Uma Metodologia de Verificação Funcional para Circuitos Digitais

Karina Rocha Gomes da Silva

Tese de Doutorado submetida à Coordenação dos Cursos de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências no domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Prof. Elmar Uwe Kurt Melcher Dr.
(Orientador)
Prof. Wolfgang Rosenstiel Dr.
(Co-Orientador)

Campina Grande, Paraíba, Brasil
©Karina Rocha Gomes da Silva, Fevereiro de 2007

Resumo

O advento das novas tecnologias VLSI e metodologias de projetos de *System On a Chip* (SoC) têm trazido um crescimento explosivo à complexidade dos circuitos eletrônicos. Como um resultado desse crescimento, a verificação funcional tem se tornado o maior gargalo no fluxo de projetos de hardware. Assim, novos métodos são requeridos para permitir que a verificação funcional seja realizada de forma mais rápida, fácil e que permita uma maior reusabilidade.

Esse trabalho propõe a criação de uma nova metodologia para verificação funcional de componentes digitais integráveis, que permite o acompanhamento do fluxo de projeto, de forma que o *testbench* (ambiente de simulação) seja gerado antes da implementação do dispositivo sendo verificado (*Design Under Verification* - DUV), tornando o processo de verificação funcional mais rápido e o *testbench* mais confiável, devido a ele ser verificado antes do início da verificação funcional do DUV.

Abstract

The advent of new VLSI technology and SoC design methodologies, has brought an explosive growth in the complexity of modern electronic circuits. As a result, functional verification has become the major bottleneck in any design flow. New methods are required that allow for easier, quicker and more reusable verification.

In this work a novel functional verification methodology for digital components is proposed, which follows the project flow, allowing the *testbench* (simulation environment) to be generated before the Design Under verification (DUV) implementation. In this way, the functional verification process become faster and the verification engineer can trust the *testbench*, because it has been verified before the DUV´s functional verification.

Agradecimentos

Agradeço à Deus, pela minha vida.

Obrigado ao professor Elmar pelo incentivo, dedicação, amizade e pelo envolvimento em todo o processo do meu doutorado.

Obrigada ao meu Co-orientador, professor Rosenstiel, por me receber na sua instituição em Tuebingen.

Meus agradecimentos especiais aos meus adoráveis familiares, a quem eu devo tudo e dedico esse trabalho: Altair, Avelar, Cássio, Avelar Filho, Fernanda, Taisa, Alana, vó Dora, vó Ana, vô Ivo, Seu Zé, Ednay, Naia, Ilza, aos meus cunhados e sobrinhos. Ao meu marido Cássio, com muito carinho, obrigada por estar ao meu lado todos os dias dessa jornada, pela inspiração, dedicação, força, amizade, incentivos e pelo seu amor.

Obrigada aos professores Joseana, Edna, Guido, Marius, Antônio Marcus, Roberto e Eustáquio, pela participação no meu doutorado.

Quero agradecer a todos os integrantes do LAD. Em especial meus agradecimentos para a minha amiga Dani, George, Henrique, Isaac, Ana Karina, Wilson, Fagner, Zurita, Matheus, Osman e Izabela.

Aos colegas e amigos de Tuebingen, obrigada pelas discussões e por todo o apoio que me deram: Djones, Fabiana, Pradeep, Prakash, Mike, Dominik, Axel, Jurgen, Fabiano, André, Tiago, Jacqueline e Simone.

Agradeço à CAPEs e ao CNPq, que financiaram o meu projeto.

Agradeço à Ângela, por me ajudar na resolução de problemas burocráticos na COPELE.

Aos meus queridos amigos, Tatiana, Tiago, Andreza, Alessandro, Almeidinha, Juliana, Fabiana, Cláudia, Ildeu, Eulina, Javé e Flávio pela força, pensamento positivo e por sua amizade.

Aos amigos feitos durante o doutorado Lauro, Nazareno, Salete, Lia e Midian.

As pessoas que não tiveram o nome citado aqui, mas que de alguma forma também contribuiram para esse trabalho, meu muito obrigado.

Conteúdo

1 Introdução **1**

1.1 Motivação e contexto 1

1.1.1 Especificação do hardware 2

1.1.2 Especificação da verificação funcional 3

1.1.3 Implementação do *testbench* 4

1.1.4 Implementação RTL 5

1.1.5 Verificação Funcional 5

1.1.6 Síntese 6

1.1.7 Simulação pós-síntese 6

1.1.8 Prototipação 6

1.2 Definição do problema 6

1.3 Organização 9

2 Conceitos de Verificação Funcional **10**

2.1 Dispositivo a ser verificado 10

2.2 Tipos de verificação 11

2.2.1 Verificação estática 11

2.2.2 Verificação dinâmica 11

2.2.3 Verificação híbrida 12

2.3 Verificação funcional 12

2.3.1 Black-box, grey-box e white-box 14

2.3.2 Testbench 15

2.3.3 Cobertura 17

2.3.4 Geração de estímulos 21

2.4 SystemC 27
2.5 Considerações sobre conceitos de verificação funcional 28

3 Metodologia VeriSC **31**
3.1 Construção do *testbench* 33
3.1.1 Templates 36
3.1.2 Source 45
3.1.3 DUV *(Design Under Verification)* 51
3.1.4 TDriver 51
3.1.5 Reset_driver 54
3.1.6 TMonitor 56
3.1.7 Checker 57
3.1.8 Arquivo de estrutura 58
3.1.9 Modelo de Referência 59
3.1.10 FIFO Buffer (*First In First Out*) 61
3.1.11 Pré-source 61
3.1.12 Sink 61
3.2 Metodologia 61
3.2.1 Testbench para o DUV completo (primeiro passo) 62
3.2.2 Decomposição hierárquica do Modelo de Referência (segundo passo) 65
3.2.3 Testbench para cada bloco do DUV (terceiro passo) 66
3.2.4 Substituição do DUV completo (último passo) 67
3.3 Cobertura funcional 68
3.3.1 Exemplo de descrição do modelo de cobertura 71
3.3.2 Biblioteca BVE-COVER 72
3.4 Resumo da metodologia VeriSC 76

4 Resultados **78**
4.1 Resultados da metodologia VeriSC tradicional 78
4.1.1 Decodificador MP3 79
4.1.2 Decodificador MPEG4 81
4.2 Resultados da metodologia VeriSC 83

4.2.1 Decodificador de Bitstream 83
4.2.2 Resultados obtidos nos cursos de verificação 85
4.2.3 Resultados obtidos com o projeto DigiSeal 86
4.3 Resultados obtidos com biblioteca BVE-COVER 89

5 Conclusão 91

5.1 Trabalhos futuros 92

A eTBc tool: A Transaction Level Testbenches Generator 98

A.1 Introduction 98
A.2 VeriSC2 Methodology 99
A.2.1 The testbench implementation steps 101
A.3 eTBc Tool 104
A.3.1 eTBc tool implementation 106
A.4 Results 106

B Analysis about VeriSC2 methodology taking in account VSI Alliance pattern report 109

B.1 Functional Verification Deliverables 109
B.1.1 Documentation 110
B.1.2 Testbench 110
B.1.3 Drivers 111
B.1.4 Monitors 112
B.1.5 Assertions 112
B.1.6 Functional Coverage 112
B.1.7 Code Coverage 113
B.1.8 Formal Methods 114
B.1.9 Documentation 114
B.1.10 Behavioural Models 115
B.1.11 Behavioural Models for Memory 116
B.1.12 Detailed Behavioural Models for I/O Pads 116
B.1.13 Stimulus 116

B.1.14 Scripts 116
B.1.15 Stub Model 116
B.1.16 Functional Verification Certificate 116
B.2 Reuse of Functional Verification Deliverables in SoC Verification 116
B.2.1 DUV Re-Verification 117
B.2.2 DUV Re-Verification in a SoC 117
B.3 Functional Verification Deliverables Rules 117
B.3.1 Drivers 117
B.3.2 Monitors 118
B.3.3 Assertions 119
B.3.4 Functional Coverage 119
B.3.5 Code Coverage 120
B.3.6 Formal Methods 120
B.3.7 Documentation 120
B.3.8 Behavioural Models 120
B.3.9 Scripts 120
B.3.10 Stub Model 121
B.3.11 Functional Verification Certificate 121

Lista de Símbolos

BFM	-	*Bus-functional model*
DUV	-	*Design Under Verification*
IP	-	*Intellectual property*
OSCI	-	*Open SystemC Initiative*
RTL	-	*Register Transfer Level*
SBCCI	-	*Symposium on Integrated Circuits and Systems Design*
SCV	-	*SystemC Verification Library*
SDL	-	*Specification Description Language*
SOC	-	*System on a chip*
TLM	-	*Transaction Level Modelling*
TLN	-	*Transaction Level Netlist*
VHDL	-	*VHSIC Hardware Description Language*
HDL	-	*Hardware Description Language*
VLSI	-	*Very Large Scale Integration*
FIFO	-	*First In First Out*
eTBc	-	*Easy Testbench Creator*
VSIA	-	*VSI Alliance*
BVE-COVER	-	*Brazil-IP Verification Extension*
COMET	-	*Coverage measurement tool*
eTL	-	*eTBc Template Language*
eDL	-	*eTBc Design Language*

Lista de Figuras

1.1 Etapas para o desenvolvimento de hardware 2

2.1 Diagrama de projeto 13

2.2 Esquema de um *testbench* genérico 15

2.3 Fluxo de projeto do trabalho de Hu 26

2.4 Diagrama de blocos do *testbench* do trabalho de Randjic 28

3.1 Fluxo de verificação 32

3.2 Diagrama do *testbench* da metodologia VeriSC 35

3.3 Exemplo de um DUV (*P_mpeg*) formado por parte do MPEG4 36

3.4 Esquema da ferramenta eTBc 37

3.5 Exemplo da divisão do *P_mpeg* em blocos 62

3.6 Testbench para o DUV completo (primeiro passo) 63

3.7 Substituição do DUV por TMonitor, Modelo de Referência e TDriver . . . 64

3.8 Decomposição hierárquica do Modelo de Referência (segundo passo) . . . 65

3.9 Testbenches para cada bloco do DUV: *PIACDC* e *QI* (terceiro passo) 66

3.10 Substituição do DUV completo (último passo) 68

3.11 Locais para medir cobertura funcional 70

3.12 Componentes da biblioteca de cobertura BVE-COVER 72

3.13 Análise do buraco de cobertura 76

3.14 Fluxo de cobertura 77

4.1 Decodificador MP3 80

4.2 Decodificador MPEG4 81

4.3 Bloco esquemático do DigiSeal 86

4.4 Comparação entre DigiSeal e MPEG4 respectivamente 88

A.1 General Testbench schema 100
A.2 Testing Source and Checker 101
A.3 Testbench construction for the top level DUV 102
A.4 Hierarchical decomposition of the RM 103
A.5 Testbench implementation for module DUV_1 103
A.6 Replace the top level DUV (Last Step) 104
A.7 eTBc tool - architecture diagram 105

B.1 Testbench to functional verification according VSIA 111
B.2 Testbench to functional verification 111

Lista de Tabelas

2.1 Comparação entre trabalhos da área 29

3.1 Modelo de cobertura 72

4.1 Características do MP3 80
4.2 Características do MPEG4 82
4.3 Comparação entre metodologia VeriSC e metodologia VeriSC tradicional . 85
4.4 Comparação entre DigiSeal e parte do MPEG4 87
4.5 Tabela de resultados da cobertura 90

A.1 3MBIP Features 108

Capítulo 1

Introdução

Esse trabalho é apresentado como uma contribuição para a área de circuitos integrados digitais. Especificamente, o trabalho aborda o tema de verificação de circuitos digitais síncronos, utilizando a técnica de verificação funcional, realizada através de simulação.

O foco do trabalho consiste em introduzir uma nova metodologia de verificação funcional de circuitos digitais integrados, denominada VeriSC.

De acordo com alguns autores [5; 29] a fase de verificação consome cerca de 70% de todos os recursos do projeto. Portanto, uma proposta de fluxo de projeto que não leve em consideração as necessidades da verificação como uma fase prioritária, pode se mostrar bastante inadequada. Nesse contexto, o trabalho aqui apresentado, vem contribuir apresentando uma proposta que priorize as necessidades da fase de verificação, de forma a torná-la mais rápida e confiável na eliminação de erros do projeto.

O restante desse capítulo é apresentado da seguinte forma: situar e motivar o trabalho de verificação em um projeto de implementação de hardware; declarar o problema abordado nesse trabalho e dar uma visão geral de como se encontra estruturado o restante do documento.

1.1 Motivação e contexto

A implementação de um *Intellectual Property core* (*IP Core*) consiste na criação de um componente de hardware que desempenhe uma determinada funcionalidade. Esse componente deve permitir a integração em um ambiente onde ele possa interagir com os demais

componentes para formar um sistema.

Para que essa integração seja possível, é necessário que esse *IP Core* contenha a menor quantidade de falhas possível. Essa confiabilidade vai depender de como é realizado o processo de geração do *IP Core*. Esse processo é composto de fases. Não se tem um consenso entre os trabalhos da área sobre os elementos conceituais de cada fase. Nem mesmo a delimitação de cada fase vem a estabelecer um consenso. No entanto, para que o leitor possa se orientar com relação ao processo de desenvolvimento de hardware e possa contextualizar o trabalho de verificação aqui relatado, serão descritas as fases necessárias para a obtenção do hardware, de acordo com os conceitos adotados para a realização deste trabalho. Essas fases são mostradas na Figura 1.1.

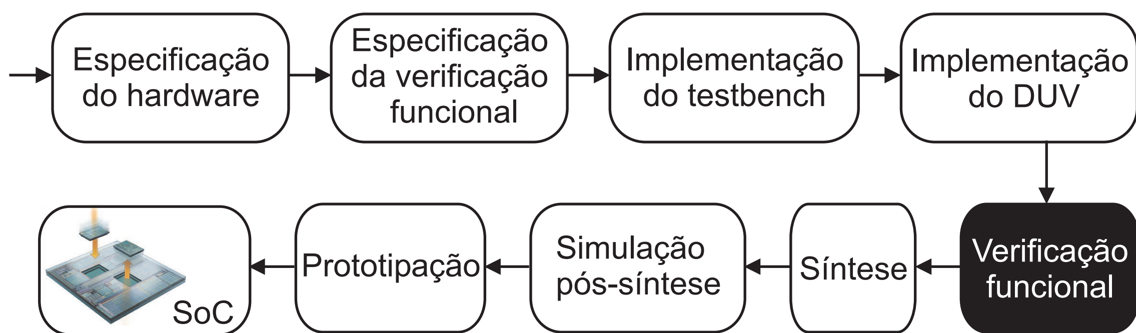

Figura 1.1: Etapas para o desenvolvimento de hardware

1.1.1 Especificação do hardware

Várias fases precedem a implementação de um código *Register Transfer Level* (RTL). Em particular, é necessário obter uma descrição completa do que se deseja construir, denominada de especificação do hardware.

A especificação do hardware é fundamental para o entendimento das necessidades do dispositivo a ser desenvolvido. Nessa fase são estudados todos os requisitos do mesmo. Tais requisitos são documentados, através da definição exata de cada funcionalidade que o hardware final deve executar. Essa especificação pode ser feita em um documento texto inicial. Existe a possibilidade de se construir uma especificação executável a partir dessa especificação em formato de texto.

A especificação deve ter um alto nível de abstração, no qual ainda não tenham

sido tomadas decisões em relação à implementação das funcionalidades, em termos da arquitetura-alvo a ser adotada, nem sobre os componentes de hardware ou software a serem selecionados. Ela contém detalhes de alto nível, tais como funcionalidades a serem executadas, bem como informações de baixo nível, tais como especificação e descrição dos pinos a serem usados.

É necessário que a especificação do hardware seja realizada por alguém que conheça os detalhes da aplicação visada. Normalmente quem faz essa especificação é o engenheiro de projeto. Outro fator importante é que qualquer mudança que ocorra dentro dessa especificação deve ser repassada para a especificação da verificação.

1.1.2 Especificação da verificação funcional

A especificação da verificação é muito importante para guiar o processo de verificação de um dispositivo. É nessa especificação onde são documentados os aspectos importantes que devem ser verificados em um determinado dispositivo. Todas as funcionalidades devem ser consideradas. O documento deve ser feito em formato de texto e normalmente quem faz essa especificação é o engenheiro de verificação, mas todos os participantes do projeto devem ser consultados. O plano de cobertura faz parte da especificação da verificação.

O plano de verificação é um conjunto de documentos que fornece os objetivos, componentes e detalhes da verificação. Esse plano é uma parte crítica para se obter sucesso durante a verificação. Tal plano precisa lidar com uma gama de questões, indo desde objetivos de alto-nível até a identificação e documentação das variáveis. O plano de verificação também serve como uma especificação funcional para o ambiente de verificação, tal como o estabelecimento de quais os tipos de estímulos que serão utilizados para a verificação do projeto. Esse plano deve ser dinâmico e refletir o estado presente do projeto. Ele deve ser revisto e atualizado sempre que algo for mudado no projeto, até mesmo após um novo erro ter sido encontrado.

É importante que os pontos críticos do projeto sejam discutidos por todos os membros da equipe, que esses pontos sejam definidos como críticos (*corner cases*) e que seja dada uma atenção especial para cada um desses pontos, pois eles podem ser fortes candidatos a apresentar erros na verificação.

A parte do plano que trata da cobertura é responsável por determinar parâmetros de

medição para o progresso da simulação. Esses parâmetros devem ser estabelecidos pela equipe de verificação, levando-se em consideração quais as funcionalidades importantes que devem ser executadas durante a simulação e qual o tipo de cobertura será realizada para melhor poder constatar que essas funcionalidades foram efetivamente simuladas.

1.1.3 Implementação do *testbench*

O *testbench* é o ambiente através do qual o dispositivo a ser verificado (*Design Under Verification* - DUV) será inserido, de forma que ele receba estímulos e que as respostas sejam comparadas com o resultado ideal.

O *testbench* deve ser implementado preferencialmente em um nível alto de abstração, o qual é denominado de nível de transação (*Transaction-level*). Esse nível de transação não se preocupa com detalhes de protocolos no nível de sinais. Ao invés disso, o seu foco é a comunicação entre blocos e a transferência de transações.

Uma transação é uma representação básica para a troca de informações entre dois blocos funcionais. Em outras palavras, é uma operação que inicia num determinado momento no tempo e termina em outro, sendo caracterizada pelo conjunto de instruções e dados necessários para realizar a operação. Um exemplo de uma transação poderia ser uma transmissão de um pacote ethernet. Ela é definida pelo seu tempo inicial, tempo final e atributos [7]. Uma transação pode ser simples como uma leitura de memória ou complexa como a transferência de um pacote inteiro de dados através de um canal de comunicação.

O *testbench* se comunica com o DUV através de algum protocolo que consiga converter dados no nível de transação para dados no nível de sinais. O *testbench* deve ser composto de um módulo capaz de gerar os tipos de estímulos necessários para a verificação do DUV. Um bom *testbench* precisa comparar automaticamente o resultado vindo do DUV com o resultado ideal, que se encontra na especificação.

Um dos problemas que podem ocorrer na verificação funcional pode ser causado por erro de implementação do *testbench*. Por esse motivo, o *testbench* deve ser um ambiente com a quantidade mínima de erros. Sendo assim, é importante que seja realizada uma implementação bem feita do *testbench*.

1.1.4 Implementação RTL

O código RTL é um código escrito em um nível mais baixo de abstração, o nível de sinais. Nesse nível, todas as operações são controladas por ciclos de relógios (*clocks*). Normalmente esse código é implementado em uma linguagem de descrição de hardware, tal como VHDL, Verilog, SystemC, entre outras.

A implementação RTL é o modelo que nas fases seguintes será convertido em hardware, por um procedimento em grande parte automatizado. Um aspecto importante que deve ser respeitado é que os erros precisam ser captados ainda no código RTL, pois quanto mais cedo erros forem detectados, menos recursos serão gastos para a correção.

1.1.5 Verificação Funcional

Verificação é algumas vezes confundido com outros termos. Para que não exista nenhuma dúvida com relação ao que é a verificação funcional, neste trabalho é adotado um conceito aceito pela comunidade de verificação e utilizado na bibliografia da área. "Verificação funcional é um processo usado para demonstrar que o objetivo do projeto é preservado em sua implementação"[5]. A verificação funcional deve ser realizada através da comparação de dois modelos, o modelo sendo desenvolvido e o modelo ideal que reflete a especificação.

A verificação funcional é realizada através de simulação. Durante a simulação, são inseridos estímulos na entrada do DUV e esses estímulos são coletados em sua saída e comparados com os resultados esperados (resultados ideais). No trabalho aqui apresentado, essa comparação é feita automaticamente através da comparação dos resultados com um Modelo de Referência. O Modelo de Referência é uma implementação executável e por definição reflete a especificação.

Os estímulos devem ser escolhidos cuidadosamente, de forma que possam vir a exercitar as funcionalidades desejadas. A verificação somente está terminada quando todas as funcionalidades especificadas forem executadas. Essas funcionalidades estão documentadas no modelo de cobertura. Por isso, a verificação estará terminada somente quando todos os critérios de cobertura forem atingidos. Esse processo de dirigir os estímulos de acordo com a cobertura e determinar o término da simulação pelos critérios de cobertura é denominado de verificação dirigida por cobertura.

1.1.6 Síntese

A síntese de RTL é realizada através da conversão de uma descrição RTL em um conjunto de registradores e lógica combinacional. Assim que o código RTL é sintetizado, é gerada uma *netlist* em nível de portas lógicas.

1.1.7 Simulação pós-síntese

Assim que a síntese é realizada, é necessário fazer uma simulação que é denominada simulação pós-síntese. A partir de então, aspectos específicos do dispositivo, como por exemplo, atrasos das portas lógicas, passam a ser considerados durante a execução. Nessa fase, tanto aspectos de funcionalidade, quanto de tempo são levados em consideração durante a simulação.

A simulação pós-síntese é essencial para determinar se os requisitos de tempo são respeitados e se pode ser alcançada um desempenho melhor do dispositivo, usando circuitos mais otimizados.

1.1.8 Prototipação

Vencida a etapa de simulação pós-síntese, pode-se passar para da prototipação. Essa etapa consiste na implantação da *netlist* gerado pela síntese em algum dispositivo de hardware.

A prototipação de um *IP Core* digital é tipicamente feita em FPGA e pode adicionalmente ser feita em silício.

1.2 Definição do problema

De acordo com o trabalho de Dueñas [13], 65% dos *IP Cores* falham em sua primeira prototipação em silício e 70% destes casos são devidos a uma verificação funcional mal feita. Por isso, a qualidade de dispositivos e o tempo de desenvolvimento de um projeto dependem muito de como a verificação é realizada. Uma verificação incompleta resulta em uma má qualidade ou falha no dispositivo, o que vem a causar erros durante o seu uso. Além disso, qualquer problema funcional ou comportamental que escape da fase de verificação do projeto poderá não ser detectado nas fases de prototipação e irá emergir somente depois que o

primeiro silício estiver integrado no sistema alvo [8].

Outro grande problema a ser considerado é o reuso do *IP Core* em outros projetos. É comum que empresas comprem um *IP Core* pronto para integrar em seus projetos de SoC. O maior obstáculo do reuso é a confiabilidade. Os engenheiros têm dúvidas em incorporar em seus projetos, partes de outros projetos que eles não conheçam e/ou nos quais eles não confiem. No entanto, a confiabilidade não é algo que pode ser adicionado ao projeto no final dele. Essa confiabilidade precisa ser construída através de boas práticas de projeto. Essa confiabilidade pode também ser demonstrada através de um bom processo de verificação [5]

Com relação ao tempo de projeto, deve-se considerar que o tempo total para finalização de um projeto é muito significativo porque determina o momento de introdução do produto no mercado (*time to market*). Esse problema de tempo ocorre, em parte, devido ao engenheiro de verificação precisar esperar até que o engenheiro de projeto termine a implementação para iniciar a verificação. Isso faz com que aumente o tempo para que o projeto fique pronto.

Outro fator a ser considerado em um projeto é saber qual o momento em que a verificação do projeto está completa.

Na literatura, existem vários trabalhos relacionados com verificação. No entanto, a maioria deles aborda a verificação funcional parcialmente, sem conseguir englobar todos os aspectos necessários para realizar uma verificação funcional completa. Os trabalhos de Grinwald [20], Asaf [3] e Lachish [25] introduzem técnicas para definir critérios de cobertura funcional. Os trabalhos de Dudani [28] e Hekmatpour [22], definem formas de melhorar os vetores de testes. Esses trabalhos serão discutidos no Capítulo 2.

A definição do nosso problema consiste de alguns pontos que podem ser assim resumidos:

a) A verificação funcional consome um esforço considerável, estimado em 70% dos recursos do projeto completo; b) muitas vezes o testbench não é projetado para ser reusável; c) algumas abordagens fazem a decomposição hierárquica do projeto sem considerar o processo de verificação funcional, causando um esforço extra para implementar o testbench e/ou mais testbenches precisam ser criados; d) a verificação começa somente quando todo o RTL hierárquico já foi implementado e e) testbenches são depurados juntamente com o código RTL no momento da verificação funcional. Quando um erro aparece durante a si-

mulação, ele pode ser devido a um erro de implementação do código RTL, porém pode ser também um erro no testbench.

Um grande desafio vem da necessidade de adaptar o *testbench* para o dispositivo que está sendo verificado (DUV), porque o DUV normalmente é implementado sem considerar a verificação [5]. O DUV deve ser implementado de forma a facilitar o processo de verificação, pois o maior esforço de projeto se encontra exatamente na fase de verificação. Por esse motivo, o DUV deve ser adaptado ao ambiente de verificação. Uma forma para melhor adaptar o projeto ao *testbench* é criar o *testbench* antes da criação do projeto, dessa forma a verificação pode influenciar no fluxo de implementação do projeto.

Diante do exposto, os objetivos desse trabalho podem ser resumidos da seguinte forma:

Criação de uma metodologia de verificação funcional, para verificar componentes digitais síncronos, através da comparação do DUV com seu Modelo de Referência, permitindo a geração do testbench antes mesmo da implementação do DUV, de forma a facilitar a verificação, dando ênfase à fase de verificação. Com isso, a metodologia se propõe a reduzir o tempo total de verificação e encontrar erros mais cedo, quando o DUV começa a ser implementado.

Os objetivos específicos desse trabalho são os seguintes:

- Criar uma metodologia de verificação;
- Gerar o *testbench* antes do DUV;
- Mudar a ênfase do projeto para a fase de verificação ao invés da fase de projeto;
- Reduzir o tempo total de verificação;
- Encontrar erros mais cedo na fase de projeto;
- Reusar os módulos do próprio *testbench* para a geração do *testbench* antes do DUV.

De forma a realizar esse intento, usando a metodologia proposta nesse trabalho, denominada VeriSC, é possível fazer a verificação do RTL em todas as fases de sua implementação, até mesmo no início. Além disso, essa metodologia propõe o reuso dos próprios elementos do *testbench* para criar os *testbenches* hierárquicos e para testar os próprios componentes do *testbench* e assegurar que ele não contenha erros.

1.3 Organização

Esse documento está organizado da seguinte forma:

No Capítulo 2 são apresentados os conceitos fundamentais referentes à verificação funcional, definindo dessa forma, as nomenclaturas e conceitos adotados nesse trabalho, bem como alguns trabalhos relacionados.

No Capítulo 3 é apresentada a metodologia VeriSC, que é a proposta resultante desse trabalho de doutorado. No Capítulo são apresentados todos os passos necessários para a criação de um *testbench* baseado nessa metodologia. Nesse capítulo é descrita a biblioteca de cobertura funcional BVE-COVER, bem como uma introdução à ferramenta de geração de *testbenches* eTBc (*Easy Testbench Creator).*

No Capítulo 4 são apresentados os resultados obtidos.

