados e as entradas de dados são modificadas para atingir precisão de pinos. O *driver* faz a tradução entre os tipos de dados gerados pelo gerador de *estímulos* e os esperados pelo *sistema sob verificação* enquanto o *monitor* traduz as respostas do *sistema sob verificação* para padrões compreendidos pelo *verificador*.

2.4.4 Níveis de Verificação

É comum que um sistema seja dividido em partes menores para facilitar o desenvolvimento durante o ciclo de projeto. Da mesma forma, a verificação de um sistema pode ser fracionada, caminhando gradativamente das partes para o todo. A divisão da verificação em níveis traz diversas vantagens:

1. A identificação do erro por meio de seu sintoma é mais fácil em pequenos módulos ou em conjuntos pequenos de módulos do que no sistema inteiro;

2. A criação de estímulos para os módulos de maneira individual é mais fácil, pois exige um conjunto de vetores de entrada menor do que o do sistema como um todo;

3. A verificação pode começar mais cedo, antes de todos os módulos do sistema serem desenvolvidos;

4. A verificação do sistema completo é facilitada, pois os módulos já foram verificados individualmente e, dessa forma, grande parte da verificação está relacionada a transações entre unidades, como ordem de execução de processos, disparo de eventos, etc. (Gawanmeh et al., 2004).

Do mesmo modo que nos níveis de abstração, não existe um consenso para definir os níveis de verificação, mesmo porque projetos diferentes requerem um conjunto de níveis diferente. Alguns níveis, apresentados em (Wile et al., 2005), são descritos a seguir:

- **Projetista** - É o nível mais baixo de verificação e é realizado pelo projetista, compreendendo funções específicas e pequenas porções de código. Geralmente trata-se de verificar se uma parte do código pode ser carregada e simulada pelo ambiente de verificação. As interfaces nesse nível são muito dinâmicas, por não existir uma especificação formal de como elas devem interagir com o mundo exterior e, por esse motivo, existe pouca automação do processo de verificação. Em partes críticas do projeto, tende-se a utilizar verificação formal, uma vez que o conjunto de estados neste nível é menor do que no sistema como um todo;

- **Unidade** - Em grandes sistemas, costuma-se empregar o conceito de testes de unidade. Porções de código desenvolvidos são agrupadas em unidades com especificação de interface detalhada. Como a interface é mais estável do que no nível do projetista, pode-se criar ferramentas automáticas para geração de teste, principalmente testes baseados em geração aleatória de estímulos;

- **Core** - Um *core* é um módulo que é reutilizado várias vezes em diversos projetos. Nesse caso, ele possui uma documentação, contendo sua especificação funcional e detalhamento de suas interfaces. A vantagem de se utilizar um *core* é que, uma vez verificado em um projeto, ele não precisa ser verificado se utilizado em outro. Para que isso ocorra, no entanto, é preciso que o *core* seja bem verificado durante o seu primeiro uso, pois ele pode ser utilizado de um modo totalmente diferente nos projetos seguintes. O teste de um *core* deve ser feito cuidadosamente, tentando prever suas possíveis futuras utilizações;

- ***Sistema*** - Um sistema é composto por um conjunto de unidades. Neste nível, já se considera que existam interfaces muito bem definidas. O propósito da verificação nesse nível é determinar se as conexões e as transações entre as unidades foram definidas de maneira correta;

- **Co-verificação de *Hardware/Software*** - É o nível onde a interação entre os componentes de *hardware* e *software* é verificada. O *software* é o código que é executado utilizando-se a plataforma de *hardware* como base, e compreende os *drivers* de dispositivo, os códigos de inicialização do sistema, micro-códigos e aplicativos. O principal objetivo desse nível de verificação é encontrar erros devido a interpretações diferentes das especificações do projeto.

A ferramenta de análise de cobertura, que será apresentada no capítulo 5, pode ser empregada em qualquer um dos níveis de verificação descritos, desde que o sistema tenha sido modelado utilizando-se o SystemC. O método para geração automática de vetores de teste para sistemas combinatórios, que será descrito no capítulo 4, teoricamente também

pode ser utilizado em qualquer nível, porém, sua limitação de ser aplicável somente em sistemas combinatórios o restringe, na prática, aos níveis *Projetista*, *Unidade* e *Core*.

2.4.5 Análise de Cobertura

Para garantir o funcionamento correto de um sistema por meio de simulação, é necessário que este responda corretamente a todos os elementos de seu conjunto de entradas possíveis. No entanto, aplicar todos os elementos do conjunto de entradas possíveis de um sistema é uma tarefa que só pode ser realizada para sistemas muito simples, sendo que apenas uma fração desse conjunto é utilizada durante os testes. A cobertura de um teste é um valor que indica qual fração de um determinado critério foi satisfeita e é utilizada para determinar quando os testes atingiram um resultado satisfatório.

Existem dois tipos de modelos de cobertura: funcionais e estruturais. Os modelos funcionais se concentram na semântica do sistema, tentando determinar quanto da sua funcionalidade foi exercitada pelo ambiente de verificação (Singh et al., 2004) e, normalmente, empregam observação do tipo *caixa-preta*. Apesar de largamente utilizada na indústria de SoCs, a cobertura funcional não é o foco deste trabalho, pois não há como criar, de modo automático, modelos para cobertura funcional (Wile et al., 2005), por esse motivo seus modelos não serão tratados aqui.

Os modelos estruturais, por sua vez, verificam aspectos da implementação do sistema que foram cobertos pela verificação como, por exemplo, linhas de código exercitadas pelos casos de teste. A utilização de modelos estruturais é bem difundida na indústria de *software* devido a sua simplicidade e ao grande número de ferramentas disponíveis. O maior problema dos modelos estruturais está relacionado ao fato de que eles são inteiramente dependentes da estrutura de controle do sistema sob verificação. Um problema óbvio é quando a própria estrutura de controle deste sistema está errada (Ntafos, 1988), como nos Erros-de-Omissão (*Faults-of-Omission*), quando alguma característica necessária ao funcionamento correto do dispositivo não é implementada no código.

Apesar desse problema, a intuição sugere que existe uma relação entre cobertura estrutural e confiabilidade do sistema. Essa intuição é confirmada em (Malaiya et al., 2002), onde é proposto um modelo para prever a confiabilidade a partir de medidas de cobertura

de código. Com o aumento do nível de abstração na descrição de SoCs e uma vez que a criação automática de modelos de cobertura estrutural é possível, esta abordagem será utilizada no decorrer deste trabalho. Cobertura de código e cobertura de fluxo-de-dados serão mais detalhados nos itens seguintes.

2.4.5.1 Cobertura de Código

Análise de cobertura de código é uma técnica de verificação estrutural cujos resultados podem ser utilizados como uma medida da qualidade do conjunto de testes aplicados, indicando partes do código não exercitadas (ou exercitadas de modo insatisfatório) pelos testes. Pode-se determinar quão completo foi o teste por meio da análise da fração de unidades de fluxo, tais como ramos, cobertas por ele.

Muitos trabalhos (Lawrance et al., 2005; Yin et al., 1997; Malaiya et al., 1994) definem métricas para cobertura de código. Neste trabalho, são utilizados os conceitos de cobertura de enunciados (*statement coverage*), cobertura de funções (*function coverage*), cobertura de ramos (*branch coverage*), cobertura de termos (*condition coverage*), cobertura exaustiva de termos (*multi-condition coverage*), cobertura relacional (*relational coverage*), cobertura de laços (*loop coverage*), e cobertura de caminhos (*path coverage*).

Para a definição dos critérios de cobertura, considere a equação 5, que representa uma forma geral de uma condição de um desvio condicional.

$$\Phi = \mu_1 \; \Psi_1 \; \mu_2 \; \Psi_2 \; ... \; \Psi_{n-1} \; \mu_n, \quad n > 0 \tag{5}$$

Uma expressão lógica, Φ, de um desvio condicional pode ser considerada como um conjunto de termos booleanos, μ, agrupados e submetidos aos operadores lógicos Ψ.

> **Definição 20.** Um operador lógico, Ψ, é definido com um operador que recebe dois operandos booleanos e computa um resultado também booleano. Um operador Ψ não pode fornecer um resultado constante, que seja independente dos operandos (tautologia do operador).

A definição 20 determina que não existe operador lógico, Ψ, cujo resultado seja sempre verdadeiro (*TRUE*) nem operador Ψ cujo resultado seja sempre falso (*FALSE*) para qualquer valor de seus operandos. Os operadores *E* (*AND*) e *OU* (*OR*) respeitam a defi-

nição do operador lógico Ψ.

No restante desta seção, considera-se P um programa que é representado pelo grafo de fluxo $G_P(V_P, A_P)$ e B_P é o conjunto dos blocos básicos de P. O programa P é descrito em uma linguagem que admite o emprego dos operadores relacionais "<", "<=", ">" e ">=", que denotam as comparações de *menor*, *menor ou igual*, *maior* e *maior ou igual*, respectivamente, entre dois operandos quaisquer retornando um valor booleano.

As definições 21 a 28 formalizam os critérios de cobertura de código empregados neste trabalho.

Definição 21. Para **cobertura total de enunciados** é necessário que todos os blocos $b \in B_P$ sejam exercitados durante a verificação. Pela definição 6, isto equivale a dizer que todos os vértices $v \in V_P$ devem ser visitados durante a verificação.

Definição 22. Para **cobertura total de ramos** é necessário que todos os arcos $a \in A_P$ sejam percorridos durante a verificação.

Definição 23. Seja p o prefixo de um enunciado condicional de controle de fluxo de P, cuja expressão lógica da condição é Φ.
Para **cobertura total de termos** é necessário que cada um dos termos μ da expressão lógica Φ, apresentada na equação 5, assuma ao menos uma vez o valor verdadeiro (*TRUE*) e ao menos uma vez o valor falso (*FALSE*) durante a verificação para todo $p \in P$.

Definição 24. Seja p um prefixo de um enunciado condicional de controle de fluxo de P.
Para **cobertura exaustiva de termos** é necessário que todas as 2^n possíveis combinações para os valores dos termos μ da expressão lógica Φ, apresentada na equação 5, sejam exercitadas durante a verificação para todo $p \in P$.

Definição 25. Seja p o prefixo de um enunciado condicional de controle de fluxo de P cuja expressão lógica, Φ, é da forma apresentada na equação 5.
Para **cobertura relacional total** é necessário que cada um dos termos μ de Φ que empregue um operador relacional "<", "<=", ">" ou ">=" assuma ao menos uma vez o valor verdadeiro (*TRUE*), ao menos uma vez o valor falso (*FALSE*) e que seja atingida uma condição de igualdade entre seus operandos durante a verificação, para todo $p \in P$.

Definição 26. Seja C_P o conjunto de todos os caminhos de G_P.
Para **cobertura total de caminhos** é necessário que todos os caminhos $c \in C_P$ sejam exercitados durante a verificação.

Definição 27. Seja F_P o conjunto de todas as funções de P.
Para **cobertura total de funções** é necessário que todas as funções $f \in F_P$ sejam exercitadas durante a verificação.

Definição 28. Seja L_P o conjunto de todos os laços de P.
Para **cobertura total de laços** é necessário que todos os laços $l \in L_P$ sejam exercitados durante a verificação.

Um conjunto de caminhos C de G_P pode ou não satisfazer um critério de cobertura q.

Definição 29. Seja C um conjunto de caminhos do grafo de fluxo G_P que representa o programa P e c um caminho de C.
O conjunto C **satisfaz** o critério q se, depois de percorridos todos os caminhos c de C, os enumeráveis do programa P foram exercitados de forma a atender a definição de q.

Uma métrica de cobertura indica a fração de um determinado critério que foi coberta durante a verificação de um sistema.

Definição 30. Seja C um conjunto de caminhos do grafo de fluxo G_P que representa o programa P e c um caminho de C.
A **métrica** s_q denota a fração dos enumeráveis do programa P necessários para que um critério q seja satisfeito, que foi exercitada depois que todos os caminhos c de C foram percorridos.

Verifica-se que existe uma relação entre os critérios de cobertura. Um critério de cobertura q_1 *contém* um critério q_2, ou $q_1 \supseteq q_2$, se qualquer conjunto de caminhos C que satisfaça q_1 também satisfizer q_2.

Definição 31. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa o programa P. Sejam q_1 e q_2 dois critérios de cobertura e C o conjunto de todos os caminhos de G_D.
Um critério $\mathbf{q_1} \supseteq \mathbf{q_2} \Leftrightarrow \forall\, C_q \subset C \mid C_q$ satisfaz q_1, C_q também satisfaz q_2.

Um critério q_1 *inclui* um critério q_2, ou $q_1 \supset q_2$, se $q_1 \supseteq q_2$ e existir um conjunto de caminhos C que satisfaça q_2 mas não q_1.

Definição 32. Seja $G_D(V_D, A_D)$ um grafo de definição e uso que representa um programa P. Sejam q_1 e q_2 dois critérios de cobertura e C o conjunto de todos os caminhos de G_D.
Um critério $\mathbf{q_1} \supset \mathbf{q_2} \Leftrightarrow q_1 \supseteq q_2$ e $\exists\, C_q \subset C \mid C_q$ satisfaz q_2, mas não satisfaz q_1.

Definição 33. Dois critérios q_1 e q_2 são **incomparáveis** se nem $q_1 \supset q_2$ nem $q_2 \supset q_1$.

A figura 13 mostra a relação entre os critérios de cobertura apresentados. Para a construção da figura considerou-se que todos os enunciados do programa são *virtualmente* alcançáveis. A palavra *virtualmente* foi empregada no sentido de que esse enunciado pode ser isolado por um desvio condicional com uma condição impossível de ser atingida, mas não devem existir desvios incondicionais que isolem um determinado enunciado.

Figura 13: Relação entre as Coberturas de Código

Na figura 13, a relação $q_1 \supset q_2$ é representada como $q_1 \Rightarrow q_2$. Pode-se verificar que a *cobertura total de ramos* inclui a *cobertura total de enunciados*, ou seja, caso haja cobertura completa de ramos haverá cobertura completa de enunciados. A *cobertura total de enunciados* é conhecida por ser um critério fraco de cobertura (Rapps e Weyuker, 1985). Apesar disso, é um critério muito utilizado por ser simples de ser implementado. O critério mais forte é o de *cobertura total de caminhos*, que inclui todos os outros. Porém o número de caminhos em um programa cresce exponencialmente com cada laço ou ramo, tornando-se impraticável a sua utilização em qualquer programa que não seja extremamente simples (Lawrance et al., 2005).

A relação entre *cobertura total de enunciados* e *cobertura total de ramos* é verificada utilizando-se o conceito de grafos de fluxo, apresentado na definição 6. A *cobertura total de enunciados* requer que cada um dos vértices do grafo de fluxo seja visitado, enquanto a *cobertura total de ramos* requer que todos os arcos sejam percorridos. Visitar todos os vértices não garante que todos os arcos sejam percorridos, pois mais de um arco pode ter como *cabeça* o mesmo vértice. No entanto, percorrer todos os arcos do grafo faz com que todos os vértices sejam percorridos, uma vez que determinou-se que todos os enunciados do programa são *virtualmente* alcançáveis.

A relação entre *cobertura total de ramos*, *cobertura total de termos* e *cobertura exaustiva de termos* será provada a seguir. Cada um dos ramos em um programa é criado por um desvio condicional ou um desvio incondicional. Levando-se em conta a restrição de que todos os enunciados do programa devem ser *virtualmente* alcançáveis, os desvios incondicionais não são importantes para a prova, pois eles não podem isolar nenhuma parte do código. Assim, nossa análise se restringe aos desvios condicionais.

Pela equação 5, uma expressão lógica, Φ, de um enunciado condicional de controle de fluxo pode ser considerada como um conjunto de termos booleanos μ agrupados e submetidos a operadores lógicos Ψ. Considerando-se o conceito de *cobertura total de ramos*, apresentado na definição 22, verifica-se que os dois ramos de um desvio condicional são cobertos se sua expressão lógica, Φ, assumir os valores verdadeiro (*TRUE*) e falso (*FALSE*). A partir disto, pode-se verificar que a *cobertura total de ramos* e a *cobertura total de termos* são critérios incomparáveis por dois contra exemplos.

Lema 1. O critério *cobertura total de ramos* não *inclui* o critério *cobertura total de termos*.

Prova por contra exemplo 1. Seja q_1 o critério de *cobertura total de ramos* e q_2 o critério *cobertura total de termos*. Provando-se que a relação $q_1 \supseteq q_2$ é falsa, prova-se que q_1 não inclui um critério q_2, de acordo com a definição 32. O retângulo cinza, em evidência na figura 14, mostra uma situação onde ocorre *cobertura total de termos* mas não ocorre

cobertura total de ramos. Portanto, a relação $q_1 \supseteq q_2$ é falsa e q_1 não *inclui* q_2.

Φ	μ_2	μ_1
FALSE	FALSE	FALSE
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE

Figura 14: Contra Exemplo 1

Lema 2. O critério *cobertura total de termos* não *inclui* o critério *cobertura total de ramos*.

Prova por contra exemplo 2. Seja q_1 o critério de *cobertura total de termos* e q_2 o critério *cobertura total de ramos*. Provando-se que a relação $q_1 \supseteq q_2$ é falsa, prova-se que q_1 não inclui um critério q_2, de acordo com a definição 32. O retângulo cinza, em evidência na figura 15, mostra uma situação onde ocorre *cobertura total de ramos* mas não ocorre *cobertura total de termos*, pois o termo μ_2 não assumiu o valor falso (*FALSE*). Portanto, a relação $q_1 \supseteq q_2$ é falsa e q_1 não *inclui* q_2.

Φ	μ_2	μ_1
FALSE	FALSE	FALSE
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE

Figura 15: Contra Exemplo 2

Lema 3. O critério *cobertura total de termos* e o critério *cobertura total de ramos* são incomparáveis.

Prova 3. Pela definição 33, se nem $q_1 \supset q_2$, nem $q_2 \supset q_1$, os critérios q_1 e q_2 são incomparáveis. Seja q_1 o critério de *cobertura total de ramos* e q_2 o critério *cobertura total de termos*. A prova 1 mostra que a relação $q_1 \supset q_2$ é falsa e a prova 2 mostra que a relação $q_2 \supset q_1$ é falsa. Portanto, q_1 e q_2 são critérios incomparáveis.

Pela figura 13, pode-se verificar que *cobertura exaustiva de termos* inclui *cobertura total de ramos* e também *cobertura total de termos*.

Lema 4. *Cobertura exaustiva de termos* contém o critério *cobertura total de termos*.