No Capítulo 5 são apresentadas as conclusões e sugeridos alguns trabalhos futuros.

Capítulo 2

Conceitos de Verificação Funcional

No capítulo anterior, foram descritas as fases de um projeto de hardware, de forma a dar ao leitor uma visão geral do contexto para a apresentação da metodologia de verificação VeriSC. As demais fases de implementação de hardware estão diretamente ligadas à verificação e são influenciadas por essa fase, no entanto, não serão abordadas profundamente no escopo desse documento.

Nesse capítulo são relatados os conceitos básicos nos quais se baseia esse trabalho, de forma a facilitar o entendimento, e os trabalhos relacionados da área. A descrição dos trabalhos relacionados possui o objetivo de dar uma visão geral dos vários aspectos abordados por outros pesquisadores.

2.1 Dispositivo a ser verificado

A literatura adota nomenclaturas diferentes para denominar o dispositivo a ser verificado. Nesse trabalho, esse dispositivo será denominado DUV (*Device Under Verification*).

O dispositivo a ser verificado pode ser implementado em níveis diferentes de abstração. Esse dispositivo pode começar a ser implementado a partir de um modelo comportamental e ter a sua implementação refinada até um nível de portas lógicas. No entanto, nesse trabalho, será considerado um dispositivo implementado no nível RTL (*Register Transfer Level*) ou linguagem de descrição de hardware. A metodologia aqui apresentada pode ser usada pelos demais níveis de abstração, da mesma forma.

2.2 Tipos de verificação

Os erros lógicos nos dispositivos são causados pelas discrepâncias ocorridas entre o comportamento pretendido e o comportamento observado. Esses erros podem ocorrer devido à especificação ambígua, interpretação errada da especificação ou devido a algum erro inserido na implementação do DUV. Esses erros podem ser captados através da verificação, que pode ser realizada de forma estática, dinâmica ou híbrida.

2.2.1 Verificação estática

Na classe de mecanismos de verificação estática, também chamada de verificação formal, de acordo com a definição de Bergeron [5] tem-se: verificação de modelos, verificação de equivalência e prova de teoremas.

A verificação de modelos demonstra que propriedades definidas pelo usuário nunca são violadas para todas as possíveis seqüências de entradas. A verificação de equivalência, por sua vez, compara dois modelos para determinar se eles são logicamente equivalentes ou não. Finalmente, a prova de teoremas demonstra que o teorema está provado ou não pode ser provado.

A verificação formal pode provar a inexistência de erros através de equações matemáticas e verificação de modelos. No entanto, esse processo pode ser complicado e podem ocorrer limitações do tamanho do circuito a ser verificado.

2.2.2 Verificação dinâmica

A verificação dinâmica, também denominada verificação funcional, é realizada através de simulação. Verificação funcional não prova a ausência de erros e sim a presença dos mesmos. No entanto, não há limitações de tamanho de modelos a serem verificados, desde que não haja empecilho com relação ao tempo gasto na simulação. Além disso, existem métodos para certificar quanto das funcionalidades de um projeto foram testadas, o que pode dar uma certa garantia de quanto a verificação foi abrangente. Esses métodos são chamados de cobertura.

2.2.3 Verificação híbrida

A verificação híbrida, também conhecida como semi-formal, combina técnicas das verificações estática e dinâmica, tentando sobrepor as restrições de capacidade impostas pelos métodos estáticos e as limitações de completude dos métodos dinâmicos.

2.3 Verificação funcional

A verificação funcional foi a técnica escolhida para esse trabalho. O conceito de verificação funcional adotado nesse trabalho se baseia no livro de Bergeron [5] e possui a seguinte definição: verificação funcional é um método utilizado para comparar o DUV com sua especificação.

O ponto de partida de um projeto de verificação é que exista uma especificação, considerada o modelo ideal e essa especificação deve ser respeitada durante toda a fase de projeto. A verificação funcional é um processo que acompanha o dispositivo em busca de uma completa verificação de todas as suas funcionalidades especificadas.

Um aspecto importante é decidir a granularidade do circuito a ser verificado. Quando se faz a verificação de um projeto, ele é normalmente dividido em blocos para facilitar a sua verificação. O processo de verificação funcional torna-se mais complexo quando empregado em um bloco muito grande, por ser mais difícil encontrar um erro ao fazer a simulação do projeto inteiro de uma única vez. Quando o projeto é muito grande, ao aparecer um erro, é necessário fazer uma investigação de todo o modelo para descobrir onde está a causa desse erro. Com um modelo menor, essa investigação se torna mais fácil. Por outro lado, com essa divisão, mais *testbenches* devem ser criados para a verificação de cada bloco em que o DUV foi dividido. Para efeito de comparação, o Modelo de Referência e o DUV devem ser divididos da mesma forma para possibilitar a verificação. Considerações mais detalhadas sobre o número de hierarquias a serem empregadas e do tamanho adequado de um DUV podem ser encontrados na literatura [5].

Em um projeto de hardware é necessário que a implementação do dispositivo a ser verificado e a verificação sejam feitos por pessoas diferentes, respectivamente pelos engenheiros de projeto e verificação. Isso é necessário para que o erro introduzido por algum deles não seja absorvido pelo outro. É aconselhável que eles tenham o mínimo contato possível.

Com relação à sua concepção, existem três aspectos da verificação funcional que devem ser cuidadosamente observados: intenção do projeto, especificação e implementação. De acordo com Piziali [29] esses aspectos podem ser representados através do diagrama da Figura 2.1.

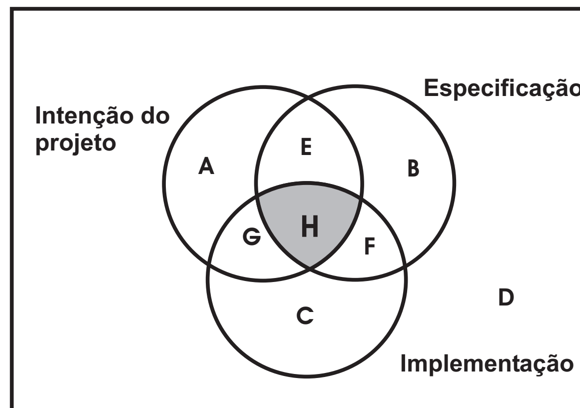

Figura 2.1: Diagrama de projeto

Esse diagrama é composto de três círculos que se sobrepõem. Cada círculo representa um aspecto do projeto.

O círculo intenção do projeto (A, E, H, G) representa o comportamento do projeto, da forma como o engenheiro de projeto o imagina. O círculo especificação (B, F, H, E) representa a documentação da especificação funcional do dispositivo. O círculo implementação (F, C, G, H) é a parte do projeto que o engenheiro consegue implementar a partir da especificação. Além disso, o comportamento não pretendido, não especificado e não implementado, está representado fora do círculo, com a letra D.

Nessa figura, é importante observar que os círculos se sobrepõem em algumas áreas, onde existem pontos comuns. Um exemplo desses pontos comuns é o ponto simbolizado pela letra E. Nesse ponto, estão todos os aspectos capturados pela intenção e especificação do projeto e não satisfeitos pela implementação. Outro exemplo pode ser visto na letra F. Ele mostra aspectos especificados, implementados, mas que não estavam contidos na intenção do engenheiro. No entanto, o mais importante desses espaços é representado pela letra H, pois aí se encontra a sobreposição dos três círculos, representando a interconexão do intento do projeto, a sua especificação e a implementação. Isso quer dizer que os três aspectos do projeto foram cobertos nessa área. Quanto maior for a área da interconexão dos três círculos,

melhor será o projeto construído. Um modelo ideal seria aquele em que os três círculos se sobrepusessem de forma integral.

2.3.1 Black-box, grey-box e white-box

Do ponto de vista da visibilidade de um *testbench* para a verificação funcional, existem três abordagens: Black-box, grey-box e white-box.

A abordagem black-box permite que a verificação funcional seja realizada sem nenhum conhecimento da implementação real do DUV. A verificação é acompanhada nas interfaces do DUV, sem acesso direto aos estados internos dele e sem conhecimento de sua estrutura e implementação. Essa abordagem possui a vantagem de não depender de qualquer detalhe de implementação. Mesmo que o DUV seja modificado durante a fase de verificação, o *testbench* não precisa ser alterado se a interface não for mudada. O ponto negativo dessa abordagem é que se perde um pouco do controle da parte interna da implementação do DUV, uma vez que não se tem acesso aos detalhes de implementação.

Com a abordagem white-box, é possível ter uma visibilidade e controlabilidade completa da estrutura interna do DUV. A vantagem de se usar white-box é que se pode testar diretamente funções do dispositivo sendo verificado, uma vez que se tem acesso a ele e rapidamente localizar o local onde ocorreu uma falha. A desvantagem dessa abordagem é que ela é altamente acoplada com detalhes de verificação. Uma vez que o DUV seja modificado, todo o *testbench* deve ser modificado também. Além disso, é necessário ter conhecimento de detalhes de implementação para criar cenários de testes e saber quais as respostas que devem ser esperadas.

A abordagem grey-box possui características das duas abordagens. Ela procura resolver o problema da falta de controlabilidade do black-box e da dependência de implementação do white-box. Um exemplo de um caso de teste típico grey-box pode ser usado para aumentar as métricas de cobertura. Nesse caso, os estímulos são projetados para linhas de código específicas ou para verificar funcionalidades específicas.

2.3.2 Testbench

A verificação funcional utiliza simulação para verificar o DUV. Para que essa simulação seja possível, é necessário que haja um ambiente de verificação que possa receber o DUV, inserir estímulos e comparar suas respostas com as respostas de um modelo ideal. Esse ambiente é denominado de *testbench*.

Em uma simulação, o DUV tem seus resultados comparados com os resultados da especificação. Essa especificação pode ser implementada em um modelo executável. Esse modelo recebe várias denominações, tais como Golden Model e Modelo de Referência. Para a verificação funcional, esse modelo é considerado como sendo a implementação da especificação e por isso é livre de erros. Nesse trabalho a denominação usada será Modelo de Referência.

A Figura 2.2 mostra a estrutura genérica de um *testbench*. Nessa figura, o *testbench* é representado pelo U invertido, ou seja, a região que envolve o DUV.

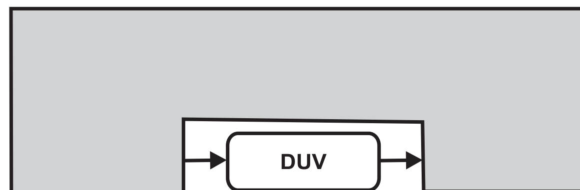

Figura 2.2: Esquema de um *testbench* genérico

Um bom *testbench* deve possuir as seguintes características básicas: ser dirigido por coberturas, possuir randomicidade direcionada, ser autoverificável e baseado em transações [5].

- O *testbench* é dirigido por coberturas se o seu tempo de simulação e os estímulos escolhidos dependem de critérios de cobertura pré-estabelecidos. Durante a especificação da verificação são traçadas todas as metas de cobertura que devem ser alcançadas durante a simulação. Cobertura é um mecanismo para detectar se todas as funcionalidades especificadas foram testadas e será explicado mais adiante.

- Ele possui randomicidade controlada quando são utilizados estímulos randômicos direcionados para testar todas as funcionalidades do projeto e atingir uma determinada cobertura especificada. Os estímulos randômicos não são completamente randômicos, pois a randomicidade é direcionada para atingir valores pré-determinados para

um dado bloco do DUV. É importante observar que esses estímulos, uma vez que são colocados para simular, vão gerar valores dentro da distribuição de probabilidade escolhida e somente vão parar de gerar valores quando a cobertura funcional for atingida. Caso se perceba que a cobertura não foi atingida dentro de um determinado tempo e que ela não está progredindo, deve-se mudar a distribuição de probabilidade de acordo com as funcionalidades não atingidas na cobertura para que as funcionalidades não exercitadas venham a ser exercitadas e a cobertura atingida.

- O *testbench* deve possuir autoverificação, ou seja, ele deve ter mecanismos que permitam a comparação automática das respostas obtidas da simulação com as respostas esperadas a partir da especificação do projeto. O trabalho de verificar manualmente todos os resultados obtidos da simulação é muito árduo e propenso a erros, pois é necessário comparar cada saída do DUV com o resultado esperado. Dessa forma, o *testbench* deve ter uma forma de fazer essa comparação de forma automática, evitando assim esse trabalho manual. O Modelo de Referência é muito importante nessa operação de autoverificação, pois ele permite que os resultados do DUV possam ser automaticamente comparados com os resultados de um modelo ideal, diminuindo as possibilidades de que um erro causado por uma comparação mal feita venha a ocorrer.

- O *testbench* também não possui entradas e nem saídas, imprime mensagens quando o DUV apresenta comportamento inesperado e pode criar arquivos de logs para uma análise posterior.

- Finalmente, o *testbench* deve ser implementado no nível de transações. Uma transação é uma transferência de informações de alto nível, de dados ou controle entre o *testbench* e o DUV sobre, uma interface.

  É muito importante que esses estímulos sejam criados em um alto nível de abstração para que o trabalho seja realizado de forma mais eficiente, sem considerar detalhes de baixo nível, pois os testes são realizados em um nível de sistema. Essa capacidade melhora a depuração e análise da cobertura, apresentando informações em termos de transação e seus relacionamentos, ao invés de sinais e formas de onda.

  O *testbench* possui um nível de implementação diferente do DUV, pois o DUV é im-

plementado no nível de sinais e eles se comunicam com protocolos no nível de sinais.

A metodologia de verificação baseada em transação foi introduzida em 1998 por Richard Goering [18] e integrada em um uma ferramenta de verificação em 1999, vide Santarini [36]. Essa metodologia foi primeiramente usada no laboratório da Cadence [40] e sua descrição pode ser encontrada no trabalho de Brahme [7]. Nesse trabalho é apresentada a metodologia para se trabalhar com transações. Basicamente ela é dividida em duas partes: a camada de cima que lida com as transações sem se preocupar com detalhes específicos de protocolos no nível de sinais e a camada de baixo que faz o mapeamento entre o nível de transação e os protocolos nos níveis de sinais. Com o uso dessa metodologia existem ganhos com reuso, ferramentas de depuração, facilidade de implementação e entendimento por parte dos desenvolvedores. O artigo mostra também os passos utilizados para realizar uma verificação funcional completa, utilizando a biblioteca de C++ *Testbuilder*. Ele aborda todos os principais conceitos para realizar uma verificação funcional completa, tais como: simulação no nível de transações, geração de todos os tipos de estímulos, inclusive o randômico direcionado, autoverificação e coberturas. Além disso, ele mostra que uma verificação deve ter uma forma de ordenamento das transações, tais como FIFOs e semáforos.

O trabalho proposto nesse documento de tese, também utiliza todos esses conceitos de verificação funcional descritos na metodologia do trabalho de Brahme, embora utilize o *Testbuilder-SC* no lugar do *C++ Testbuilder*. Difere também na abordagem, pois gera o *testbench* antes do DUV, ao contrário da abordagem proposta no trabalho de Brahme.

2.3.3 Cobertura

Na verificação funcional, um grande problema é saber quando parar a simulação de forma que a verificação esteja completa, ou seja, quando o DUV está totalmente verificado. Esse problema decorre do fato de que a simulação em si não consegue afirmar se todas as funcionalidades implementadas foram verificadas. É muito importante que exista um mecanismo para detectar se todas as funcionalidades especificadas foram testadas.

Pode-se tentar resolver esse problema de testar as funcionalidades especificadas através do uso de geradores de testes randômicos. Os geradores de testes randômicos estão se tornando cada vez mais avançados e realmente tem ajudado a melhorar a qualidade da verifica-

ção dos dispositivos, pois eles geram valores aleatórios, que se simulados durante um tempo suficiente, podem cobrir todos os valores necessários para estimular o dispositivo. Porém esses geradores de testes sozinhos não podem garantir e nem mostrar quanto das funcionalidades especificadas foram testadas ou ainda quando elas foram testadas mais de uma vez. Para responder estas perguntas, tem sido usada cobertura funcional.

Cobertura Funcional é uma técnica usada para medir o progresso da simulação e reportar quais funcionalidades deixaram de ser testadas. Cobertura pode ajudar no monitoramento da qualidade de testes e direcionar esses testes para criar geradores que cobrem áreas que não foram adequadamente testadas. A parte do espaço de cobertura não testado é chamada de buraco de cobertura.

Uma das formas de realização da cobertura funcional é através do modelo de cobertura *cross-product*, também denominado de cross-coverage. Esse modelo é criado testando-se todas as possibilidades existentes dentro de um conjunto de funcionalidades definidas. Por exemplo, o par de funcionalidades <pedido, resposta>, em que pedido é um dos possíveis pedidos que podem ser mandados para a memória (ex. escrita de memória, leitura de memória, etc) e resposta é uma das possíveis respostas (aceita, não aceita, rejeita, etc). O espaço de cross-coverage é formado pelo produto cartesiano de todas essas funcionalidades e deve ter cada uma delas testada.

Um problema que deve ser descoberto pela cobertura funcional é o buraco de cobertura. O buraco de cobertura é definido como sendo as funcionalidades que não foram testadas durante a simulação do dispositivo.

Esse buraco de cobertura pode ter três causas:

- Funcionalidades não exercitadas porque o simulador precisa de mais tempo para exercitá-las.

- Funcionalidades não exercitadas porque não foram gerados estímulos suficientes para estimular todas as funcionalidades do DUV.

- Existem erros no dispositivo que não permitem que as funcionalidades sejam testadas.

O buraco de cobertura deve ser descoberto e solucionado, pois ele pode levar a uma verificação incompleta.

A principal técnica para mostrar que uma simulação obteve sucesso, é analisar a cobertura. Isso significa criar uma lista de tarefas (funcionalidades especificadas a serem testadas) e observar se cada uma dessas funcionalidades foi testada durante a fase de simulação.

O primeiro passo para implementar a cobertura funcional é criar um modelo de cobertura. O modelo de cobertura faz parte da especificação da verificação e é o documento que se deve ter como base para a implementação da cobertura funcional. Não é recomendável que se inicie a cobertura funcional diretamente na implementação do código da cobertura. Inicialmente deve-se ter certeza de ter entendido toda a funcionalidade do dispositivo a ser verificado e estar certo de ter extraído todas as informações que devem ser verificadas no mesmo.

O modelo de cobertura consiste em isolar os atributos "interessantes"e construir a sua história. Entenda-se aqui por atributos interessantes, todos aqueles que devem ser analisados no momento de se fazer uma análise de cobertura funcional, por serem considerados difíceis ou propensos a erros. Além disso, devem ser definidas restrições para o modelo. As restrições são partes da simulação, são combinações de atributos que nunca devem ocorrer. Caso a parte de atributos definidos como ilegais seja simulada, isso pode demonstrar que existe um erro na implementação do projeto.

Alguns autores discutem e oferecem soluções para a cobertura funcional e para a análise e solução dos buracos de coberturas identificados na verificação do DUV.

No trabalho de Lachish [25] são descritos métodos para descobrir, reportar e analisar grandes espaços não cobertos, ou seja, buracos de cobertura. Esses buracos de cobertura são calculados baseando-se nos espaços de cobertura criados através do modelo de cobertura funcional *cross-product*. Os buracos de cobertura agrupam propriedades que não foram cobertas e que compartilhem propriedades em comum. Basicamente, essa técnica busca encontrar tarefas não cobertas que tenham valores comuns em alguns de seus atributos e achar também algum denominador comum nos valores dos atributos que os distinguem. O espaço de cobertura é composto de quatro componentes:

- Descrição semântica do modelo (história do modelo);
- Lista de atributos mencionados na história;
- Conjunto de todos os valores possíveis para cada atributo;

- Lista de restrições e combinações nos valores de atributos do cross-coverage.

Cada atributo pode ser particionado em um ou mais conjuntos de valores similares semanticamente. Os modelos de cobertura funcionais podem ser black_box ou white_box. Esses modelos podem ser também aplicados em um ponto no tempo ou modelos temporais, cujas tarefas correspondem a um estado da aplicação. As três formas propostas para a descoberta e análise desses buracos de coberturas são as seguintes:

- Buracos agregados. Esse tipo de buraco é descoberto entre tarefas que não são quantitativamente similares. Por exemplo, entre tarefas que possuem a mesma distância de Hamming entre elas (o número de atributos nos quais os dois buracos diferem é igual).

- Buracos particionados. Essa segunda técnica agrupa buracos que são conceitualmente similares. Valores que são agrupados juntos, normalmente são analisados juntos e buracos que são descobertos em um valor, podem ocorrer para o grupo todo.

- Buracos projetados. Esses buracos são definidos baseados no grupo a que eles pertençam. A dimensão desse buraco projetado é o numero de atributos que não possuem valores específicos.

Uma idéia quando se está implementando o modelo de cobertura é separá-lo da ferramenta de análise, como no trabalho de Grinwald [20], em que o foco principal é separar o modelo de cobertura da ferramenta para fazer a análise posterior. De acordo com esse trabalho, tendo uma ferramenta independente do modelo de cobertura, pode-se elaborar um modelo de cobertura que mais se encaixe dentro do dispositivo que se deseja verificar, pois o usuário pode mudar o escopo ou a profundidade da cobertura durante o processo de verificação. Isso se deve ao fato de que, em algumas ferramentas, o modelo de cobertura é tão genérico que não consegue ser aplicado para uma necessidade específica, sendo necessário por isso que o usuário implemente a sua própria ferramenta de cobertura. A ferramenta de análise utilizada é denominada COMET (Coverage measurement tool). Essa ferramenta gera todas as funcionalidades que são comuns aos modelos de cobertura específicos e aos modelos genéricos, tais como relatórios de tarefas e relatórios de eventos ilegais. Esses relatórios mostram o progresso de cada atributo e o progresso da cobertura como uma função do tempo. O modelo de cobertura criado é baseado em atributos. Ele é dividido em duas

partes: *snapshot* e temporal. O *snapshot* se preocupa somente com eventos instantâneos, enquanto o temporal pode fazer o cruzamento de dois eventos em tempos distintos.

O trabalho de Asaf [3], visa resolver o problema causado pela grande dimensão da cobertura funcional em uma verificação. Um dos problemas que ocorrem ao tentar implementar a cobertura funcional é causado pela tentativa de cobrir todos os casos da cobertura de uma só vez, o que torna o número de eventos a serem cobertos muito grande e difícil de analisar. A solução apresentada nesse trabalho é definir vistas para a realização de cobertura funcional *cross product*, enfocando em partes específicas do modelo. A cobertura *cross-product* compreende todas as possíveis combinações de valores para um dado conjunto de atributos.

São apresentadas as três operações básicas para a definição de vistas de cobertura funcional. Essas operações, utilizadas na restrição e seleção do que deve ser analisado na cobertura, são: projeção, seleção e agrupamento. Na projeção os dados cobertos são projetados em um subconjunto de valores escolhidos do *cross-product*. Essa operação deve ser realizada porque procurar erros em todos os valores gerados pode ser muito incômodo e propenso a erros. A segunda operação seleciona eventos específicos, isso permite ignorar partes que não são interessantes para a verificação. Essa operação permite selecionar e filtrar eventos. A terceira operação cobre eventos juntos, podendo agrupar sempre eventos que pertençam a uma mesma classe de funcionalidades e podendo olhar suas respostas combinadas, por exemplo, agrupar sempre eventos de E/S. Usando vistas definidas para cobertura pode-se facilitar bastante o trabalho da realização da cobertura funcional de componentes muito grandes.

Outro problema que se encontra diretamente ligado à cobertura é a geração de estímulos. Os estímulos devem ser gerados de forma a direcionar a simulação para alcançar os critérios de cobertura estabelecidos para que a simulação exercite todas as funcionalidades especificadas.

2.3.4 Geração de estímulos

Durante a simulação para a realização da verificação funcional, o DUV deve ser estimulado por uma ampla variedade de combinações de valores. O ideal é que sejam previstas situações que venham a testar todas suas funcionalidades. De forma a conseguir esse objetivo, o DUV deve ser simulado com os seguintes estímulos: estímulos citados na especificação, situações críticas, estímulos randômicos e casos de testes reais. Com o uso desses estímulos, a simu-

lação deve atingir a cobertura proposta na especificação da cobertura. Caso essa cobertura não seja alcançada, é importante que esses estímulos sejam mudados de forma a alcançar tal cobertura.

A geração de estímulos pode ser feita de forma manual ou através de alguma ferramenta. A geração manual pode ser pouco eficiente para alcançar a cobertura, pois nem sempre cobre todas as possibilidades necessárias. A geração através de ferramentas especializadas possui uma chance maior de detectar erros, que possam até mesmo escapar da especificação. Existem diversos trabalhos que procuram melhorar a qualidade dos estímulos em uma verificação, esses trabalhos focam principalmente na geração de estímulos e na obtenção de uma boa cobertura para a simulação.

O trabalho de Ferrandi [14] apresenta uma solução para a criação de vetores de testes para o DUV, através da técnica ATPG (*Automatic Test Pattern Generation*), baseada em controlabilidade e observabilidade. O DUV é descrito como sendo uma máquina de estados finitos. O método para realizar a geração de testes é baseado na simulação de uma seqüência de entradas. O algoritmo proposto introduz o conceito de seqüência de transição, como sendo um caminho finito, para resolver o problema de ter que gerar um número infinito de caminhos. De acordo com as técnicas mostradas no algoritmo a seguir, pode ser obtido um conjunto de seqüências, em que cada seqüência é um conjunto ordenado de restrições. Esse algoritmo ajuda a alcançar o maior caminho de cobertura possível.

O algoritmo de geração de testes é composto de cinco tarefas principais:

- Aquisição de dados: Todas as informações sobre o sistema são processadas. Essas informações são as listas de portas, sentença, instruções condicionais e transições obtidas para a análise do código fonte em SystemC;

- Análises das transições: Consiste em encontrar os estados iniciais e finais de cada transição;

- Enumeração de seqüência: Enumera vários caminhos de execução em potenciais no SystemC, para os quais, vetores de teste são necessários.

- Análises de seqüências e produção de restrições: para cada seqüência considerada, um conjunto de restrições é produzido. Esse conjunto de restrições corresponde a

todas as instruções condicionais que devem ter como resultado o valor TRUE durante a execução da seqüência. Quando essas restrições são aplicadas para variáveis e sinais (não para portas), não é possível forçar o valor das variáveis e sinais para um dado valor, portanto deve-se retroceder no código, até encontrar qual porta ou constante deu origem a essa seqüência de variáveis e sinais.

- Resolvedor de restrições e extração de testes: Finalmente, o resolvedor de restrições gera uma solução, caso exista, para satisfazer um conjunto de restrições associado com a seqüência.

Esse trabalho [14] possui o foco voltado para a geração de vetores de teste com o objetivo de alcançar um critério de cobertura. Ele utiliza o resolvedor de restrições denominado GProlog. No entanto, não existe nenhuma especificação de como os critérios de cobertura podem ser medidos.

Outro exemplo de geração de estímulos aliado com a controlabilidade da cobertura pode ser visto no trabalho de Benjamin [4]. Nesse trabalho, o autor descreve a importância da cobertura para a geração de estímulos. Além disso, ele apresenta uma metodologia para verificação funcional que se diferencia das demais através da forma de geração de estímulos. Essa metodologia é híbrida, são integrados os métodos de simulação e verificação de modelos (formal).

O gerador de testes dirigido pela cobertura é um programa que acha um caminho através da máquina de estados finitos do projeto que satisfaz cada tarefa do modelo. O autor denominou esse caminho de teste abstrato. Ele utiliza algoritmos para essa geração de testes abstratos, utilizando modelos de cobertura de transição e modelos de coberturas de estados. Esses algoritmos são implementados na ferramenta GOTCHA. Os resultados de testes esperados são produzidos por um simulador arquitetural, que é o Modelo de Referência para a verificação. Isso significa que nenhum teste gerado contém erros. O autor define também testes concretos. Esses testes são seqüências de instruções que forçam o projeto através de um caminho pré-definido nos testes abstratos, formando a especificação de testes. Esses testes são utilizados na simulação do DUV.

O trabalho de Dudani [28] também aborda formas de gerar testes para preencher o espaço existente entre a cobertura real obtida através de dados de testes fornecidos e os objetivos de

cobertura requeridos. A solução adotada é baseada na implementação de quatro técnicas muito importantes para melhorar a eficiência da verificação: especificação de *assertions*; cobertura funcional; simulação randômica e *testbenches* reativos. Nesse trabalho são discutidas as seguintes definições:

- Especificação de *assertions*: *Assertions* normalmente são chamadas de checker ou monitor. Elas são a descrição do comportamento esperado quando o projeto é estimulado com uma entrada. Para verificar uma característica do DUV, primeiro a sua funcionalidade precisa ser entendida, depois ela precisa ser explicitamente descrita e finalmente as condições sob as quais o comportamento é aplicado precisam ser estabelecidas. Essa descrição é implementada em uma *assertion*.

- Cobertura funcional: A cobertura funcional é baseada na funcionalidade do projeto. Ela mede quanto da funcionalidade do projeto está sendo verificada. Usando *assertions* como ponto inicial da cobertura funcional, pode-se descrever cenários funcionais mais elaborados, para capturar *corner cases* e todas as combinações de dados e controle.

- Simulação randômica: a simulação randômica é criada para exercitar as funcionalidades básicas do projeto. Simulações randômicas podem ser manipuladas para exercitar áreas críticas e são normalmente ajustadas através de "tentativa e erro".

- *Testbenches* reativos: Normalmente os estímulos randômicos são ajustados manualmente trocando a "semente"de geração. O uso de cobertura funcional durante a verificação pode dar um *feedback* de quais as funcionalidades que ainda precisam ser verificadas. O *testbench* reativo funciona de forma que o *feedback* recebido da cobertura funcional dirige o *testbench* para fazer os ajustes necessários para criar estímulos mais efetivos para a verificação funcional.

Alguns trabalhos são discutidos como forma de implementação de cada uma dessas definições e no final é proposta uma metodologia baseada nesses trabalhos. A metodologia usa *assertions* para exercitar vários caminhos dentro de um cenário e a cobertura funcional enumera esses caminhos e monitora cada um deles. No final, têm-se quais os caminhos que

foram exercitados, com o objetivo de encontrar erros não cobertos pela simulação. Os testes são direcionados com o feedback da cobertura.

No trabalho de Hekmatpour [22] são propostos métodos para melhorar a qualidade dos geradores de testes nos vários estágios da verificação funcional. Ele descreve métodos para calibrar o gerador de testes e melhorar a cobertura funcional. O trabalho propõe que a cobertura de 100% dos testes seja feita através das seguintes técnicas:

- Uso de gerador de testes baseado em modelos, onde o gerador de teste utiliza randomicidade e controle. O usuário deve poder enriquecer a saída do gerador por um conhecimento prévio dos testes;

- Utilização da randomicidade de forma a controlar a quantidade de erros sendo encontrada. Quando essa quantidade for decrescendo, mudar a randomicidade de forma controlada para capturar novos erros;

- Geração de testes randômicos de forma incremental. Sempre entre um teste e outro deve ser feita uma análise da cobertura.

- Utilização de testes randômicos dirigidos pelos erros encontrados. Essa técnica deve ser utilizada para garantir que esse erro não volte a acontecer e que não há mais nenhum erro próximo a esse e/ou relacionado a ele.

- Existência de um plano de verificação dinâmico, que reflita o estado presente do projeto. Ele deve ser revisto e atualizado sempre que algo seja mudado no projeto, até mesmo após um novo erro ser encontrado.

- Escolha dos requisitos para o gerador de testes. Deve-se elaborar um plano para o gerador de testes.

- Deve-se calibrar o gerador de testes. Para isso é necessário desenvolver um qualificador, que possa detectar que o comportamento do gerador de testes está consistente, de acordo com o esperado.

- Gerenciamento da verificação, guiando-se pela cobertura.

A metodologia, apresentada no trabalho de Hu [23], é um fluxo de projeto para a verificação funcional de componentes de hardware, que vai desde a especificação até a prototipação do projeto. Nessa metodologia são definidos papéis para os engenheiros de projeto e verificação e cada parte do projeto é definido de acordo com a metodologia proposta por eles. Há uma proposta de paralelização do trabalho dos engenheiros de projeto e verificação para uma maximização e melhor aproveitamento do tempo da equipe. Na Figura 2.3 o fluxo de projeto é dividido entre as equipes de projeto e verificação. As partes cinzas são as tarefas do engenheiro de verificação, as partes brancas são as atividades do engenheiro de projeto e os quadros pretos são as atividades realizadas por ambos.

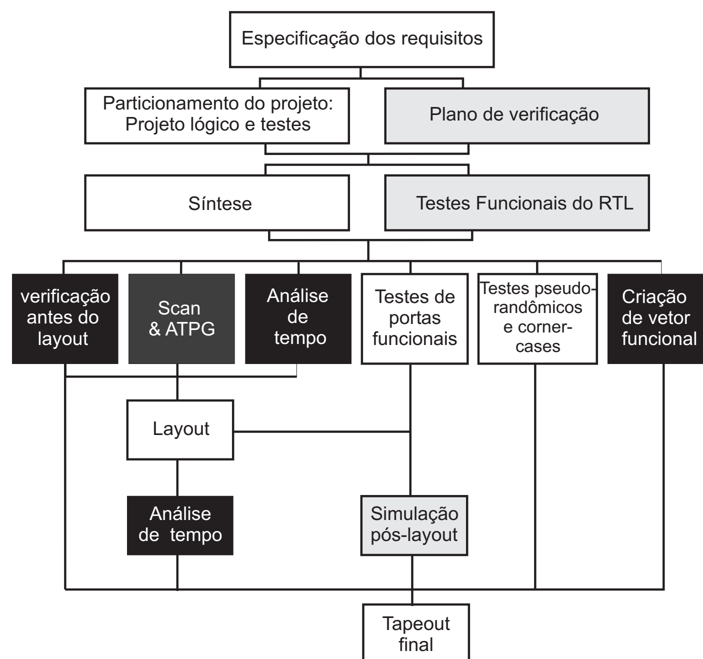

Figura 2.3: Fluxo de projeto do trabalho de Hu

Com a apresentação desses trabalhos é possível observar que existem diversas formas de criação de testes para estimular o DUV durante a verificação funcional, bem como diferentes abordagens que podem ser empregadas durante essa verificação.

A próxima Seção mostra a biblioteca de classes SystemC, utilizada nesse trabalho para a construção do *testbench* para a verificação funcional.

2.4 SystemC

SystemC [6; 21] é uma biblioteca de classes da linguagem de programação C++ padronizada pela OSCI (*Open SystemC Initiative*), uma organização composta por uma variedade de companhias, universidades e indivíduos. Ela é orientada a objetos e estende a capacidade do C++ para permitir a modelagem de hardware. Para isso, ela adiciona importantes conceitos como concorrência, eventos e tipos de dados de hardware, tais como sinais, portas e módulos.

Para permitir que o SystemC tivesse recursos para a verificação, foi criada uma biblioteca de classes chamada de *SystemC Verification Library*-SCV. A biblioteca SCV fornece suporte para: criação de APIs para a verificação baseada em transações, randomização direcionada, manipulação de exceções e outras características de verificação. A SCV permite programação no nível de transações, uma metodologia que habilita abstração de alto nível, reutilização de código e alta velocidade de simulação.

Outro fator importante para a escolha do SystemC está descrito no trabalho de Fin [15]. Esse trabalho mostra que SystemC pode ser usado para geração de testes para verificação funcional, descrever módulos em diferentes níveis, tais como módulos em RTL e módulos de software, sendo possível passar de um nível de abstração para o outro sempre usando a representação do SystemC. Com essa possibilidade torna-se viável o procedimento de teste aplicável para todas as fases de projeto.

No trabalho de Randjic [30] é descrita uma forma de verificação do processador de rede *PcomP*, através da biblioteca de C++, SystemC.

O dispositivo *PcomP* é um processador de comunicação totalmente programável. O DUV é ligado ao *testbench* via funções BFM *(Bus Functional Model)* que representam APIs, como mostrado na Figura 2.4. O módulo Master é responsável pelos parâmetros de entrada e por iniciar os testes apropriados, chamando os devidos seqüenciadores. O seqüenciador, por sua vez, prepara os dados de entrada para os processos stimuli/checker, faz uma análise dos dados, espera a execução e retorna os dados do processo. Os dados passados para o módulo CLK podem ser gerados utilizando muitos tipos de randomização. Os seqüenciadores obtêm esses dados do processo Master. Todos os dados de geração e controle são deixados para o bloco RND. A troca de dados entre seqüenciadores e CLK é feita via FIFOs. Em seguida, o processo CLK chama as funções BFM e faz todas as ações necessárias nos parâmetros. Elas

usualmente estimulam e checam o comportamento do DUV. Os dados de entrada são lidos de um arquivo de entrada e os dados de saída são escritos em um arquivo de saída.

Através desse método chegou-se a conclusão de que é possível realizar qualquer teste funcional chamando o processo CLK de forma apropriada.

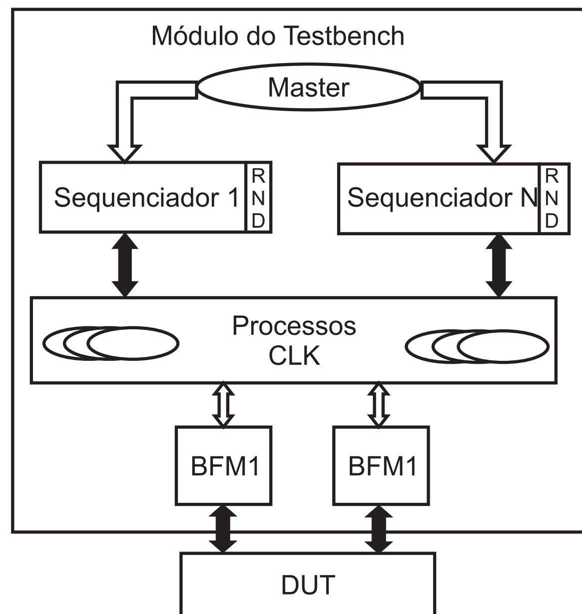

Figura 2.4: Diagrama de blocos do *testbench* do trabalho de Randjic

O trabalho descrito em Romero [34], faz a comparação entre dois métodos de verificação hierárquicos: o método desenvolvido pelo VeriSC tradicional [12] e um método de aceleração da simulação, baseado em um filtro para retirar estímulos redundantes na simulação do código RTL e acelerar o processo de verificação. O trabalho relata que a simulação do Modelo de Referência é mais rápida que a simulação do DUV. Por isso, ele propõe que o Modelo de Referência seja simulado com os dados e que esses dados sejam analisados, de forma a verificar se eles são ou não redundantes (já atingiram a cobertura esperada). Os dados redundantes são então enviados para um filtro, o qual vai impedir que o RTL venha a ser simulado com esses dados redundantes.

2.5 Considerações sobre conceitos de verificação funcional

Todos os trabalhos apresentados no estado da arte implementam o DUV antes do *testbench* para a realização da verificação funcional. Isso faz com que a parte de maior impacto em um

Autor	Trabalho	Objetivo
Cadence	The transaction-based verification Methodology.	Metodologia que estabelece transações.
Lachish	Hole Analysis for Functional Coverage Data.	Encontra buracos de cobertura.
Asaf	Defining Coverage Views to improve Functional coverage.	Define formas de agrupar as funcionalidades para melhorar os critérios de cobertura.
Ferrandi	Functional Verification for SystemC Descriptions using Constraint Solving.	Cria de vetores de testes através do uso de ATPG.
Benjamin	A study in Coverage-Driven Test Generation.	Usa técnica híbridas (funcional + formal) para implementar cobertura.
Grinwald	User Defined coverage: a tool supported methodology for design verification.	Separa a ferramenta de cobertura da metodologia de verificação.
Dudani	High Level Functional Verification Closure.	Trabalho baseado na implementação de: assertions, cobertura funcional, simulação randômica e *testbenches* reativos.
Hekmatpour	Coverage-Directed Management and Optimization of Random Functional Verification.	Propõe métodos para melhorar a qualidade da geração de testes para melhorar a cobertura funcional.
Hu	A methodology for Design Verification.	Propõe um fluxo de paralelização do trabalho dos engenheiros de projeto e verificação.
Fin	SystemC: a homogeneous environment to test embedded systems.	Mostra as formas de uso do SystemC em diferentes níveis de abstração.
Randjic	Complex ASICs Verification with SystemC.	Descreve a verificação de um processador usando SystemC.
Romero	Comparing two *testbenches* methods for Hierarchical Functional methods for hierarchical Functional verification of a Bluetooth Baseband Adaptor.	Propõe um filtro na metodologia VeriSC tradicional para retirar estímulos redundantes e acelerar o processo de simulação.

Tabela 2.1: Comparação entre trabalhos da área

projeto de componentes digitais, a verificação funcional, não seja priorizada no momento da implementação do DUV. A tabela 2.1 mostra o resumo desses trabalhos e a área de abordagem de cada um deles.

Muitos trabalhos [32; 3; 14; 4; 16; 28; 35] são específicos para uma parte da metodologia de verificação funcional, tal como a elaboração de estímulos eficientes ou implementação de cobertura funcional para tentar exercitar ao máximo as funcionalidades do projeto. As ferramentas comerciais são mais completas na sua abordagem [41; 10; 19; 40], no entanto são somente plataformas para verificação, não possuem uma metodologia para guiar a verificação funcional, embora contenha todos os requisitos para realizá-la.

A metodologia VeriSC, apresentada neste trabalho de doutorado, diverge do fluxo de verificação tradicional dos trabalhos apresentados nesse capítulo, propondo um fluxo de verificação funcional melhor integrado no desenvolvimento do projeto. Ela aborda as partes de geração de estímulos, implementação da cobertura funcional e fluxo de projeto.

A metodologia VeriSC resolve o problema de reuso de código, uma vez que todas as interfaces que são iguais ou que se comunicam de alguma forma, podem reusar código que já foi gerado, como pode ser visto no Capítulo 3.

Outro problema resolvido pela metodologia, que não é proposto nos demais trabalhos, é que como ela gera o testbench antes do DUV, é possível usar esse testbench para fazer testes na implementação do DUV durante sua implementação pelo engenheiro de projeto.

Além disso, a metodologia VeriSC faz testes exaustivos no ambiente de verificação para garantir que ao encontrar erros durante a simulação, esses erros sejam devido ao DUV e não ao ambiente de verificação.

Capítulo 3

Metodologia VeriSC

A metodologia de verificação VeriSC pode ser aplicada para a verificação funcional de DUVs digitais síncronos que utilizam um único sinal de relógio. A utilização de um único relógio não inviabiliza a aplicação em circuitos que tenham mais de um relógio, se este pode ser dividido em vários circuitos com um único relógio.

Em um trabalho anterior [12], encontra-se a descrição da primeira parte da metodologia VeriSC (metodologia VeriSC tradicional), que utiliza a mesma estrutura de blocos da metodologia VeriSC, porém com uma abordagem tradicional, onde a implementação do código RTL precede a verificação. Essa metodologia VeriSC tradicional resultou em uma ferramenta para geração de *prototypes* para *testbenches* e foi utilizada no projeto BrazilIP [24] para a verificação de parte de um decodificador MPEG4.

Através dos resultados obtidos com essa metodologia VeriSC tradicional, surgiu a idéia da implementação da metodologia VeriSC apresentada nesse trabalho, onde o *testbench* é implementado antes do DUV. Dessa forma, o *testbench* está pronto para ser utilizado antes mesmo do desenvolvimento do DUV ser iniciado.

A metodologia VeriSC é composta de um novo fluxo de verificação, que não se inicia pela implementação do DUV. Nesse fluxo, a implementação do *testbench* e do Modelo de Referência antecedem a implementação do DUV. Para permitir que o *testbench* seja implementado antes do DUV, a metodologia implementa um mecanismo para simular a presença do DUV com os próprios elementos do *testbench*, sem a necessidade da geração de código adicional que não é reutilizado depois. Com esse fluxo, todas as partes do *testbench* podem estar prontas antes do início desenvolvimento do DUV. Esse fluxo será explicado na

Seção 3.2.

A metodologia VeriSC é uma metodologia que adota o conceito de projetos com hierarquia, de forma que um projeto pode ser dividido em partes a serem implementadas e verificadas. O fluxo para a realização da verificação funcional seguindo a metodologia VeriSC, abordado nesse trabalho, pode ser visto na Figura 3.1 e é composto das seguintes partes:

- Implementação de um plano de verificação.
- Implementação de todos os *testbenches* necessários para o DUV completo e os DUVs resultantes de hierarquização.
- Implementação do DUV. Essa implementação é seguida pela simulação do DUV, juntamente com o *testbench*.
- Captação dos dados de simulação através da coleta dos itens de cobertura e dos logs da simulação.
- Análise da cobertura funcional, que pode levar a uma mudança de estímulos no *testbench* e a uma nova simulação.

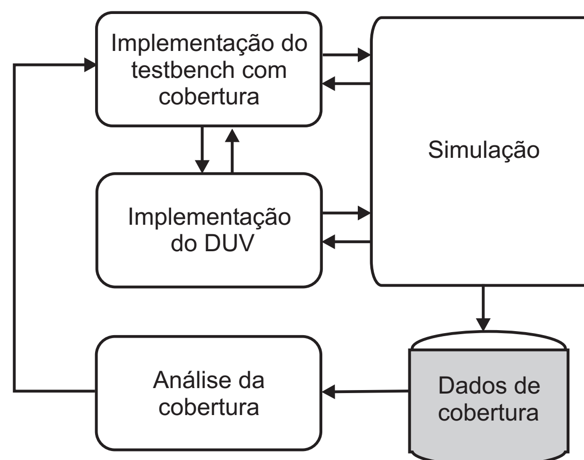

Figura 3.1: Fluxo de verificação

Essa abordagem possui várias vantagens se comparada com a abordagem tradicional onde o engenheiro de verificação espera até que o DUV esteja pronto para iniciar o desenvolvimento do *testbench*. Entre as vantagens podem ser citadas:

1. O tempo de integração do *testbench* com o DUV será praticamente nulo, pois o *testbench* é gerado antes do DUV e também é gerado um *prototype* para a implementação do DUV, com as interfaces já integradas às interfaces do *testbench* (como explicado na próxima seção).

2. Tendo um *testbench* para fazer a verificação a partir do início do projeto do DUV, os testes preliminares, que deveriam ser feitos no DUV, podem ser feitos utilizando o *testbench*, o que economiza um tempo significativo, como será visto na seção de resultados.

3. O *testbench* passa por uma fase de depuração muito bem elaborada, composta de vários passos, até o *testbench* pronto, para que a quantidade de erros nele possa ficar próxima a zero.

4. A metodologia faz uso de muito reuso durante a implementação do *testbench*. Isso faz com que alguns blocos já considerados corretos não tenham que ser novamente implementados, economizando dessa forma um tempo significativo de trabalho. A metodologia também verifica a decomposição hierárquica como parte da implementação do DUV, minimizando problemas no momento da junção de sub-blocos do DUV para formar um bloco completo.

A metodologia oferece vários *templates* para a criação do *testbench*, que serão explicados na próxima seção.

3.1 Construção do *testbench*

O *testbench* é o ambiente de simulação no qual o DUV é inserido para realizar a verificação. Esse ambiente estimula o DUV e compara suas respostas com as respostas de um modelo ideal. O *testbench* da metodologia VeriSC é implementado no nível de transações.

Na Figura 3.2 pode-se ver a composição do *testbench* que foi usado nesse trabalho. O *testbench* é a parte da figura em forma de U invertido. Ele é composto pelos seguintes blocos: Source, TDriver(s), TMonitor(es), Modelo de Referência e Checker, sendo que cada bloco do *testbench* é ligado ao outro por FIFO(s), representadas na figura como setas mais largas.

O *testbench* é ligado ao DUV por interfaces formadas por sinais, representados na figura por setas mais finas.

- O testbench é ligado por FIFOs. As FIFOs exercem uma tarefa muito importante no testbench, pois elas são responsáveis por controlar o **sequenciamento** e o sincronismo dos dados que entram e saem delas. Os dados das FIFOs devem ser retirados de um em um e comparados, de forma que sempre estarão na ordem correta de comparação.
- O DUV *(Design Under Verification)* é o projeto que está sendo verificado. Ele deve ser implementado em RTL, ou seja, no nível de sinais. Por isso, ele precisa de algum mecanismo para se comunicar com o testbench que trabalha em *Transaction Level Modelling* (TLM). Essa comunicação se faz com a ajuda dos blocos TDriver(s) e TMonitor(es) que traduzem sinais para transações e vice-versa.
- O Source é responsável por criar estímulos para a simulação. Esses estímulos devem ser cuidadosamente escolhidos para satisfazer os critérios de cobertura especificados. Todos os estímulos criados pelo Source são transações.
- O *testbench* possui um TDriver para cada interface de entrada do DUV (as interfaces de entrada são as comunicações do DUV com outros blocos que enviam dados para ele). O TDriver é responsável por converter as transações, recebidas pelo Source, em sinais e submetê-los para o DUV. Ele também executa o protocolo de comunicação (*handshake*)com o DUV, através de sinais.
- O *testbench* possui um TMonitor para cada interface de saída do DUV (cada interface de saída representa um bloco que recebe dados do DUV). O TMonitor executa o papel inverso do TDriver. Ele converte todos os sinais de saída do DUV para transações e repassa as mesmas para o Checker, via FIFO. Além disso, ele também executa um protocolo de comunicação com o DUV, através de sinais.
- O módulo Checker é o responsável por comparar as respostas resultantes do TMonitor e Modelo de Referência, para saber se são equivalentes.
- O Modelo de Referência contém a implementação ideal (especificada) do sistema. Por isso, ao receber estímulos, ele deve produzir respostas corretas.

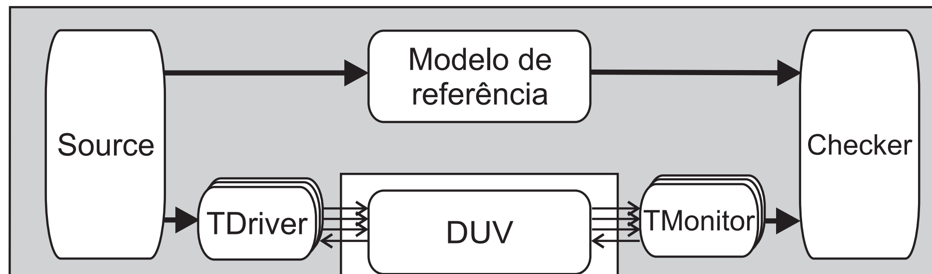

Figura 3.2: Diagrama do *testbench* da metodologia VeriSC

O *testbench* da metodologia VeriSC possui as seguintes características básicas:

- É dirigido por coberturas. A verificação funcional deve parar apenas quando os critérios de cobertura funcional forem alcançados. Além disso, os estímulos são direcionados pelo progresso da cobertura funcional. Se a cobertura não estiver evoluindo, os estímulos devem ser mudados e direcionados para que a cobertura funcional seja atingida.

- Possui randomicidade direcionada. Os estímulos randômicos são direcionados para que a cobertura seja atingida, como explicado anteriormente.

- É autoverificável. O *testbench* compara automaticamente os resultados vindos do Modelo de Referência e do DUV.

- O *testbench* é baseado em transações. Os estímulos gerados pelo módulo Source do *testbench* são todos no nível de transações.

Essas características fazem com que a metodologia permita a criação de um *testbench* compatível com o estado da arte [5] e de cenários de testes que possibilitem exercitar as funcionalidades especificadas em um plano de verificação [11].

O *testbench* deve simular ao mesmo tempo o Modelo de Referência e o DUV e comparar se as respostas de ambos são equivalentes. O DUV precisa ser implementado em nível de sinais (RTL) e o Modelo de Referência em nível de transações para facilitar a sua depuração e implementação.

3.1.1 Templates

A metodologia VeriSC estabelece uma implementação padrão para cada elemento do *testbench* que é denominada de *template*. Esse padrão pode ser usado como um roteiro para a geração do *testbench*, sendo que as partes específicas e particulares a cada DUV devem ser manualmente implementadas pelo engenheiro de verificação. Essa padronização dos elementos do *testbench* viabiliza o reuso, diminui a quantidade de erros no código e diminui o tempo de implementação, já que várias partes do *testbench* já se encontram semiprontas.

Exemplos dos *testbenches* serão mostrados no decorrer dessa seção. Para entender melhor o funcionamento dos *templates*, será utilizado um exemplo, que é parte do decodificador MPEG4, desenvolvido pelo projeto BrazilIP [24]. Esse exemplo é um DUV composto por dois blocos: Predição Inversa AC/DC *PIACDC* e Quantização Inversa *QI*, como apresentado na Figura 3.3. Essa figura mostra os dois blocos e suas interfaces.

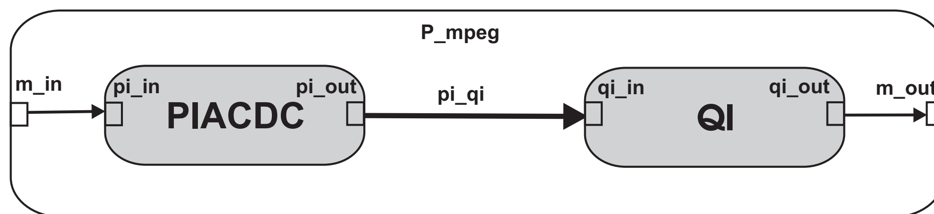

Figura 3.3: Exemplo de um DUV (*P_mpeg*) formado por parte do MPEG4

Nesse exemplo, é importante observar que o bloco *PIACDC* possui como interface de entrada *pi_in* e interface de saída *pi_out*. O bloco *QI*, por sua vez possui *qi_in* como interface de entrada e *qi_out* como interface de saída. O DUV completo (*P_mpeg*) possui como interface de entrada *m_in* e como interface de saída *m_out*. A FIFO que liga os dois blocos é denominada de *pi_qi*. Pela figura é possível deduzir que as interfaces *m_in* e *pi_in* são iguais. Da mesma forma as interfaces *qi_out* e *m_out* também são iguais.

O *testbench* possui partes que são comuns para todos os *testbenches*. As partes do *testbench* que são comuns são denominadas de *templates*. Nesses *templates* estão implementadas as partes dos módulos Source, TDriver, TMonitor, Modelo de Referência, Checker, FIFOs, Reset-driver, Pré-source e Sink que podem ser aproveitadas todas as vezes que for necessário implementar um novo *testbench*, independente de qual DUV será verificado. Esses *templates* não podem ser executados antes de serem adaptados ao DUV. Essa adap-

tação é feita por uma ferramenta, descrita a seguir, que cria *prototypes* a partir de *templates*. Nos *prototypes* já se encontram propriedades estruturais específicas do DUV a ser verificado, como sinais de entrada, sinais de saída e etc. Os *prototypes* podem ser compilados e executados como tal. No entanto, para que tenham utilidade no processo de verificação é necessária ainda a intervenção manual do engenheiro de verificação, para inserir elementos funcionais específicos do DUV a ser verificado.

Os *templates* foram definidos como parte desse trabalho e parte do trabalho de um trabalho de mestrado. O trabalho de dissertação é implementar a ferramenta eTBc (Easy Testbench Creator), que gera automaticamente *prototypes* para um DUV específico.