Prova 4. Considerando a equação 5, o critério *cobertura exaustiva de termos* requer que todas as 2^n combinações possíveis para os termos μ sejam avaliadas durante o processo de teste. Se todas as combinações forem avaliadas, automaticamente o critério *cobertura total de termos* será satisfeito, pois cada um dos termos terá assumido individualmente os valores verdadeiro (*TRUE*) e falso (*FALSE*).

Lema 5. O critério *cobertura exaustiva de termos* contém o critério *cobertura total ramos*.

Prova 5. Considere as equações 6 e 7, apresentadas a seguir.

$$\Phi^{1 \to m} = \mu_1 \; \Psi_1 \; \mu_2 \; \Psi_2 \; ... \; \Psi_{m-1} \; \mu_m, \quad m > 0 \tag{6}$$

$$\Phi^{m+1 \to n} = \mu_{m+1} \; \Psi_{m+1} \; \mu_{m+2} \; \Psi_{m+2} \; ... \; \Psi_{n-1} \; \mu_n, \quad n \geq m \tag{7}$$

Pela definição 20, o operador Ψ é aplicado sobre operandos booleanos fornecendo um resultado booleano, sendo $\Phi^{1 \to m}$ e $\Phi^{m+1 \to n}$ valores booleanos. Empregando-se este conceito, pode-se combinar as equações 6 e 7 com a equação 5 obtendo-se a equação 8.

$$\Phi = \Phi^{1 \to m} \; \Psi_m \; \Phi^{m+1 \to n}, \; 0 < m \leq n \text{ e } n > 0 \tag{8}$$

As equações 6 e 7 podem ser aplicadas recursivamente às expressões $\Phi^{1 \to m}$ e $\Phi^{m+1 \to n}$, obtendo-se a equação 9

$$\Phi^{1\to n} = \Phi^{1\to m}\ \Psi_m\ \Phi^{m+1\to n},\ 0 < m \leq n \text{ e } n > 0 \qquad (9)$$

Empregando-se o conceito dos operadores lógicos, apresentado na definição 20, verifica-se que se todas as 4 combinações para os valores das expressões $\Phi^{1\to m}$ e $\Phi^{m\to n}$ forem aplicadas, $\Phi^{1\to n}$ assumirá, ao menos uma vez, o valor verdadeiro (*TRUE*) e ao menos uma vez o valor falso (*FALSE*).

Portanto, o critério *cobertura exaustiva de termos* contém o critério *cobertura total ramos*.

Lema 6. O critério *cobertura exaustiva de termos* inclui o critério *cobertura total ramos*.

Prova por contra exemplo 6. Seja q_1 o critério de *cobertura de termos* e q_2 o critério *cobertura de ramos*. Pela prova 5 foi visto que $q_1 \supseteq q_2$. De acordo com a definição 32, para que $q_1 \supset q_2$, basta um contra exemplo que satisfaça q_2 e não q_1. O retângulo cinza, em evidência na figura 16, mostra uma situação onde ocorre *cobertura total de ramos* mas não ocorre *cobertura exaustiva de termos*, pois o termo μ_2 não assumiu o valor falso (*FALSE*). Portanto, a relação $q_1 \supset q_2$ é verdadeira.

Φ	μ_2	μ_1
FALSE	FALSE	FALSE
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE

Figura 16: Contra Exemplo 3

Os critérios *cobertura total de laços*, *cobertura relacional total* e *cobertura exaustiva de termos* (e todos os critérios incluídos nesta última) são incomparáveis. Isso pode ser facilmente comprovado por meio de contra-exemplos e pela própria definição dos critérios e a prova não será oferecida neste trabalho.

2.4.5.2 *Cobertura de Fluxo de Dados*

Uma abordagem mais elaborada para determinar a cobertura da verificação é a utilização de técnicas baseadas em análise de fluxo de dados. Cada ocorrência de uma variável em um programa pode ser classificada como sendo uma *definição*, um *uso-computação* ou um *uso-prefixo* (Rapps e Weyuker, 1985). Classificando-se os tipos de uso das variáveis ao longo do programa, cria-se um grafo de definição e uso, que é descrito na seção 2.2.2, onde também são apresentados os conjuntos *def(i)*, *uso-c(i)*, *uso-p(i,j)*, *dcu(x,i)* e *dpu(x,i)*.

Apesar de uma coletânea de técnicas e trabalhos sobre análise de fluxo de dados ser apresentada em (Ntafos, 1988), neste trabalho, os critérios empregados são limitadas aos apresentados em (Rapps e Weyuker, 1985) por ser um trabalho considerado referência na área. Sendo G_D um grafo de definição e uso e C um conjunto de caminhos completos de G_D, determinam-se os seguintes critérios de cobertura de fluxo de dados:

- **Todos vértices** - Todos os vértices de G_D estão presentes em C. Este critério é equivalente a *cobertura total de enunciados* apresentada na seção 2.4.5.1;

- **Todos os arcos** - Todos os arcos de G_D estão presentes em C. Este critério é equivalente a *cobertura total de ramos* apresentado na seção 2.4.5.1;

- **Todos *defs*** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui um *caminho livre de definições da variável* x de v até algum elemento de $dcu(x, v)$ ou $dpu(x, v)$;

- **Todos *usos-p*** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui um *caminho livre de definições da variável* x de v até todos os elementos de $dpu(x, v)$;

- **Todos os *usos-c*/alguns *usos-p*** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui algum *caminho livre de definições da variável* x de v até todos os vértices presentes em $dcu(x, v)$. Se $dcu(x, v)$ for vazio, então C deve incluir algum *caminho livre de definições da variável* x de v até algum arco

presente em $dpu(x, v)$. Ou seja, todas as definições que foram usadas no sistema devem ser incluídas em algum caminho de C, sendo que os *usos-c* são priorizados;

- **Todos os *usos-p*/alguns *usos-c*** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui algum *caminho livre de definições da variável* x de v até todos os arcos presentes em $dpu(x, v)$. Se $dpu(x, v)$ for vazio, então C deve incluir algum *caminho livre de definições da variável* x de v até algum vértice presente em $dcu(x, v)$. Ou seja, todas as definições que foram usadas no sistema devem ser incluídas em algum caminho de C, sendo que os *usos-p* são priorizados;

- **Todos os usos** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui algum *caminho livre de definições da variável* x de v até todos os elementos de $dpu(x, v)$ e todos os elementos de $dcu(x, v)$;

- **Todos *caminhos-du*** - C satisfaz a esse critério se para cada vértice v de V_D e para todo $x \in def(v)$, C inclui todos os *caminhos-du* da variável x;

- **Todos os caminhos** - C satisfaz a esse critério se C incluir todos os caminhos de G_D.

A relação entre estes critérios, provada em (Rapps e Weyuker, 1985), é apresentada na figura 17.

2.5 Geração de Vetores de Teste baseada em Execução Simbólica

Execução simbólica é utilizada para determinar os resultados de um programa sem que ele seja executado e pode ser utilizada para testar de forma efetiva um programa (Coward, 1991). Um modo comum de se realizar uma execução simbólica é percorrer um caminho no grafo de fluxo que leva de um ponto de entrada até um ponto de saída do programa extraindo uma lista de atribuições e condições. Durante essa execução, cada variável de entrada recebe um símbolo, em vez de um valor, e cada atribuição é resolvida em termos desses símbolos.

Figura 17: Relação entre as Coberturas de Fluxo de Dados

No fim da execução simbólica, as variáveis de saída do programa serão representadas por meio dos símbolos das variáveis de entrada, e estarão sujeitas a um conjunto de restrições, determinadas pelos prefixos dos ramos pertencentes ao caminho executado simbolicamente.

A figura 18 mostra a execução simbólica, passo a passo, de um programa simples. A coluna *Condição do Caminho* mostra quais condições devem ser respeitadas para que o caminho seja executado. As demais colunas mostram o valor atribuído às variáveis nas linhas de código correspondentes. Nessa figura, o caminho que compreende os enunciados 0, 1, 2, 3, 5, 6 e 7 foi executado simbolicamente.

		Condição do Caminho	Valor X	Y	Z	O
0:	if X = Y then	-	a	b	c	d
1:	Z = y	a = b	a	b	b	d
2:	end if	a = b	a	b	b	d
3:	if X < Z then	a = b	a	b	b	d
4:	O = X - Z					
5:	else	a = b && a >= b	a	b	b	a + b
6:	O = X + Z	a = b && a >= b	a	b	b	a + b
7:	end if	a = b && a >= b	a	b	b	a + b

Figura 18: Execução simbólica para o caminho 0, 1, 2, 3, 5, 6 e 7.

A figura 19 mostra a execução de outro caminho do programa. O caminho executado,

que compreende os enunciados 0, 1, 2, 3, 4 e 7, é impossível de ser executado. Isso pode ser visto pela condição necessária para que o caminho seja executado: $a = b \ \&\& \ a < b$. Essa condição é claramente impossível de ser resolvida.

		Condição do Caminho	Valor X	Y	Z	O
0:	if X = Y then	-	a	b	c	d
1:	Z = y	a = b	a	b	b	d
2:	end if	a = b	a	b	b	d
3:	if X < Z then	a = b	a	b	b	d
4:	O = X - Z	a = b && a < b	a	b	b	a - b
5:	else					
6:	O = X + Z					
7:	end if	a = b && a < b	a	b	b	a - b

Figura 19: Execução simbólica para o caminho 0, 1, 2, 3, 4 e 7.

Uma vez executado simbolicamente, pode-se determinar vetores de testes que exercitam o caminho a partir das condições. Estas condições, expressas em termos das variáveis de entrada do programa, são organizadas de forma a criarem um sistema de inequações que, quando resolvido, terá como resultado um conjunto de vetores de teste que exercitam o caminho. Se a solução não for possível de ser encontrada, o caminho selecionado é impossível.

Apesar de poderosa, a execução simbólica é uma tarefa computacional complexa, pois utiliza símbolos no lugar de números. Além disso, torna-se difícil lidar com laços que dependem de valores de variáveis de entrada, chamadas de função e ponteiros. Por esses motivos, execução simbólica não é largamente utilizada.

2.6 Trabalhos Relacionados

Validar sistemas descritos em altos níveis de abstração por meio de simulação ou verificação formal é uma tarefa que se torna cada vez mais difícil, à medida que cresce a sua complexidade. Apesar da verificação formal ser uma técnica poderosa, a verificação baseada em simulação ainda será a base da indústria de microeletrônica em um futuro próximo (Wolfsthal e Gott, 2005). Por esse motivo, métodos que combinam verificação formal com verificação baseada em simulação tornam-se cada vez mais importantes.

Muitos métodos para verificação de projetos de SoCs foram baseados em metodo-

logias para teste de software. Gawanmeh et al. (Gawanmeh et al., 2004) propõem um método que combina análise estática de código e descrição da semântica do modelo SystemC por meio de Máquinas de Estado Abstratas (ASMs). O trabalho considera que grande parte dos SoCs são compostos por vários blocos IP, que já foram testados individualmente e afirma que, dessa forma, grande parte da verificação está relacionada a validar as transações entre blocos (ordem de execução de processos, disparo de eventos, etc.). Essas transações podem ser modeladas de maneira eficiente por meio de máquinas de estado. Em tese, qualquer sistema computacional pode ser descrito por meio de ASMs no seu nível de abstração natural, ou seja, sem modificações. O objetivo do trabalho de Gawanmeh et al. é modelar um sistema concreto descrito em SystemC por meio de ASMs que podem ser utilizadas para verificar algumas propriedades, preservando o comportamento do modelo SystemC original no que diz respeito à execução de processos e ativação de eventos. Uma vez compiladas, as ASMs podem ser utilizadas para verificação formal do sistema. A maior limitação desse método está relacionada ao fato que todos os canais utilizados no modelo SystemC devem ser modelados por meio de ASMs, bem como o núcleo de simulação da linguagem. O projetista deve, portanto, modelar quaisquer *custom channels* que por ventura ele tenha criado.

Outro trabalho (Patel e Shukla, 2007) utiliza ASMs descritas na linguagem AsmL para criar um modelo de referência para o desenvolvimento de projetos em SystemC, que pode ser utilizado tanto para a especificação quanto para a verificação do sistema. O modelo em AsmL é utilizado, juntamente com o Microsoft Spec Explorer (Microsoft, 2007), para gerar testes que levam à execução de determinados estados no projeto SystemC, possibilitando o teste de casos com baixa alcançabilidade. Como o Spec Explorer não possui suporte nativo ao SystemC, os autores criaram construções em C# que permitem controlabilidade e observabilidade do código. Nesse trabalho, não existe automação do processo de geração de código em SystemC a partir das máquinas de estado finito descritas em AsmL, nem mesmo a geração de código AsmL a partir do modelo SystemC. Portanto, uma mudança no ciclo de projeto é necessária, iniciando-se da descrição do sistema por meio de ASMs seguida pela tradução para um modelo SystemC.

O trabalho (Silva et al., 2004) propõe uma ferramenta para geração automática de *testbenches*. VeriSC gera *templates* para os módulos do *testbench* de acordo com as características do modelo em SystemC. O trabalho do engenheiro de verificação fica reduzido a definir os critérios de cobertura, as distribuições dos valores dos estímulos e o *handshaking* dos sinais envolvidos. Classes para a Biblioteca de Verificação do SystemC (SCV) (OSCI, 2007) são criadas automaticamente para gerar as entradas para o modelo sob verificação e para o modelo de referência, que pode estar descrito em qualquer linguagem executável de alto nível. As respostas do modelo sob verificação e do modelo de referência são comparadas de forma a determinar se o modelo se comporta corretamente.

Inúmeros trabalhos também foram publicados para a geração de vetores de teste *black-box* em projetos de *software*. Nesse tipo de teste, a estrutura interna do sistema a ser verificado é ignorada e o seu comportamento é comparado com uma especificação de requisitos. Um exemplo para esse tipo de trabalho foi apresentado em (Malaiya, 1995), onde é proposta a geração automática de testes usando *codificação de pontos de verificação e testes anti-randômicos*[2]. Vetores de teste parecidos exercitam o código de maneira similar. Uma seqüência anti-randômica é aquela na qual o teste t_i é escolhido para satisfazer alguma relação com os testes $t_0, t_1, ..., t_{i-1}$ aplicados anteriormente. Dessa forma, cada seqüência de teste é definida para ser o mais distante de outras seqüências, possuindo um maior potencial para aumentar a cobertura do teste. Para a utilização desse método, o projetista cria uma partição para cada uma das variáveis de entrada do sistema, dividindo em casos típicos, casos ilegais e casos de limite. Essas partições são codificadas em binário. Vetores de teste são gerados de acordo com essa partição, aplicando-se o método anti-randômico de modo que cada teste seja o máximo possível diferente de todos aplicados anteriormente. Esse tipo de geração de testes mostrou resultados melhores do que seqüências puramente aleatórias (Yin et al., 1997).

Uma abordagem estrutural para geração de testes para *software* é apresentada em (Clarke, 1976), onde um sistema para geração de vetores de teste para programas em ANSI Fortran é descrito. O sistema executa um caminho do programa simbolicamente de

[2]Versão para *checkpoint encoding and antirandom testing*

forma a criar um conjunto de restrições para os valores das variáveis de entrada. A partir dessas restrições, é possível criar os vetores de teste que levam a execução do caminho, caso o este seja executável. Essa abordagem permite uma grande controlabilidade do código, porém, executar um programa simbolicamente é uma tarefa complexa e que possui alguns problemas técnicos, como apresentado na seção 2.5.

Outra opção é tratar o problema de geração de vetores de teste como um problema de otimização numérica, como é mostrado em (Gallagher e Narasimhan, 1997). Esta abordagem não sofre das restrições relacionadas à execução simbólica. Nessa abordagem, o código fonte do programa é instrumentado e um caminho é forçado a executar enquanto um algoritmo de otimização é aplicado. Utilizando-se de técnicas de função de penalidade, pode-se extrair o vetor de teste que exercita o caminho forçado.

Recentemente, um trabalho introduz a utilização de técnicas de cobertura de código para medir a qualidade de um *testbench* SystemC (Große et al., 2007). A ferramenta desenvolvida cria um relatório que identifica as partes do código não cobertas pelo teste, informando sobre cobertura de ramos. Este é o primeiro trabalho que se tem notícia que apresenta uma ferramenta que trata a cobertura de testes em SystemC do ponto de vista estrutural, sendo o trabalho mais parecido com a ferramenta apresentada no capítulo 5. Porém, a ferramenta apresentada aqui permite o emprego de métricas de cobertura mais elaboradas, além de possuir uma estrutura mais flexível.

3 METODOLOGIA HÍBRIDA PARA VERIFICAÇÃO DE SOCS

No capítulo anterior, foi feita uma revisão sobre verificação em projetos de SoCs, descrevendo o ciclo de verificação, os níveis de observação, tipos e níveis de verificação que podem ser empregados durante o projeto de um sistema. Neste capítulo, serão apresentadas considerações sobre a utilização de métricas de análise de cobertura estrutural durante a verificação de SoCs, definindo-se uma metodologia híbrida para aumentar a eficiência da verificação.

Primeiro, será discorrido sobre como o emprego incorreto da análise de cobertura estrutural pode encobrir falhas no código fonte, levando a uma falsa impressão de uma verificação completa. A partir destas análises, uma metodologia de verificação, que combina técnicas funcionais e estruturais para melhorar a eficiência da verificação, será apresentada.

Durante todo esse capítulo, será considerado o modelo do *testbench* apresentado na figura 12. Também assume-se que existe um oráculo, um modelo de referência infalível, que determina qual seria a resposta correta do sistema quando submetido aos testes propostos.

3.1 Considerações Sobre o Emprego de Cobertura Estrutural na Verificação de SoCs

As técnicas de teste estrutural baseiam-se na idéia de que uma porção de código executada tem menos chances de conter erros do que uma porção de código nunca exercitada e utilizam a própria estrutura de controle do sistema como guia para a elaboração da

verificação. Os problemas relacionados ao emprego somente de técnicas estruturais na verificação de um sistema são bem conhecidos. O trabalho de (Marick, 1999) aponta muitos desses problemas, onde o uso incorreto da cobertura de código pode levar a uma ineficiência da verificação, quebrando a redundância entre a *Especificação Funcional* e o ciclo de verificação, descrito na seção 2.4.1.