Após todos os *prototypes* para o DUV serem gerados, o engenheiro de verificação deve acrescentar expressões que não podem ser inferidas automaticamente na geração dos *prototypes*, como por exemplo, critérios de cobertura, distribuições de probabilidade para a geração randômica de estímulos, etc.

A ferramenta eTBc possui o esquema mostrado na Figura 3.4. Ela recebe como entrada os *templates* e uma descrição, chamada descrição TLN (*Transaction Level Netlist*) e gera os *prototypes* adaptados para o DUV. Mais informações sobre essa ferramenta podem ser encontradas em [11] e no Apêndice A.

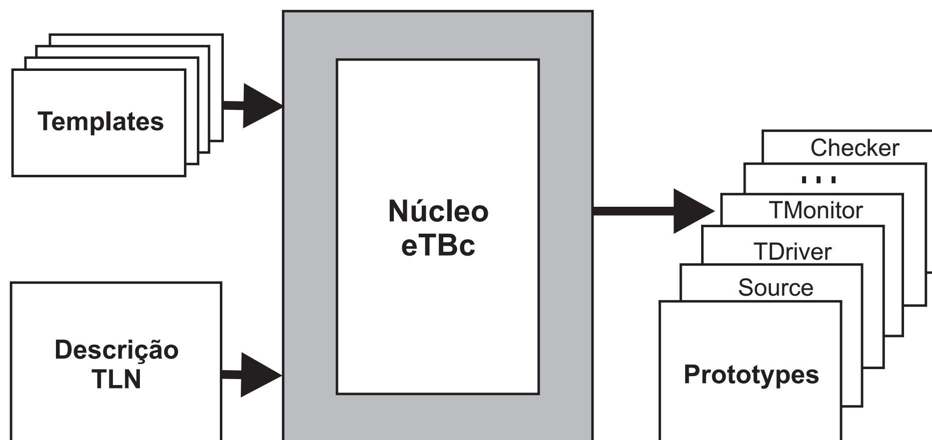

Figura 3.4: Esquema da ferramenta eTBc

Para possibilitar que a ferramenta eTBc gere *prototypes* para o DUV automaticamente, somente com duas entradas (TLN e *templates*), foram criadas duas linguagens para essa ferramenta: Linguagem para o TLN (eDL) e Linguagem para o *template* (eTL).

eDL - *eTBc Design Language*

Inicialmente, para o uso da ferramenta eTBc e geração dos *prototypes*, o engenheiro precisa definir uma estrutura denominada de TLN (*Transaction Level Netlist*), que irá conter toda a descrição do DUV que será verificado, como mostrado no Código 4. A TLN contém a descrição hierárquica do DUV já dividido em blocos. É através dessa descrição que a ferramenta será capaz de inferir todas as informações para automaticamente gerar *prototypes* para os *testbenches*. Esse arquivo deve especificar todas as interfaces que o DUV contém, blocos que se comunicam, sinais usados em cada bloco do DUV, variáveis no nível de transação e FIFOs que serão usadas para ligar blocos do *testbench*. A TLN é definida de acordo com a linguagem eDL.

Inicialmente é necessário definir os tipos de comunicação que irão compor o DUV, juntamente com as interfaces dos sub-blocos resultantes da divisão hierárquica. Cada tipo de comunicação deve ser descrito uma única vez, mesmo que sejam comuns a várias interfaces. Cada tipo de comunicação é formado por dados no nível de transação, que especificam os parâmetros que irão compor a transação da interface e pelos sinais indicados no barramento, que definem o grupo de sinais que vão compor essa interface.

O formato genérico para a definição de um tipo de comunicação pode ser visto no Código 1.

Código 1 : Descrição do tipo de comunicação da TLN

```
01        struct <nome-tipo-de-comunicação> {
02          trans {
03            <tipo> <nome>;
04          }
05          signals{
06            <tipo> <nome>;
07          }
08        }
```

A seguir, cada sub-bloco do DUV, que foi obtido pela divisão hierárquica deve ser definido. Para cada bloco, deve ser definido o seu nome, da seguinte forma:

$$module < nome - do - bloco >,$$

seguido da definição de qual(is) interface(s) de entrada será(ão) usada(s), a qual será identificada pela palavra chave *input*, o tipo de comunicação para essa interface, seguido pelo

nome da interface de entrada desse sub-modulo. Da mesma forma a interface de saída deve ser definida. Portanto para cada sub-bloco deve haver uma definição, como no Código 2.

Código 2 : Descrição de cada submódulo da TLN

```
01         module <nome-sub-bloco> {
02           input  <nome-tipo-de-comunicação> <nome-interface-entrada>;
03           output  <nome-tipo-de-comunicação> <nome-interface-saída>;
04         }
```

Por último, deve ser definido o DUV completo, como mostrado no Código 3. Essa definição deve mostrar o nome do DUV (linha 01), seguido pela definição das interfaces de entrada e saída (linhas 02 e 03). Todas as FIFOs que serão usadas para a interconexão do DUV e de seus sub-blocos devem ser instanciados em seguida. A última parte (linha 05) deve conter a definição de como todos os módulos serão ligados. Todos as ligações de blocos devem ser realizadas em uma expressão como essa. Para cada interface deve haver uma ligação com outra interface de outro bloco. Essa ligação pode ser mais bem compreendida através do exemplo do Código 4.

Código 3 : Descrição do módulo DUV

```
01 module <nome-do-DUV>  {
02  input   <nome-tipo-de-comunicação > <nome-interface-entrada>;
03  output  <nome-tipo-de-comunicação> <nome-interface-saída>;
04  fifo    <nome-tipo-de-comunicação>  <nome-da-FIFO>;
05 <nome-sub-bloco> <nome-sub-bloco>_i(.<interface1>(<interface2>),.<interface3>(<interface4>));
06 }
```

O exemplo de TLN do Código 4 é baseado no DUV da Figura 3.3. Nesse exemplo são utilizadas apenas uma interface de entrada e uma de saída, ambas as interfaces possuem um único tipo de comunicação denominado *coeffs*. No entanto, a metodologia poderia ser aplicada a um exemplo com mais interfaces, da mesma forma.

As linhas de 1 a 10 descrevem o tipo de comunicação (*coeffs*) a ser enviado pelo *testbench*. Ele é formado pela estrutura, com dados no nível de transação (linhas 2 a 4) que especificam as transações que irão compor o tipo de comunicação *coeffs* e pelos sinais indicados no barramento (linhas 5 a 9), que definem os sinais que vão compor esse tipo de comunicação.

As linhas 11 a 18 descrevem respectivamente os blocos *PIACDC* e *QI*. Essa descrição

Código 4 : Descrição TLN da Figura 3.3

```
01          struct coeffs {
02            trans {
03              short coeff [64];
04            }
05            signals{
06              signed[8] in_pqf;
07              bool valid;
08              bool ready;
09            }
10          }
11          module PIACDC {
12            input   coeffs  pi_in;
13            output  coeffs  pi_out;
14          }
15          module QI {
16            input  coeffs  qi_in;
17            output coeffs  qi_out;
18          }
19          module P_mpeg  {
20            input   coeffs  m_in;
21            output  coeffs  m_out;
22            fifo    coeffs  pi_qi;
23            PIACDC   piacdc_i(.pi_in(m_in),.pi_out(pi_qi));
24            QI       qi_i    (.qi_in(pi_qi),.qi_out(m_out));
25          }
```

mostra que o módulo *PIACDC* contém como interfaces de entrada e saída, *pi_in* e *pi_out* e que ambas interfaces receberão dados definidos no tipo de comunicação coeffs. Da mesma forma o bloco *QI* é especificado. Como esse exemplo possui uma divisão hierárquica de dois sub-blocos do DUV (*PIACDC* e *QI*), somente esses blocos são definidos.

A linha 19 declara o nome do DUV completo (*P_mpeg*) e as linhas 20 e 21 mostram as interfaces de entrada e saída do DUV completo, juntamente com qual tipo de comunicação será utilizado para a sua interface de entrada e saída. A linha 22 define a FIFO que liga os módulos PIACDC e QI. Finalmente, as linhas 23 e 24 definem como os blocos internos do DUV se ligam uns aos outros.

eTL - *eTBc Template Language*

Essa linguagem (eTL) está inserida nos *templates* e não aparece nos *prototypes* para o DUV, sendo somente uma linguagem auxiliar para a ferramenta eTBc. A linguagem foi criada para separar os *templates* do código fonte do eTBc, facilitando a manutenção da ferramenta, uma vez que basta mudar os *templates* para adaptar uma nova funcionalidade a eles. A ferramenta faz leitura dos *templates* e substitui as palavras reservadas da linguagem pelas variáveis definidas na TLN.

As palavras reservadas na linguagem eTL foram definidas de forma a permitir pesquisas nos elementos descritos na TLN. Posteriormente a essa pesquisa, a ferramenta realiza uma substituição, caso encontre, do elemento pesquisado pela palavra reservada existente no *template*. Todas as outras palavras no *template* que forem colocadas fora dos delimitadores $$ serão apenas replicadas para o arquivo de *prototypes*. Todas as palavras reservadas nessa linguagem são definidas sintaticamente na seguinte forma: $$(palavra-reservada).

Na linguagem eTL existem dois tipos de palavras reservadas, as de controle e as de substituição. As palavras de controle são usadas para gerenciar o fluxo das informações e as palavras de substituição causam a sua substituição por uma expressão.

A linguagem permite dois níveis de aninhamento de laços, portanto, existem duas variáveis de controle de índice, que são: "i"e "j". O índice "i"indica o laço mais interno, enquanto o índice "j"indica o laço mais externo. Quando se tem apenas um laço sem aninhamento, deve-se usar o índice "i", no caso, o índice "j"poderá ser usado quando se tem dois ou mais níveis de aninhamento de laço.

O significado de cada palavra reservada pode ser visto abaixo:

Palavras de substituição

As palavras reservadas de substituição detêm a função de causar a sua substituição por uma expressão.

- $$(structname): palavra causa a sua substituição por um nome de arquivo de estrutura de dados. Geralmente o nome do arquivo de estrutura de dados, no caso de SystemC é "structs.h"

  Exemplo :

  ```
  $$(file) $$(driver.h)
  #include $$(structname)
  $$(endfile)
  ```

  Esse trecho de código irá criar um arquivo chamado "driver.h"com o seguinte conteúdo: #include "structs.h"

  Neste caso, o nome do arquivo de estruturas tem o nome "structs.h".

- $$(i.*var*): essa palavra causa sua substituição por um nome de uma variável e deve ser usada juntamente com a palavra chave $$(foreach). Essa variável é definida na TLN. O índice "i"se refere ao laço mais interno da estrutura $$(foreach).

  Exemplo:

  ```
  $$(foreach) $$(duv.out)
  $$(i.nome)
  $$(endfor)
  ```

  Esse trecho de código itera entre todas as interfaces(portas) de saída de um dado módulo provocando a substituição da palavra $$(i.nome) pelo nome daquela interface.

- $$(j.*var*): essa palavra causa sua substituição por um nome de uma variável e deve ser usada juntamente com a palavra chave $$(foreach). Essa variável é definida no escopo da TLN. O índice "j"se refere ao laço mais externo da estrutura $$(foreach).

  Exemplo:

$$(foreach) $$(duv.out)

$$(foreach) $$(i.signal)

$$(j.name) $$(i.name)

$$(endfor)

$$(endfor)

Esse trecho de código itera entre todas os sinais de todas as interfaces (portas) de saída de um dado módulo provocando a substituição da palavra $$(j.name) pelo nome da interface e da palavra $$(i.name) pelo nome do sinal.

- $$(i.type): essa palavra causa sua substituição pelo tipo associado a um elemento. Esse tipo é declarado no escopo da TLN.

  Exemplo:

  $$(foreach) $$(duv.out)

  $$(i.type) ...

  $$(endfor)

  Esse trecho de código itera entre todas as interfaces (portas) de saída de um dado módulo provocando a substituição da palavra $$(i.type) pelo tipo daquela interface.

- $$(j.type): funciona como i.type, mas para o índice do laço exterior.

- $$ (i.signal): essa palavra causa sua substituição por um nome de sinal. Um exemplo pode ser visto no seguinte trecho de código:

  $$(foreach) $$(duv.out)

  $$(i.signal)

  $$(endfor)

  Esse trecho de código itera entre todas as interfaces(portas) de saída de um dado módulo provocando a substituição da palavra $$(i.signal) por cada sinal referente à lista de sinais daquela interface.

- $$ (j.signal): funciona como i.signal, mas para o índice do laço exterior.

- $$(duv.name): induz a ferramenta a fazer busca pelo DUV, especificado na linha de comando, quando o usuário estiver usado a ferramenta eTBc.

- $$(i.name): pode ser usado para designar nomes de interfaces, estruturas e de variáveis, dependendo do contexto no qual ela seja usada.

Palavras de controle

As palavras de controle são usadas para gerenciar o fluxo do código gerado.

1. $$(file) $$(<nome_do_arquivo>): Palavra usada para criar um arquivo. Deve ser sucedida de um nome de arquivo. Esse arquivo vai conter o código do *prototype* sendo gerado na seqüência. Essa palavra é usada em conjunto com $$(endfile), que define o fim desse arquivo. Exemplo :

   $$(file) $$(<nome_do_arquivo>)

   ...

   $$(endfile)

2. $$(foreach) $$(<lista>) <código a ser iterado> $$(endfor), indica uma estrutura de laço, criado para repetir porções do código que devem aparecer mais de uma vez. Os indexadores que são utilizados são "i"e "j". O indexador de laço i é usado para o laço mais interno, o "j"é usado para o laço mais externo. Nesse laço, o termo "lista"pode ser um dos seguintes valores:

   - $$(duv.in).
   - $$(duv.out).
   - $$(i.signal).
   - $$(i.name).
   - $$(i.type).

3. $$ (duv.in): essa palavra deve ser usada junto com a estrutura de laço $$(foreach) e serve para iterar entre todas as portas de entrada de um dado módulo.

   Exemplo:

   $$(foreach) $$(duv.in)

   ...

   $$(endfor)

Esse trecho de código indica que para toda interface de entrada do DUV, ele irá repetir o trecho de código "...".

4. $$ (duv.out): essa palavra deve ser usada junto com a estrutura de laço $$(foreach) e indica que o laço será executado para todas as interfaces de saída do DUV. Exemplo:
   $$(foreach) $$(duv.out)
   ...
   $$(endfor)
   Esse trecho de código indica que cada interface de saída do DUV, ele irá repetir o código "...".

5. $$(if) *condição* ... $$(endif): essas palavras reservadas indicam o início e o fim de uma estrutura condicional, respectivamente. Um exemplo dessa condição pode ser vista no seguinte trecho de código:
   $$(if)
   ...
   $$(endif)

6. $$(i.isnotlast):é usada em conjunto com a estrutura condicional $$(if) e serve para sinalizar que o código será executado enquanto não seja a última ocorrência de um determinado símbolo.

   Exemplo:

   $$(if) $$(i.isnotlast) , $$(endif)

   Esse código será executado se o símbolo ","não for o último de uma sequência de símbolos.

Tendo a descrição das linguagens, o próximo passo é mostrar como são implementados os blocos do *testbench*. As seções seguintes detalham cada um desses blocos.

3.1.2 Source

O Source é o bloco que gera estímulos para a simulação. Esses estímulos são criados no nível de transação e devem ser cuidadosamente escolhidos pelo engenheiro de verificação

de acordo com o plano de verificação, para que a cobertura funcional estabelecida seja alcançada. Os estímulos recomendados são: estímulos direcionados, situações críticas, estímulos randômicos e casos de teste reais.

Os estímulos direcionados são casos de teste escolhidos manualmente para estimular um grupo de funcionalidades. É importante frisar que esses casos de teste podem ser usados somente se forem poucas as funcionalidades a serem verificadas, pois seria oneroso demais gerar casos de teste para uma quantidade muito grande de funcionalidades.

Situações críticas são estímulos criados especificamente para estimular uma determinada funcionalidade na qual existe suspeita de haver erro.

Os estímulos randômicos são gerados dentro de uma faixa de valores escolhida. Esses estímulos randômicos tipicamente são capazes de revelar muitas falhas, uma vez que podem gerar valores que estimulem pontos que não foram antecipados, devido ao não-determinismo da geração de seus valores.

Finalmente, casos de teste reais, são estímulos recolhidos em uma situação real da aplicação.

O Source envia esses estímulos através da comunicação com o Modelo de Referência e TDriver por meio de FIFOS.

O Código 5 mostra o *template* do Source. Os *templates* dos demais blocos seguem o mesmo princípio e por isso não serão explicados nesse nível de detalhamento.

A ferramenta eTBc recebe o *template*, de acordo com o Código 5. Nesse código, todas as linhas que contiverem os símbolos $$ devem ser consideradas pela ferramenta como elementos que devem ser substituídos, de acordo com as informações recebidas pela descrição TLN. De posse das informações da descrição TLN, a ferramenta vai modificar o *template* para adaptá-lo ao DUV. Os símbolos $$ dos *templates* não aparecem nos *prototype*, como mostra o Código 6.

O primeiro grupo de dados do *template* (linhas 2 a 13) indica os dados randômicos que serão gerados pelo Source. Para cada interface do DUV, a ferramenta gera automaticamente uma classe de geração de dados randômicos, já adaptada para a transação que será utilizada por essa interface, como pode ser visto nas linhas 4 a 10 do Código 6. Esse código é descrito a seguir:

- Linha 01: declaração da classe que descreve o gerador da transição randomizada.

Código 5 : Template padrão do Source

```
01    $$(file) $$(source.h)
02    $$(foreach) $$(duv.in)
03       class $$(i.name)_constraint_class: public scv_constraint_base {
04         //  scv_bag<int> ?_distrib;
05         public:
06         scv_smart_ptr<$$(i.type)> $$(i.type)_sptr;
07         SCV_CONSTRAINT_CTOR($$(i.name)_constraint_class) {
08            //  ?_distrib.push(?, ?);
09            // $$(i.type)_sptr->?.set_mode(?_distrib);
10            //  SCV_CONSTRAINT($$(i.type)_sptr->?() > ?);
11          }
12       };
13    $$(endfor)
14    SC_MODULE(source) {
15       $$(foreach)  $$(duv.in)
16          sc_fifo_out<$$(i.type)_ptr> $$(i.name)_to_refmod;
17          sc_fifo_out<$$(i.type)_ptr> $$(i.name)_to_driver;
18          $$(i.name)_constraint_class $$(i.name)_constraint;
19       $$(endfor)
20       $$(foreach) $$(duv.in)
21          void $$(i.name)_p() {
22             ifstream ifs("$$(i.name).stim");
23             string type;
24             $$(i.type) stim;
25             while( !ifs.fail() && !ifs.eof() ) {
26                ifs >> type;
27                if ( type == "$$(i.type)" ) {
28                   ifs >> stim;
29                   $$(i.name)_to_refmod.write(new $$(i.type)( stim ) );
30                }
31                ifs.ignore(225, '\n');
32             }
33             while(1) {
34                $$(i.name)_constraint.next();
35                stim =  $$(i.name)_constraint.$$(i.type)_sptr.read();
36                $$(i.name)_to_refmod.write(new  $$(i.type)( stim ) );
37                $$(i.name)_to_driver.write(new  $$(i.type)( stim ) );
38             }
39          }
40       $$(endfor)
41       SC_CTOR( source ):
42             $$(foreach) $$(duv.in)
43                $$(i.name)_constraint("$$(i.name)_constraint") $$(if) $$(i.isnotlast) , $$(endif)
44             $$(endfor)
45             {$$(foreach) $$(duv.in) SC_THREAD($$(i.name)_p); $$(endfor)}
46    };
47    $$(endfile)
```

Código 6 : Template do Source adaptado para o DUV *PIACDC*

```
01        class pi_in_constraint_class: public scv_constraint_base {
02           //  scv_bag<int> ?_distrib;
03           public:
04           scv_smart_ptr<coeffs> coeffs_sptr;
05           SCV_CONSTRAINT_CTOR(pi_in_constraint_class) {
06              //  ?_distrib.push(?, ?);
07              // coeffs_sptr->?.set_mode(?_distrib);
08              //  SCV_CONSTRAINT(coeffs_sptr->?() > ?);
09           }
10        };
11        SC_MODULE(source) {
12           sc_fifo_out<coeffs_ptr> pi_in_to_refmod;
13           sc_fifo_out<coeffs_ptr> pi_in_to_driver;
14           pi_in_constraint_class pi_in_constraint;
15           void pi_in_p() {
16              ifstream ifs("pi_in.stim");
17              string type;
18              coeffs stim;
19             while( !ifs.fail() && !ifs.eof() ) {
20                ifs >> type;
21                if ( type == "coeffs" ) {
22                   ifs >> stim;
23                   pi_in_to_refmod.write(new coeffs( stim ) );
24                }
25                ifs.ignore(225, '\n');
26             }
27             while(1) {
28                pi_in_constraint.next();
29                stim =  pi_in_constraint.coeffs_sptr.read();
30                pi_in_to_refmod.write(new  coeffs( stim ) );
31                pi_in_to_driver.write(new  coeffs( stim ) );
32             }
33           }
34           SC_CTOR( source ):
35           pi_in_constraint("pi_in_constraint")
36           {
37              SC_THREAD(pi_in_p);
38           }
39        };
```

- Linha 02: declaração de um porção de dados (chamada de estrutura do tipo bag), do tipo especificado (inteiro no exemplo).
- Linha 04: declaração de um ponteiro para essa transação.
- Linha 06: coloca o tipo de dado que se deseja randomizar e quantas vezes.
- Linha 07: associa a estrutura bag a parâmetros de transação.
- Linha 08: opcional para realizar a randomização limitada, como explicado mais adiante.

A randomização direcionada especifica exatamente e a quantidade e a proporção de geração para determinados valores. Ela pode ser realizada por um ponteiro denominado de "apontador esperto". Ele gera valores aleatórios, distribuídos por igual, na faixa associado ao tipo do parâmetro. Quando é necessário fazer uma distribuição específica, é necessário criar uma estrutura do tipo bag:

$$scv_bag < int >< parametro > _dist.$$

Logo em seguida, encher esta estrutura:

$$< parametro > _dist.push(< valor >, < ndevezes >).$$

Finalmente, é necessário pendurar a estrutura bag em um parâmetro:

$$tr_sptr- >< parametro > .set_mode(< parametro > _dist).$$

Exemplo em que são gerados 55 vezes o valor 7 e 44 vezes o valor 8:

$$param_dist.push(7, 55);$$

$$param_dist.push(8, 44);$$

Pode-se criar também uma estrutura bag com faixas de valores. Exemplo em que são gerados valores compreendidos entre 0 e 7 em 60% dos casos e valores compreendidos entre 8 e 10 em 40% dos casos:

$$scv_bag < pair < int, int >> param_dist;$$

$$param_dist.push(pair < int, int > (0, 7), 60);$$

$$param_dist.push(pair < int, int > (8, 10), 40);$$

Para realizar uma randomização mais limitada, é possível usar constraints. Exemplo:

$$SCV_CONSTRAINT(tr_sptr- > param() < 10);$$

Nesse caso, todos os valores gerados serão menores que 10, dentro do valor especificado pelo parâmetro.

O segundo grupo de dados do *template*, entre as linhas 15 e 27 irá conter a opção de fazer a leitura a partir de um arquivo que contém os estímulos direcionados, os críticos e os reais.

Das linhas 28 a 31, existe um laço infinito que gera as próximas transações randomizadas, de acordo com a randomização definida.

$$pi_in_constraint.next();$$

Essas transações geradas são enviadas para a FIFO que se liga ao Modelo de Referência e para a FIFO que se liga ao TDriver.

$$pi_in_to_refmod.write(newcoeffs(stim));$$

$$pi_in_to_driver.write(newcoeffs(stim));$$

A FIFO recebe a próxima transação com os dados a serem usados nos estímulos e os envia respectivamente para o Modelo de Referência e para o TDriver que a envia para o DUV.

Nas linhas finais está definido o construtor do Source, que irá ativar a *thread* para executar o código implementado no processo *p_in_p*.

O Código 6 é o *prototype* que será passado para o engenheiro de verificação. Nesse *prototype* todas as informações referentes a interfaces e FIFOs já estão devidamente preenchidas, como pode ser visto na linha 04, onde o nome do grupo de dados *coeffs* já foi devidamente substituído. As linhas que devem ser alteradas pelo engenheiro de verificação estão comentadas para que elas sejam descomentadas em caso de uso das mesmas. No caso do módulo Source, o engenheiro de verificação precisa definir as distribuições de probabilidade dos estímulos randômicos que serão gerados para a simulação, usando o SCV, como explicado

acima, ou usando outro tipo de distribuição de probabilidade que fica a critério do engenheiro de verificação. Além disso, ele pode definir outros tipos de estímulos que não sejam os estímulos randômicos.

3.1.3 DUV *(Design Under Verification)*

O DUV *(Design Under Verification)* é o projeto que está sendo verificado. Ele não é parte do *testbench*, no entanto, ambos estão diretamente ligados um ao outro, sendo preciso por isso, que ele seja definido nesse capítulo.

O DUV é implementado no nível de sinais, possuindo uma interface de comunicação que somente permite a recepção e envio de dados no nível de sinais. Devido a isso, é necessário algum mecanismo para que ele se comunique com o *testbench* que trabalha em TLM (transaction Level Modelling). Essa comunicação se faz com a ajuda dos blocos TDriver(s) e TMonitor(es) que traduzem sinais para transações e vice-versa.

No *prototype* do DUV é gerada apenas ¨uma casca¨ que contém a definição da interface que será utilizada para a comunicação do *testbench* com os blocos TDriver(s) e TMonitor(es). O código do *prototype* do DUV *PIACDC* da Figura 3.3, pode ser visto no Código 7, já adaptado para o exemplo e com todos os sinais já declarados. O engenheiro de projeto precisa implementar as funcionalidades que irão compor o DUV.

3.1.4 TDriver

Cada *testbench* possui um ou mais blocos denominados TDriver. É necessário que seja implementado um TDriver para cada interface de entrada do DUV, pois cada bloco TDriver vai ser responsável pela comunicação de uma determinada interface de entrada do DUV com o Source. As interfaces de entrada são os canais por onde o DUV recebe dados de outros blocos. Cada interface de entrada possui o seu próprio protocolo de comunicação (*handshake*), que nos *prototypes* do DUV deve ser implementado pelo engenheiro de verificação. O TDriver recebe transações vindas do Source via FIFOs e as converte em sinais a serem enviados para o DUV.

Existe a possibilidade de serem verificados DUVs que possuam interfaces bidirecionais e precisam se comunicar tanto com o TDriver quanto com o TMonitor, como por exemplo,

Código 7 : Template do DUV adaptado para o exemplo

```
01          #include "systemc.h"
02          SC_MODULE( PIACDC ) {
03             sc_in<bool> clk;
04             sc_in<bool> reset;
05             sc_in<sc_int<8> > pi_in_in_pqf;
06             sc_in<bool > pi_in_valid;
07             sc_in<bool > pi_in_ready;
08             sc_out<sc_int<8> > pi_out_in_pqf;
09             sc_out<bool > pi_out_valid;
10             sc_out<bool > pi_out_ready;
11             void p(){
12
13             }
14             SC_CTOR ( PIACDC )
15             {
16             SC_METHOD(p);
17                sensitive << clk.pos();
18             }
19          };
```

para controle de um barramento. Para que esse *testbench* possa funcionar, o TDriver deve se comunicar com o TMonitor através de sinais de controle. No entanto, essa abordagem não foi ainda usada na prática, sendo por isso deixada como trabalho futuro.

O *prototype* gerado pela ferramenta para o TDriver do módulo PIACDC, possui o formato indicado no Código 8. Nas linhas 2 a 8 estão todas os sinais gerados pela ferramenta, bem como a FIFO que liga o TDriver com o Source. Nesse *prototype*, na linha 16, o engenheiro de verificação precisa preencher as informações referentes ao protocolo de comunicação que será usado para se comunicar com o DUV. Além disso, nesse módulo o engenheiro precisa preencher os critérios de cobertura dos dados de entrada que devem ser medidos, caso seja necessário.