O erro-de-omissão, muito comum em projetos de *software* (Glass, 1981), é um exemplo de erro indetectável por testes estruturais. Nele, uma característica do sistema não foi implementada, e não há evidências no código fonte que denunciem a presença da falha. Considere o seguinte trecho de código que apresenta um algoritmo recursivo para o cálculo do valor de *a* elevado a *b-ésima* potência:

```
1 double power(int a, int b)
2 {
3   if (b == 0)
4     return 1;
5   else
6     return a * power(a, b-1);
7 }
```

Por se tratar de uma operação matemática simples, possivelmente este trecho de código será exercitado apenas incidentalmente pelos casos de teste gerados para a verificação de partes mais críticas do sistema. No entanto, pode-se verificar que a função não trata corretamente valores negativos de *b*, que resultam em uma falha de segmentação, uma vez que o algoritmo é recursivo e será chamado indefinidamente. Ferramentas de cobertura estrutural não são capazes de ajudar na detecção deste tipo de erro, pois quaisquer valores para *a* e *b*, desde que *b* não seja negativo, retornarão um resultado correto para o *testbench* e resultarão em cobertura completa para os critérios estruturais. Erros-de-omissão, como este, serão detectados na verificação baseada em simulação somente por meio de uma geração adequada de valores aleatórios para a entrada do sistema, que resultem em um valor negativo de *b*.

Quando partes documentadas do código não são cobertas durante os testes, significa que o plano de teste não foi criado corretamente e deve ser revisto, contemplando todas as funcionalidades previstas. Dessa forma, evita-se um erro-de-omissão em uma funcio-

nalidade do sistema, que será descoberto pelo *testbench* quando os casos de teste gerados para verificar a funcionalidade forem aplicados.

No entanto, a principal falha na utilização de técnicas de cobertura estrutural está relacionada à mudança do foco da equipe de verificação, passando a perseguir metas de cobertura em vez de criar testes que cubram partes importantes do sistema. Um exemplo ocorre quando se define um valor de cobertura para que o sistema esteja pronto para ser enviado para a próxima etapa no ciclo de projeto, o que pode levar a duas ineficiências na verificação:

1. Quando a meta é atingida, a verificação para, mesmo que ainda existam partes do sistema que não foram suficientemente testadas;

2. Os engenheiros de teste priorizam casos mais fáceis de serem cobertos, tentando aumentar a cobertura, em vez de criarem casos de teste que exercitem partes críticas e com pouca controlabilidade, mais difíceis de serem elaborados.

Qualquer um desses dois vícios mostra uma mudança do foco da verificação do sistema para o aumento da cobertura, que resulta em uma verificação pobre e ineficiente.

3.2 Metodologia Híbrida para Verificação

Se, por um lado, um alto valor de cobertura não é garantia da funcionalidade do sistema, por outro, exercitar o código é um modo simples de reduzir a possibilidade de que ele contenha erros. Ferramentas de cobertura estrutural, notadamente cobertura de código, vêm sendo largamente empregadas na indústria de *software*, e têm apresentado resultados satisfatórios, como mostra o trabalho de (Malaiya et al., 1994).

Mesmo em projetos onde o plano de verificação foi realizado de forma cuidadosa, casos especiais criados durante a implementação podem ser ignorados pelos testes funcionais. Um exemplo clássico é quando uma variável de ponto flutuante precisa atingir especificamente o valor 0 (zero) para que um ramo do programa seja executado. A possibilidade da ocorrência deste valor utilizando-se geração aleatória de estímulos é mínima.

A partir das informações de cobertura, o engenheiro de teste pode planejar melhor o ambiente de verificação, de modo a verificar as porções não cobertas do código fonte.

Considerando a inabilidade da cobertura estrutural detectar um erro de omissão e a presença de casos especiais no código fonte do sistema, uma forma de aumentar a eficiência da verificação é combinar técnicas funcionais e estruturais em uma metodologia híbrida de verificação. A figura 20 mostra o fluxograma da metodologia proposta, que é composta de 5 passos, que são:

1. **Testes funcionais** - Gerar vetores de teste funcionais e aplicá-los ao sistema;

2. **Análise de cobertura** - Identificar partes do código que não foram exercitadas pelos testes funcionais, ou foram exercitadas apenas de modo ineficiente;

3. **Escolha de caminho** - Escolher, de acordo com algum critério, o melhor caminho do código fonte que leve à execução das porções não exercitadas;

4. **Caminho é possível?** - Determinar se o caminho escolhido é executável. Se não for, voltar ao passo 3;

5. **Testes estruturais guiados por cobertura** - Determinar quais vetores de entrada exercitam o caminho escolhido e aplicá-los ao sistema;

Figura 20: Fluxograma da Metodologia Proposta

Tendo como partida o ciclo de projeto apresentado na figura 7, pode-se detalhar a etapa de verificação para incluir a metodologia proposta. Este detalhe é mostrado na figura 21, onde a verificação é quebrada em duas etapas, uma etapa funcional e uma etapa estrutural. Em cada uma das etapas, são utilizadas ferramentas de análise de cobertura de forma a determinar a qualidade dos testes aplicados. Enquanto a cobertura funcional não atingiu um nível satisfatório, os vetores de teste funcionais são trabalhados para cobrir as funcionalidades ainda não exercitadas. Quando o resultado desta etapa é considerado satisfatório, pode-se realizar a etapa estrutural, onde é verificado se os critérios estruturais foram satisfeitos.

Figura 21: Detalhe da Verificação no Ciclo de Projeto de SoC

Sempre que ocorrer uma mudança no sistema, seja por meio de *refinamento* ou correção de erros de implementação, o conjunto de vetores funcionais é aplicado novamente, na forma de testes de regressão, que verificam se erros não foram inseridos no código fonte do sistema durante a alteração. Geralmente, não é necessário que este conjunto de testes seja redefinido cada vez que for utilizado, pois, idealmente, as funcionalidades do

sistema variam pouco ao longo do projeto. Os testes estruturais, por sua vez, estão intimamente ligados à estrutura de controle do sistema e devem ser redefinidos sempre que esta estrutura de controle for alterada. Isto ocorre, principalmente, depois de uma etapa de refinamento do modelo, quando mais detalhes são incluídos e estruturas inseridas no código fonte.

Para a aplicação da metodologia, é necessário que ferramentas e métodos que ofereçam suporte à execução dos passos apresentados sejam utilizados. Uma revisão da literatura mostra que muitos trabalhos estão relacionados à geração de vetores funcionais, utilizada no passo 1. Um deles é o trabalho de (Malaiya, 1995), que apresenta resultados melhores do que a geração de estímulos puramente aleatórios (Yin et al., 1997). Por este motivo, este passo não será tratado neste trabalho.

Nota-se, no entanto, uma carência de ferramentas e métodos que ofereçam suporte à execução dos outros passos da metodologia. Ferramentas que automatizam o passo 2 estão disponíveis. Porém, elas não lidam corretamente com as macros e bibliotecas do SystemC, referenciando o código de modo errôneo, ou empregam métricas de cobertura muito simples. O GCOV (GNU, 2007b) implementa apenas a cobertura de enunciados que, como visto na seção 2.4.5.1, é uma métrica simples de cobertura. Além disso, as bibliotecas do SystemC são incluídas em sua análise de cobertura, dificultando a ligação entre o resultado da análise e o código fonte original do sistema. Outra ferramenta, apresentada em (Große et al., 2007), faz a ligação correta entre o código fonte do sistema e o resultado da análise de cobertura, mas emprega somente o conceito de cobertura de ramos. No capítulo 5, é proposta e desenvolvida uma ferramenta para análise de cobertura de código para sistemas descritos em SystemC que lida corretamente com a ligação entre o código fonte original e o resultado da análise de cobertura e oferece suporte ao emprego de diversas métricas de análise de cobertura de código. Esta ferramenta permite a execução do passo 2 da metodologia proposta.

Não foram encontradas ferramentas que ofereçam suporte para os passos 3, 4 e 5. Sem estas ferramentas a validação da metodologia proposta não é possível. As direções atuais da pesquisa, no entanto, indicam possíveis soluções para o suporte a 2 destes passos. O

capítulo 4 apresenta um método numérico para geração de vetores de teste para módulos descritos em SystemC, que abrange os passos 4 e 5. Este método foi publicado em (Dias Junior e Silva, 2007) e, apesar de ainda não se encontrar em estágio final de pesquisa, já apresentou resultados promissores.

4 MÉTODO NUMÉRICO PARA GERAÇÃO DE VETORES DE TESTE PARA MÓDULOS DESCRITOS EM SYSTEMC

No capítulo 3, foi apresentada uma metodologia composta de cinco etapas que combina técnicas funcionais e estruturais para uma verificação mais efetiva do sistema. A quinta etapa desta metodologia requer uma ferramenta que permita a geração de vetores de teste que exercitem um determinado caminho no código fonte do sistema. Uma revisão da literatura mostrou que tais ferramentas não estão disponíveis e métodos e técnicas que ofereçam suporte ao seu desenvolvimento são escassos e incompletos, principalmente para sistemas descritos em SystemC.

Neste capítulo é apresentado um método numérico para geração de vetores de teste para módulos descritos em SystemC. A aplicação deste método está restrita a sistemas combinatórios, uma vez que o desenvolvimento de um método possa ser aplicado em módulos seqüenciais é uma tarefa complexa, que está além das restrições de tempo impostas ao desenvolvimento desta dissertação. As direções apontadas neste capítulo serão utilizadas como base para pesquisas futuras.

4.1 Descrição do Método Numérico

Uma alternativa à execução simbólica de caminhos, apresentada na seção 2.5, é uma abordagem baseada em otimização numérica, proposta em (Gallagher e Narasimhan, 1997) para sistemas desenvolvidos na linguagem Ada, que foi adaptada para o SystemC em (Dias Junior e Silva, 2007). Nessa abordagem, o código fonte é instrumentado, inserindo-se chamadas de funções com o intuito de controlar o fluxo de execução. Um

método numérico de otimização é empregado para se obter um dos vetores que exercita o caminho desejado. Considere o seguinte fragmento de código:

```
1  x = 0;
2  if ( a > 2 )
3  {
4    x = a - 2;
5    if ( b == 3 )
6    {
7      foo();
8    }
9  }
10
11 if ( b != 2 )
12 {
13   if ( x <= 4 )
14     bar();
15 }
```

Deseja-se gerar um vetor de teste para exercitar um determinado caminho deste código, por exemplo o caminho que inclui as funções *foo* e *bar*. Para a execução deste caminho, todas as condições testadas nos prefixos devem ser verdadeiras. Analisando-se o caminho simbolicamente, chega-se a conclusão que, para sua execução, são impostas as seguintes restrições:

$$a > 2 \; e \; b == 3 \; e \; b! = 2 \; e \; x <= 4 \tag{10}$$

Considerando a e b como variáveis de entrada do programa, pode-se simplificar a equação 10:

$$b == 3 \; e \; a > 2 \; e \; a <= 6 \tag{11}$$

Assim, um método de otimização numérica deverá fornecer valores para os vetores de entrada que respeitem as restrições apresentadas na equação 11 sem que seja necessária uma análise simbólica do caminho.

Para a aplicação do método numérico, a primeira etapa é a instrumentação do código fonte. Nessa etapa, cada teste de condição é substituído por uma chamada a uma função global, da forma apresentada no trecho de código abaixo:

```
1  x = 0;
2  if ( store( gt, a - 2, a > 2, 1 ))
3  {
4    x = a - 2;
5    if ( store( eq, b - 3, b == 3, 2 ))
6    {
7      foo();
8    }
9  }
10
11 if ( store( neq, b - 2, b != 2, 3 ))
12 {
13   if ( store( ge, 4 - x, x <= 4, 4 ))
14     bar();
15 }
```

A função *store* recebe quatro argumentos:

1. **Tipo** - representa os operadores relacionais $==, \geq, >, \neq$ através do tipo *enumerate* {*eq, ge, gt, ne*} respectivamente. Qualquer condição que envolva os outros operadores relacionais deve ser reorganizada para utilizar um desses operadores, como na condição $x \leq 4$ que foi reorganizada para $4 \geq x$;

2. **Restrição Numérica** - cada condição deve ser reformulada para criar restrições da forma $r == 0$, $r \neq 0$, $r \geq 0$ ou $r > 0$. Assim, a condição $4 \geq x$ foi reformulada para $4 - x \geq 0$. Somente o valor de r é representado na chamada de função;

3. **Condição Original** - a condição do prefixo na sua forma original;

4. **Identificação do Ramo** - um número que identifica unicamente cada um dos ramos.

A restrição numérica r indica quão próxima de ser satisfeita está a condição de um determinado ramo. Usando essas restrições é possível resolver os prefixos de um determinado caminho sem saber sua forma exata, apenas determinando valores para as entradas do programa que satisfaçam as restrições. A determinação dos valores de entrada que exercitam um determinado caminho pode ser feita por meio da resolução de um problema de otimização com restrições.

Seja c um caminho do programa P que deseja-se exercitar. Seja $\mathbf{x}$ o vetor que representa os valores para as n entradas de P. Seja $r_i(\mathbf{x})$ a restrição imposta à execução

do *i-ésimo* predicado p_i de P quando a entrada do sistema é o vetor $\mathbf{x}$. O problema de otimização com restrições para se determinar o valor do vetor $\mathbf{x_o}$ que exercita c pode ser formulado como:

$$f(\mathbf{x}), \mathbf{x} \in \Re^n \tag{12}$$

Sujeito ao seguinte conjunto de restrições não lineares:

$$r_i(\mathbf{x}) \left\{ \begin{array}{c} > \\ \geq \\ = \\ \neq \end{array} \right\} 0, \qquad i = 1, 2, ..., m \tag{13}$$

Onde $f(\mathbf{x})$ é a função cujo resultado representa a distância do vetor $\mathbf{x}$ atual do vetor $\mathbf{x_o}$ ótimo. A função $r_i(\mathbf{x})$ indica o valor da restrição da *i-ésima* condição do caminho c escolhido, cujo número total de restrições é m. Minimizando-se o valor de $f(\mathbf{x})$, desde que as restrições $r_i(\mathbf{x})$ sejam atendidas, obtém-se $\mathbf{x} = \mathbf{x_o}$.

Pode-se transformar esse problema de otimização sujeito a restrições em um problema sem restrições por meio da utilização da técnica de função de penalidade, de acordo com a equação abaixo:

$$f'(\mathbf{x}, \mathbf{w}) = \sum_{i}^{m} R_{tipo_i}(r_i(\mathbf{x}), w_i) \tag{14}$$

Onde $\mathbf{w} = \{w_1, w_2, ..., w_m\}$ é um conjunto de pesos positivos e $R_{tipo_i}(r_i(\mathbf{x}), w_i)$ representa a penalidade imposta à *i-ésima* condição do caminho, c, escolhido, cujo valor da restrição é $r_i(\mathbf{x})$.

A função de penalidade R_{tipo_i} é escolhida de forma que seja sempre positiva e que seu valor seja pequeno quando a restrição r_i está próxima de ser satisfeita. As equações 15 a 18 apresentam as funções de penalidade propostas em (Gallagher e Narasimhan, 1997) para os operadores $\geq$, $>$, $==$ e $\neq$.

$$R_{ge}(r_i(\mathbf{x})) = e^{-w_i(r_i(x)+\delta)}, \text{ para } r_i(\mathbf{x}) \text{ da forma } r \geq 0 \tag{15}$$

$$R_{gt}(r_i(\mathbf{x})) = e^{-w_i(r_i(x))}, \text{ para } r_i(\mathbf{x}) \text{ da forma } r > 0 \tag{16}$$

$$R_{eq}(r_i(\mathbf{x})) = e^{w_i|(r_i(x))|-1}, \text{ para } r_i(\mathbf{x}) \text{ da forma } r = 0 \tag{17}$$

$$R_{neq}(r_i(\mathbf{x})) = ke^{-w_i|(r_i(x))|}, \text{ para } r_i(\mathbf{x}) \text{ da forma } r \neq 0 \tag{18}$$

Onde δ é uma constante positiva de valor pequeno que garante que a origem não é penalizada e k é uma constante que mantém a proporção entre a equação 18 e as demais. Os gráficos para cada uma destas equações é mostrado na figura 22.

Figura 22: Funções de Penalidade para o Método Numérico

Caso a expressão Φ da condição do prefixo p_i inclua operadores lógicos, suas restrições podem ser calculadas utilizando-se das seguintes propriedades:

- $R_{tipo_i}(\mu_1 AND\ \mu_2) = R_{tipo_i}(\mu_1) + R_{tipo_i}(\mu_2)$
- $R_{tipo_i}(\mu_1 OR\ \mu_2) = min(R_{tipo_i}(\mu_1), R_{tipo_i}(\mu_2))$

- $R_{tipo_i}(NOT\ \mu) = R'_{tipo_i}(\mu)$

- $R_{tipo_i}(\mu_1 XOR\ \mu_2) = min(R_{tipo_i}(\mu_1) + R'_{tipo_i}(\mu_2), R'_{tipo_i}(\mu_1) + R_{tipo_i}(g_2))$

Onde μ_i é o termo cuja restrição é r_i e $R'(r_i)$ é a função de penalidade para testar o inverso da operação de r_i.

O valor de $r_i(\mathbf{x})$ é calculado forçando-se a execução do caminho por meio da função *store*. O código fonte da função *store* é apresentado a seguir. Algumas partes da função foram omitidas, por não tratarem especificamente do tópico abordado nessa seção.

```
1  bool store(r_type type, double r, bool value, int id)
2  {
3    m_type[id] = type;
4    m_r[id] = r;
5
6    (...)
7
8    return (value && !contExec) ||
9        (contExec && Branch[id]);
10 }
```

A variável (*contExec*) permite que a execução do caminho seja controlada. Caso o valor de *contExec* seja falso (*FALSE*), o caminho é executado de acordo com os valores dos vetores de entrada, como numa execução normal do programa. Caso o valor dessa variável seja verdadeiro (*TRUE*), o fluxo do programa é controlado pelos valores armazenados no vetor de booleanos *Branch*. Cada elemento desse vetor indica se um determinado ramo deve ser forçado a executar.

O caminho cujo o vetor de entrada deseja-se obter deve ser executado forçadamente. Aplicando-se um método numérico de otimização não linear multi-variável para minimizar o valor da função da equação 14, obtém-se o vetor de entrada $\mathbf{x} = \mathbf{x_o}$ que exercita o caminho c, se ele existir. Se este vetor não for encontrado, o caminho é impossível, a busca do método de otimização terminou em um mínimo local, ou o número de iterações executadas foi insuficiente para determiná-lo.