O *prototype* do TDriver é implementado da seguinte forma:

- Linha 02: entrada de FIFO para transações do Source.
- Linha 03: sempre precisa do sinal de clock que é uma entrada do TDriver.
- Linha 04 a 07: sinais ligados à interface de entrada do DUV.
- Linha 08: visualizador de transações.

Código 8 : Template do TDriver gerado para o exemplo

```
01         SC_MODULE(pi_in_driver) {
02            sc_fifo_in<pi_in *> stim;
03            sc_in  <bool> clk;
04            sc_in <bool> reset;
05            sc_out<sc_int<8> > in_pqf;
06            sc_out<bool > valid;
07            sc_out<bool > ready;
08            scv_tream stream;
09            scv_tr_generator<pi_in,pi_in> gen;
10            pi_in *tr_ptr;
11            void p() {
12               while ( ! reset ) wait();
13               while(1) {
14                  tr_ptr = stim.read();
15                  scv_tr_handle h = gen.begin_transaction(*tr_ptr);
16                  wait();
17                  gen.end_transaction(h);
18                  delete tr_ptr;
19               }
20            }
21            SC_CTOR(pi_in_driver):
22            stream(name(), "Transactor"),
23            gen("pi_in_driver", stream){
24                 SC_THREAD(p); sensitive << clk.pos();
25            }
26         };
```

- Linha 10: apontador para uma transação.
- Linha 11: processo que vai receber transações e modificar os sinais.
- Linha 14: recebe transações da FIFO.
- Linha 15: marca o momento de início da transação.
- Linha 16: demora pelo menos um clock para realizar a transação.
- Linha 17: marca o momento de fim da transação.
- Linha 18: descarta a transação usada.
- Linha 23: título de transação que aparecerá no visualizador.
- Linha 24: o TDriver é síncrono com DUV.

3.1.5 Reset_driver

O Reset_driver serve para "resetar"o sistema de tempos em tempos. Esse bloco se comporta como um TDriver que gera somente valores de *reset*. O tempo que ele espera até que o próximo sinal seja enviado para o DUV pode ser configurado manualmente pelo engenheiro de projeto, na linha 05, de acordo com a especificação. O código desse Reset_driver pode ser visto no Código 9.

Código 9 : Template do Reset_driver gerado para o exemplo

```
01        SC_MODULE(reset_driver) {
02           sc_out <bool> rst;
03           void p() {
04              rst=1;
05              wait(555, SC_NS);
06              rst=0;
07           }
08           SC_CTOR(reset_driver){
09              SC_THREAD(p);
10           }
11        };
```

Código 10 : Template do TMonitor gerado para o exemplo

```
01        SC_MODULE(pi_out_monitor) {
02           sc_fifo_out<pi_out *> reply;
03           sc_in<bool> clk;
04           sc_in<sc_int<8> > in_pqf;
05           sc_in<bool > valid;
06           sc_in<bool > ready;
07           scv_tream stream;
08           scv_tr_generator<pi_out,pi_out> gen;
09           pi_out *tr_ptr;
10           void p() {
11              while(1) {
12                 tr_ptr = new pi_out();
13                 scv_tr_handle h =  gen.begin_transaction ();
14                 wait();
15                 gen.end_transaction(h, *tr_ptr);
16                 reply.write(tr_ptr);
17              }
18           }
19           SC_CTOR(pi_out_monitor):
20           stream(name(), "Transactor"),
21           gen("pi_out_monitor", stream){
22              SC_THREAD(p); sensitive << clk.pos();
23           }
24        };
```

3.1.6 TMonitor

O TMonitor da metodologia VeriSC diverge do conceito de Monitor estabelecido pela VSIA [17], para maiores informações, veja apêndice B. O TMonitor do VeriSC executa o papel inverso do TDriver. Ele converte todos os sinais de saída do DUV para transações e repassa as mesmas para o Checker, via FIFO. Além disso, ele também executa um protocolo de comunicação com o DUV, através de sinais.

O *testbench* possui um TMonitor para cada interface de saída do DUV.

O *prototype* gerado pela ferramenta para o TMonitor do módulo PIACDC, possui o formato indicado no Código 10. Novamente, o engenheiro de verificação precisa estabelecer qual o protocolo de comunicação que será usado para a comunicação com o DUV e os critérios de cobertura dos dados de saída que devem ser medidos, caso seja necessário.

Seguem explicações sobre o código do TMonitor:

- Linha 02: saída de FIFO para transações do Source.
- Linha 03: sempre precisa do sinal de clock.
- Linha 04 a 06: sinais ligados à interface de saída do DUV
- Linha 07: visualizador de transações.
- Linha 09: apontador para uma transação.
- Linha 10: processo que vai pegar transações e agitar sinais.
- Linha 12: cria nova transação vazia.
- Linha 13: marca o momento de início da transação.
- Linha 14: demora pelo menos um clock para realizar a transação.
- Linha 15: marca o momento de fim da transação.
- Linha 16: envia a transação pela FIFO.
- Linha 21: título de transação que aparecerá no visualizador.
- Linha 22: o TDriver é síncrono com DUV.

3.1.7 Checker

Código 11 : Template do Checker adaptado para o exemplo

```
01        #include <sstream>
02        SC_MODULE( checker ){
03           sc_fifo_in<coeffs_ptr> pi_out_from_refmod;
04           sc_fifo_in<coeffs_ptr> pi_out_from_duv;
05           sc_signal<unsigned int> error_count;
06           void pi_out_p() {
07              while(1) {
08                 coeffs_ptr  trans_refmod = pi_out_from_refmod.read();
09                 coeffs_ptr  trans_duv = pi_out_from_duv.read();
10                 if ( !(*trans_refmod == *trans_duv) ) {
11                   ostringstream ms;
12                   ms <<  "expected: " << *trans_refmod << endl
13                   << "received: " << *trans_duv;
14                   SCV_REPORT_ERROR("out_access", ms.str().c_str());
15                   error_count = error_count.read()+1;
16                 }
17                 delete ( trans_refmod );
18                 delete ( trans_duv );
19              }
20           }
21           SC_CTOR( checker ):
22           pi_out_cv("pi_out_cv"){
23              SC_THREAD(pi_out_p);
24           }
25        };
```

O módulo Checker é o responsável por comparar as respostas resultantes do DUV e do Modelo de Referência, para saber se são equivalentes. Ele compara esses dados automaticamente e, no caso de encontrar erros, emite uma mensagem mostrando quando aconteceu o erro. Essa mensagem de erro é importante, porque dessa forma, o engenheiro recebe uma alerta de qual dado seria o esperado e qual dado foi recebido. Isso pode facilitar bastante a encontrar um erro na verificação, uma vez que a metodologia VeriSC é *black-box* e não permite a visão dos componentes internos do DUV.

O Checker compara toda a transação que chega, através da comparação de variável por variável, e as respostas devem ser funcionalmente correspondentes. Tipicamente elas devem ser iguais, mas caso exista alguma especificação que determine uma tolerância para a diferença entre o Modelo de Referência e o DUV, essa tolerância pode ser implementada. A

implementação dessa tolerância deve ser feita no Checker, quando é realizada a comparação das transações que chegam do Modelo de Referência e DUV, no exemplo, na linha 10 do Código 11.

O módulo Checker não precisa ser modificado pelo engenheiro de verificação. Esse módulo é gerado pela ferramenta da forma necessária para a simulação. O exemplo do *prototype* gerado para o Checker pode ser visto no Código 11.

A seguir encontra-se a descrição do *prototype* do Checker:

- Linha 03 e 04: entradas de FIFO de transações do Modelo de Referência e do TMonitor.
- Linha 05: contador de erros de comparação que possam vir a ocorrer.
- Linha 06: processo que vai comparar transações.
- Linhas 08 e 09: pega transações das FIFOs.
- Linha 10: compara parâmetros das transações.
- Linhas 12, 13 e 14: relatam erro se transações não estiveram equivalentes.
- Linha 15: adiciona mais um erro ao contador.
- Linhas 17 e 18: descartam as transações usadas.
- Linha 23: o Processo é um thread que não obedece ao clock, mas somente à chegada de transações pelas FIFOs.

3.1.8 Arquivo de estrutura

O arquivo de estrutura define todos os dados que serão usados em uma determinada transação, bem como o tipo desses dados. Ele deve ser definido no início da implementação do *testbench*. Esse arquivo deve ser único e conter o formato mostrado no Código 12

Além disso, dentro desse arquivo de estrutura, foi desenvolvido um *prototype* com sobrecarga de operadores para realizar operações de comparação automática de transações e impressão de valores esperados e recebidos, no módulo Checker do *testbench*.

Código 12 : Estrutura para uma transação

```
01          struct <nome_da_estrutura>{
02              <tipo> <variável>;
03          };
```

Um exemplo dessas operações de sobrecarga pode ser visto no Código 13. As linhas 03, 04 e 05 implementam a sobrecarga do operador "==" para que compare automaticamente as transações vindas do Modelo de Referência e do DUV. As linhas 07, 08 e 09 implementam a sobrecarga do operador "<<" para que possa imprimir as transações com os valores esperado e recebido.

Código 13 : Sobrecarga de operadores no arquivo de estruturas

```
01          struct bs {
02              unsigned char byte;
03              inline bool operator== (const bs& arg) const {
04                 return( (byte == arg.byte) );
05              }
06          };
07          inline ostream& operator<< (ostream& os, const bs& arg) {
08             os << "bs=" << hex << (int)arg.byte << dec;
09             return os;
10          }
```

Sobrecargas de operadores são utilizadas no módulo Checker para que a comparação das transações completas que chegam do Modelo de Referência e do TMonitor possam ser realizadas sem ser necessário mudar o código do Checker para cada tipo de transação.

Além disso, em caso de erro, o Checker imprime para o usuário os valores da transação esperada e da transação errada que acaba de ser recebida. Para isso também se usa a sobrecarga no arquivo de estruturas. O Checker imprime o valor da transação esperada (transação que vem do Modelo de Referência) e da transação recebida (transação vinda do DUV).

3.1.9 Modelo de Referência

O Modelo de Referência contém a implementação ideal (especificada) do sistema. Por isso, ao receber estímulos, ele deve produzir respostas corretas. Para efeito de comparação dos dados, após a conclusão das operações ele deve passar os dados de saída para o módulo Checker via FIFO(s). O Modelo de Referência deve ser implementado no nível de transações.

Código 14 : Template do Modelo de Referência adaptado para o exemplo

```
01        SC_MODULE(refmod) {
02           sc_fifo_in <coeffs_ptr  > pi_in_stim;
03           sc_fifo_out <coeffs_ptr > pi_out_stim;
04           coeffs_ptr pi_in_ptr;
05           coeffs_ptr pi_out_ptr;
06           void p() {
07              while (1) {
08                 pi_in_ptr = pi_in_stim.read();
09                 pi_out_ptr = new coeffs;
10                 pi_out_stim.write(pi_out_ptr);
11                 delete( pi_in_ptr );
12              }
13           }
14           SC_CTOR(refmod){
15              SC_THREAD(p);
16           }
17        };
```

É importante salientar que o Modelo de Referência pode ser gerado em qualquer linguagem que possa se comunicar com o SystemC no nível de transações. Esse Modelo de Referência pode se comunicar através de FIFOs ou de *pipes*, conforme ele seja implementado em C++/SystemC ou usando outra linguagem de programação que permita o acesso aos recursos de *pipe* do sistema operacional, respectivamente.

A cobertura pode ser colocada no Modelo de Referência para checar a execução de determinadas funcionalidades. O *prototype* gerado pela ferramenta contém apenas uma "casca"do Modelo de Referência e pode ser visto no Código 14. O engenheiro de verificação precisa preencher o *prototype* com todo o código necessário para compor suas funcionalidades.

A seguir é mostrada a descrição do Código 14:

- Linha 02: entrada de FIFO de transações do Source.
- Linha 03: saída de FIFO de transações para o Checker.
- Linha 08: leitura de transações de entrada.
- Linhas 09 e 10: envia transações de saída.
- Linha 11: descarta transação.

3.1.10 FIFO Buffer (*First In First Out*)

Os elementos do *testbench* são ligados entre si por meio de FIFOs, que são representadas na Figura 3.2 como setas largas. Cada FIFO é responsável por ligar uma interface do *testbench* e por realizar o transporte de transações de um bloco para outro.

Como o próprio conceito indica, o primeiro elemento (transação) a entrar na FIFO será o primeiro a sair. Assim, as FIFOs possuem um papel muito importante no *testbench* que é o de controle do sequenciamento e do sincronismo dos dados que entram e saem delas. Dessa forma, pode-se comparar os dados do DUV com os dados do Modelo de Referência, na ordem correta de chegada.

3.1.11 Pré-source

O bloco Pré-source é necessário para a geração dos passos da metodologia e também é gerado pela ferramenta. O Pré-source é um sub-conjunto do módulo Source, com quase as mesmas funcionalidades. A diferença é que ele gera estímulos somente para o Modelo de Referência e não para o DUV. Como conseqüência, ele só tem FIFOs que o ligam ao Modelo de Referência.

Depois que o Pré-source é utilizado, a sua parte que implementa os estímulos pode ser reusada no módulo Source que será gerado pela ferramenta nos passos seguintes.

3.1.12 Sink

O bloco Sink possui um subconjunto das funcionalidades do Checker. Ele recebe os dados de saída apenas do Modelo de Referência, tendo assim somente FIFOs de entrada.

Com a definição de todos os blocos do *testbench* e seu funcionamento, é possível definir como será a implementação do *testbench* passo a passo conforme a metodologia VeriSC.

3.2 Metodologia

No início do projeto, o engenheiro de verificação, juntamente com o engenheiro de projeto devem dividir o DUV e o Modelo de Referência em blocos correspondentes a serem implementados e verificados, como mostrado na Figura 3.5. Nesse caso, o DUV que é parte do

mpeg *P_mpeg*, foi dividido em 2 blocos a serem verificados. Cada bloco *PIACDC* e *QI* é uma parte do *P_mpeg*, que foi dividido, formando novos DUVs menores. O bloco *PIACDC* se comunica com o bloco *QI* através da FIFO *pi_qi*.

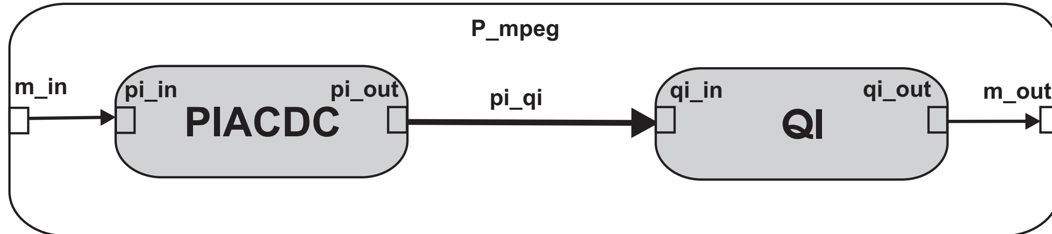

Figura 3.5: Exemplo da divisão do *P_mpeg* em blocos

Para a implementação dessa metodologia e geração dos *testbenches* para o DUV principal e os DUVs hierárquicos, são necessários alguns passos para garantir, que tanto o Modelo de Referência, quanto as demais partes do *testbench*, não insiram erros durante a simulação do DUV. Os passos a serem seguidos servem tanto para verificação dos sub-blocos do DUV, quanto para a verificação do DUV completo, depois da junção dos sub-blocos.

Para que o Testbench seja gerado antes do DUV é necessário algum mecanismo que use as próprias partes do *testbench* para que a simulação seja realizada da mesma forma como se existisse um DUV inserido no processo. Esse mecanismo de substituição é incremental, sendo que a implementação de cada parte do *testbench* vai sendo construída com passos. A Substituição do DUV que seria inserido juntamente com o *testbench* é realizada utilizando-se de uma tripla composta de TMonitor, Modelo de Referência, TDriver. Sendo que o TMonitor usado na substituição deve ter o papel inverso ao TDriver usado no *testbench* e o TDriver da substituição deve ter o papel inverso ao TMonitor usado no *testbench*.

3.2.1 Testbench para o DUV completo (primeiro passo)

De forma a implementar os *testbenches* necessários, o primeiro passo é a geração do *testbench* para o DUV completo. Esse *testbench* é construído em 3 subpassos, os quais são explicados a seguir e mostrados na Figura 3.6.

1. O Modelo de Referência é testado em sua capacidade de interagir com o *testbench* (receber e produzir dados no nível de transações). Um Pré_source é usado para es-

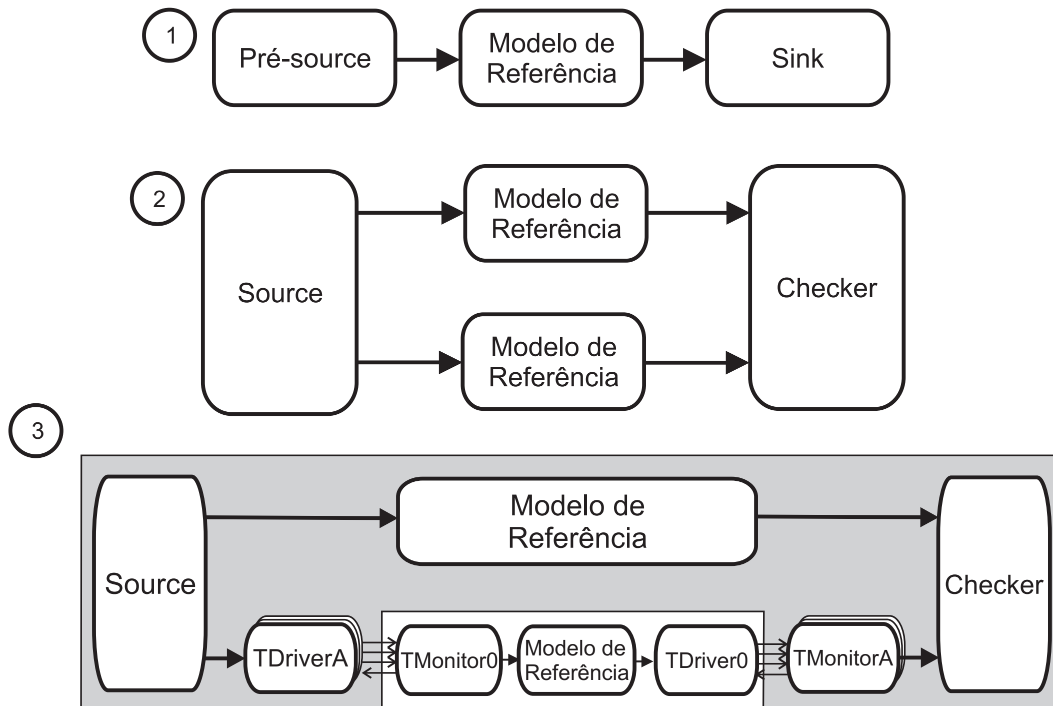

Figura 3.6: Testbench para o DUV completo (primeiro passo)

timular o Modelo de Referência e realizar este teste. Um Sink é usado para receber as transações vindas a partir do Modelo de Referência. Estímulos direcionados e randômicos já podem ser implementados no Pré-source.

2. Após testar a capacidade de interação do Modelo de Referência com o Pré_Source e Sink, deve-se implementar o Source e o Checker, que serão utilizados no *testbench*, para saber se estão corretos. O Modelo de Referência deve ser usado duas vezes. Dessa forma, o Source estimula os dois Modelos de Referência e em seguida, o Checker verifica se os resultados são iguais. No caso em que todos os estímulos forem utilizados e não for encontrado erro, alguns erros artificiais podem ser inseridos em um dos Modelos de Referência para checar a habilidade do Checker em reconhecê-los. Uma forma automática de gerar esses estímulos é objeto de um trabalho de mestrado em curso.

3. Os blocos TDriver e TMonitor devem ser testados nesse subpasso. Para realizar esse teste, deve-se simular todo o *testbench* de forma a verificar se os blocos TDriver e TMonitor estão realizando os seus papéis de forma correta, ou seja, passar as transações para sinais e os sinais recebidos de volta para transações, respectivamente.

No entanto, para simular o *testbench*, é necessário que haja um DUV para receber os dados vindos do TDriver e enviar dados para o TMonitor. Porém, o DUV não foi ainda implementado, por isso é necessário que seja "inventado"um mecanismo que venha a substituir o DUV durante a simulação. Esse é a parte mais importante desse passo, pois mostra como o *testbench* pode ser implementado independente do DUV sem nenhum prejuízo para a sua construção. Esse mecanismo de substituição pode ser criado através da utilização de uma tripla composta de TMonitor, Modelo de Referência, TDriver, que exerce o mesmo papel de um DUV, como mostrado na Figura 3.7.

Essa tupla recebe os sinais vindos do TDriverA, que os passa para o TMonitor0. O TMonitor0 por sua vez converte-os para transação e passa para o Modelo de Referência. O Modelo de Referência recebe a transação e passa a sua saída para o TDriverO, que transforma novamente para sinal e passa para o TMonitorA.

Nesse exemplo do DUV, existe somente um TDriver e um TMonitor (TDriverA e TMonitorA), porque o DUV *P_mpeg* possui somente uma interface de entrada e uma interface de saída. A tripla utilizada para substituir o DUV é denominada de (TMonitor0, Modelo de Referência e TDriver0), sendo que o TMonitor0 tem a mesma interface do TDriverA e o TDriver0 possui a mesma interface de sinais do TMonitorA.

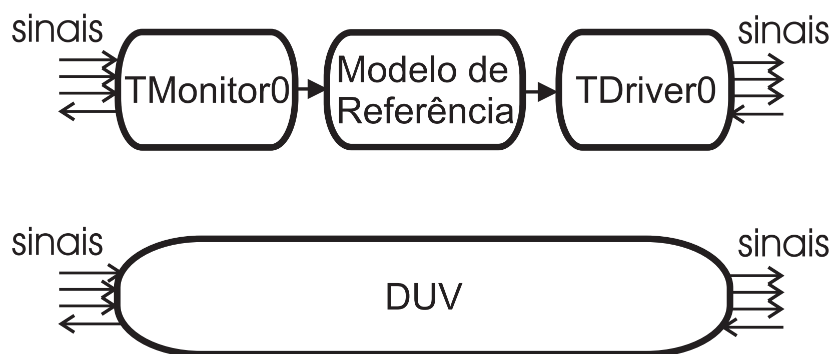

Figura 3.7: Substituição do DUV por TMonitor, Modelo de Referência e TDriver

3.2.2 Decomposição hierárquica do Modelo de Referência (segundo passo)

Considerando que projetos mais complexos são divididos em níveis de hierarquia, é necessário dividir o Modelo de Referência em blocos que correspondam aos blocos do DUV. Dessa forma, o segundo passo consiste na decomposição hierárquica do Modelo de Referência, como mostrado na Figura 3.8.

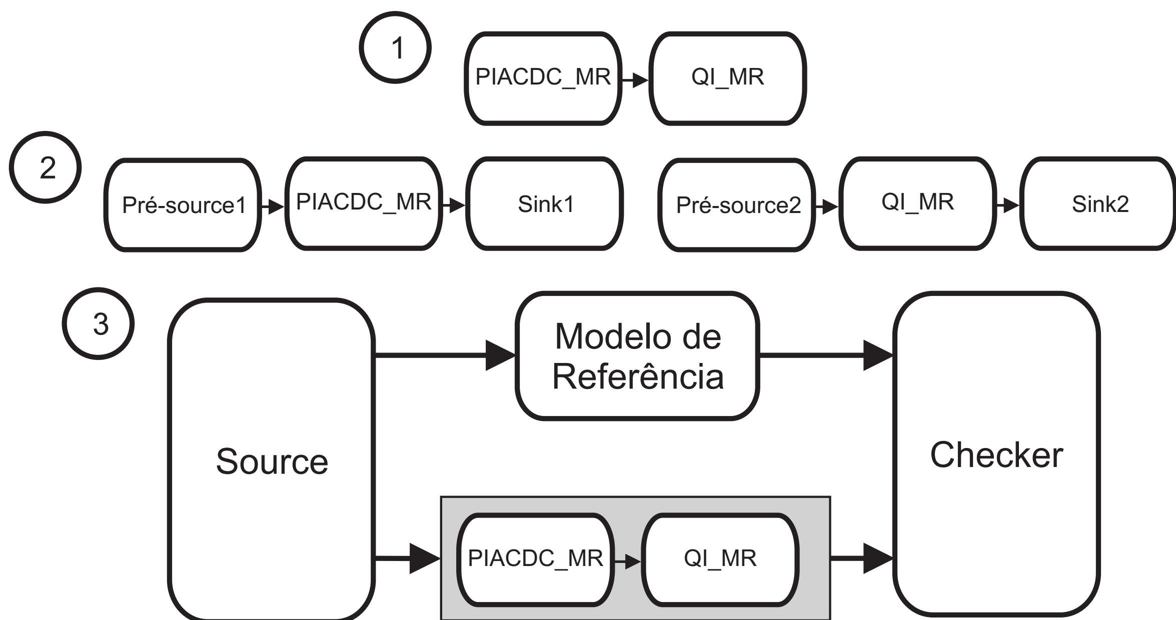

Figura 3.8: Decomposição hierárquica do Modelo de Referência (segundo passo)

O Modelo de Referência, dividido em blocos, deve ser testado da seguinte forma:

1. O Modelo de Referência é dividido hierarquicamente em PIACDC_MR e QI_MR. A decomposição do Modelo de Referência precisa ser equivalente à decomposição hierárquica pretendida para o DUV.

2. Cada bloco resultante da decomposição do Modelo de Referência deve ser tratado como um Modelo de Referência independente e deve ter sua entrada e saída testadas, utilizando o Pré_source e Sink.

3. A junção resultante dos Modelos de Referências que foram particionados devem ser comparados com o Modelo de Referência original, para verificar se a divisão deles em blocos ainda é equivalente ao modelo original. De forma a fazer essa verificação,

o Source insere dados no Modelo de Referência original e no bloco composto pelos módulos PIACDC_MR e QI_MR. O bloco Checker compara o resultado de ambos, como mostrado no número 3 da Figura 3.8.

Após esse passo, cada bloco *PIACDC* e *QI* terá seu próprio Modelo de Referência e deve-se então criar um *testbench* para cada um dos blocos.

3.2.3 Testbench para cada bloco do DUV (terceiro passo)

O terceiro passo gera um *testbench* para cada bloco resultante da decomposição hierárquica do DUV. Nesse exemplo, são gerados dois *testbenches*, um *testbench* para o bloco *PIACDC* e outro para o bloco *QI*. Esse passo é similar a criar um *testbench* para o DUV completo (primeiro passo - Seção 3.2.1).

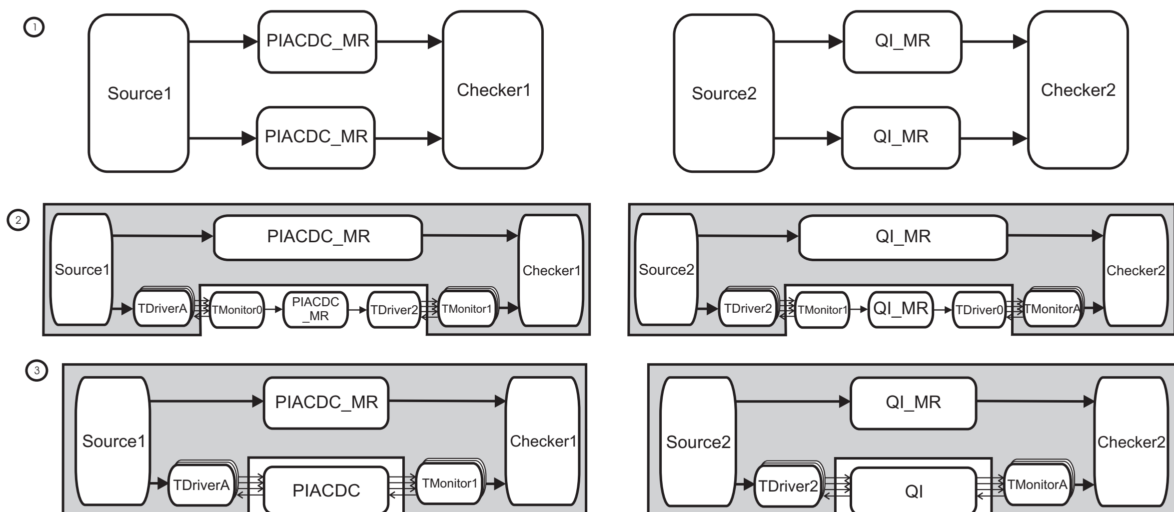

Figura 3.9: Testbenches para cada bloco do DUV: *PIACDC* e *QI* (terceiro passo)

1. Para criar o *testbench* de cada bloco (*PIACDC* e *QI*), devem ser seguidas as mesmas etapas de criação do *testbench* para o DUV completo. O Source e Checker de cada bloco devem ser testados, seguidos pelo(s) TDriver(s) e TMonitor(es). Esses substeps são mostrados nos números 1 e 2 da Figura 3.9, respectivamente. Nesse passo o reuso é aplicado para criar os *testbenches*.

   No *testbench* do primeiro bloco (*PIACDC*) pode-se reusar o TDriverA e TMonitor0, que são do *testbench* do DUV completo. Esses blocos podem ser reusados porque

a interface de entrada do primeiro bloco é a mesma interface de entrada do DUV completo. Além disso, o TDriver2 pode ser reusado pelo *PIACDC*, porque a interface de saída do primeiro bloco (*PIACDC*) é a mesma interface de entrada do segundo bloco (*QI*).

No *testbench* do segundo bloco (*QI*) é possível reusar o TMonitor1, do primeiro bloco, devido a interface de saída do primeiro bloco ser igual a interface de entrada do segundo bloco. Além disso, também é possível reusar TDriver0 e TMonitorA do *testbench* do DUV completo, pois a interface de saída do segundo bloco (*QI*) é igual à interface de saída do *testbench* completo.

Dessa forma, a metodologia pode ser usada para qualquer número de blocos e qualquer topologia da divisão hierárquica, respeitando-se as mesmas regras estabelecidas acima. No caso de outro passo de decomposição hierárquica, devem ser repetidos os passos 2 (Seção 3.2.2) e 3 (Seção 3.2.3).

2. O segundo passo consiste de substituir o grupo {TDriver(s), Modelo de Referência(s), TMonitor(es)} pelos DUV(s) correspondentes, como mostrado no número 3 da Figura 3.9.

3.2.4 Substituição do DUV completo (último passo)

Finalmente, após verificar todos os blocos do *testbench*, é necessário ligá-los e verificar se essa ligação não introduziu erros. Mesmo que cada bloco esteja isento de erros, ao juntá-los pode-se descobrir que alguma funcionalidade não esteja sendo realizada da forma especificada e pode ser que algum erro de interface seja inserido durante essa ligação.

Nesse passo, o grupo composto da tripla (TMonitor0, Modelo de Referência, TDriver0), do *testbench* completo (primeiro passo (Seção 3.2.1)), é substituído pelo DUV completo, formado pelos DUVs que foram divididos em blocos (no exemplo, *PIACDC* e *QI*), para realizar a completa verificação do DUV, como mostrado na Figura 3.10.

Dessa forma, cada bloco do DUV é verificado. Os blocos são ligados e um teste de regressão deve ser feito usando o *testbench* global, para ver se a comunicação dos blocos não introduziu nenhum erro.

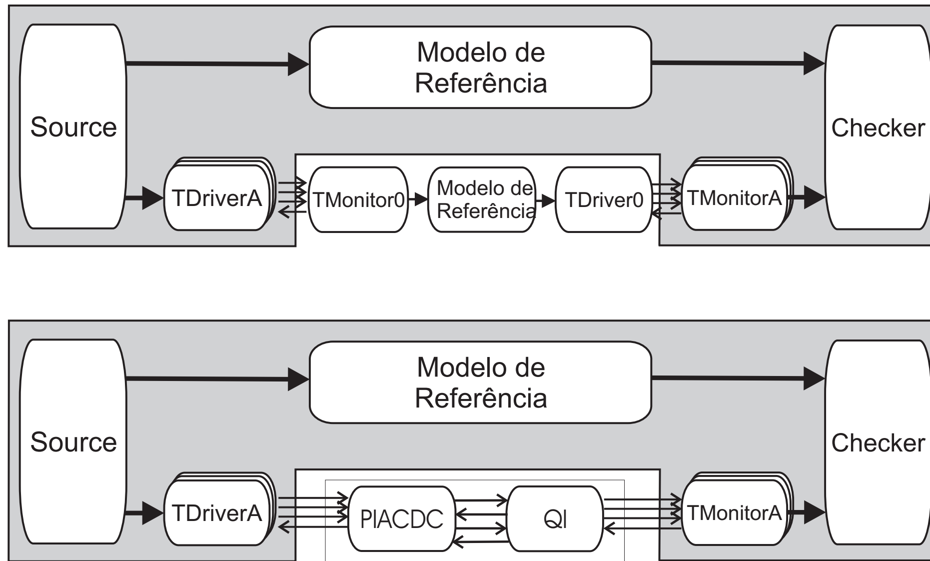

Figura 3.10: Substituição do DUV completo (último passo)

Usando essa metodologia, o *testbench* pode ser usado em todas as fases da implementação. Os *testbenches* criados são bastante confiáveis, permitindo que o engenheiro de projeto mantenha o seu foco na depuração do RTL, tendo mais confiança nos erros acusados pelo Checker do *testbench*.

3.3 Cobertura funcional

O princípio básico da verificação funcional é simular o design através da inserção de estímulos na entrada do mesmo e da comparação de sua saída com a especificação. No entanto, uma das grandes questões existentes dentro da verificação de hardware é se todas as funcionalidades especificadas foram testadas, ou seja, se o dispositivo já foi testado e a simulação pode ser encerrada.

Algumas técnicas procuram resolver esse problema de testar todas as funcionalidades usando um gerador de testes randômicos [27] e [1]. Esses geradores de testes randômicos estão se tornando cada vez mais avançados e realmente têm ajudado a melhorar a qualidade da verificação de dispositivos. Porém, esses geradores de testes sozinhos não podem garantir e nem mostrar quando as funcionalidades especificadas foram ou não testadas ou ainda quando elas foram testadas mais de uma vez. Para responder a pergunta se todas as funcionalidades especificadas foram testadas, tem sido usada uma técnica denominada de cobertura

funcional.

Cobertura é responsável por medir o progresso da verificação através de várias métricas pré-estabelecidas e ajudar o engenheiro a se localizar com relação ao término da verificação [29]. Ela mede o progresso da simulação e reporta quais funcionalidades não foram exercitadas ou foram exercitadas mais de uma vez. Além disso, ela pode também ajudar a inspecionar a qualidade da verificação e direcionar os estímulos de forma a alcançar as funcionalidades não cobertas. Essas funcionalidades não cobertas são normalmente denominadas de buracos de cobertura. É necessário fazer uma análise cuidadosa dos resultados das simulações para distinguir os buracos de cobertura válidos dos inválidos.

Um buraco de cobertura pode ocorrer devido a três diferentes causas:

- Funcionalidades não exercitadas porque o simulador precisa de mais tempo para exercitá-las.
- Funcionalidades não exercitadas porque o Source não está gerando estímulos suficientes para estimular todas as funcionalidades do DUV. Nesse caso o Source precisa ser mudado de forma a gerar esses estímulos.
- Existem erros no dispositivo, que não permitem que as funcionalidades sejam testadas.

Esse trabalho possui a implementação de uma biblioteca de cobertura denominada BVE-COVER (*BVE=Brazil-IP Verification Extension*) implementada em C++, podendo ser usada juntamente com as bibliotecas SystemC e SCV. A biblioteca de cobertura BVE-COVER é parte da metodologia de verificação VeriSC. A biblioteca de cobertura BVE-COVER foi desenvolvida como parte desse trabalho, na Universitat Tuebingen, Alemanha, sob orientação dos profs. Dr. Elmar U. K. Melcher e Dr. Wolfgang Rosenstiel.

A biblioteca de cobertura aqui apresentada se diferencia das demais por fornecer cobertura funcional para ser adicionada ao SystemC. De acordo com a revisão bibliográfica não existem bibliotecas de cobertura públicas que podem ser usadas para a cobertura funcional usando SystemC. A biblioteca aqui apresentada contém os requisitos básicos para a realização da cobertura funcional nos componentes de verificação da metodologia VeriSC.

Como já foi dito em seções anteriores, esse trabalho possui a abordagem black-box. Dessa forma, a verificação é realizada através das interfaces disponíveis, sem conhecimento

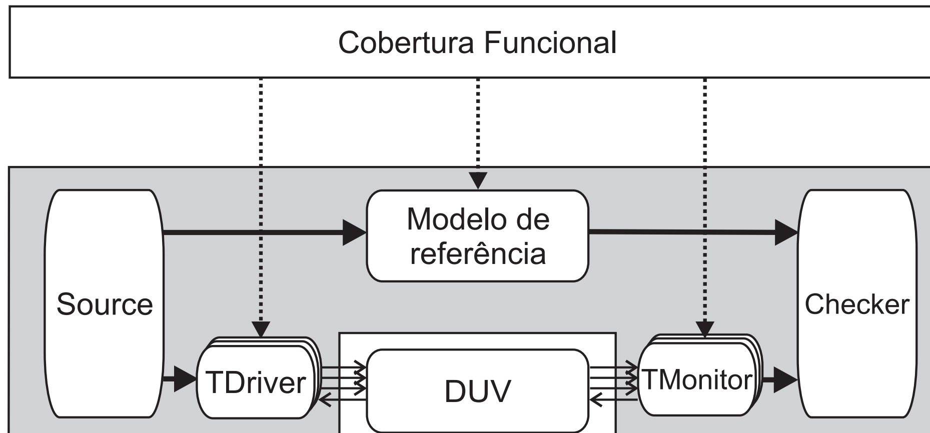

Figura 3.11: Locais para medir cobertura funcional

da estrutura interna. Portanto, a cobertura realizada no *testbench* pode ser medida nos blocos, como mostrada na Figura 3.11:

- TDriver: para medir dados de entrada. Esses dados devem ser medidos quando é necessário saber se o Source randômico está gerando os dados necessários para o dispositivo.

- TMonitor: para medir dados de saída. Quando é necessário medir dados de saída do DUV. Os requisitos de saída especificam o conjunto completo de dados a serem observados.

- Modelo de Referência: no caso de se desejar medir funcionalidades internas do dispositivo no nível de transação.

  Introduzir pontos de cobertura no Source e Checker não fornece nenhuma informação adicional que não possa ser obtida dos TDriver(s), TMonitor(es) e Modelo de Referência.

O primeiro passo para implementar a cobertura funcional é criar um modelo de cobertura. O modelo de cobertura é o documento que deve se ter como base para a implementação da cobertura funcional. Não é recomendado que se inicie a cobertura funcional diretamente na implementação do código da cobertura. Inicialmente deve-se ter certeza de ter entendido toda a funcionalidade do dispositivo a ser verificado e estar certo de ter extraído todas as informações que devem ser verificadas no mesmo.

O modelo de cobertura é formado por todos os valores importantes do dispositivo, que se deseja verificar. Além dos valores importantes, deve-se especificar o relacionamento existente entre cada um dos componentes da verificação (caso esse relacionamento exista). Outros pontos a serem especificados no modelo de cobertura são as condições ilegais do sistema e as condições que devem ser ignoradas na medição da cobertura.

3.3.1 Exemplo de descrição do modelo de cobertura

Para efeito de demonstração de como pode ser feito um modelo de cobertura, será usado o bloco Scan Inverso (SI), parte do decodificador MPEG4.

O módulo SI (Scan Inverso) recebe um vetor unidimensional de 64 elementos e os reordena em um array bidimensional 8x8, seguindo três diferentes seqüências de ordenação (scan vertical, scan horizontal e scan zig-zag), de acordo com os flags da entrada de configuração. Sua especificação para cobertura pode ser feita da seguinte forma:

1. Entrada de configuração: Três flags: acpred_flag e intra_block, vindos do módulo Bitstream e dcpred_direction, vindo do módulo *QI*. Esses três flags indicam qual seqüência de ordenação será utilizada e devem ser gerados para eles valores aleatórios binários entre 0 e 1.

2. Entrada: O módulo SI recebe um array unidimensional de 64 elementos, QFS[64], sendo cada elemento um coeficiente de 12 bits cada. Portanto, é preciso gerar aleatoriamente valores de coeficientes contidos no intervalo [-2048, 2047].

3. Saída: O módulo SI escreve um array de coeficientes de 12 bits reordenados em uma matriz PQF[8][8].

4. Restrição: Não há nenhuma restrição quanto aos valores deste módulo. Relacionamento entre pontos de cobertura: Há um relacionamento entre os flags recebidos por esse módulo, que determinam a seqüência de ordenação que será utilizada para rearranjar os coeficientes.

O modelo de cobertura pode ser visto na Tabela 3.1. Essa tabela descreve os possíveis valores que devem ser alcançados e qual a correlação entre esses atributos durante a simulação. Nessa tabela, o símbolo * significa que não existem restrições de correlação dos atributos.

Atributo	intra_block	acpred_flag	dcpred_direction	QFS	PQF
Possíveis valores	true, false	true, false	true, false	[-2048; 2047]	[-2048; 2047]
Correlação entre atributos	*	*	*	*	*

Tabela 3.1: Modelo de cobertura

3.3.2 Biblioteca BVE-COVER

A biblioteca de cobertura funcional BVE-COVER tem o objetivo de encontrar buracos de cobertura e mostrar o progresso da simulação. Ela é composta de 4 componentes básicos: BVE-COVER Bucket, BVE-COVER Illegal, BVE-COVER Ignore e BVE-COVER Cross-coverage, como mostrado na Figura 3.12.

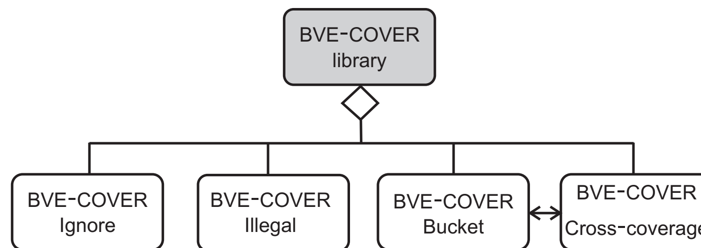

Figura 3.12: Componentes da biblioteca de cobertura BVE-COVER

Essa biblioteca foi implementada em C++ e integrada com a biblioteca SystemC [38]. As próximas subseções explicam cada componente da biblioteca de cobertura.

BVE_COVER Bucket

Todas as funcionalidades importantes que precisam ser cobertas na simulação são especificadas usando o BVE_COVER Bucket. A simulação depende da definição dos *buckets*, uma vez que existem duas formas de parar a simulação: através do enchimento de todos os *buckets* ou através da especificação de um limite de tempo na simulação.

As funcionalidades são especificadas usando a instrumentação de código, como mostrado no Código 15. O engenheiro de verificação deve especificar quais funcionalidades devem ser executadas e quantas vezes elas devem ser executadas para alcançar 100% de cobertura.

Os eventos são coletados diretamente da simulação sendo executada. Cada vez que um evento acontece a biblioteca faz uma atualização na lista de cobertura. O BVE_COVER Bucket avalia os pontos de cobertura em cada novo evento da simulação. Cada vez que um ponto de cobertura é executado, ele atualiza os dados da porcentagem de cobertura.

O BVE_COVER Bucket então calcula a porcentagem do total de cobertura desejado que foi coberto. O algoritmo para calcular a porcentagem usa a seguinte fórmula:

$$cobertura = \frac{\sum_{i=1}^{N} cobertura_i}{N}$$

, onde $cobertura_i$ é uma medida de cobertura particular de cada ponto de cobertura e N é o número de pontos de cobertura.

Código 15 : BVE_COVER_BUCKET

```
01         BVE_COVER_BUCKET  Cv_bucket_intra;

02         Cv_bucket_intra.begin( );
03            BVE_COVER_BUCKET(Cv_bucket_intra, (intra_block==true, 100));
04            BVE_COVER_BUCKET(Cv_bucket_intra, (intra_block==false, 100));
05         Cv_bucket_intra.end( );
```

Os resultados da simulação são armazenados em um arquivo texto para fazer uma análise dos dados armazenados após a simulação.

O BVE_COVER Bucket também é responsável por medir o progresso da simulação. Existe uma barra de progresso que mostra o progresso de cada ponto de cobertura da simulação. Quando todos os pontos de cobertura alcançarem 100% a simulação pára.

De forma a mostrar como a biblioteca funciona, será utilizado o exemplo do Scan Inverso, com partes de suas funcionalidades. Nesse exemplo, a simulação irá terminar quando a variável intra_block executar pelo menos cem vezes cada um de seus possíveis valores, ou seja, *true* e *false*, como mostrado no Código 15.

BVE_COVER Ignore

Existem dois tipos de buracos de cobertura que podem ser descobertos ao analisar a cobertura funcional: buracos de cobertura válidos e buracos inválidos. Um buraco de cobertura válido representa uma observação pretendida que não foi observada. Um buraco de cobertura inválido é um requisito funcional não pretendido em um dos modelos de cobertura.

O BVE_COVER Ignore é usado para evitar o buraco de cobertura inválido e para especificar as funcionalidades que não são consideradas relevantes na simulação. Ele é importante para achar buracos de cobertura que sejam reais, como será explicado na Seção de análise.

O BVE_COVER Ignore pode está exemplificado no Código 16. Nesse código está especificado que sempre que a variável intra_block for igual a 0 e acpred_flag for igual a 1, ela deve ser ignorada, pois essa conbinação de valores não configura uma funcionalidade no bloco Scan Inverso.

Essa funcionalidade faz sentido para a descoberta de buracos de cobertura. Caso essa condição não seja simulada, isso não vem a configurar buraco de cobertura, pois esse é um valor que pode ser ignorado durante a simulação.

Código 16 : BVE-COVER Ignore

```
01          BVE_COVER_IGNORE  Cv_ignore;

02          Cv_ignore.begin( );
03             BVE_COVER_IGNORE(Cv_ignore, ((intra_block==0) && (acpred_flag == 1)));
04          Cv_ignore.end( );
```

BVE_COVER Illegal

Outras características importantes a serem observadas e especificadas são as funcionalidades que são consideradas ilegais, isto é, as funcionalidades que não devem acontecer durante a simulação. Esse componente da biblioteca de cobertura deve ser visto como a combinação de BVE_COVER Ignore e uma *assertion* para achar erros no projeto. Ele tem um papel similar ao Ignore-coverage, com relação a achar buracos de cobertura, mas também identifica as funcionalidades Ilegais como uma assertion e mostra para o usuário que essas funcionalidades ocorreram.

A instrumentação do código é feita, como mostrada no Código 17. Nesse código é possível observar que, se o valor de QFS em algum momento for -2049, esse valor será considerado ilegal, pois está fora da faixa estabelecida para esse valor, no plano de cobertura.

Código 17 : BVE-COVER Illegal

```
01          BVE_COVER_ILLEGAL  Cv_illegal;

02          Cv_illegal.begin( );
03             BVE_COVER_ILLEGAL(Cv_illegal, (QFS==-2049));
04          Cv_illegal.end( );
```

BVE_COVER Cross_coverage

O Cross-coverage é um modelo, cujo espaço é definido pela permutação completa de todos os valores de todos os atributos, mais precisamente conhecido como uma matriz de cobertura multidimensional. Cross-coverage mede a presença de uma combinação de valores. Ela ajuda a responder questões como: "injetei pacotes corrompidos em todas as portas? Executei todas as operações de combinações de leitura em todos os endereços?"

Alguns ou todos os *buckets* podem ser selecionados para criar o Cross-coverage. Com essa informação, a tabela é criada com o cruzamento das informações de todos os *buckets*. A cada novo evento, o simulador verifica se algumas dessas informações cruzadas ocorreram. Em caso de uma ocorrência, os dados são atualizados na tabela. A instrumentação do código é feita, como mostrado no Código 18. A simulação gera um arquivo texto, para análise posterior.

Código 18 : BVE-COVER Cross-coverage

```
01          BVE_COVER_CROSS_COVERAGE  Cv_cc;

02          Cv_cc.begin( );
03             BVE_COVER_CROSS_COVERAGE(Cv_cc, Cv_bucket_intra);
04             BVE_COVER_CROSS_COVERAGE(Cv_cc, Cv_bucket_inter);
05          Cv_cc.end( );
```

Análise

A saída da cobertura pode mostrar muitos aspectos da execução da simulação. Ela pode mostrar o progresso da simulação, buracos de cobertura e atributos ilegais que foram simulados. Analisando esses resultados, pode-se fazer uma adaptação nos estímulos e também fazer bons testes de regressão, conforme será explicado na Seção 3.4.

De forma a analisar os resultados, é muito importante identificar se o buraco de cobertura é um buraco real, ou seja, se as funcionalidades não são parte dos conjuntos BVE_COVER Illegal e/ou BVE_COVER Ignore. Tal fato pode ser descoberto diminuindo das funcionalidades especificadas as funcionalidades BVE_COVER Illegal, BVE_COVER Ignore e funcionalidades executadas. Para ilustrar esse processo de procura do buraco de cobertura, a Figura 3.13 será usada para explicar o processo. O retângulo de fora indica as funcionalida-

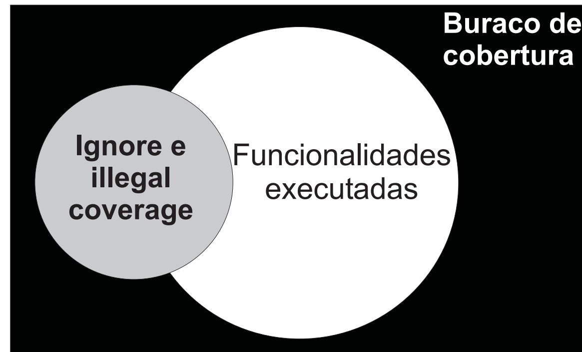

Figura 3.13: Análise do buraco de cobertura

des especificadas para serem executadas na cobertura.

As funcionalidades simuladas durante o processo de verificação são representadas na Figura 3.13 como sendo o círculo interno de cor branca. O círculo de cor cinza representa as funcionalidades especificadas por BVE_COVER Ignore e BVE_COVER Illegal. Os buracos de cobertura são calculados como a parte preta da Figura 3.13.

3.4 Resumo da metodologia VeriSC

No processo de verificação, um ponto crucial é a escolha dos estímulos, para que todas as funcionalidades sejam exercitadas e possíveis erros sejam capturados e corrigidos. Quando um erro é encontrado na simulação, deve-se ficar atento e observar as funcionalidades que se encontram relacionadas a essa funcionalidade que apresentou o erro por duas razões [22]:

- A primeira delas é que pode ser que algum erro que já foi retirado seja inserido novamente ao fazer a correção de outro erro. Por isso, sempre que houver um erro, é necessário que sejam feitos todos os testes anteriores, de forma a verificar que erros anteriores não venham a ocorrer novamente. Esse tipo de teste é denominado de teste de regressão. Os testes de regressão são testes que são aplicados para realizar a simulação sempre que alguma coisa dentro do DUV for modificada. Eles devem ser reaplicados para ter a certeza de que essas novas mudanças não introduziram erros em áreas anteriormente consideradas isentas de erros.

- Outro fator importante, é que podem ocorrer erros nas funcionalidades adjacentes e relacionadas a essa funcionalidade que ocorreu o erro. Dessa forma, é importante que

possam ser gerados testes para áreas adjacentes a essa área onde ocorreu tal erro.

Os estímulos devem ser cuidadosamente escolhidos e redirecionados de acordo com a cobertura. Se durante a verificação a cobertura não evoluir ou não for alcançada, é necessário que os estímulos sejam redirecionados para atingir a cobertura especificada. Deve ser feito um estudo de quais funcionalidades não estão sendo cobertas para que os estímulos sejam direcionados na intenção de captar essas funcionalidades.

O primeiro conjunto de testes que deve ser criado pelo engenheiro de verificação deve ser composto de testes simples para detectar erros básicos [23], seguidos de testes reais. Em seguida pode-se aplicar testes randômicos e observar se a cobertura está evoluindo. Se a cobertura não for alcançada com esses estímulos randômicos, pode-se aplicar testes direcionados para captarem erros específicos.

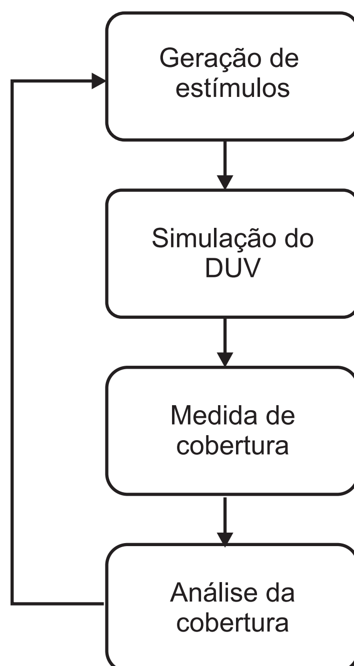

Figura 3.14: Fluxo de cobertura

O processo que deve ser seguido com a análise da cobertura está descrito na Figura 3.14. Esse processo consiste em gerar estímulos, simular o DUV, fazer a coleta e análise dos dados de cobertura e caso esses dados não tenham sido satisfatórios, mudar a geração de estímulos no Source e voltar a simular novamente.

O Capítulo seguinte descreve os estudos de caso realizados para avaliar a metodologia VeriSC.

Capítulo 4

Resultados

A metodologia VeriSC evoluiu de uma metodologia anterior, que propunha a verificação de acordo com o fluxo tradicional, onde o DUV é implementado e, a partir dele o *testbench* é gerado e a verificação é então realizada. Essa metodologia anterior e tradicional foi denominada de VeriSC tradicional.

Outra diferença da metodologia anterior (VeriSC tradicional) para a metodologia atual (VeriSC) é que para empregar a metodologia VeriSC tradicional é necessário haver um DUV para a geração dos *prototypes*. Não há um método que permita a geração dos *prototypes* antes do DUV na metodologia VeriSC tradicional.

A comparação do código reusado pelas metodologia VeriSC tradicional e VeriSC mostra que a metodologia VeriSC reusa o código de todos os TDrivers e TMonitores que possuem interfaces iguais ou que possuem interfaces que são semelhantes e estão no lado oposto de recepção e envio de dados. Em termos de porcentagem de reuso, cada módulo reusado corresponde a 25% do código gerado para o testbench.

Esse capítulo é composto pelos resultados obtidos em todas as partes desenvolvidas para a construção da linguagem de geração de templates, para a metodologia de verificação VeriSC, bem como da biblioteca de cobertura.

4.1 Resultados da metodologia VeriSC tradicional

Para dar suporte ferramental à metodologia VeriSC tradicional, foi implementada uma ferramenta para geração automática de *prototypes*, que recebe como entrada *templates*, o arquivo

do DUV e uma descrição das interfaces do DUV. A saída da ferramenta é composta por *prototypes* para o *testbench*. Maiores informações sobre essa ferramenta podem ser encontradas no artigo publicado no evento SBCCI 2004 [12].