4.2 Resultados da Aplicação do Método Numérico

Considere o seguinte fragmento de código, já instrumentado com a função *store*:

```
1  if (store(eq, ctl - 1, ctl == 1,  1))
2  {
3    c = a + b;
4  }else{
5    if (store(eq, ctl - 2, ctl == 2,  2))
6    {
7      c = a - b;
8    }else{
9      if (store(eq, ctl - 3, ctl == 3,  3))
10     {
11       if (store(eq, b - 2, b == 2,  4))
12       {
13         c = a << 1;
14       }else{
15         c = 0;
16         int i = 0;
17         for (i = 0; store(gt, b - i, i < b,  5); ++i)
18         {
19           c += a;
20         }
21       }
22     }else{
23       if (store(eq, ctl - 4, ctl == 4,  6))
24       {
25         if (store(eq, b - 2, b == 2,  7))
26         {
27           c = a << 1;
28         }else{
29           if (store(ne, b, b != 0,  8))
30           {
31             c = 0;
32             int a_v = abs(a);
33             int b_v = abs(b);
34             while (store(gt, a_v, a_v > 0, 9))
35             {
36               a_v -= b_v;
37               ++c;
38             }
39           }else{
40             error = DIVISION_BY_ZERO;
41           }
42         }
43       }
44     }
45   }
46 }
```

Trata-se de um código de uma unidade aritmética que calcula as quatro operações básicas sobre valores inteiros. Existem três entradas para o código: *a*, *b* e *ctl*, respectivamente o primeiro operando da operação, o segundo operando e um sinal de controle que seleciona qual operação será realizada. A multiplicação e a divisão foram implementadas de forma a conter enunciados de laço, para ilustrar a instrumentação deste tipo de construção. Existe uma especialização para a multiplicação e divisão por 2. Se o operando *b*

é igual a 2, a operação é tratada como um deslocamento de *bits*.

Analisando cuidadosamente o código, são encontrados dois erros:

1. A especialização para a divisão está errada, pois deve ser implementada como um deslocamento à direita;

2. Ainda na divisão, quando os operandos possuem sinais diferentes, o resultado calculado estará errado. Existe um erro de omissão no código, pois os sinais dos operandos deveriam ser testados para se determinar o sinal do resultado.

Como visto no capítulo 3, o erro 2 é facilmente identificado com uma geração adequada de valores aleatórios para as entradas do sistema. Porém, o erro 1 não é tão simples de ser identificado pois só ocorre quando *ctl* = 4 e *b* = 2. O grafo de fluxo para o código é apresentado na figura 23. Para este exemplo, considere que queira-se gerar um vetor de teste para a execução do caminho $c = \{1, 2, 3, 6, 7, end\}$.

Figura 23: Grafo de Fluxo para Exemplo de Aplicação do Método Numérico

Para a geração dos vetores, o valor da variável *contExec* da função *store* é feito verdadeiro (*TRUE*). Os valores de *Branch[i]*, para $i = \{1, 2, 3\}$ são feitos iguais a falso (*FALSE*), enquanto os valores de *Branch[i]* para $i = \{6, 7\}$ são feitos iguais a verdadeiro (*TRUE*). Os valores de *Branch[i]* para $i = \{8, 9\}$ não importam, pois seus enunciados não serão atingidos.

Tabela 1: Resultado das Iterações do exemplo

Iter	a	$\partial f/\partial a$	b	$\partial f/\partial b$	ctl	$\partial f/\partial ctl$
1	10	4.2	14	4.8e+015	-4	-2.9e+010
2	10	6.3	14	4.1e+015	-4	-2.9e+010
3	10	-0.52	14	1.6e+015	-4	-2.9e+010
4	10	0	13	8.8e+014	-4	-2.9e+010
...	...	...	...	...	...	...
53	10	-3.7e-015	2.1	1.3	4.2	2.2
54	10	1.9e-015	1.9	-1.3	4.1	1.5
55	10	-1.9e-015	2	-1.2	4	0.47
56	**10**	**-1.9e-015**	**2**	**-1.3**	**4**	**-1.4**

Cada vez que a função *store* é invocada no predicado p_i, ela armazena os valores da restrição r_i. Depois de todo o caminho executado, o valor da função $f(\mathbf{x}, \mathbf{w})$, apresentada na equação 14, é calculado e um método de otimização numérica é executado. Este valor é utilizado pelo método para determinar o próximo vetor de entrada $\mathbf{x}$ que será aplicado ao sistema.

A tabela 1 mostra o resultado para as primeiras e as últimas iterações da aplicação do método numérico para geração de vetores de teste. As colunas *a*, *b* e *ctl* representam os valores das entradas do sistema e as colunas $\partial f/\partial a$, $\partial f/\partial b$ e $\partial f/\partial ctl$ indicam, respectivamente, o valor dos gradientes destas entradas em cada uma das iterações. Esta informação é utilizada pelo método de otimização numérica não linear empregado neste trabalho. Os resultados foram obtidos utilizando-se o método quasi-Newton com atualização BFGS.

Analisando os resultados, verifica-se que a entrada *a* não influencia nas restrições do caminho e que os valores para exercitar o caminho são *b* = 2 e *ctl* = 4, como esperado.

4.3 Análise do Método

O método apresentado usa otimização numérica para gerar vetores de vetores de teste para sistemas descritos em SystemC. Esta abordagem não sofre das restrições relacionadas à execução simbólica, como definir valores de ponteiros, uma vez que o código é realmente executado durante a otimização. Um exemplo simples foi apresentado para de-

monstrar o funcionamento e, apesar dos resultados serem promissores, mais estudos são necessários para determinar a sua aplicabilidade. Até o momento, o método é aplicável somente a sistemas combinatórios. Como o caminho precisa ser executado a cada iteração do algoritmo de otimização numérica, o sistema não pode atingir estados que impeçam a execução deste caminho. Trabalhos futuros serão desenvolvidos para que o método possa ser empregado também em sistemas seqüenciais e ofereça suporte completo ao passo 5 da metodologia proposta no capítulo 3.

5 SCCOV: FERRAMENTA PARA ANÁLISE DE COBERTURA DE TESTE

Na seção 2.4, foi dada uma introdução à verificação em projetos de SoCs. Foi visto que, para uma verificação efetiva, é preciso que o engenheiro de teste possua informações sobre a cobertura que os testes aplicados atingiram. Essa informação determina o momento em que o sistema foi suficientemente testado e pode-se seguir para a próxima etapa do ciclo de projeto. Nesse capítulo, será apresentada uma ferramenta para análise de cobertura de teste em projetos de alto nível de abstração. Esta é a primeira ferramenta, da qual tem-se conhecimento, que permite que diversas métricas de cobertura estrutural sejam aplicadas aos testes de projetos descritos em SystemC e é a principal contribuição deste trabalho. Para a sua elaboração, os seguintes requisitos foram considerados:

1. A ferramenta não deve requerer nenhuma modificação no código fonte do compilador GCC nem no código fonte das bibliotecas do SystemC;

2. A ferramenta deve poder ser aplicada a qualquer código SystemC, sem que seja necessária a utilização de estilos de código específicos;

3. A ferramenta deve aplicar os conceitos de *cobertura total de ramos*, *cobertura total de laços*, *cobertura total de termos*, *cobertura exaustiva de termos*, *cobertura relacional total* e *cobertura total de funções*, conforme descritos na seção 2.4;

4. A ferramenta deve fornecer informações exatas sobre a cobertura do código fonte, referenciando o código original do sistema antes da instrumentação;

5. Os resultados obtidos pelo sistema original devem ser equivalentes aos obtidos pelo sistema depois da aplicação da ferramenta.

Para analisar a cobertura dos testes aplicados ao sistema, é necessário que, de alguma forma, sejam inseridas construções em seu código executável, permitindo a extração das informações de cobertura. A figura 24 mostra o fluxo detalhado da compilação de um código SystemC genérico. Construções para extração de informação podem ser inseridas em vários pontos diferentes no fluxo de compilação. Porém, inserir construções durante a compilação do programa iria requerer uma versão modificada do compilador e, portanto, fere o requisito 1. Por esse motivo, este trabalho utiliza instrumentação do código fonte SystemC diretamente, atuando no ponto *A* da figura 24.

Figura 24: Fluxo de Compilação Detalhado do Código SystemC

Para se alterar diretamente o código fonte do sistema, é preciso que seja construído um *analisador léxico* que permita extrair informações sobre o fluxo do programa e então inserir construções no código fonte. Esse *analisador léxico* deve ser genérico o bastante para não ferir o requisito 2, que indica que a ferramenta não deve obrigar o projetista a utilizar determinados estilos de código. No entanto, como será visto nas seções seguintes, as construções a serem inseridas são muito simples e atendem aos padrões do C++. Dessa forma, um *analisador léxico* simples atenderá a esse requisito sem maiores problemas.

Outro cuidado importante no desenvolvimento da ferramenta é que ela não deve ferir, de forma alguma, o requisito 5, que indica que o código fonte original e o código fonte instrumentados devem ser equivalentes do ponto de vista dos resultados gerados. Apesar de bastante intrusiva, a instrumentação do código não pode alterar o comportamento da simulação do sistema sob pena de invalidar totalmente os testes. Exceto por um possível aumento do tempo gasto pela simulação devido às chamadas instrumentadas, o comportamento do modelo executável original e do instrumentado devem ser equivalentes.

5.1 Estrutura da Ferramenta ScCov

Com a análise desses requisitos, obtém-se o o novo fluxo de compilação do código fonte SystemC utilizando-se da ferramenta proposta, que é mostrado na figura 25. Nela, verifica-se que a ferramenta é aplicada no código fonte original desenvolvido pelo projetista e tem como resultado um código fonte instrumentado, que possui construções específicas para se medir a cobertura dos testes.

Figura 25: Fluxo de Compilação do Código SystemC usando a Ferramenta ScCov

O processamento da ferramenta se dá em duas etapas. Primeiro, é realizada uma

análise léxica do código onde são extraídas informações acerca das construções presentes no modelo original. Essas informações são utilizadas para guiar a etapa de instrumentação de código e para fazer a ligação entre os relatórios gerados pela ferramenta e o código fonte original do sistema. Após a análise léxica, o código fonte é instrumentado. Em cada ponto onde é necessário extrair informações sobre a execução dos testes, um *wrapper* é criado em volta da construção original. Durante a execução do modelo, os *wrappers* coletam as informações necessárias para a construção dos relatórios de cobertura. Depois de instrumentado, o código fonte SystemC é compilado normalmente e o resultado é um modelo executável com capacidade de medir a cobertura dos testes aplicados.

A estrutura da ferramenta é apresentada na figura 26, onde seus componentes estão destacados em cinza. Trata-se de uma representação de todo o fluxo de utilização, englobando as etapas de análise e instrumentação do código, execução do teste pelos modelos executáveis, e geração de relatórios pela interface de usuário.

Figura 26: Diagrama de Blocos da Ferramenta ScCov

O código fonte SystemC, fornecido pelo projetista, passa inicialmente por um *extrator de símbolos*, que examina e separa as construções da linguagem e as envia para o *analisador léxico*. As informações acerca das construções e estrutura do código são extraídas e enviadas para o *servidor de dados*, para que possam ser utilizadas por outros componentes da ferramenta. Uma vez analisado, o código fonte é instrumentado pelo *editor de código*, que insere chamadas de funções, bibliotecas e outras construções necessárias para extrair

as informações de cobertura dos testes.

O código fonte instrumentado é compilado normalmente, como qualquer outro código SystemC. Quando o *Modelo Executável* do sistema é executado, ele passa a enviar dados sobre a cobertura dos testes aplicados para o *servidor de dados*. Qualquer acesso ao servidor de dados, durante toda a aplicação da ferramenta, é efetuado por meio de comunicação inter-processos (IPC). Essa abordagem permite que várias instâncias do *Modelo Executável* do sistema estejam ativas ao mesmo tempo, em processadores diferentes, o que aumenta a eficiência dos testes. Utilizar IPC também garante uma maior escalabilidade da ferramenta, permitindo que funcionalidades sejam inseridas quando necessário.

Em seguida, cada um dos componentes básicos da ferramenta será descrito e os relatórios fornecidos serão apresentados.

5.1.1 Extrator de Símbolos

O *extrator de símbolos* percorre cada arquivo do projeto e extrai os símbolos, ou *tokens*, que serão utilizados pelos demais componentes da ferramenta. O funcionamento básico do componente é simples: o arquivo é percorrido acumulando-se os caracteres em uma variável intermediária até o momento em que um delimitador é encontrado. Quando o delimitador é encontrado, os caracteres acumulados na variável são marcados como um *token*, e o próprio delimitador também é marcado como um *token*. Uma função, *GetNextToken()*, retorna os *tokens* presentes no arquivo. A tabela 2 mostra a relação dos delimitadores utilizados pelo *extrator de símbolos*.

Além dos delimitadores comuns, o *extrator de símbolos* permite a definição de delimitadores atômicos. Nesse caso, pares de delimitadores são relacionados de forma que os caracteres entre o primeiro delimitador e o segundo, incluindo os próprios delimitadores, são considerados como um único *token*. Esse tipo de delimitador é encontrado, por exemplo, no início e fim de blocos de comentário. O *extrator de símbolos* interpreta o caractere de escape do C++ (\). Sempre que esse caractere é encontrado, o próximo caractere é desconsiderado para efeitos de marca de fim de delimitadores atômicos. A tabela 3 apresenta os delimitadores atômicos e para qual finalidade cada um deles é utilizado no código C++.

Tabela 2: Conjunto de Delimitadores do extrator de símbolos

Delimitador	Descrição
Espaço	Caractere de Espaço
\n	Alimentação de Linha
\r	Retorno de Carro
\t	Tabulação
(	Abre Parênteses
)	Fecha Parênteses
[	Abre Colchete
]	Fecha Colchete
{	Abre Chave
}	Fecha Chave
,	Vírgula
;	Ponto-e-vírgula
:	Dois Pontos
::, ++, - -, ., ->, +, -, *, /, !, , -=, +=, ** >>, <<, >, <, >=, <=, ==, !=, &&, ||, &, |, ^	Operadores do C++

Tabela 3: Conjunto de Delimitadores Atômicos do Extrator de Símbolos

Marca de Início	Marca de Fim	Descrição
//	\n	Comentário de linha
/*	*/	Bloco de comentário
"	"	Delimitadores de *strings* C
'	'	Delimitadores de caracteres

5.1.2 Analisador Léxico

Os *tokens* da saída do *extrator de símbolos* são alimentados diretamente no *analisador léxico*. Tanto o *analisador léxico* quanto o *editor de código*, descrito a seguir, funcionam utilizando um princípio de regras em cascata. Cada uma das regras baseia-se em uma máquina de estado para determinar quando ela deve ser aplicada e, em geral, é independente da regra seguinte. O *analisador léxico*, portanto, é um conjunto de regras independentes que recebem como entrada *tokens* extraídos pelo *extrator de símbolos* e interpretam as seqüências de símbolos, gerando informações que serão armazenadas para utilização futura, sem alterar qualquer *token* que recebe. A figura 27 mostra a disposição das regras em cascata no interior do *analisador léxico*.

As regras do *analisador léxico* podem ser inseridas ou retiradas mesmo durante a

Figura 27: Regras em Cascata no Analisador Léxico

execução do programa. Essa característica permite que a ferramenta seja expandida facilmente para analisar construções além daquelas previstas durante o seu desenvolvimento, sem ser necessária a modificação de outras partes do código.

A seguir, as regras do *analisador léxico* são apresentadas, na ordem em que são aplicadas, com uma breve descrição do que é feito em cada uma das regras.

1. Identifica ***if*** - Identifica o início, fim, cabeçalho e hierarquia dos enunciados *if*;
2. Identifica ***for*** - Identifica o início, fim, cabeçalho e hierarquia dos enunciados *for*;
3. Identifica ***do*** - Identifica o início, fim, cabeçalho e hierarquia dos enunciados *do*;
4. Identifica ***while*** - Identifica o início, fim, cabeçalho e hierarquia dos enunciados *while*;
5. Identifica ***switch*** - Identifica o início, fim, cabeçalho e hierarquia dos enunciados *switch*;
6. Identifica **operadores relacionais** - Identifica condições que utilizam operadores relacionais $>$ e $<$;
7. Identifica **funções** - Identifica o início, fim e argumentos das funções.

5.1.3 Editor de Código

O *editor de código* é responsável pela instrumentação do código fonte do programa SystemC, alterando-o de forma a permitir que informações sejam extraídas e inseridas durante a simulação do sistema. Em cada ponto onde é necessário extrair informações sobre a execução dos testes, um *wrapper* é criado em volta da construção original. Os *wrappers* são chamadas de função que, quando executadas, coletam informações de execução e enviam para o *servidor de dados*.

Da mesma forma que o *analisador léxico*, o *editor de código* é baseado em regras em cascata. A diferença entre os dois é que no *editor de código* uma regra pode alterar o valor do *token* recebido antes de enviá-lo para a regra seguinte. Entre cada uma das regras, existe uma instância de um *extrator de símbolos*, responsável por separar os *tokens* da modificação inserida pela regra anterior. Dessa forma, a regra pode alterar o *token* sem ser necessário entregá-lo separado para a regra seguinte. A figura 28 mostra a disposição das regras e dos *Extratores de Símbolos* no interior do *editor de código*.

Figura 28: Regras e Extratores de Símbolos do Editor de Código

Cada uma das regras é responsável por inserir um tipo diferente de *wrapper* em uma construção do SystemC. Diferentemente das regras do *analisador léxico*, a ordem das regras do *editor de código* influencia no resultado final da instrumentação. As seguintes regras estão presentes no *editor de código* da ferramenta, e são apresentadas na ordem em que são aplicadas.

1. Instrumenta **Funções** - Instrumenta as funções para determinar quais foram executadas pelos testes;

2. Instrumenta **sc_main** - Insere construções de inicialização da ferramenta na função inicial do sistema;

3. Insere ***Include*** - Insere uma referência para o arquivo de cabeçalho (*header file*) da biblioteca da ferramenta em cada um dos arquivos instrumentados;

4. Instrumenta ***if*** - Instrumenta os enunciados *if*;

5. Instrumenta ***for*** - Instrumenta os enunciados *for*;

6. Instrumenta ***while*** - Instrumenta os enunciados *while*;

7. Instrumenta ***do*** - Instrumenta os enunciados *do*;

8. Instrumenta ***switch*** - Instrumenta os enunciados *switch*;

9. Instrumenta **operadores relacionais** - Instrumenta condições que utilizam operadores relacionais $>$ e $<$.

A seguir, cada uma das regras para instrumentação do código fonte do projeto SystemC será detalhada, analisando como cada tipo de enunciado deve ser alterado pela ferramenta.

5.1.3.1 Instrumenta Funções

O padrão C++ usa protótipos de função (Eckel, 2000) para identificar funções no programa. Quando cria-se uma função, é necessário que seja determinado o tipo de retorno e quais argumentos a função deve receber. Assim, a forma geral de uma função em C++ é:

```
1 tipo_retorno NOME_DA_FUNCAO(tipo1 arg1, tipo2 arg2, ...)
2 {
3   // código a ser executado
4 }
```

Onde a primeira linha, contendo o nome da função, é chamada de protótipo da função, e o trecho compreendido entre `{ . . . }` é chamado de corpo da função.

Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, uma função interfere apenas na *cobertura total de funções*. Pode-se acreditar, de maneira equivocada, que uma função interfere na *cobertura total de ramos*. Uma função, no entanto, não permite por si só, que o fluxo de execução de um programa se altere, sendo necessário o emprego de enunciados de controle de execução. O fluxo de execução pode ser alterado internamente na função por meio do emprego de um enunciado de controle de execução, bem como devido à utilização do retorno de uma função como condição para um enunciado de controle de execução. A chamada pura de uma função não altera o fluxo de execução de um programa.

Para a instrumentação de uma função, insere-se no início de seu corpo uma chamada à função *sc_cov::functionWrapper*, cujo protótipo é:

```
1 void sc_cov::functionWrapper(unsigned funcId);
```

O argumento *funcId* recebe um número que identifica aquela função de forma única no programa em teste. Essa função, quando executada, permite a contagem do número de vezes que a função instrumentada foi invocada. Assim, a função apresentada anteriormente, depois de instrumentada, teria a seguinte forma:

```
1 tipo_retorno NOME_DA_FUNCAO(tipo1 arg1, tipo2 arg2, ...)
2 {
3   sc_cov::functionWrapper(id);
4   // código a ser executado
5 }
```

O padrão C++ permite a utilização de sobrecarga de funções. Uma função pode ter o mesmo nome de outra função, desde que seus argumentos sejam de tipos e/ou quantidades diferentes. Funções sobrecarregadas não alteram o funcionamento da ferramenta, pois elas recebem identificadores diferentes durante a etapa de instrumentação, sendo tratadas como funções completamente distintas nas etapas seguintes. Nos relatórios de cobertura de função, detalhados ao final do capítulo, funções sobrecarregadas são diferenciadas pelo arquivo e linha no arquivo onde foram definidas, e pela assinatura de seus argumentos, descrita no relatório.

5.1.3.2 Instrumenta sc_main

Assim como todo código C++ possui um ponto de entrada, o código SystemC também possui um ponto de entrada, que é a função *sc_main*. Esta é a primeira função invocada quando o modelo executável do sistema é executado e, geralmente, é o ponto onde são instanciados os objetos da simulação, como os *Modules*, *Signals*, etc.

Além do tratamento dado a todas as funções do sistema, a função *sc_main* recebe uma instrumentação especial, sendo inserida uma chamada de função que realiza as inicializações necessárias à ferramenta *ScCov* e também uma função de finalização. Uma chamada à função *sc_cov::init* é inserida no início do corpo da função e uma chamada à função

sc_cov::end é inserida antes de todos os enunciados *return*, conforme o código abaixo:

```
1  int sc_main (int argc , char *argv[])
2  {
3    sc_cov::init(config_file);
4    sc_cov::functionWrapper(id);
5
6    // código a ser executado
7
8    sc_cov::end();
9    return 0;
10 }
```

O argumento *config_file* recebe o nome do arquivo de configuração que fornece informações de parâmetros para a ferramenta. Este arquivo é gerado automaticamente para cada projeto. É possível notar que a inicialização por meio da função *sc_cov::init* é feita antes da própria chamada da função *sc_cov::functionWrapper*, utilizada para verificar a cobertura de funções. Isso acontece porque a função *sc_cov::functionWrapper* necessita que a ferramenta seja inicializada antes de ser invocada.

5.1.3.3 Insere Include

A instrumentação de código insere apenas as chamadas de função no corpo do código fonte do projeto SystemC. É necessário que seja inserida uma referência às declarações e definições dessas funções. A regra *Insere Include* insere uma diretiva *#include "sccov.h"* no início de cada programa instrumentado. O arquivo de cabeçalho *sccov.h* contém as declarações dos protótipos das funções instrumentadas e é a ligação entre o módulo sob teste e a biblioteca de cobertura ScCov.

5.1.3.4 Instrumenta if

O enunciado *if* permite que o fluxo de execução de um programa seja desviado de acordo com o teste de uma condição. Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, um enunciado *if* interfere na *cobertura total de ramos*, *cobertura total de termos* e *cobertura exaustiva de termos*. Seja o seguinte trecho de código, que apresenta a forma geral de um enunciado *if*:

```
1 if (CONDICAO)
2 {
3    // código a ser executado se CONDICAO for verdadeira
4 }
5 else
6 {
7    // código a ser executado se CONDICAO for falsa
8 }
```

Nesse exemplo, *CONDICAO* pode ser considerada uma expressão Φ, na forma apresentada na equação 5. Para analisar a cobertura de ramos, basta verificar se *CONDICAO* atingiu os valores verdadeiro (*TRUE*) e falso (*FALSE*) durante a execução dos testes. Caso esses dois valores sejam atingidos, ocorreu *cobertura total de ramos* no que se refere ao enunciado *if* em questão.

No caso da cobertura de termos, é necessário que seja verificado quais dos termos μ_i presentes em *CONDICAO* atingiu, individualmente, os valores verdadeiro (*TRUE*) e falso (*FALSE*) durante a execução dos testes. Já a *cobertura exaustiva de termos* requer que cada uma das combinações possíveis de valores para os termos de *CONDICAO* seja verificada. Considerando-se que *CONDICAO* seja composta por n termos, é necessário verificar quais das 2^n possíveis combinações de valores para os termos μ_i foram exercitadas pelos testes.

Para a instrumentação de um enunciado *if*, utilizam-se duas chamadas de função distintas, cujos protótipos são apresentados a seguir:

```
1 bool sc_cov::branchWrapper(unsigned branchId,
2              baseBranchData::branchType type, bool value);
3
4 bool sc_cov::conditionWrapper(unsigned branchId,
5              unsigned conditionId, bool value);
```

O argumento *branchId* recebe um número que identifica aquele ramo de forma única no programa. A função *sc_cov::branchWrapper* possui mais dois argumentos. O argumento *type* indica qual o tipo do ramo que está sendo instrumentado, nesse caso receberá o valor *baseBranchData::ifBranch*, que informa que aquele ramo é um *if*. O terceiro argumento, *value*, é o valor da condição original do enunciado de controle de fluxo. No exemplo dado anteriormente, *value* receberá o valor de CONDICAO. A chamada

sc_cov::branchWrapper retorna o valor de *value* sempre que é chamada.

A utilização de *sc_cov::branchWrapper* permite somente a verificação de cobertura de ramos, mas não possibilita a verificação de cobertura de termos e cobertura exaustiva de termos. Para estes dois tipos de cobertura, é preciso que um *wrapper* também seja criado em torno de cada um dos termos separados por um operador lógico.

A função *sc_cov::conditionWrapper* envolve cada um dos termos presentes em *CONDICAO*. Ela possui dois argumentos além do identificador do ramo. O argumento *conditionId* identifica qual o número do termo que a função envolve, e começa do valor 1 e é incrementado a cada operador lógico subseqüente. O terceiro argumento, *value*, recebe o valor do termo envolvido. A função retorna o valor de *value* sempre que é chamada.

Assim, substituindo-se *CONDICAO* pela expressão lógica $x == 1 \mid\mid b < 3$ na forma geral do enunciado *if* apresentada anteriormente, chega-se ao seguinte resultado para o código instrumentado:

```
1  if (sc_cov::branchWrapper(id, baseBranchType::ifBranch,
2      sc_cov::conditionWrapper(id, 1, x == 1) ||
3      sc_cov::conditionWrapper(id, 2, b < 3)))
4  {
5     // código a ser executado se CONDICAO for verdadeira
6  }
7  else
8  {
9     // código a ser executado se CONDICAO for falsa
10 }
```

5.1.3.5 Instrumenta for

O enunciado *for* permite que um determinado trecho de código seja executado repetidamente, enquanto uma determinada condição for verdadeira. Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, um enunciado *for* interfere na *cobertura total de ramos*, *cobertura total de laços*, *cobertura total de termos* e *cobertura exaustiva de termos*. A sua forma é apresentada a seguir:

```
1 for(INICIO ; CONDICAO ; LACO )
2 {
3    // código a ser executado
4    // enquanto CONDICAO for verdadeira
5 }
```

Da mesma forma que em um enunciado *if*, *CONDICAO* pode ser considerada uma expressão Φ, na forma apresentada na equação 5. O enunciado *for* influencia em um tipo de cobertura que não está presente nos enunciados *if*. A *cobertura total de laços*, que requer que o laço seja iterado uma vez, várias vezes e também não seja iterado durante os testes.

Para a instrumentação de um enunciado *for*, utilizam-se três chamadas de função distintas, cujos protótipos são apresentados a seguir:

```
1 bool sc_cov::loopWrapper(unsigned branchId,
2              baseBranchData::branchType type, bool value);
3
4 bool sc_cov::conditionWrapper(unsigned branchId,
5              unsigned conditionId, bool value);
6
7 void sc_cov::resetLoop(unsigned branchId,
8              baseBranchData::branchType type);
```

O argumento *branchId* recebe um número que identifica aquele ramo de forma única no programa. A função *sc_cov::loopWrapper* possui mais dois argumentos. O argumento *type* indica qual o tipo do ramo que está sendo instrumentado, nesse caso receberá o valor *baseBranchData::forBranch*, que informa que aquele ramo é um *for*. O terceiro argumento *value*, é o valor da condição original do enunciado de controle de fluxo. No exemplo dado anteriormente, *value* receberá o valor de CONDICAO. A chamada *sc_cov::loopWrapper* retorna o valor de *value* sempre que é chamada.

A função *sc_cov::resetLoop* indica o fim daquela execução do laço. Ela é utilizada para determinar quantas iterações do laço foram realizadas a cada execução.

Assim, utilizando-se a forma geral do enunciado *for* apresentada anteriormente e, substituindo-se *CONDICAO* pela expressão lógica $i < 30$, chega-se ao seguinte resultado para o código instrumentado:

```
1 for(INICIO; sc_cov::loopWrapper(id, baseBranchType::forBranch,
2     sc_cov::conditionWrapper(id, 1, i < 30)) ; LACO )
3 {
4    // código a ser executado
5    // enquanto CONDICAO for verdadeira
6
7 } sc_cov::resetLoop(id, baseBranchType::forBranch);
```

5.1.3.6 Instrumenta while

O enunciado *while* permite que um determinado trecho de código seja executado repetidamente, enquanto uma determinada condição for verdadeira. Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, um enunciado *while* interfere na *cobertura total de ramos*, *cobertura total de laços*, *cobertura total de termos* e *cobertura exaustiva de termos*. A sua forma é apresentada a seguir:

```
1 while (CONDICAO)
2 {
3    // código a ser executado enquanto CONDICAO for verdadeira
4 }
```

Da mesma forma que em um enunciado *if*, *CONDICAO* pode ser considerada uma expressão Φ, na forma apresentada na equação 5. Para a instrumentação de um enunciado *while*, utilizam-se três chamadas de função distintas, assim como em um enunciado *for*. Os protótipos das funções são apresentados a seguir:

```
1 bool sc_cov::loopWrapper(unsigned branchId,
2             baseBranchData::branchType type, bool value);
3
4 bool sc_cov::conditionWrapper(unsigned branchId,
5             unsigned conditionId, bool value);
6
7 void sc_cov::resetLoop(unsigned branchId,
8             baseBranchData::branchType type);
```

O argumento *branchId* recebe um número que identifica aquele ramo de forma única no programa. A função *sc_cov::loopWrapper* possui mais dois argumentos. O argumento *type* indica qual o tipo do ramo que está sendo instrumentado, nesse caso receberá o valor *baseBranchData::whileBranch*, que informa que aquele ramo é um *while*. O terceiro argumento *value*, é o valor da condição original do enunciado de controle de fluxo.

No exemplo dado anteriormente, *value* receberá o valor de CONDICAO. A chamada *sc_cov::loopWrapper* retorna o valor de *value* sempre que é chamada.

Assim, utilizando-se a forma geral do enunciado *while* apresentada anteriormente e, substituindo-se *CONDICAO* pela expressão lógica $x == 1 \,||\, b < 3$, chega-se ao seguinte resultado para o código instrumentado:

```
1 while(sc_cov::loopWrapper(id, baseBranchType::whileBranch,
2     sc_cov::conditionWrapper(id, 1, x == 1) ||
3     sc_cov::conditionWrapper(id, 2, b < 3)))
4 {
5    // código a ser executado enquanto CONDICAO for verdadeira
6
7 };
8 sc_cov::resetLoop(id, baseBranchType::whileBranch);
```

5.1.3.7 *Instrumenta* do

O enunciado *do* permite que um determinado trecho de código seja executado repetidamente, enquanto uma determinada condição for verdadeira. Sua diferença em relação ao *while* é que o trecho de código é executado ao menos a primeira vez. Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, um enunciado *do* interfere na *cobertura total de ramos*, *cobertura total de laços*, *cobertura total de termos* e *cobertura exaustiva de termos*. A sua forma é apresentada a seguir:

```
1 do
2 {
3    // código a ser executado enquanto CONDICAO for verdadeira
4 }while (CONDICAO);
```

Da mesma forma que em um enunciado *if*, *CONDICAO* pode ser considerada uma expressão Φ, na forma apresentada na equação 5. Para a instrumentação de um enunciado *do*, utilizam-se três chamadas de função distintas, da mesma forma que nos enunciados *for* e *while*. Os protótipos destas funções são apresentados a seguir:

```
1 bool sc_cov::loopWrapper(unsigned branchId,
2              baseBranchData::branchType type, bool value);
3
4 bool sc_cov::conditionWrapper(unsigned branchId,
5              unsigned conditionId, bool value);
6
7 void sc_cov::resetLoop(unsigned branchId,
8              baseBranchData::branchType type);
```

O argumento *branchId* recebe um número que identifica aquele ramo de forma única no programa. A função *sc_cov::loopWrapper* possui mais dois argumentos. O argumento *type* indica qual o tipo do ramo que está sendo instrumentado, nesse caso receberá o valor *baseBranchData::doBranch*, que informa que aquele ramo é um *do*. O terceiro argumento *value*, é o valor da condição original do enunciado de controle de fluxo. No exemplo dado anteriormente, *value* receberá o valor de CONDICAO. A chamada *sc_cov::loopWrapper* retorna o valor de *value* sempre que é chamada.

Assim, utilizando-se a forma geral do enunciado *do* apresentada anteriormente e, substituindo-se *CONDICAO* pela expressão lógica $x == 1 \mid\mid b < 3$, chega-se ao seguinte resultado para o código instrumentado:

```
1 do
2 {
3    // código a ser executado enquanto CONDICAO for verdadeira
4
5 }while(sc_cov::loopWrapper(id, baseBranchType::doBranch,
6     sc_cov::conditionWrapper(id, 1, x == 1) ||
7     sc_cov::conditionWrapper(id, 2, b < 3)));
8     sc_cov::resetLoop(id, baseBranchType::doBranch);
```

5.1.3.8 Instrumenta switch

O enunciado *switch* permite que um, ou vários, trechos de código sejam executados de acordo com o valor de uma variável. Considerando as definições dos critérios de cobertura, apresentadas na seção 2.4.5.1, um enunciado *switch* interfere na *cobertura total de ramos*. A sua forma é apresentada a seguir:

```
1  switch ( TESTE ) {
2
3    case 1 :
4      // código a ser executado se TESTE = 1
5      ...
6      break;
7
8    case 5 :
9      // código a ser executado se TESTE = 5
10     ...
11     break;
12
13   default :
14     // código a ser executado em todos outros casos.
15     ...
16
17 }
```

Para avaliar um enunciado *switch*, primeiramente, o valor de *TESTE* é calculado. Caso o valor obtido esteja explicitamente descrito em algum dos *cases* do *switch*, o código correspondente é executado. Caso nenhum *case* possua o valor calculado, o código padrão, sob o descritor *default*, é executado. Quando um *case* é identificado, seu código é executado até que uma chamada *break* seja encontrada. Se não houver chamada *break*, cada um dos próximos *cases* é executado até que uma chamada *break* ou o fim do *switch* seja atingido, o que acontecer primeiro.

O *switch* é um enunciado que permite que o fluxo do programa se divida em vários ramos, sendo cada *case* um possível ramo de execução. Para verificar a cobertura atingida por um *switch*, instrumenta-se cada um de seus *cases*. O protótipo da função instrumentada é mostrado a seguir:

```
1  bool sc_cov::switchWrapper(unsigned branchId);
```

O argumento *branchId* é um número que identifica aquele ramo de forma única no programa.

Assim, utilizando-se a forma geral do enunciado *switch* apresentada anteriormente, chega-se ao seguinte resultado para o código instrumentado:

```
1  switch ( TESTE ) {
2
3    case 1 : sc_cov::switchWrapper(id_1);
4      // código a ser executado se TESTE = 1
5      ...
6      break;
7
8    case 5 : sc_cov::switchWrapper(id_2);
9      // código a ser executado se TESTE = 5
10     ...
11     break;
12
13   default : sc_cov::switchWrapper(id_3);
14     // código a ser executado em todos outros casos.
15     ...
16
17 }
```

O identificador do ramo é diferente para cada um dos *cases* do programa. A utilização de *cases* com condições iguais em diferentes *switches* ao longo do programa não interfere na instrumentação de código, uma vez que cada *case* recebe um identificador que depende de sua posição no programa.

5.1.3.9 Nota Sobre Operadores de Incremento, Decremento e Funções em Testes de Condição

O padrão C++ permite que operandos sejam modificados ao mesmo tempo em que seus valores são testados. Esse é o caso dos operadores de incremento ($++$), decremento ($--$) e funções em condições e termos de um enunciado condicional de controle de fluxo. Esse é um obstáculo para a instrumentação do código, uma vez que não é possível replicar o trecho do código que contém operações do tipo sem que se altere o funcionamento do código instrumentado.

Esse problema foi contornado na ferramenta ScCov por meio da utilização da função *sc_cov::conditionWrapper*, que armazena o valor de cada um dos termos presentes na condição original do prefixo do enunciado condicional de controle de fluxo para que possa ser utilizada em várias operações internas da ferramenta sem que seja necessária replicação de código. Com essa abordagem, operações que testam e modificam o valor dos operandos ao mesmo tempo não impedem o emprego da ferramenta.

Tabela 4: Exemplo Seqüência de Valores para os Termos do Enunciado *if*

Número da Execução	Resultado da Operação	$x == 1$	$b < 3$
1	FALSE	FALSE	FALSE
2	TRUE	TRUE	FALSE
3	TRUE	TRUE	TRUE

Porém, os operadores lógicos do C++ possuem a característica de antecipar o valor do resultado da operação lógica sem testar todos os termos, desde que o valor da saída seja definido pelos termos já verificados. Considere o trecho de código do exemplo a seguir:

```
1 if (x == 1 || b < 3)
2 {
3     // código a ser executado
4 }
```

Nesse exemplo, caso $x == 1$ seja verdadeiro (*TRUE*), o termo $b < 3$ não será testado pois somente o primeiro termo determina que o valor do resultado da operação lógica seja verdadeiro (*TRUE*). Considerando o código instrumentado da seção 5.1.3.4, pode ser verificado que a função *sc_cov::conditionWrapper* não será executada para o segundo termo e, portanto, não receberá o valor do teste $b < 3$. Apesar de não causar erro durante a execução da ferramenta, essa característica pode gerar resultados equivocados nos relatórios de cobertura de termos e cobertura exaustiva de termos.