A metodologia VeriSC tradicional e a ferramenta de geração de *prototypes* foram utilizadas para a verificação de um decodificador de vídeo MPEG4, na Universidade Federal de Campina Grande. Além disso, a metodologia foi também usada para a verificação de um decodificador de áudio MP3, na Universidade Federal de Campinas, além de outros projetos parceiros, desenvolvidos nas universidades UFPE, USP, UnB e UFMG, todos integrantes do Brazil IP [24].

Nessa Seção serão mostrados os resultados obtidos com a metodologia tradicional de verificação que foi implementada no início desse trabalho. Serão apresentados dois projetos, o decodificador de áudio MP3 e o decodificador de vídeo MPEG4.

4.1.1 Decodificador MP3

O decodificador de áudio MP3 foi desenvolvido pela Universidade Federal de Campinas e empregou a metodologia VeriSC tradicional em sua verificação funcional. O padrão MPEG-1 layer III, também conhecido como MP3, é um formato para sinais digitais de áudio comprimido e foi padronizado em 1991 pelo MPEG (*Moving Pictures Expert Group*). O Cine-IP 8051 tem 255 instruções, uma RAM interna com 256 bytes de e uma RAM externa de 64 bytes. A frequência de clock usada é de 33 MHz.

O processo de decodificação MP3 recebe um *bitstream* MP3 e gera em sua saída *samples*. MP3 está baseado em pequenos pacotes *(frames)* para os quais são produzidos alguns milisegundos de sinais de áudio. Os pacotes são auto contidos, ou seja, contém as informações de como eles são decodificados bem como os dados de áudio codificado.

O MP3 possui as características descritas na Tabela 4.1:

Na implementação e verificação desse projeto, foi empregado apenas um nível hierárquico, como pode ser observado no esquema do decodificador MP3, descrito na Figura 4.1, onde cada bloco foi verificado separadamente. Os resultados obtidos na verificação funcional foram publicados em conjunto com a Unicamp, no Congresso SBCCI 2004 [12]. Esses resultados estão descritos nessa Seção.

Durante a implementação foi realizado um conjunto de testes pelos projetistas do MP3.

Número de linhas de código do Modelo de Referência	12.896
Número de linhas de código do RTL	933
Linhas de código do testbench	28.548
Vetores de teste	81.762.300

Tabela 4.1: Características do MP3

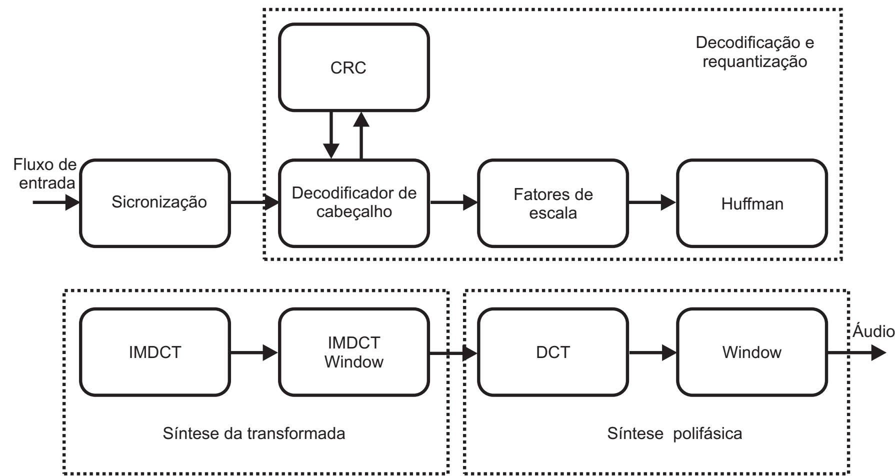

Figura 4.1: Decodificador MP3

Ao final da implementação, esses testes prévios não conseguiram detectar nenhum problema de implementação do MP3.

Porém, durante a verificação funcional, alguns erros foram encontrados, os quais não foram encontrados durante o conjunto de testes iniciais. Esses erros são relevantes e foram encontrados no módulo Window.

Para a realização da verificação desse módulo, o Source utilizou ponto flutuante para fornecer estímulos randômicos. Os dados randômicos foram gerados utilizando a biblioteca SCV_CONSTRAINT do SCV (SystemC Verification Library).

Os erros encontrados na simulação foram os seguintes:

1. Na máquina de estados finitos (FSM), o estado *reset* não reiniciou o vetor nt[2][512] para o valor zero;

2. Os primeiros 15 blocos de saída causaram erros quando o sinal de *reset* foi iniciado;

3. O módulo somente decodificava corretamente dados estereofônicos, não sendo capaz de decodificar corretamente dados monofônicos.

Portanto, o uso da metodologia de verificação conseguiu captar erros na implementação que não puderam ser captados pelos testes realizados pelo engenheiro de projeto. O projeto do decodificador MP3 foi transformado em um chip de Silício que funcionou sem falhas na primeira rodada de Silício.

4.1.2 Decodificador MPEG4

O MPEG4 é um padrão aberto de codificação de vídeo que foi criado pelo grupo MPEG (*Motion Picture Experts Group*), o qual veio substituir o padrão MPEG2.

O decodificador de vídeo MPEG4, apresentado nesse trabalho, é parte do projeto Brazil IP e foi desenvolvido na Universidade Federal de Campina Grande.

Na verificação do MPEG4, foi usado somente um nível de decomposição hierárquica. O DUV (MPEG4), foi dividido em blocos. Cada bloco teve o seu código RTL implementado por um membro da equipe. Após a codificação RTL, os membros do grupo foram alocados com blocos do MPEG4 para realizar a verificação funcional. Os blocos verificados pelos membros da equipe foram diferentes dos blocos codificados por eles, de forma a garantir que a verificação realizada não estivesse "viciada". A Figura 4.2 descreve a divisão do MPEG4 em blocos.

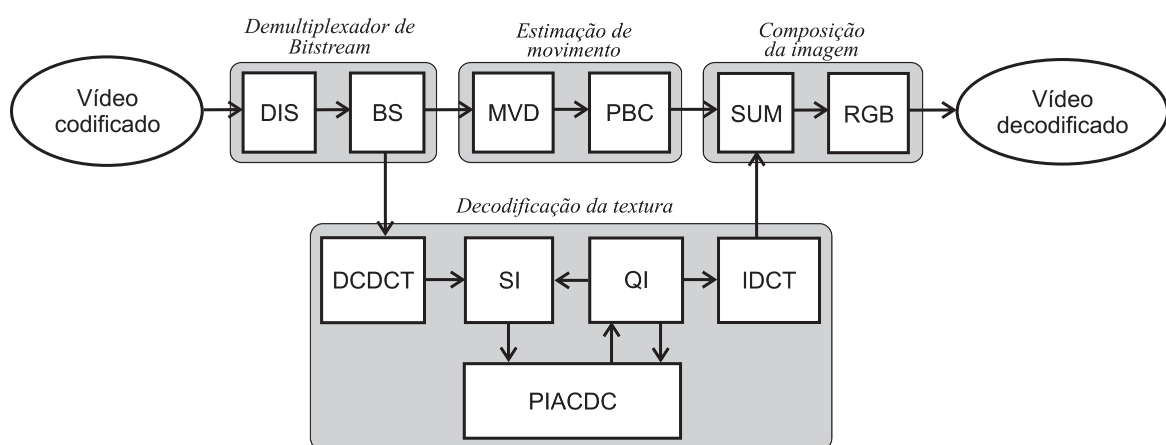

Figura 4.2: Decodificador MPEG4

O Modelo de Referência usado para verificar o MPEG4 foi o Software XVID [39]. A re-

Número de linhas de código do Modelo de Referência	12.718
Número de linhas de código do RTL	21.473
Linhas de código do testbench	51.382
Vetores de teste	50.789.096

Tabela 4.2: Características do MPEG4

alização da verificação do MPEG4 foi realizada em 75% do total verificado com a metodologia VeriSC tradicional. Os demais 25% (Decodificador de Bitstream) foram verificados utilizando a metodologia VeriSC, tema desse trabalho.

Algumas características do MPEG4 podem ser vistas na tabela 4.2.

Através do uso dessa metodologia (VeriSC tradicional) e da ferramenta muitos resultados positivos foram alcançados.

1. Os membros do grupo na verificação funcional foram guiados, já que nenhum deles tinha uma grande experiência com verificação.

2. Redução do tempo de verificação, pois os *prototypes* continham uma boa parte do código já implementado e adaptado para o DUV.

3. Segurança para o verificador com relação à ordem na qual o trabalho estava sendo realizado.

Alguns pontos negativos também foram observados nessa metodologia tradicional:

1. Não garantia de que após preencher os *prototypes* eles estariam funcionando sem erros.

2. As interfaces de sinais dos blocos do DUV apresentaram alguns erros quando da junção para a formação do MPEG4.

3. Trabalho do implementador de RTL de criar um ambiente de simulação durante a sua implementação para estimular e analisar as respostas do DUV.

4. A verificação funcional e implementação do *testbench* somente podem ser iniciadas ao final da implementação do RTL de cada bloco.

A metodologia foi muito útil para ajudar a localizar erros durante a verificação funcional. O projeto do decodificador MPEG4 já foi convertido em um chip de Silício que funcionou sem falhas na primeira rodada de silício e é um dos chips mais complexos já desenvolvidos no Brasil [33].

4.2 Resultados da metodologia VeriSC

A metodologia VeriSC, apresentada nesse trabalho, foi bastante testada e obteve bons resultados em seus testes. Ela foi empregada em 25% da verificação do MPEG4, que representa o processador Bitstream. Além disso, foi também utilizada no projeto Lacre Digital (DigiSeal), desenvolvido e verificado na Universidade federal de Campina Grande. Outra aplicação não menos importante foi o uso da metodologia em cursos de verificação funcional, no qual os alunos desenvolveram alguns módulos acadêmicos para aprenderem como usar a metodologia.

4.2.1 Decodificador de Bitstream

O Decodificados de Bitstream é um demultiplexador do sistema MPEG4. É ele que envia os dados para os demais blocos do MPEG4. Esse processador e sua arquitetura foram criados especificamente para decodificar os fluxos de entrada e mandar os dados extraídos para os diferentes módulos de processamento do decodificador MPEG4. O Bitstream tem 1625 linhas de código em SystemC e foi verificado em 90 dias.

Com o uso da metodologia no processador de Bitstream foi possível comparar a performance das duas metodologias adotadas na verificação do MPEG4. Para fazer essa comparação foram coletados dados de verificação e desenvolvimento de todos os blocos. O controlador de versões Subversion (SVN) [9] foi empregado na coleta de dados. Subversion guarda a versão de cada atualização feita, além do dia e horário em que essa atualização foi realizada.

Durante a verificação dos blocos do *testbench* que seguiram a metodologia VeriSC tradicional, a verificação de cada bloco foi realizada separadamente, sem nenhum reuso. Após a verificação de todos os blocos, eles foram ligados de forma a obter o DUV completo MPEG4. Nessa fase de junção muitos problemas foram detectados:

- As interfaces dos blocos em RTL e TL (*transaction level)* introduziram erros nas comunicações com os blocos vizinhos, porque não houve reuso e foram implementados separadamente; O tempo estimado, baseado em logs do SVN para depurar esses erros de comunicação e para ligar os blocos no mesmo DUV, funcionando sem erros, foi de 11% do tempo total de projeto.
- Foi estimado 20% do tempo de projeto para depurar o completo *testbench*, até que ele chegasse ao ponto em que não houvesse erros de compilação e nem erros semânticos.
- Tempo elevado gasto pelo codificador RTL para criar um ambiente de simulação durante a sua implementação para estimular e analisar as respostas do DUV antes de passar para o responsável pela verificação. Esse ambiente foi descartado após essa fase e não foi utilizado na verificação.

Essa mesma avaliação realizada com a metodologia VeriSC mostrou que:

- O tempo gasto para fazer a ligação entre os blocos foi próximo de zero, porque todas as interfaces são integradas nos TDriver(s) e TMonitor(es) do *testbench* e eles são sempre reusados, de forma que as interfaces estarão compatíveis.
- Com a implementação hierárquica do *testbench*, os erros do *testbench* vão sendo depurados no decorrer da sua implementação. Nessa abordagem, um pouco mais da metade do tempo de depuração é economizado.
- A criação de um ambiente para fazer a pré-verificação do DUV não é necessária, uma vez que o *testbench* está pronto antes mesmo do início da implementação do RTL.

A seguinte Tabela 4.3 resume a comparação entre a metodologia VeriSC apresentada nesse trabalho e a metodologia tradicional VeriSC tradicional (trabalho anterior), com relação ao estudo de caso do MPEG4.

Dessa forma, é possível concluir que a metodologia VeriSC leva a um ganho em torno de 30% de produtividade comparada com a metodologia VeriSC tradicional

	Metodologia VeriSC tradicional	**Metodologia VeriSC**
Tempo gasto para ligar os blocos do DUV	11%	Próximo de zero
Tempo gasto para a depuração do *testbench*	20%	Aproximadamente 10%
Construção de um ambiente para simulação prévia do DUV	10%	Não necessário

Tabela 4.3: Comparação entre metodologia VeriSC e metodologia VeriSC tradicional

4.2.2 Resultados obtidos nos cursos de verificação

Outros resultados foram obtidos com testes realizados com estudantes em cursos de verificação na Universidade Federal de Campina Grande. Esses cursos foram ministrados pelo prof. Elmar Melcher. O experimento foi realizado com estudantes como mesmo grau de conhecimento e a mesma experiência. Os estudantes tinham pouca experiência em desenvolvimento Orientado a Objetos e quase nenhuma experiência com a linguagem C++. Devido aos estudantes terem pouca experiência com hardware e não terem utilizado nenhuma outra metodologia de verificação anteriormente, foi utilizado um simples conversor PCM/DPCM, que consiste em calcular a diferença de duas amostras de áudio digital subseqüentes, realizando a operação de saturação no resultado dessa diferença [11].

O primeiro grupo de alunos que realizou o curso de verificação, empregou a metodologia tradicional para verificar o seu projeto. Esse grupo não fez um *testbench* hierárquico, eles implementaram somente um *testbench* depois de haver implementado o dispositivo RTL em um dos blocos do DUV.

O segundo grupo de alunos, realizou o segundo curso ministrado na UFCG e usou a metodologia VeriSC e implementou 3 *testbenches*. Um *testbench* para o DUV completo, um *testbench* para o bloco diferença e um *testbench* para o bloco de saturação. Eles empregaram a abordagem hierárquica da metodologia para a realização dessa verificação funcional.

Os resultados experimentais indicaram que houve um aumento de 30% da produtividade com os estudantes que utilizaram a metodologia VeriSC e empregaram a abordagem

hierárquica. Observações realizadas com os estudantes durante o curso revelaram que essa melhor produtividade foi devida aos seguintes fatores:

- Elevado grau de reuso de código do *testbench* em VeriSC e o oposto na metodologia VeriSC tradicional.
- Menos erros de compilação devido à abordagem incremental do VeriSC.
- Menos erros de execução, devido à mesma razão.
- Menos tempo para depurar o RTL, uma vez que o *testbench* já estava pronto antes mesmo do início da implementação do RTL.

4.2.3 Resultados obtidos com o projeto DigiSeal

O projeto Lacre Digital tem como objetivo conceber e implementar um dispositivo destinado à detecção da violação de dispositivos instalados em redes aéreas de distribuição de energia elétrica. O circuito digital desenvolvido para estes dispositivos se chama DigiSeal.

Os blocos que constituem o DigiSeal apresentados na Figura 4.3 foram codificados em Linguagem de Descrição de Hardware (Verilog e VHDL), para a criação do DUV de cada bloco, e nas Linguagens C/C++ e SystemC, na confecção dos modelos de referência e dos seus *testbenches*.

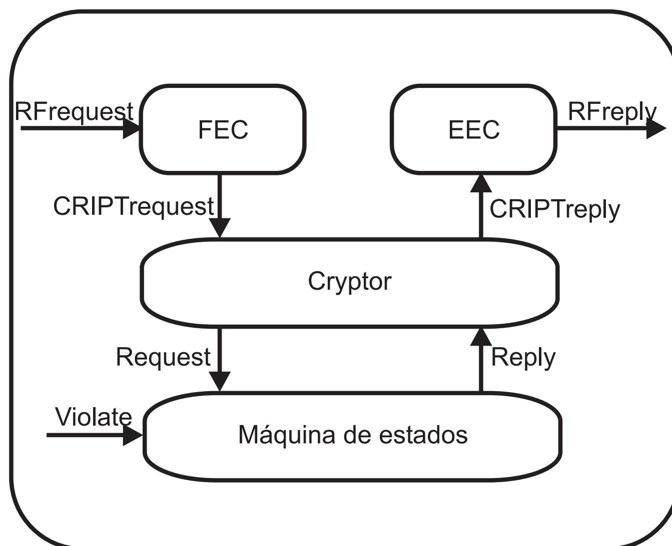

Figura 4.3: Bloco esquemático do DigiSeal

-	**Blocos do MPEG4**	**DigiSeal**
Número de linhas do código do *testbench*	6682	4534
Número de linhas do código do RTL	3005	3112
Número de portas lógicas	6876	6176
Área de Silício	2.2 x 2.2 mm	2,41x2,41 mm

Tabela 4.4: Comparação entre DigiSeal e parte do MPEG4

Os resultados consistem na análise da metodologia VeriSC para o DigiSeal. Esses resultados são comparados a uma aplicação que utilizou a metodologia tradicional, que é uma parte do MPEG4 que possui uma complexidade aproximada à do DigiSeal [33].

A verificação funcional foi realizada em cada um dos blocos hierárquicos, conforme a metodologia estabelece. Após a verificação de cada bloco, foi realizada uma regressão com a junção dos blocos para formar o DUV completo e nessa fase, houve somente um erro devido a funcionalidades que não foram completamente cobertas na verificação dos blocos individuais.

Após a construção do FPGA, o mesmo se mostrou completamente funcional na primeira tentativa e não apresentou nenhum erro, o que veio a confirmar a eficiência da metodologia adotada na verificação.

Para efeito de comparação, a Tabela 4.4 mostra dados dos dois projetos a serem comparados nessa fase.

De forma a avaliar a eficiência do VeriSC com mais detalhes, o projeto DigiSeal foi comparado com um conjunto de blocos do projeto do decodificador MPEG4. Os blocos escolhidos do MPEG4 para comparar com o DigiSeal foram *PIACDC* e *QI*, ambos verificados utilizando a metodologia VeriSC tradicional *PIACDC* implementa a predição inversa AC/DC e *QI* implementa a quantização inversa.

Os valores na Tabela 4.4 indicam que os dois projetos possuem uma complexidade equivalente. Entretanto, a análise do respectivo tamanho do código dos *testbenches* revela que o código do *testbench* do VeriSC é mais eficiente que da metodologia VeriSC tradicional

devido ao fato de haver muito reuso na metodologia VeriSC.

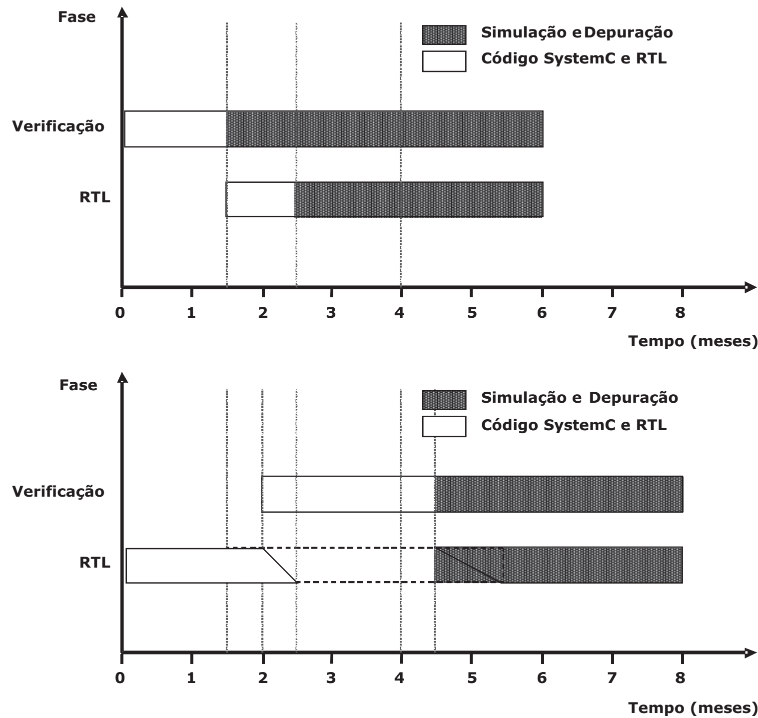

Figura 4.4: Comparação entre DigiSeal e MPEG4 respectivamente

Os dois gráficos da Figura 4.4 mostram o ganho de tempo no projeto usando VeriSC. O primeiro gráfico mostra o tempo de projeto para o DigiSeal e o segundo gráfico o tempo de parte do projeto MPEG4. As fases de simulação e depuração são mostradas como sendo as partes da figura que estão hachuradas. A implementação do *testbench* e a codificação do RTL são mostradas como as partes da figura que estão de branco.

O Projeto DigiSeal começou com verificação funcional, isto é, construindo o *testbench*. Após um mês, o código RTL teve início usando o primeiro *testbench* disponível para a StateMachine. Desse ponto em diante, a escrita do RTL e a sobreposição da simulação com a escrita do *testbench* e a simulação para verificação funcional. Essa sobreposição foi possível porque o primeiro bloco do *testbench* já estava completo.

Os blocos PIACDC e QI iniciaram a partir do código RTL. Nessa fase, código extra foi necessário para que o projetista estivesse apto a testar o código RTL. Após 2 meses, quando o RTL já estava pronto, o *testbench* foi construído para simulá-lo. Somente após o *testbench* estar pronto, após 4 meses, a verificação funcional teve início.

Comparando os dois gráficos da Figura 4.4, pode-se perceber que o projeto DigiSeal teve

um tempo de desenvolvimento em torno de 30% menor que o *PIACDC+QI*. As razões para isso ter acontecido são:

- A completa superposição da implementação RTL e verificação funcional;
- A necessidade de código RTL extra para testar o código;
- Menos tempo gasto na escrita do *testbench* por causa de reuso de código;
- Consistência das interfaces entre blocos da decomposição hierárquica facilmente realizada pelo reuso sistemático de código.

4.3 Resultados obtidos com biblioteca BVE-COVER

Os experimentos com a biblioteca de cobertura foram feitos em todos os módulos do decodificador MPEG4.

Foi especificado um modelo de cobertura para cada módulo do decodificador MPEG4: DVM (Decodificador do Vetor de Movimento), DVM_VOP (*Vídeo Object Plane* do Decodificador de Movimento), CBP (Copiador do Bloco de Predição), DCDCT (Decodificador de coeficiente da Transformada Inversa do Cosseno), SI (Scan Inverso), PIACDC (Predição Inversa AC/DC), QI (Quantização Inversa) e BS (Bitstream). Foram escolhidos vídeos reais para a medição da cobertura especificada nos TMonitores, TDrivers e Modelo de Referência. A maioria dos blocos alcançaram 100%, com excessão do Bitstream. O módulo Bitstream apresentou uma cobertura baixa, como mostrado na Tabela 4.5. Essa tabela mostra os resultados de cobertura de todos os módulos. A primeira coluna mostra cada bloco do MPEG4 que foi definido um modelo de cobertura e a primeira linha mostra os vídeos que foram aplicados na simulação.

Esses modelos de coberturas foram especificados individualmente, de forma que cada bloco tivesse o seu próprio modelo de cobertura.

O bloco Bitstream se comunica com quase todos os módulos do decodificador MPEG4. Ele possui 8 TMonitor(s), 2 com envio de dados puros para os demais blocos e 6 com dados de configuração. No entanto, o Bitstream recebe como dados de entrada um fluxo de vídeo codificado, que deve ser decodificado e enviado para os demais blocos. Como o bitstream

	Deep %	**Drift %**	**Beer %**	**Cobertura média %**
DVM	100	100	100	100
DVM_VOP	100	100	100	100
CBP	100	100	100	100
DCDCT	100	100	100	100
SI	100	100	100	100
PIACDC	100	100	100	100
QI	100	100	100	100
BS	86.1	84.4	85.7	88

Tabela 4.5: Tabela de resultados da cobertura

não alcançou 100% de cobertura, foi realizada uma investigação para descobrir se havia algum erro de funcionalidade nesse bloco. Foi descoberto que o vídeo randômico sendo gerado para simular o bitstream não estava estimulando o bitstream de forma a obter 100% de cobertura. Por isso, outro vídeo randômico foi criado para simular o bitstream, de forma a obter 100% de cobertura. O novo vídeo é um trabalho da disciplina Projeto 2, do Curso de Ciência da Computação, DSC, UFCG.

Capítulo 5

Conclusão

Nesse trabalho de doutorado foi discutida a problemática que envolve o desenvolvimento de hardware, com o foco voltado para os problemas relacionados à verificação funcional. Foram introduzidos conceitos referentes às fases de desenvolvimento de um projeto de hardware, bem como onde se localiza a parte de verificação funcional dentro desse projeto. Em seguida, foram detalhados os conceitos básicos necessários, bem como os trabalhos relacionados e relevantes para a verificação funcional. O trabalho desenvolvido como tese de doutorado foi então explicado e finalmente foram mostrados todos os resultados alcançados.

Devido a existirem diversas definições para alguns termos apresentados, o Capítulo 2 foi destinado à introdução de conceitos básicos e dos trabalhos relacionados.

A metodologia VeriSC, diverge do fluxo de verificação tradicional dos trabalhos apresentados no estado da arte, propondo um fluxo de verificação funcional melhor integrado no desenvolvimento do projeto. Ela aborda as partes de geração de estímulos, implementação da cobertura funcional e fluxo de projeto. Ela propõe uma solução para o problema de reuso de código, uma vez que todas as interfaces que são iguais ou que se comunicam de alguma forma, podem reusar código que já foi gerado, como pode ser visto no Capítulo 3.

Outro problema que a metodologia propõe solução e que não é proposto nos demais trabalhos, é a geração do testbench antes do DUV, possibilitando o uso desse testbench para fazer testes na implementação do DUV durante o trabalho do engenheiro de projeto.

Além disso, a metodologia VeriSC faz testes exaustivos no ambiente de verificação para garantir que ao encontrar erros durante a simulação, esses erros sejam devido ao DUV e não ao ambiente de verificação.

Um problema encontrado na metodologia VeriSC é que ela não implementa os critérios de cobertura de forma automática, sendo que isso gera um trabalho considerável para inserir esses critérios manualmente.

Um outro ponto a ser analisado é que a metodologia VeriSC adota uma metodologia Black-box. Isso traz a desvantagem de ser difícil descobrir erros diretamente no local, sem uma investigação maior. A vantagem do Black-box é que não há uma dependência de código, pois não há nenhum código inserido no DUV, de forma que uma mudança no DUV não causa impacto no código do testbench.

Finalmente, a metodologia propõe que o DUV seja verificado utilizando uma abordagem hierárquica, onde ele é dividido em blocos para que seja possível realizar a verificação em partes do DUV. A metodologia acompanha esta decomposição hierárquica e mantém a coerência das interfaces entre blocos.

Usando a metodologia VeriSC é possível economizar tempo de implementação tanto para o *testbench* quanto para o DUV, resultando num ganho no tempo global do projeto. Os resultados mostram um aumento em torno de 30% na produtividade de projetos.

5.1 Trabalhos futuros

Existem melhorias que podem ser implementadas como trabalhos futuros. Entre essas melhorias podemos citar:

- Implementar uma forma de colocar cobertura automaticamente.
- Assertions com tempo para verificar algumas características funcionais.
- Melhor acoplamento de VeriSC ao ipProcess [2; 26].
- Testar a metodologia para DUVs com interfaces bidirecionais.
- Gerar relatórios de cobertura que possam ser analisados diretamente em um banco de dados.
- Gerar formas de criar estímulos para um DUV automaticamente.
- Pesquisar o uso da metodologia para DUVs Gray-box.