Considere a tabela 4 onde é apresentado um exemplo com uma seqüência de valores para cada um dos termos do enunciado *if* anterior. Nela, verifica-se que ocorreu cobertura total de termos, pois cada um dos termos assumiu, individualmente, os valores verdadeiro (*TRUE*) e falso (*FALSE*). Para ocorrer *cobertura exaustiva de termos*, faltou apenas uma combinação de valores, onde o primeiro termo é falso (*FALSE*) e o segundo é verdadeiro (*TRUE*). Levando-se em conta a característica dos operadores lógicos do C++, o segundo termo da operação lógica somente será testado na primeira execução do trecho de código, pois na segunda e terceira execuções, o valor da operação lógica é determinado apenas pelo primeiro termo cujo resultado é verdadeiro (*TRUE*).

Como o segundo termo foi executado somente na primeira linha da tabela, para a biblioteca de cobertura ele somente foi executado com o valor falso (*FALSE*). Dessa forma,

o relatório de cobertura de termos acusará, erroneamente, que não houve *cobertura total de termos* e o relatório de cobertura exaustiva de termos acusará que duas combinações não foram testadas, desconsiderando a execução da linha 3 da tabela 4.

Um modo de contornar esse problema é substituir os operadores lógicos *&&* (*and*) e || (*or*) das condições pelos operadores de *bit &* (*bitand*) e | (*bitor*) respectivamente. Nesses operadores, todos os termos são testados, não importa o resultado dos testes anteriores. Essa substituição, no entanto, não deve ser feita de modo arbitrário. Substituir operadores lógicos por operadores de *bit* pode resultar em falhas catastróficas de execução em alguns programas. Considere o trecho de código abaixo:

```
1 if (vetor.size() >= 30 && vetor[29] == 5)
2 {
3     // código a ser executado
4 }
```

Nesse caso, somente se o tamanho do vetor for maior ou igual a 30 (trinta), o segundo termo será executado, o que evita uma falha de segmentação do segundo teste. Se o operador lógico desse trecho for substituído por um operador de *bit*, o programa pode resultar em falha de segmentação, tentando alcançar uma posição do vetor que não existe.

Dessa forma, a decisão pela substituição dos operadores lógicos por operadores de *bit* foi deixada a cargo do usuário da ferramenta, e pode ser diferente para cada projeto. Como situações onde o teste de todas as condições pode ser prejudicial são incomuns em projetos de *hardware*, o padrão da ferramenta é substituir operadores lógicos por operadores de *bit*.

5.1.3.10 Instrumenta operadores relacionais

Um erro comum em programas é o emprego de um operador de “>” ou “<” onde deveria ser utilizado um “>=” ou “<=”, ou vice-versa. Para verificar se o operador foi escolhido de forma correta, é necessário que se teste a condição de igualdade entre os dois operandos, além do teste das condições de verdadeiro (*TRUE*) e falso (*FALSE*) da comparação. Uma vez testada a condição de igualdade, pode-se assegurar que o operador foi empregado de forma correta.

Para que essa cobertura seja atingida é necessário que dois testes sejam feitos: o teste

original, com o operador utilizado pelo projetista, e um teste com o operador de igualdade (==). Trata-se do mesmo problema descrito no item 5.1.3.9, pois seria necessário uma replicação do código com a substituição dos operadores. Para resolver o problema, a solução apresentada anteriormente foi utilizar o valor dos operandos no lugar dos próprios operandos. Uma função *wrapper* foi construída para receber o valor dos operandos, tratar esse valor, e devolvê-lo para a execução normal do programa. Porém, o problema apresentado em 5.1.3.9 é mais simples do que o problema descrito aqui, pois, por tratar expressões lógicas, possui apenas operandos booleanos. No caso do teste de relação, os operandos podem ter qualquer tipo básico do C++, além de tipos definidos pelo usuário. Construir uma função *wrapper* diferente para cada tipo de operando é uma tarefa impossível sem a utilização de conceitos mais avançados da linguagem C++.

A solução para o problema foi a utilização de funções *template*. Assim como a herança e o polimorfismo são modos de reutilização de código objeto, os *templates* são uma forma de reutilização de código fonte (Eckel, 2000). Pode-se definir uma classe ou função onde um ou mais parâmetros não são resolvidos em tempo de projeto, mas em tempo de compilação. A partir da utilização de *templates*, é possível resolver o problema da replicação de código nos testes de operadores relacionais com a criação de apenas cinco funções. Abaixo, estão representadas três destas funções. As funções para os operadores "<" e "<=" foram omitidas, pois são semelhantes às apresentadas.

```
1  static bool sc_cov::relation(unsigned relationId,
2                               bool trueCondition,
3                               bool equalCondition);
4
5  template <class C1, class C2>
6  static bool sc_cov::gtRelationWrapper(unsigned relationId,
7                                        const C1& p1, const C2& p2)
8  {
9    return relation(relationId, p1 > p2, p1 == p2);
10 };
11
12 template <class C1, class C2>
13 static bool sc_cov::geRelationWrapper(unsigned relationId,
14                                        const C1& p1, const C2& p2)
15 {
16   return relation(relationId, p1 >= p2, p1 == p2);
17 };
```

Onde o argumento *relationId* identifica de forma única a relação no projeto SystemC. As funções *gtRelationWrapper*, *geRelationWrapper*, *ltRelationWrapper* e *leRelationWrapper* representam, respectivamente, os operadores >, >=, < e <=.

Pode ser verificado que, os tipos dos argumentos *p1* e *p2* não foram especificados em nenhuma das funções, sendo deixados para serem resolvidos pelo compilador. Quando encontra chamadas à funções *template*, o compilador cria uma nova definição da função substituindo *C1* e *C2* pelos tipos adequados.

Como exemplo, considere o trecho de código abaixo:

```
1 if (b < 3)
2 {
3   // código a ser executado
4 }
```

A instrumentação completa desse trecho de código, incluindo *wrappers* para análise cobertura de ramos, termos, exaustiva de termos e relacional é apresentada a seguir:

```
1 if (sc_cov::branchWrapper(id, baseBranchData::ifBranch,
2     sc_cov::conditionWrapper(id, 1,
3     sc_cov::ltRelationWrapper(r_id, b, 3))))
4 {
5   // código a ser executado
6 }
```

Onde *id* e *r_id* são, respectivamente, os identificadores de ramo e de relação.

5.1.4 Servidor de Dados e Interface de Usuário

Os dados recolhidos pelo *analisador léxico* são enviados para o *servidor de dados*, que é responsável por armazenar e disponibilizar os dados coletados por todos os outros componentes do sistema. A utilização de um servidor de dados independente permite uma gerência mais centralizada dos testes e garante uma maior flexibilidade na utilização dos recursos disponíveis, pois diversas instâncias de modelos executáveis podem ser verificados ao mesmo tempo. As seguintes características são suportadas pelo *servidor de dados*:

- **Comunicação por meio de *TCP/IP*** - Permite que o servidor esteja em uma estação diferente do cliente;

- **Suporte a várias conexões simultâneas** - O sistema foi desenvolvido utilizando conceitos de *multithreading* e implementando mecanismos de exclusão mútua que permitem que vários clientes coletem e armazenem dados no servidor simultaneamente;

- **Dados separados por projeto** - Os dados são divididos, internamente, por projeto. Por este motivo, testes de módulos diferentes podem ser executados ao mesmo tempo, desde que esses módulos sejam configurados como projetos diferentes;

- **Dados salvos em arquivo** - Os dados armazenados no servidor podem ser gravados em um arquivo para que sejam utilizados no futuro, ou para que o teste prossiga do ponto onde foi interrompido;

- **Dados acumuláveis** - Diversas instâncias de um modelo podem ser executadas ao mesmo tempo, atualizando os mesmos parâmetros. Os dados serão acumulados, sem que seja necessária a criação de uma entrada diferente para cada execução. Isso garante que os relatórios serão gerados corretamente sem que o tamanho dos arquivos de dados cresça a cada execução do modelo.

O usuário conecta-se ao *servidor de dados* utilizando a *Interface de Usuário*. A *Interface de Usuário* possui opções para a instrumentação do código fonte do projeto SystemC e para a geração dos relatórios de cobertura, descritos a seguir.

5.1.5 Relatórios de Cobertura

Os relatórios de cobertura são o resultado da ferramenta ScCov. O objetivo dos relatórios é fornecer para o projetista uma visão detalhada da qualidade dos testes aplicados ao modelo do sistema descrito em SystemC. São 6 (seis) as possibilidades para relatórios:

1. Cobertura de Funções;

2. Cobertura de Ramos;

3. Cobertura de Termos;

4. Cobertura Exaustiva de Termos;

5. Cobertura Relacional;

6. Cobertura de Laços.

Os relatórios apresentam valores para as métricas de cobertura aplicadas sobre os critérios definidos na seção 2.4.5.1. Cada relatório possui um resumo, no qual são destacados a quantidade de unidades de cobertura existentes no sistema, o número de unidades de cobertura exercitadas pelos testes aplicados e a taxa de cobertura atingida, em termos de porcentagem das unidades de cobertura existentes no sistema. Os relatórios também apresentam informações exatas sobre a localização das unidades de cobertura não exercitadas no código fonte original do sistema, não sendo necessário que o projetista entre em contato, em nenhum momento, com o código fonte instrumentado.

Todos os relatórios são gerados utilizando o padrão XML. Como o XML é um padrão bem difundido, diversas ferramentas existem para acessar seus dados. Nesse trabalho, escolheu-se utilizar o acesso ao arquivo XML por meio de folhas de estilo XSLT. As folhas de estilo fornecem regras para criar uma visualização do arquivo XML que pode ser acessada em um navegador WEB comum. Folhas de estilo padrão são fornecidas juntamente com a ferramenta, mas o usuário pode modificá-las para atender aos seus requisitos de visualização e estética.

A seguir, cada um dos relatórios será detalhado.

5.1.5.1 Relatório de Cobertura de Funções

O relatório de cobertura de funções indica quais funções não foram exercitadas durante os testes. O resumo do relatório contém o número de funções presentes no sistema, o número de funções exercitadas pelo teste e a taxa de cobertura atingida. A figura 29 mostra um exemplo de um relatório de cobertura de funções. Verifica-se que, de um total de 8 funções, 7 foram cobertas neste exemplo, resultando em uma cobertura de 87,50%. A data e hora de geração do relatório também são mostradas para um registro mais completo.

Abaixo do resumo, está a relação das funções não cobertas. Um quadro indicativo mostra o arquivo e a linha onde se encontra a definição da função e, ao lado do quadro, o

RESUMO

Número de Funções: 8
Funções Cobertas: 7
Cobertura Atingida: 87.50 %

Relatório Gerado em: Sat Jan 26 09:46:27 2008

FUNÇÕES NÃO COBERTAS

Arquivo: ula.cpp	**Linha:** 42

```
double ula::sum(double a, double b)
{
    return a + b;
}
```

Figura 29: Relatório de Cobertura de Funções

corpo da função é apresentado para consultas.

5.1.5.2 Relatório de Cobertura de Ramos

O relatório de cobertura de ramos relaciona os ramos que não foram cobertos completamente durante os testes. Para que um ramo seja coberto completamente, ele precisa ser executado tanto em seu caminho caminho verdadeiro (*TRUE*), quanto no caminho falso (*FALSE*). Um exemplo para este relatório é apresentado no anexo A.1.

5.1.5.3 Relatório de cobertura de termos

O relatório de cobertura de termos relaciona cada um dos termos presentes nos prefixos dos enunciados de controle de fluxo que não foram cobertos completamente pelos testes. Para que um termo seja coberto completamente, ele precisa ser executado com valor verdadeiro (*TRUE*) e também com valor falso (*FALSE*). Um exemplo para este relatório é apresentado no anexo A.2.

5.1.5.4 Relatório de Cobertura Exaustiva de Termos

O relatório de cobertura exaustiva de termos relaciona cada uma das combinações possíveis para os termos presentes nos prefixos dos enunciados de controle de fluxo que

não foram cobertas pelos testes. Um exemplo para este relatório é apresentado no anexo A.3.

5.1.5.5 Relatório de Cobertura Relacional

O relatório de cobertura relacional apresenta todas as relações presentes nos prefixos dos enunciados de controle de fluxo que não foram cobertas totalmente pelos testes. O objetivo deste relatório é indicar relações onde operadores “>” e “<” podem ter sido usados no lugar de “>=” e “<=”, ou vice-versa. Para que seja atingida uma cobertura completa, a relação deve ser testada na sua forma verdadeira (*TRUE*), na forma falsa (*FALSE*) e no limite entre a forma verdadeira e a forma falsa que é a igualdade entre os termos da relação. Um exemplo para este relatório é apresentado no anexo A.4.

5.1.5.6 Relatório de Cobertura de Laços

O relatório de cobertura de laços relaciona todos os laços presentes no projeto que não foram cobertos pelos testes. Para que um laço seja totalmente coberto, ele precisa ser iterado uma única vez, várias vezes, e também não ser iterado durante os testes aplicados. Um exemplo para este relatório é apresentado no anexo A.5.

6 RESULTADOS DA FERRAMENTA SCCOV

Neste capítulo, serão apresentados resultados do emprego da ferramenta proposta no capítulo 5 em três módulos descritos em SystemC. O primeiro, uma unidade aritmética simples para cálculo de operações matemáticas e do máximo divisor comum entre dois números inteiros, descrito no nível de *Especificação Executável*. O segundo módulo, uma CPU RISC, desenvolvida pela Synopsys e fornecida como exemplo com o SystemC, descrita no nível *Comportamental*. Por último, um módulo para controle de um LCD descrito no nível RTL. Os sistemas serão analisados de forma a assegurar que a ferramenta atende aos requisitos apresentados no capítulo 5 e para mensurar o impacto das chamadas instrumentadas no desempenho da simulação dos modelos.

Primeiramente, será apresentada a metodologia de cada um dos experimentos realizados para coleta dos resultados. Em seguida, cada um dos sistemas será descrito, apresentando-se informações sobre a estrutura interna do código fonte, e os resultados obtidos com os experimentos. Uma análise dos resultados obtidos é apresentada ao fim do capítulo.

6.1 Metodologia

O primeiro objetivo dos experimentos apresentados é a validação da ferramenta, determinando se ela atende aos requisitos do capítulo 5. Como objetivo secundário, pretende-se avaliar qual o impacto da aplicação da ferramenta no desempenho da simulação dos modelos. Abaixo, os requisitos do capítulo 5 são novamente apresentados para consulta:

1. A ferramenta não deve requerer nenhuma modificação no código fonte do compila-

dor GCC nem no código fonte das bibliotecas do SystemC;

2. A ferramenta deve poder ser aplicada a qualquer código SystemC, sem que seja necessária a utilização de estilos de código específicos;

3. A ferramenta deve aplicar os conceitos de *cobertura total de ramos*, *cobertura total de laços*, *cobertura total de termos*, *cobertura exaustiva de termos*, *cobertura relacional total* e *cobertura total de funções*, conforme descritas na seção 2.4;

4. A ferramenta deve fornecer informações exatas sobre a cobertura do código fonte, referenciando o código original do sistema antes da instrumentação;

5. Os resultados obtidos pelo sistema original devem ser equivalentes aos obtidos pelo sistema depois da aplicação da ferramenta.

Quatro experimentos foram realizados. O primeiro destina-se a verificar se a ferramenta atende ao requisito 4. O segundo experimento é destinado a verificar se o requisito 3 foi satisfeito. Apesar de a ferramenta fornecer relatórios sobre os tipos de cobertura relacionados, deve-se verificar se esses relatórios foram gerados de maneira correta. O terceiro experimento determina se o requisito 5 foi atendido. O quarto experimento mede a diferença de desempenho entre a descrição original e a descrição instrumentado.

Não será necessária a elaboração de experimentos para o requisito 1, uma vez que este requisito refere-se ao desenvolvimento e não à execução da ferramenta em si. Também não foi elaborado um experimento para o requisito 2, pois espera-se que este requisito seja verificado como um efeito colateral da aplicação da ferramenta nos outros experimentos. Como os sistemas utilizados como exemplo estão descritos em níveis de abstração diferentes e foram desenvolvidos por grupos diferentes, espera-se que eles contemplem uma grande variedade de estilos de programação.

6.1.1 Descrição do Experimento 1

Este experimento destina-se a verificar se a ferramenta fornece informações exatas sobre o código fonte do sistema nos relatórios de cobertura, referenciando o código ori-

ginal antes da instrumentação. Para a execução do experimento, os seguintes passos são utilizados:

1. Aplica-se a ferramenta ao código fonte do sistema;

2. Gera-se todos os relatórios de cobertura, sem que o modelo executável com o código instrumentado seja executado;

3. Verifica-se, para cada um dos relatórios, se todos os enumeráveis presentes no código fonte original estão presentes nos relatórios e se as indicações de *nome de arquivo* e *linha* estão corretas.

Uma vez que o modelo executável do sistema não foi executado, espera-se que os relatórios contenham todos os enumeráveis (ramos no relatório de ramos, funções no relatório de funções, etc.) presentes no código fonte original. A verificação é feita por inspeção visual do código fonte e do relatório.

6.1.2 Descrição do Experimento 2

Este experimento destina-se a verificar se a ferramenta aplica o conceito dos diferentes tipos de cobertura de código, apresentadas no item 2.4.5.1, de forma correta. Para a execução do experimento, os seguintes passos são realizados:

1. Vários conjuntos de teste são gerados;

2. Por inspeção do código, determina-se quais enumeráveis de cada relatório serão cobertos por cada conjunto de teste;

3. Aplica-se cada conjunto de teste em separado ao modelo executável do sistema;

4. Verifica-se se o relatório de cobertura indica que os enumeráveis relacionados por inspeção foram cobertos pelo conjunto de teste;

5. Entre as aplicações de cada um dos conjuntos de teste, apaga-se os dados de cobertura do servidor de dados;

Em sistemas complexos, como é o caso da CPU RISC da Synopsys e do Controlador de LCD, a inspeção visual para casos de teste é impraticável, havendo uma cobertura de vários enumeráveis mesmo com a aplicação dos mais simples vetores de entrada. Por este motivo, este experimento será realizado somente para Unidade Aritmética Simples, pois ela oferece um ambiente mais controlável e passível de se verificar se as estruturas de dados internas da ferramenta são atualizadas corretamente durante a execução do modelo instrumentado.

6.1.3 Descrição do Experimento 3

Este experimento destina-se a verificar se os resultados apresentados pelo modelo executável instrumentado são equivalentes aos obtidos pelo sistema original. Para a execução deste teste, utiliza-se o modelo do *testbench* apresentado na figura 12, onde o *Modelo de Referência* é substituído pelo modelo executável original do sistema e o *Sistema sob Verificação* é substituído pelo modelo executável instrumentado. Um conjunto de teste é gerado e aplicado ao *testbench*. Espera-se que o resultado dos dois sistemas sejam equivalentes.

6.1.4 Descrição do Experimento 4

Este experimento destina-se a analisar o impacto da instrumentação do código no desempenho dos sistemas. Para isso, conjuntos de teste de tamanhos diferentes são gerados e aplicados ao modelo original do sistema e ao modelo instrumentado. Dentro de cada um dos modelos, é inserida uma chamada para se medir o tempo de execução, como mostrado abaixo:

```
1  int sc_main(int ac, char *av[])
2  {
3    clock_t t1=clock();
4
5      // código do sc_main
6
7    clock_t t2=clock();
8    std::cout << "Time␣for␣simulation␣=␣"
9         << double(t2-t1) / CLOCKS_PER_SEC
10        << "␣seconds" << std::endl;
11
12   return 0;
13 }
```

Tabela 5: Enumeráveis da UA

Linhas de Código	Funções	Ramos	Termos	Combinações de Termos	Relações	Laços
86	8	29	31	31	3	6

Com isso, ao final da execução, obtém-se o tempo gasto por cada modelo para cada um dos conjuntos de teste aplicados e pode-se determinar a alteração na ordem de complexidade do sistema.

6.2 Unidade Aritmética Simples

A unidade aritmética utilizada como exemplo é apresentada na figura 30.

Figura 30: Unidade Aritmética Simples

Trata-se de um sistema combinatório descrito em nível de *Especificação Executável*, com três entradas e uma saída. A entrada *ctl* determina qual operação deve ser efetuada sobre os operandos das entradas *A* e *B*. O resultado é apresentado na saída *result*. Possíveis operações são *soma*, *subtração*, *multiplicação*, *divisão*, *potenciação* (*A* elevado à *B-ésima* potência) e o máximo divisor comum entre *A* e *B*.

A tabela 5 mostra a relação dos enumeráveis presentes na UA. Para as colunas *Relações* e *Laços*, o total de métricas pode ser determinada multiplicando-se o valor apresentado por 3, conforme descrito no capítulo 5.

6.2.1 Resultados do Experimento 1

Todos os enumeráveis presentes no código fonte estavam relacionados nos relatórios correspondentes. As referências ao código fonte (nome e linha do arquivo de origem) estavam corretas. A tabela contendo o código fonte do enumerável, quando aplicável,

Tabela 6: Vetores de Teste Aplicados no Experimento 2

Execução	*ctl*	*A*	*B*
1	1	100	15
2	2	5	-2
3	3	10	9
4	4	-9	3
5	4	28	0
6	5	-5	2
7	5	9	-3
8	6	81	-27
9	6	12	0
10	6	0	256
11	6	0	0

estava corretamente apresentada no relatório.

6.2.2 Resultados do Experimento 2

Neste experimento, os 11 (onze) vetores de teste apresentados na tabela 6 foram aplicados às entradas da UA. Estes vetores foram escolhidos para que todas as métricas de cobertura fossem testadas e todos os *wrappers* instrumentados no código fonte fossem executados. Ao final do teste, foi verificado que os relatórios comportaram-se da forma esperada, listando corretamente os enumeráveis cobertos.

6.2.3 Resultados do Experimento 3

Para a execução deste experimento, foram gerados 100.000 (cem mil) estímulos aleatórios para a UA. Mesmo sendo um sistema extremamente simples, notou-se que este conjunto de teste não foi capaz de atingir 100% da cobertura alcançável do modelo. Foi necessário gerar 4 (quatro) vetores por meio de inspeção do código para que todos os enumeráveis alcançáveis fossem cobertos.

Por enumeráveis alcançáveis, entende-se os enumeráveis que são possíveis de serem cobertos. Um exemplo de enumerável inalcançável é um laço *while* cuja condição de teste seja *TRUE*. Neste caso, algumas coberturas não são aplicáveis.

Depois de todo o código coberto, foi realizada a comparação entre o modelo original e o instrumentado, por meio de um *testbench*. Os modelos responderam de forma equi-

valente a todos os estímulos aplicados. Os modelos são, portanto, equivalentes dentro da metodologia proposta para o experimento.

6.2.4 Resultados do Experimento 4

Para medir o impacto da instrumentação de código no desempenho da simulação da Unidade Aritmética, foram gerados 12 conjuntos de estímulos com tamanhos variados. Os valores foram gerados aleatoriamente dentro da faixa permitida por cada uma das entradas do modelo. Para cada um dos conjuntos gerados, foi medido o tempo gasto para se simular todos os vetores, tanto para o modelo original quanto para o modelo instrumentado. Os resultados são apresentados no gráfico da figura 31, que mostra a relação *Tempo de Execução* X *Número de Estímulos*.

Figura 31: Impacto no Desempenho da Unidade Aritmética

A reta marcada com os círculos foi obtida a partir dos resultados colhidos da execução do modelo original. Nota-se um aumento na inclinação da reta *Tempo de Execução* X *Número de Estímulos*, o que indica um aumento no tempo de execução, mas não uma alteração da ordem de complexidade do sistema, que continua sendo linear. Neste caso, o aumento do tempo de execução da simulação foi da ordem de 16 (dezesseis) vezes depois que o código foi instrumentado.

Tabela 7: Enumeráveis da CPU RISC da Synopsys

Linhas de Código	Funções	Ramos	Termos	Combinações de Termos	Relações	Laços
1690	14	258	280	520	37	20

6.3 CPU RISC da Synopsys

Um diagrama de blocos da descrição SystemC da CPU RISC desenvolvida pela Synopsys e utilizada como como exemplo é apresentada na figura 32.

Figura 32: CPU RISC da Synopsys

A tabela 7 mostra a relação dos enumeráveis presentes na CPU RISC. Para as colunas *Relações* e *Laços*, o total de métricas pode ser determinada multiplicando-se o valor apresentado por 3, conforme descrito no capítulo 5.

6.3.1 Resultados do Experimento 1

Todos os enumeráveis presentes no código fonte estavam relacionados nos relatórios correspondentes. As referências ao código fonte (nome e linha do arquivo de origem) estavam corretas. A tabela contendo o código fonte do enumerável, quando aplicável, estava corretamente apresentada no relatório.

6.3.2 Resultados do Experimento 2

O experimento 2 não foi aplicado a este sistema devido a complexidade da inspeção visual dos enumeráveis cobertos, mesmo para casos de teste simples.

6.3.3 Resultados do Experimento 3

Para a execução deste experimento, foram gerados 2 programas diferentes para a CPU RISC. Estes programas atingiram uma cobertura de 74% dos enumeráveis dos relatórios de cobertura de ramos e termos. No relatório de cobertura exaustiva de termos, foi atingida apenas 40% de cobertura. Porém, foi verificado, por inspeção de código, que muitas das combinações presentes nas condições eram impossíveis de serem atingidas. Recalculando a cobertura, ignorando-se essas combinações impossíveis, obteve-se um resultado de 74% da cobertura atingível. O relatório de cobertura relacional apresentou 90% de cobertura. O relatório de cobertura de funções alcançou 100% de cobertura. Porém, o relatório de cobertura de laços mostrou que apenas 33.33% das métricas foram alcançadas, ou apenas um terço do total.

Excetuando-se a cobertura de laços, a cobertura atingida pelos programas foi considerada satisfatória para se executar os testes entre o sistema original e o sistema instrumentado. Os modelos responderam de forma equivalente aos programas aplicados e, portanto, são equivalentes dentro da metodologia proposta para o experimento.

Quanto ao valor baixo para a cobertura de laços, verificou-se que os laços são iterados sempre da mesma forma em todas as execuções do programa. Como o total de métricas para um laço é 3 (iterado uma vez, várias vezes e não iterado), somente um terço das métricas foram cobertas, ou seja, uma por laço. Este foi o motivo para o valor baixo desta cobertura.

6.3.4 Resultados do Experimento 4

Para medir o impacto da instrumentação de código no desempenho da simulação da CPU RISC da Synopsys, foram gerados 13 programas. A alteração do número de instruções executadas se deu por uma variação do número de iterações do laço principal do programa. Para cada um dos programas, foi medido o tempo gasto para a execução da simulação, tanto para o modelo original quanto para o modelo instrumentado. Os resultados são apresentados no gráfico da figura 31, que mostra a relação *Tempo de Execução* X *Número de Instruções*.

Figura 33: Impacto no Desempenho da CPU RISC da Synopsys

A reta marcada com os círculos foi obtida a partir dos resultados colhidos da execução do modelo original. Nota-se um aumento na inclinação da reta *Tempo de Execução* X *Número de Instruções*, o que indica um aumento no tempo de execução, mas não uma alteração da ordem de complexidade do sistema, que continua sendo linear. Neste caso, o aumento do tempo de execução da simulação foi da ordem de 7 (sete) vezes depois que o código foi instrumentado.

6.4 Controlador de LCD

Um diagrama de blocos do controlador LCD utilizado como como exemplo é apresentado na figura 34.

A tabela 8 mostra a relação dos enumeráveis presentes no Controlador de LCD. Para as colunas *Relações* e *Laços*, o total de métricas pode ser determinada multiplicando-se o valor apresentado por 3, conforme descrito no capítulo 5. O único laço presente no sistema é para a impressão de valores na tela para fins de depuração.

Figura 34: Controlador de LCD

Tabela 8: Enumeráveis do Controlador de LCD

Linhas de Código	Funções	Ramos	Termos	Combinações de Termos	Relações	Laços
622	10	141	155	157	10	1

6.4.1 Resultados do Experimento 1

Todos os enumeráveis presentes no código fonte estavam relacionados nos relatórios correspondentes. As referências ao código fonte (nome e linha do arquivo de origem) estavam corretas. A tabela contendo o código fonte do enumerável, quando aplicável, estava corretamente apresentada no relatório.

6.4.2 Resultados do Experimento 2

O experimento 2 não foi aplicado a este sistema devido a complexidade da inspeção visual dos enumeráveis cobertos, mesmo para casos de teste simples.

6.4.3 Resultados do Experimento 3

Para a execução deste experimento, utilizou-se um arquivo com uma seqüência de comandos para construção da imagem apresentada na figura 35. O arquivo utilizado atinge mais de 92% dos enumeráveis dos relatórios, o que representa uma cobertura significativa.

Este arquivo foi aplicado tanto ao modelo original, quanto ao modelo instrumentado. Utilizando-se o *testbench*, foi possível verificar que os modelos responderam de forma

Figura 35: Imagem Reconstruída pelo Controlador de LCD

Tabela 9: Impacto no Desempenho da Simulação do Controlador de LCD

	Dimensão do Quadrado da Imagem (pixels)								
	10	20	30	40	50	60	70	80	90
Original (s)	6,11	6,24	6,36	6,53	6,66	6,80	6,93	7,04	7,15
Instrumentado (s)	236	233	235	237	237	235	238	235	237
Impacto (%)	3862	3734	3694	3630	3559	3456	3449	3338	3314

equivalente aos arquivos de comando aplicados. Os modelos são, portanto, equivalentes dentro da metodologia proposta para o experimento.

6.4.4 Resultados do Experimento 4

Por estar descrito no nível RTL, o módulo do LCD possui um tempo de simulação fixo, de 65 ms. Mesmo assim, foram aplicados 9 diferentes vetores de teste para a construção de imagens de diferentes tamanhos. Os resultados destes experimentos são apresentados na tabela 9. A primeira linha da tabela indica o tempo gasto para a execução do modelo original do sistema e a segunda linha o tempo gasto depois de instrumentado. Uma terceira linha foi inserida, onde é indicado o número de vezes que o tempo de execução do modelo aumentou depois da instrumentação de código.

Por ter um tempo de simulação fixo, o tamanho da imagem tem pouca influência no tempo de execução do sistema, principalmente para o tempo de execução do modelo ins-

trumentado. Observando-se o valor do impacto para as várias imagens, verifica-se um decréscimo na relação entre o tempo da execução da simulação da descrição instrumentado e o tempo da descrição original, partindo da ordem de 38 vezes para a menor imagem e chegando a 33 vezes para a imagem maior.

A explicação para isso está no modo como funciona o Controlador de LCD. Após ter recebido os comandos por meio da *NOC*, a imagem já foi construída na memória interna do controlador. Com a imagem construída, somente é necessário enviar sinais para o LCD indicando como cada um dos *pixels* deve ser mostrado, o que requer poucos cálculos por parte do modelo. O sistema passa a executar mais enunciados de controle de fluxo do que enunciados de computação, passando a ser quase que totalmente orientado a controle. Como a instrumentação do código é realizada principalmente sobre os enunciados de controle de fluxo, a densidade de chamadas a funções instrumentadas é maior para imagens pequenas, que gastam menos tempo de simulação para a reconstrução da imagem. Por este motivo, em imagens maiores, onde o número de cálculos para reconstrução da imagem é maior, a relação decresce.

6.5 Análise dos Resultados dos Experimentos

Com os experimentos, verificou-se que a ferramenta atende aos requisitos determinados no capítulo 5, aplicando corretamente os conceitos de análise de cobertura de código. Deve-se ressaltar que os resultados apresentados aqui foram obtidos utilizando-se uma versão já verificada da ferramenta, onde os erros de implementação já haviam sido removidos.

Uma elevação do tempo de execução do modelo foi observada. Esse aumento era esperado, uma vez que chamadas de função são inseridas em cada enunciado de controle de fluxo do programa. Os dados são enviados para o servidor somente uma vez, ao fim do teste. Isto evita que perca-se tempo com envio de informações para o servidor de dados a cada chamada de uma função instrumentada. Portanto, o aumento no tempo de execução é quase somente devido ao tempo gasto para a execução das funções instrumentadas mais um valor constante de tempo ao fim de cada execução do modelo, onde os dados

são enviados para o servidor. A elevação do tempo de execução está relacionada com o número de enunciados de controle por linha de código executada durante a simulação. Porém, não é trivial deduzir um único modelo para a elevação do tempo de execução do sistema, pois o tempo pode variar dependendo do caminho do código fonte percorrido a cada simulação. A dedução de tal modelo está além dos objetivos deste trabalho e não parece oferecer material para pesquisas futuras.

Uma observação dos resultados obtidos mostra que o relatório de cobertura de laços não fornece informações significativas para projetos de SoCs, principalmente em níveis mais baixos de abstração, onde os laços são quase completamente inexistentes. Neste tipo de sistema, a maioria dos laços são implementados de forma a executarem um número determinado de vezes, o que sempre resulta na cobertura de uma métrica, *várias iterações*. Este relatório de cobertura foi mantido na ferramenta pois ele pode ser de alguma utilidade em modelos descritos em altos níveis de abstração.

7 CONCLUSÕES

Neste trabalho, foram analisadas técnicas e métodos que permitem um maior controle sobre a qualidade do ambiente de verificação em projetos de SoCs descritos em SystemC. No capítulo 3, foi apresentada uma metodologia híbrida, que combina técnicas funcionais e estruturais de forma a aumentar a eficiência da verificação. A proposta da metodologia é analisar a cobertura estrutural atingida pelos testes funcionais e, a partir desta análise, gerar vetores de teste que permitam que as partes exercitadas de modo ineficiente pelos testes funcionais sejam cobertas. Combinando estes dois tipos de técnicas, a metodologia não sofre dos problemas relacionados à aplicação somente de técnicas estruturais, que são incapazes de detectar erros de omissão, nem dos problemas relacionados às técnicas funcionais, que não permitem um controle da cobertura da estrutura do sistema atingida pelos testes aplicados.

Durante o desenvolvimento do trabalho, foi verificada uma carência de ferramentas e métodos que ofereçam suporte à automação dos passos da metodologia, o que cria um obstáculo para a sua validação em sistemas reais. Esta carência leva à segunda contribuição deste trabalho, que é o desenvolvimento de uma ferramenta capaz de analisar a cobertura de código atingida pelo ambiente de verificação. Utilizando de instrumentação de código fonte, a ferramenta permite que unidades de código, como ramos, sejam monitoradas durante a execução do modelo, verificando se foram executadas de modo eficiente pelos vetores de teste aplicados.

A ferramenta apresentada foi aplicada em três sistemas desenvolvidos utilizando SystemC e descritos em diferentes níveis de abstração. Foram realizados experimentos que

verificaram que foram atendidos os requisitos de projeto, determinados durante a sua idealização. Como a ferramenta insere construções no código fonte do sistema sob verificação, também foram realizados experimentos para verificar qual o impacto da sua utilização no tempo de execução dos modelos executáveis dos sistemas. A análise dos resultados mostrou uma elevação no tempo de execução da ferramenta, mas nenhuma alteração na ordem de complexidade da execução dos modelos executáveis. Como não houve alteração na ordem de complexidade do sistema, a elevação do tempo de execução pode ser contornada com a utilização da própria estrutura da ferramenta, que permite que instâncias do modelo do sistema sejam executadas em computadores diferentes simultaneamente, sendo os resultados acumulados em um servidor de dados único para consultas futuras. Apesar de ter sido notada uma relação entre o aumento do tempo de execução e o número de enunciados condicionais de controle de fluxo por unidade de código, não foi deduzido um modelo para determinar a elevação do tempo de execução, pois esta elevação é variável dentro de um mesmo modelo, dependendo do caminho do código fonte percorrido a cada simulação.

Uma conclusão secundária obtida a partir dos resultados dos experimentos aplicados é que a cobertura de laços, que requer que cada laço do código fonte do programa seja iterado uma vez, várias vezes e também não seja iterado durante os testes, não fornece resultados significativos para sistemas descritos em níveis de abstração mais baixos do que o de *Especificação Executável*. Nestes níveis, os laços passam a ter número de iterações definido, sendo que apenas uma das métricas requeridas costuma ser atingida. No nível RTL, a utilidade desta cobertura é praticamente nula, pois os laços são quase que completamente inexistentes nos sistemas descritos neste nível de abstração.

Uma última contribuição do trabalho foi o desenvolvimento de um método para geração de vetores de teste para sistemas descritos em SystemC. O método utiliza uma etapa de instrumentação de código seguida pela utilização de um algoritmo de otimização numérica para a geração de vetores de teste que exercitam um caminho no código fonte. Este método foi desenvolvido como uma alternativa para a execução simbólica de código, que também foi revisada durante este trabalho. Até o momento, o método é aplicável

somente a sistemas combinatórios, pois o caminho precisa ser executado a cada iteração do algoritmo de otimização numérica e o sistema não pode atingir estados que impeçam a execução deste caminho. Pretende-se tratar desta limitação em trabalhos futuros.

Este trabalho também deixa perguntas a serem respondidas. A primeira delas é se a metodologia proposta realmente resulta em um impacto positivo sobre a verificação do sistema. A intuição indica que sim, uma vez que as vantagens das técnicas estruturais e funcionais são combinadas em um único modelo. Porém, foi observado que a aplicação da metodologia proposta deve ser feita de modo cuidadoso de forma a evitar a ineficiência do *foco na cobertura*, quando os testes deixam de ser direcionados para a verificação da funcionalidade do sistema e passam a ser implementados de forma a aumentar métricas de cobertura. Por este motivo, a sua utilização nos primeiros estágios de desenvolvimento do sistema deve ser evitada, pois nestes estágios as funcionalidades ainda estão sendo implementadas e o ambiente de verificação está sendo moldado para uma verificação funcional eficiente. Em estágios mais avançados de desenvolvimento, onde os vetores de teste funcional não são mais modificados de maneira corriqueira, a aplicação da metodologia pode ser bastante proveitosa, principalmente nos casos onde ocorre geração automática de código por meio do emprego de ferramentas de síntese. Códigos gerados automaticamente são difíceis de serem entendidos e podem esconder implementações especiais de funcionalidades, criando sub-conjuntos dos casos de entrada que não são cobertos pela especificação funcional. Para se ter certeza sobre as suposições das vantagens da metodologia, no entanto, é necessária a realização de experimentos em sistemas reais, o que é material para trabalhos futuros.

Outra pergunta deixada por este trabalho é *qual método de geração de vetores de teste é mais indicado para a verificação de SoCs?* Uma das desvantagens da utilização de execução simbólica de código é que não é possível se determinar os valores armazenados em ponteiros, nem mesmo os limites dos laços que dependem de valores de entrada do sistema. No entanto, a importância destes problemas se reduz na medida em que o sistema passa a ser descrito em níveis mais baixos de abstração, onde ponteiros e laços são evitados por não serem construções sintetizáveis pelas ferramentas atuais. Por ou-

tro lado, a geração de vetores de teste baseada em execução simbólica de código é um problema NP-completo, o que pode tornar os métodos numéricos vantajosos mesmo nos níveis mais baixos de abstração, onde a complexidade do código tende a ser muitas vezes superior a dos níveis mais altos. Responder a esta pergunta requer que os dois métodos sejam aplicados a diferentes sistemas e o desempenho de cada um deles seja comparado.

Por fim, tendo este trabalho utilizado como referência estudos realizados tanto para a indústria de *software* quanto para a indústria de microeletrônica, uma pergunta final levantada é *quais as características que diferem o desenvolvimento de SoCs em alto nível de abstração do desenvolvimento de sistemas puramente de software?* E *a partir de que momento do ciclo de verificação, e até mesmo do ciclo de projeto, estas duas indústrias necessitam de técnicas diferentes?* Respostas para estas duas perguntas devem ser buscadas em trabalhos futuros, para direcionar as pesquisas de ferramentas e métodos para as duas indústrias.

REFERÊNCIAS

(Abramovici et al., 1990) ABRAMOVICI, M.; BREUER, M. A.; FRIEDMAN, A. D. *Digital Systems Testing and Testable Design*. [S.l.]: Computer Science Press, 1990.

(Aho et al., 1986) AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0201100886.

(Bergamaschi e Cohn, 2002) BERGAMASCHI, R. A.; COHN, J. The a to z of socs. In: *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM Press, 2002. p. 790–798. ISBN 0-7803-7607-2.

(Bergeron, 2003) BERGERON, J. *Writing Testbenches: Functional Verification of HDL Models*. [S.l.]: Kluwer Academic Publishers, 2003.

(Black e Donovan, 2005) BLACK, D. C.; DONOVAN, J. *SystemC: From the Ground Up*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 0387292403.

(Bricaud, 1999) BRICAUD, P. J. Ip reuse creation for system-on-a-chip design. In: *Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999*. New York, NY, USA: IEEE Computer Society, 1999. p. 395–401. ISBN 0-7803-5443-5.

(Chang et al., 1999) CHANG, H.; COOKE, L.; HUNT, M.; MARTIN, G.; MCNELLY, A.; TODD, L. *Surviving the SoC Revolution - A Guide to Platform-Based Design*. [S.l.]: Kluwer Academic Publishers, 1999.

(Claasen, 2003) CLAASEN, T. A. System on a chip: Changing ic design today and in the future. *IEEE Micro*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 3, p. 20–26, 2003. ISSN 0272-1732.

(Clarke, 1976) CLARKE, L. A. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, v. 2, n. 3, p. 215–222, 1976.

(Coward, 1991) COWARD, P. D. Symbolic execution and testing. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 33, n. 1, p. 53–64, 1991. ISSN 0950-5849.

(Dias Junior e Silva, 2007) Dias Junior, A.; SILVA, J. Diógenes Cecilio da. Code-coverage based test vector generation for systemc designs. In: *Proceedings of IEEE International Symposium on VLSI*. IEEE Computer Society, 2007. p. 198–206.

Disponível em: <http://dblp.uni-trier.de/db/conf/isvlsi/isvlsi2007.html#DiasS07>. Acesso em: 11/09/2007.

(Dias, 2007) DIAS, S. R. *SRD: Uma Ferramenta de Apoio ao Projetista de Sistemas de Hardware Utilizando a Linguagem SystemC*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, 2007.

(Eckel, 2000) ECKEL, B. *Thinking in C++*. 2. ed. New Jersey, USA: Prentice-Hall Inc., 2000. ISBN 0-13-979809-9.

(Gajski et al., 2000) GAJSKI, D. D.; WU, A. C.-H.; CHAIYAKUL, V.; MORI, S.; NUKIYAMA, T.; BRICAUD, P. Embedded tutorial: essential issues for ip reuse. In: *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*. New York, NY, USA: ACM Press, 2000. p. 37–42. ISBN 0-7803-5974-7.

(Gallagher e Narasimhan, 1997) GALLAGHER, M. J.; NARASIMHAN, V. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 8, p. 473–484, 1997. ISSN 0098-5589.

(Gawanmeh et al., 2004) GAWANMEH, A.; HABIBI, A.; TAHAR, S. *Enabling SystemC Verification using Abstract State Machines*. [S.l.], 2004.

(Glass, 1981) GLASS, R. L. Persistent software errors. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 7, n. 2, p. 162–168, 1981.

(GNU, 2007a) GNU. *GCC, the GNU Compiler Collection*. 2007. Disponível em: <http://gcc.gnu.org/>. Acesso em: 13/12/2007.

(GNU, 2007b) GNU. *GCOV, Using the GNU Compiler Collection (GCC)*. 2007. Disponível em: <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>. Acesso em: 13/12/2007.

(Große et al., 2007) GROSSE, D.; PERAZA, H.; KLINGAUF, W.; DRECHSLER, R. Measuring the quality of a systemc testbench by using code coverage techniques. In: *Forum on specification & Design Languages (FDL)*. Gières, France: ECSI Press, 2007.

(Grötker et al., 2002) GRÖTKER, T.; LIAO, S.; MARTIN, G.; SWAN, S. *System Design with SystemC*. [S.l.]: Kluwer Academic Publishers, 2002.

(Lawrance et al., 2005) LAWRANCE, J.; CLARKE, S.; BURNETT, M.; ROTHERMEL, G. How well do professional developers test with code coverage visualizations? an empirical study. In: *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 53–60. ISBN 0-7695-2443-5.

(Malaiya et al., 2002) MALAIYA, Y.; LI, N.; BIEMAN, J.; KARCICH, R. Software reliability growth with test coverage. In: *IEEE Transactions on Reliability*. [S.l.: s.n.], 2002. p. 420–426.

(Malaiya, 1995) MALAIYA, Y. K. Antirandom testing: Getting the most out of black-box testing. In: *Proceedings of the 1995 International Symposium On Software Reliability Engineering*. [S.l.: s.n.], 1995. p. 86–95.

(Malaiya et al., 1994) MALAIYA, Y. K.; LI, N.; KARCICH, R.; SKBBE, B. S.; BIEMAN, J. The relationship between test coverage and reliability. In: *Proceedings of 1994 International Symposium On Software Reliability Engineering*. [S.l.: s.n.], 1994. p. 186–195.

(Marick, 1999) MARICK, B. *How to Misuse Code Coverage*. 1999. Disponível em: <http://testing.com/writings/coverage.pdf>.

(Microsoft, 2007) MICROSOFT. *Microsoft SpecExplorer*. 2007. Disponível em: <http://research.microsoft.com/projects/SpecExplorer/default.aspx>. Acesso em: 01/09/2007.

(Ntafos, 1988) NTAFOS, S. C. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 14, n. 6, p. 868–874, 1988. ISSN 0098-5589.

(OSCI, 2007) OSCI. *Open SystemC Initiative*. 2007. Disponível em: <http://www.systemc.org>. Acesso em: 01/05/2007.

(Patel e Shukla, 2007) PATEL, H. D.; SHUKLA, S. K. Model-driven validation of systemc designs. In: *DAC '07: Proceedings of the 44th annual conference on Design automation*. New York, NY, USA: ACM Press, 2007. p. 29–34. ISBN 978-1-59593-627-1.

(Rapps e Weyuker, 1985) RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 367–375, 1985. ISSN 0098-5589.

(Silva et al., 2004) SILVA, K. R. G. da; MELCHER, E. U. K.; ARAUJO, G.; PIMENTA, V. A. An automatic testbench generation tool for a systemc functional verification methodology. In: *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM Press, 2004. p. 66–70. ISBN 1-58113-947-0.

(Singh et al., 2004) SINGH, L.; DRUCKER, L.; KHAN, N. *Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout*. Boston, USA: Springer, 2004. ISBN 140207672X.

(Stanley, 2001) STANLEY, D. *ISO-9000:2000 Definitions: Translated Into Plain Englis*. 2001. Disponível em: <http://praxiom.com>. Acesso em: 15/11/2007.

(Tredennick e Shimamoto, 2004) TREDENNICK, N.; SHIMAMOTO, B. Reconfigurable systems emerge. In: BECKER, J.; PLATZNER, M.; VERNALDE, S. (Ed.). *FPL*. Springer, 2004. (Lecture Notes in Computer Science, v. 3203), p. 2–11. ISBN 3-540-22989-2. Disponível em: <http://dblp.uni-trier.de/db/conf/fpl/fpl2004.html#TredennickS04>.

(Wile et al., 2005) WILE, B.; GOSS, J.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0127518037.

(Wolfsthal e Gott, 2005) WOLFSTHAL, Y.; GOTT, R. M. Formal verification: is it real enough? In: *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM Press, 2005. p. 670–671. ISBN 1-59593-058-2.

(Yin et al., 1997) YIN, H.; LEBNE-DENGEL, Z.; MALAIYA, Y. K. Automatic test generation using checkpoint encoding and antirandom testing. In: *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*. Washington, DC, USA: IEEE Computer Society, 1997. p. 84. ISBN 0-8186-8120-9.

APÊNDICE A RELATÓRIOS DE COBERTURA

A.1 Relatório de Cobertura de Ramos

A figura 36 mostra um exemplo de relatório de cobertura de ramos. O resumo do relatório contém o número de ramos existentes no sistema, o número de ramos cobertos pelo teste e a cobertura total atingida pelos testes. Verifica-se que, de um total de 27 ramos, 19 ramos foram cobertos neste exemplo, resultando em uma taxa de cobertura de 70,37%. A data e hora de geração do relatório também são mostradas para um registro mais completo.

RESUMO

Total de Ramos: 27
Ramos Cobertos: 19
Cobertura Atingida: 70.37 %

Relatório Gerado em: Sat Jan 26 09:46:33 2008

RAMOS NÃO COBERTOS

Arquivo: ula.cpp	**Linha:** 66
Tipo do Ramo: if	**Condição:** (b < 0)
TRUE: 1 vez(es)	**FALSE:** 0 vez(es)

```
if (b < 0)
    {
        neg = true;
        b = (int)multiplication(-1, b);
    }
```

Figura 36: Relatório de Cobertura de Ramos

Abaixo do resumo, está a relação dos ramos não cobertos pelos testes. Apenas o primeiro ramo da relação está sendo mostrado na figura. Um quadro indicativo mostra o arquivo e a linha onde se encontra definido o ramo. Também estão presentes o tipo do ramo, neste caso um ramo *if*, a condição de teste do ramo, e quantas vezes o ramo foi executado verdadeiro (*TRUE*) e quantas vezes foi executado falso (*FALSE*). Ao lado do quadro, o código do ramo é apresentado para consultas.

A.2 Relatório de Cobertura de Termos

A figura 37 mostra um exemplo de relatório de cobertura de termos. O resumo do relatório contém o número de termos existentes no sistema, o número de termos cobertos pelo teste e a cobertura total atingida. Verifica-se que, de um total de 28 termos presentes no sistema, 19 foram cobertos, atingindo 67,86% de cobertura. A data e hora de geração do relatório também são mostradas para um registro mais completo.

RESUMO

Total de Condições: 28
Condições Cobertas: 19
Cobertura Atingida: 67.86 %

Relatório Gerado em: Sat Jan 26 09:46:42 2008

TERMOS NÃO COBERTOS

Arquivo: ula.cpp	**Linha:** 89
Tipo: if	**Condição:** a == 0
TRUE: 0 times	**FALSE:** 1 times

```
if (a == 0 || b == 0)
    return a | b;
```

Figura 37: Relatório de Cobertura de Termos

Abaixo do resumo, está a relação dos termos não cobertos. Apenas um termo está sendo mostrada na figura. Um quadro indicativo mostra o arquivo e a linha onde se encontra definido o termo. Também estão presentes o tipo do ramo do termo, neste caso um ramo *if*, o termo testado do ramo, e quantas vezes este termo foi executado verdadeiro (*TRUE*) e quantas vezes foi executado falso (*FALSE*). No código ao lado, apenas o termo analisado está em evidência.

A.3 Relatório de Cobertura Exaustiva de Termos

A figura 38 mostra um exemplo de relatório de cobertura de cobertura exaustiva de termos. O resumo do relatório contém o número de combinações possíveis para todos os termos das condições existentes no sistema, o número de combinações cobertas pelos testes e a cobertura atingida. Verifica-se que, de um total de 28 combinações presentes neste exemplo, 18 foram cobertas, atingindo uma cobertura de 64,29%. A data e hora de geração do relatório também são mostradas para um registro mais completo.

Abaixo do resumo, está a relação das combinações não cobertas pelos testes. Apenas as combinações para os termos da primeira condição do relatório estão sendo apresentadas na figura. O nome do arquivo e a linha onde se encontra definido o ramo da condição é apresentada, juntamente com o tipo do ramo, logo acima de um quadro com todas as combinações possíveis para os termos. Cada uma das linhas indica uma possível combinação de valores para os termos, assim como apresentado em uma tabela verdade. A primeira coluna indica quantas vezes aquela combinação foi executada durante os testes.

RESUMO

Total de Múltiplas Condições: 28
Múltiplas Condições Cobertas: 18
Cobertura Atingida: 64.29 %

Relatório Gerado em: Sat Jan 26 09:42:53 2008

COMBINAÇÕES DE TERMOS NÃO COBERTAS

Arquivo: ula.cpp **Linha:** 89 **Tipo do Ramo:** if

Exercitada	a == 0	b == 0
1 vez(es)	FALSE	FALSE
0 vez(es)	TRUE	FALSE
0 vez(es)	FALSE	TRUE
0 vez(es)	TRUE	TRUE

Figura 38: Relatório de cobertura exaustiva de termos

A.4 Relatório de Cobertura Relacional

A figura 39 mostra um exemplo de relatório de cobertura relacional. O resumo do relatório contém o total de operadores relacionais existentes no projeto, o número de possibilidades para a relação entre os operandos, que é 3 (três) vezes o número de operadores relacionais, o número de possibilidades cobertas pelos testes e a cobertura atingida. Verifica-se que, de um total de 9 possibilidades, 6 foram cobertas, atingindo uma cobertura de 66,67%. A data e hora de geração do relatório também são mostradas para um registro mais completo.

RESUMO

Total de Operators Relacionais: 3
Possibilidades Relacionais: 9
Possibilidades Cobertas: 6
Cobertura Atingida: 66.67 %

Relatório Gerado em: Sat Jan 26 09:46:43 2008

OPERADORES RELACIONAIS NÃO COBERTOS

Arquivo: ula.cpp	**Linha:** 66	**Condição:** b < 0
TRUE: 1 vez(es)	**EQUAL:** 0 vez(es)	**FALSE:** 0 vez(es)

Figura 39: Relatório de Cobertura Relacional

Abaixo do resumo são apresentadas as relações não cobertas pelos testes. Um quadro indica o arquivo, a linha e a relação descrita. O número de vezes que a relação foi exercitada verdadeira (*TRUE*), falsa (*FALSE*) e igual (*EQUAL*) também é apresentado.

A.5 Relatório de Cobertura de Laços

A figura 40 mostra um exemplo de relatório de cobertura de laços. O resumo do relatório contém o total de laços do projeto, o total de métricas que devem ser cobertas, nesse caso 3 (três) vezes o número de laços, o total de métricas cobertas e a cobertura atingida pelos testes. Verifica-se que, de um total de 18 métricas, 5 foram cobertas, atingindo 27,78% de cobertura. A data e hora de geração do relatório também são mostradas para um registro mais completo.

RESUMO

Total de Laços: 6
Total de Métricas: 18
Métricas Cobertas: 5
Cobertura Atingida: 27.78 %

Relatório Gerado em: Sat Jan 26 09:46:47 2008

LAÇOS NÃO COBERTOS

Arquivo: ula.cpp	**Linha:** 73	
Tipo do Laço: while	**Condição:** (b > 0)	
Não Iterado 0 times	**Uma Vez:** 0 times	**Várias Vezes:** 1 times

```
while(b > 0)
    {
        temp = multiplication(temp, a);
        b = (int)subtraction(b, 1);
    }
```

Figura 40: Relatório de Cobertura de Laços

Abaixo do resumo, está a relação dos laços não cobertos pelos testes. Apenas o primeiro laço da relação está sendo mostrado na figura. Um quadro indicativo mostra o arquivo e a linha onde se encontra definido o laço. Também estão presentes o tipo do laço, neste caso um ramo *while*, a condição de teste do laço, quantas vezes o laço não foi iterado, quantas vezes foi iterado uma vez e quantas vezes foi iterado *várias vezes*. Ao lado do quadro, o código do laço é apresentado para consultas.