Bibliografia

[1] A. Ahi, G. Burroughs, A. Gore, S. LaMar, C.-Y. Lin, and A. Wiemann. Design verification of the hp 9000 series 700 pa-risc workstations. *Hewlett-Packard*, 43:34–42, 1992.

[2] Guido Araújo, Edna Barros, Elmar Melcher, Rodolfo Azevedo, Karina R. G. da Silva, Bruno Prado, and Manoel E. de Lima. A systemc-only design methodology and the cine-ip multimedia platform. *Design Automation for Embedded Systems*, 10(2-3):181–202, 2006.

[3] Sigal Asaf, Eitan Marcus, and Avi Ziv. Defining coverage views to improve functional coverage analysis. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 41–44, New York, NY, USA, 2004. ACM Press.

[4] Mike Benjamin, Daniel Geist, Alan Hartman, Gerard Mas, Ralph Smeets, and Yaron Wolfsthal. A study in coverage-driven test generation. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 970–975, New York, NY, USA, 1999. ACM Press.

[5] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[6] J. Bhasker. *A SystemC Primer*. Star Galaxy Publisher, first edition, June 2002.

[7] D. S. Brahme, S. Cox, J. Gallo, W. Grundmann, nad W. Paulsen C. N. Ip, J. L. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical report CDNL-TR-2000-0825, Cadence Berkeley Labs, August 2000.

[8] M. Brunelli, L. Battú, A. Castelnuovo, and F. Sforza. Functional verification of an hw block using vera. *Synopys Users Group*, 2001.

[9] Collins-Sussman, B. Fitzpatrick, Brian W., C. Pilato, and Michael. Version control with subversion. 2004.

[10] Coware. http://www.coware.com. Último acesso em Janeiro 2007.

[11] K. R. G. da Silva, E. U. K. Melcher, I. Maia, and H. do N. Cunha. A methodology aimed at better integration of functional verification and rtl design. *Design Automation for Embedded Systems*, 10(4):285–298, 2007.

[12] Karina R. G. da Silva, Elmar U. K. Melcher, Guido Araujo, and Valdiney Alves Pimenta. 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*, pages 66–70, New York, NY, USA, 2004. ACM Press.

[13] C. A. Dueñas. Verification and test challenges in soc designs. Invited Talk, September 2004.

[14] F. Ferrandi, M. Rendine, and D. Sciuto. Functional verification for systemc descriptions using constraint solving. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 744, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Alessandro Fin, Franco Fummi, Maurizio Martignano, and Mirko Signoretto. Systemc: a homogenous environment to test embedded systems. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 17–22, New York, NY, USA, 2001. ACM Press.

[16] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 286–291, New York, NY, USA, 2003. ACM Press.

[17] VSI Alliance Functional Verification Development Group. Specification for vc/soc functional verification version 1.0 (ver 21.0). 2004.

[18] Richard Goering. Dai enters transaction-based verification market. November 1998.

[19] Mentor Graphics. http://www.model.com. último acesso em Janeiro 2007.

[20] Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, and Avi Ziv. User defined coverage: a tool supported methodology for design verification. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 158–163, New York, NY, USA, 1998. ACM Press.

[21] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Massachussetts, USA, first edition, March 2002.

[22] A. Hekmatpour. Coverage-directed management and optimization of random functional verification. *International Test Conference*, pages 148–155, September 2003.

[23] E. Hu, B. Yeh, and T. Chan. A methodology for design verification. In *ASIC Conference and Exhibit*, pages 236–239, Santa Clara, CA, 1994. IEEE Computer Society.

[24] Brazil IP. www.brazilip.org.br. Último acesso em Janeiro 2007.

[25] Oded Lachish, Eitan Marcus, Shmuel Ur, and Avi Ziv. Hole analysis for functional coverage data. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 807–812, New York, NY, USA, 2002. ACM Press.

[26] Marilia Lima, Andre Aziz, Diogo Alves, Patricia Lira, Vitor Schwambach, and Edna Barros. ipprocess: Using a process to teach ip-core development. In *MSE*, pages 27–28, 2005.

[27] James Monaco, David Holloway, and Rajesh Raina. Functional verification methodology for the powerpc 604 microprocessor. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 319–324, New York, NY, USA, 1996. ACM Press.

[28] Jayant Nagda. High level functional verification closure. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers*

and Processors (ICCD'02), page 91, Washington, DC, USA, 2002. IEEE Computer Society.

[29] A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publisher, Massachusetts, USA, first edition, April 2004.

[30] A. Randjic, N. Ostapcuk, I.Soldo, P. Markovic, and V. Mujkovic. Complex asics verification with systemc. In *23rd International Conference on Microelectronics*, pages 671–674, 2002.

[31] P. Rashinkar, P. Paterson, and L. Singh, editors. *System-on-a-Chip Verification Methodology and Techniques*. Kluwer Academic Publishers, Massachusetts, 2001.

[32] S. Regimbal, J. F. Lemire, Y. Savaria, G. Bois, M. Aboulhamid, and A. Baron. Automating functional coverage analysis based on an executable specification. In *International Workshop on System-on-Chip for Real Time Aplications*, pages 228– 234, 2003.

[33] Ana Karina Rocha, Patricia Lira, Yang Yun Ju, Edna Barros, Elmar Melcher, and Guido Araujo. Silicon validated ip cores designed by the brazil-ip network. In *IP/SOC 2006*, June 2006.

[34] Edgar L. Romero, Marius Strum, and Wang Jiang Chau. Comparing two testbench methods for hierarchical functional verification of a bluetooth baseband adaptor. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 327–332, 2005.

[35] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Checking temporal properties under simulation of executable system descriptions. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 161, Washington, DC, USA, 2000. IEEE Computer Society.

[36] Michael Santarini. Cadence moves toward intelligent testbench. June 1999.

[37] IEEE Computer Society. Ieee standard verilog hardware description language. Technical report.

[38] S. Swan. An introduction to system level modelling in systemc 2.0. 2001.

[39] XviD Team. Xvid api 2.1 reference (for 0.9.x series). 2003.

[40] Cadence Design Systems Testbuilder. http://www.cadence.com. Último acesso em Janeiro 2007.

[41] Verisity. A promising approach to overcome the verification gap of modern soc designs. Technical report, 2006.

[42] Ilya Wagner, Valeria Bertacco, and Todd Austin. Stresstest: An automatic approach to test generation via activity monitors. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 783–788, New York, NY, USA, 2005. ACM Press.

Apêndice A

eTBc tool: A Transaction Level Testbenches Generator

The complexity of the verification design is the major problem in the IC (*Integrated Circuits*) projects and IP cores development time, because it consumes about 70% of project resources. Therefore, a methodology that can speed up the verification process and a tool to generate the testbench automatically is important. The focus of this paper is the presentation of a tool, denominated eTBc (*Easy Testbench Creator*), which works with a new concept of design and verification flow. This tool automatically implements testbench templates that help the verification engineer in his task. The testbenches templates are generated according to VeriSC2 methodology that allows for testbench construction and debugging before RTL code is available, without any extra work when compared to a methodology that creates testbenches only after RTL code has been written.

A.1 Introduction

According to industry experience, 65% of IC projects fail at first silicon and 70% of these failures are caused by insufficient functional verification [13]. Usually about 70% of the design effort is spent in the verification time [5].

Functional verification is a process used to demonstrate by simulation that the intent of a design is preserved in its implementation. It is currently often used in the verification of integrated circuit designs, along with other methods such as formal verification, semiformal

verification, hardware emulation, prototyping, etc. Functional verification uses a testbench to create an environment to simulate the DUV (*Design Under Verification*). The testbench simulates all functionalities of the DUV, comparing it to the specification.

The implementation of the testbench can take a considerable amount of time in the verification process. Some reasons which can increase the time for testbench implementation are the number connections between modules/blocks, the time spent to adapt the testbench to the DUV and the number of module instances, transactions data structures and transactions communication channels.

A methodology that makes the complex verification process easier and a tool to implement this methodology and generate automatically testbench templates can be a good approach to reduce the overall time of project flow.

Traditional verification methodologies [7][30][42][31] propose to implement the DUV first and to build the testbench around it. The methodology described in this paper (VeriSC2 methodology[11]), is an evolution of VeriSC1 methodology [12] and allows for testbench construction and debugging before RTL (*Register Transfer Level*) code is available. Traditional methodologies may be adapted to be able to provide a testbench before RTL design, however only at the expense of writing extra code for the DUV that takes the place of the future RTL implementation [5]. VeriSC2 does not need any extra code writing because all the elements of the pre-RTL testbench are reused.

The purpose of this paper is the presentation of eTBc (*Easy Testbench Creator*) which is a tool for automatic generation of templates for all testbenches used in VeriSC2 methodology.

The remaining of this paper is organized as follows: in Section A.2 the VeriSC2 verification flow will be explained. Section A.3 will present the eTBc tool and Section A.4 shows the results.

A.2 VeriSC2 Methodology

According to Bergeron [5], a good testbench should be: a) Transaction Level: interfaces described in terms of wires and signal transition should only be used when connecting to the RTL design. b) Coverage Driven: stimuli generation should depend on functional coverage measurements. d) Random Constrained: constraint stimuli should be generated randomly

within well defined limitations. e) Self Checking: the comparison between expected responses and actual responses should be done automatically.

In addition to the above properties, important characteristics of VeriSC2 methodology are:

a) Testbenches can be built in SystemC and SCV [38], but the approach could also be used to generate testbenches in other languages like System Verilog [37]. b) Verification leads the hierarchical refinement steps of the implementation. c) Reuse of testbench components in the verification of modules of hierarchical subdivisions. d) The process of testbench development can be partially automated.

The VeriSC2 methodology uses a basic testbench setup, like reported in Figure A.1. This schema is briefly described here. More details can be found in [12].

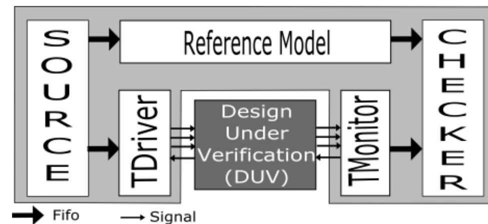

Figura A.1: General Testbench schema

Source: it is responsible for providing transaction level (TL) data to the DUV and to the Reference Model(RM). It is connected to the RM and to the TDriver by FIFOs. Each FIFO is responsible for maintaining the order of the data. There is one FIFO for each input interface.

TDriver: it receives TL data from the Source, transforms them in specified protocol signals and passes these signals along with the required data to the DUV. There is always one TDriver for each input interface to the DUV.

TMonitor: each output interface from the DUV has one TMonitor. It is responsible for receiving the protocol signals from the DUV and transforms them into TL data.

Reference Model (RM): it is the executable specification from the DUV (Golden Model) and can be written in any language. It receives TL data from the Source through FIFO(s) and sends TL data to the Checker through FIFO(s). Any compiled object code that can be linked into C++ can be used as RM. Its important to mention that the cost spent to have a RM is not associated with VeriSC2 methodology, but is part of the cost of a "good"testbench, as defined in the beginning of this section.

Checker: it is responsible for comparing TL data coming from the RM with TL data coming from TMonitor(s) to see if they are equivalent. The checker will automatically compare the outputs from RM and DUV and prints error messages if they are not equivalent.

It is important to mention that this approach has a very new concept added: it proposes the creation of the testbench before the DUV. In traditional methodologies citebrahme2000[30][42][31] and according to VSIA [17], it is not possible to implement this functionality and guarantee the testbench to be tested against errors without a DUV. In VeriSC2 methodology TMonitor(s) and TDriver(s) are used with the RM to replace the functionality of the DUV. This replacement allows the testbench to be simulated before the DUV in RTL is available, without writing any extra code that can not be reused.

The next subsections show the VeriSC2 methodology steps that should be followed by the eTBc tool, in order to generate the testbench templates, more details about the methodology can be found in [11].

A.2.1 The testbench implementation steps

For the reason of the hierarchical nature of VeriSC2 methodology and considering that more complex IP's are divided in one or more levels of hierarchy, it is necessary to create testbenches for each module as well as a testbench for the top-level design.

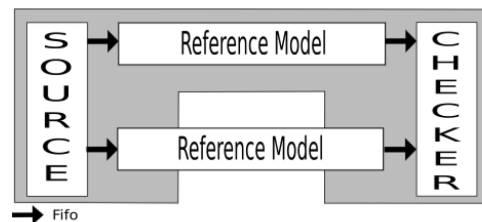

Figura A.2: Testing Source and Checker

The implementation of the testbench is done in small incremental steps, each step resulting in a working simulation that can be easily debugged. These steps consist also in reusing some modules from the testbench in order to replace the functionality of the DUV.

Testbench for the top level DUV (First Step)

Substep 1: The RM should be tested for it's ability to interact with the testbench (receive and produce TL data). A Pre_source, which is a Source that generates input only for one instance of the RM, can be used. Besides, a Pre_sink that is responsible for receiving only the output from the RM, can also be used to make the tests in the RM.

Substep 2: The Source and the Checker should be tested in its ability of create stimuli and catch errors from the simulation, respectively. In this step are generated a Source, Checker and two instances from the RM, in order to be compared by the Checker, as reported in Figure A.2.

Substep 3: Finally, the TDriver and TMonitor should be tested in this substep. This is the most interesting substep, because it shows how the testbench can be simulated independently from the DUV. In order to test TDriver and TMonitor without a DUV, one has to use one instance of the RM to make the same role as a DUV in the simulation. To use the RM some bridge is necessary to connect the TDriver and TMonitor with the RM, because they have different interface data. This bridge can be done using a TMonitor_0 and TDriver_0, which has the same interfaces as TDriver_a and TMonitor_a respectively. They are used to transform the signal data in TL data in the input and TL data in signal data in the output from the RM, as reported in Figure A.3.

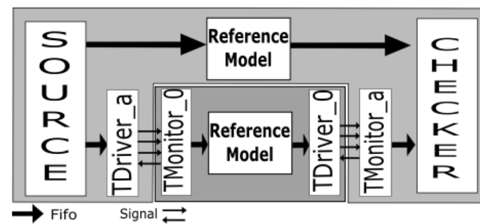

Figura A.3: Testbench construction for the top level DUV

Hierarchical decomposition of the Reference Model (Second Step)

Each module of the RTL design will have exactly one associated testbench. Therefore, the RM will be divided in several parts, corresponding to the DUV hierarchy. In this article, we have used an example composed of two RMs, like reported in Figure A.4.

Each RM should be tested, using the same method reported in Substep 1 from First Step.

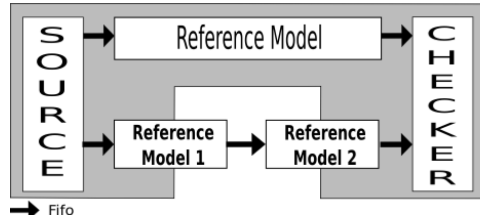

Figura A.4: Hierarchical decomposition of the RM

Testbenches for each module of the DUV: DUV_1 and DUV_2 (Third Step)

This step is identical to the First Step, but the generated testbench here is for each module from the hierarchical DUV, like reported in Figure A.5, where there is a testbench for DUV_1.

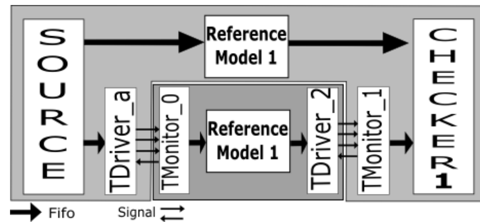

Figura A.5: Testbench implementation for module DUV_1

The implementation of the testbench follows the same rule as the testbench for the top level DUV (Substeps 2 and 3 from the First Step). In this phase some elements are reused. In DUV_1, one reuses the TDriver_a and TMonitor_0 from the complete DUV, because the DUV_1 has the same input interface as the top level DUV. On the other hand, one can reuse the TDriver_2, because it has the same output interface as the input interface from the DUV_2.

Finally, DUV_1 is replaced by the group (TMonitor_a, RM_1, TDriver_2), from Figure A.5. In this phase, testbench for DUV_1 is complete and ready to be used by DUV_1.

In the same way, the DUV_2 can reuse testbench elements from DUV_1 and from the top level DUV. This step should be repeated for all resulting sub-modules of the hierarchical decomposition.

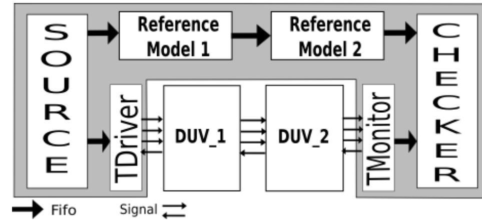

Figura A.6: Replace the top level DUV (Last Step)

Replace the top level DUV (Last Step)

The Last Step consists of the union of all modules of the hierarchy as shown in the Figure A.6. It represents a top-level design with the testbench. With this step one can make the regression tests to see if joining the modules will not introduce some errors. Thus, this is the last step to finish the verification of the top level DUV.

A.3 eTBc Tool

The eTBc(*Easy Testbench Creator*) tool implements all the verification methodology steps. With this tool one can generate testbench elements that can be transformed in actual testbenches by the verification engineer, only defining: modules, I/O interfaces and protocols between modules. This tool speeds up the verification process doing automatic generation of testbench modules: Sources, Checkers, TDrivers, RM, TMonitors, DUV and FIFOs for transactions. eTBc tool also implements connection of all testbench instances and data structures for transactions.

The eTBc tool receives as input a transaction level netlist (TLN) and template patterns. Its output is a template for specified elements from the testbench. The eTBc tool architecture is reported in Figure A.7.

The template patterns are ASCII files that contain the SystemC code common to any template instance of a testbench. It drives the eTBc tool to generate a specific testbench element. For each design project, verification and design engineers need to create one TLN file. In this file are defined all the TL and signal interfaces and how the individual modules of the design are linked together at transaction level to form the top level DUV. An example is given in source code 19.

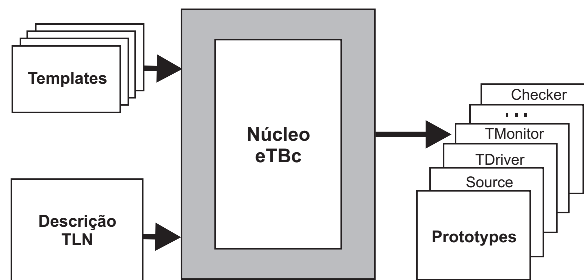

Figura A.7: eTBc tool - architecture diagram

Código 19 : Partial MPEG-4 design TLN written in eTBc language

```
struct coeffs {
     trans {
       short coeff [64];
     }
     signals{
        signed[8] in_pqf;
        bool valid;
        bool ready;
    }
}
module ( ACDCIP ) {
     input   coeffs  acdcip_in;
     output  coeffs  acdcip_out;
}
module ( IQ )   {
      input  coeffs  iq_in;
      output coeffs  iq_out;
}
module ( PARTIAL_MPEG4 )  {
      input   coeffs  p_mpeg4_in;
      output  coeffs  p_mpeg4_out;
      fifo    coeffs  acdcip_iq;
      ACDCIP   acdcip_i(.acdcip_in( p_mpeg4_in ) ,
                        .acdcip_out( acdcip_iq )) ;
      IQ       iq_i    ( .iq_in( acdcip_iq ) ,
                         .iq_out( p_mpeg4_out ));
}
```

A.3.1 eTBc tool implementation

eTBc tool is composed of two interpreters. The first interpreter is used to parse the TLN and generate a data structure that contains all information about transaction data, identifiers, range of signals, names of modules, names of transactions, etc. The second interpreter is used to scan a template pattern, insert information extracted from the TLN and generate a testbench element template. Several invocations of the tool are necessary to generate a complete testbench.

In this article a description of two modules from a MPEG4 IP-core decoder was used to report an example of a TLN file shown in source code 4. The first module is ACDCIP(Inverse Prediction AC/DC) and the second module is IQ(Inverse Quantization), so PARTIAL_MPEG4 was used as name of the top level module which creates an instance of ACDCIP and IQ.

The generated testbench elements are guaranteed to compile and to run without errors or hang-ups. The verification engineer needs to fill in manually implementation specific code into the generated element, for example, SystemC code needed to drive handshake signals.

The SystemC source code 20 shows, as an example, part of a TDriver. This TDriver was generated for the specific ACDCIP interface and contains the names from the variables and is already linked to the Source, **needing only to be filled with the necessary handshake protocol to communicate with the DUV.**

A.4 Results

The results consist of the case study prototype and an analysis of the application of the VeriSC2 methodology to the 3MBIP. The 3MBIP is a Mobile Multimedia Bitstream Processor used in MPEG4 decoder IP-core which is an OCP-IP compliant open-source SystemC-RTL called Terpsicore (referencing the muse of music and dance from the greek mythology) came from an effort of Brazilian government which, through the Ministry of Science and Technology, created a program named CI-Brasil that supports some projects. One of them is the Brazil-IP project[24] in which Terpsicore has been conceived. The 3MBIP receives the compressed video stream in the MPEG4 format and feeds the other blocks with the proper data and/or configuration parameters so that each one is able execute its function. This

Código 20 : Code generated in the eTBc Tool - TDriver SystemC code

```
SC_MODULE ( acdcip_in_Tdriver  ) {
 sc_in <bool>                 clk;
 sc_in <bool>                 reset;
 sc_fifo_in<acdcip_in_str *>  stimuli;
 sc_out<sc_int<8> >           in_pqf;
 sc_out<bool >                valid;
 sc_in<bool >                 ready;
 scv_tr_stream                stream;ConceitosVerificacaoFuncional.tex
 scv_tr_generator<acdcip_in_str,acdcip_in_str> gen;
 acdcip_in_str * tr_ptr;
 void process() {
     while ( ! reset ) wait();
         while(1) {
           tr_ptr = stimuli.read();
           scv_tr_handle h = gen.begin_transaction(*tr_ptr);
           wait();
           gen.end_transaction(h);
           delete tr_ptr;
         }
 }
 SC_CTOR(acdcip_in_Tdriver):stream(name(), "Transactor"),
      gen("acdcip_in_Tdriver", stream)
      {SC_THREAD(process); sensitive << clk.pos();}
};
```

block is a dedicated processor implemented along with Terpsicore to demultiplex multimedia streaming. The 3MBIP features are presented on the tableA.1:

RTL SystemC	1667 lines of source code
RTL Verilog	1783 lines of source code
SystemC Testbench	2607 lines of source code
ROM Size	64K
Assembly ROM size for MPEG-4 decoder	62K
Logic Elements in Altera's Stratix II EP2S60 FPGA	10%

Tabela A.1: 3MBIP Features

Today there is a working FPGA prototype and a layout that was submitted to an exhaustive functional verification process VeriSC2[11]. The chip is ready and under a test process and the layout has 22.7mm^2 at a 0.35μm CMOS 4 ML technology with a 50MHz working frequency.

Based on the experience with the other MPEG4 IP-core modules which were developed with VeriSC1 methodology [12], an increase of 30% productivity was achieved using VeriSC2 methodology [11] on development of the 3MBIP.

Apêndice B

Analysis about VeriSC2 methodology taking in account VSI Alliance pattern report

The VSI Alliance Specification for VC/SoC Functional Verification [17] is a report that improve a pattern definition about Functional Verification concepts. The defined concepts in this document are not mandatory, but if they are used, the definition should be applied. These definitions are taking into consideration as a pattern in this document, to make a comparison between VSI Alliance and adopted concepts in VeriSC2 methodology.

B.1 Functional Verification Deliverables

This section describes the concept of testbench adopted by VSI Alliance and the concept adopted by VeriSC2. The objective from the testbench is the same in both VSIA and VeriSC2 methodology.

The big difference between VeriSC2 and VSIA requirements is that VSIA does not propose anything about the way that the methodology should be followed to be verified and, on the other hand, VeriSC2 methodology is a method aimed to conduct the better integration of functional verification and RTL Design. This VeriSC2 methodology proposes to minimize the overall verification time and find the errors, as soon as possible, when the design begins to be implemented.

In order to perform this approach, VeriSC2 methodology permits the generation of the complete running testbench before the implementation of the DUV even starts. Using this proposition the design can be verified in all necessary phases of its implementation, even at the beginning of the DUV's implementation. Furthermore, VeriSC2 methodology can reuse its own elements to implement the testbench, to perform a self-test and to assure that the testbench contains no errors. On the other hand, as soon as the hierarchical decomposition is verified, RTL implementation can start and done in parallel to the design of the testbenches for the leaf cells of the hierarchy.

B.1.1 Documentation

The documentation of VeriSC2 Methodology follows some of the requirements from VSIA. There is a verification plan that specifies what kind of test case should be applied to the verification. This plan should also specify what kind of functional coverage should be applied in the verification. Although, this document has no specification about assertion methods, code coverage, property checks, standard compliance tests and validation run in a hardware platform.

B.1.2 Testbench

The testbench is responsible to execute the functionalities from the DUV (Design Under Verification) using a module to stimulate. The testbench has also a module to compare the expected and actual behaviour. The figures B.1 and B.2 represents the testbench from VSIA and VeriSC2 respectively.

Descriptions about these Figures could be find respectively in [17] and [11]. Although they have the same conception, there are some differences between the name of the modules and the functionalities. These conceptual differences will be showed in the next subsections.

The format from the testbench used in VeriSC2 methodology is SystemC language and SystemC Verification Library (SCV).

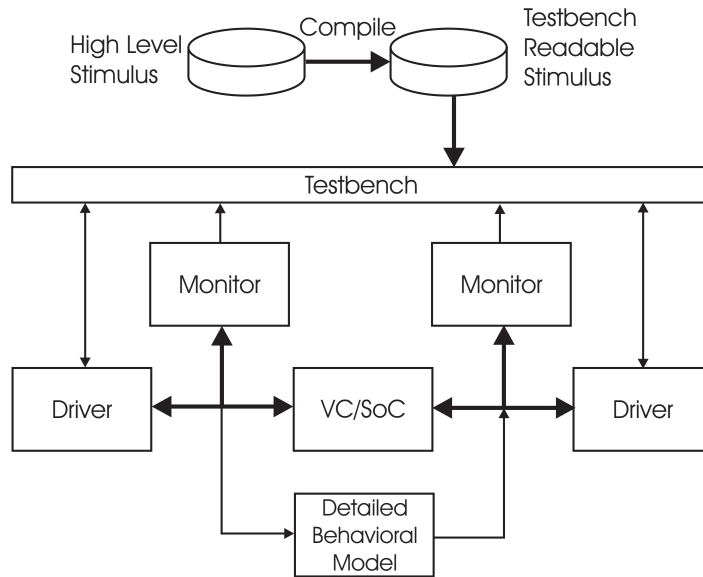

Figura B.1: Testbench to functional verification according VSIA

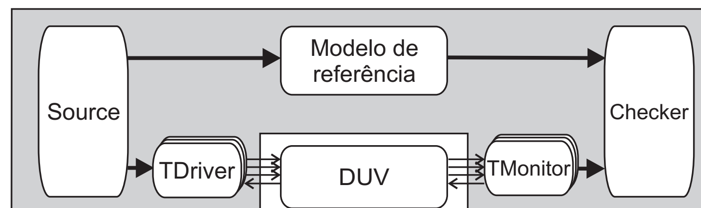

Figura B.2: Testbench to functional verification

B.1.3 Drivers

There are differences between the used concepts of Drivers in VeriSC2 methodology testbench. VeriSC2 methodology uses the concepts of TDriver (Transaction Driver) and TMonitor (Transaction Monitor) to improve the functionalities of the Drivers from VSIA. They play quite the same role as the Drivers from VSIA. The differences are:

- The position that they are used. There are one TDriver for each input interfaces and one TMonitor for each output interfaces from the DUV.
- Both of them have two interfaces: The DUV side connects the TDriver and TMonitor to the DUV; The testbench side which connects the TDriver and TMonitor to the testbench.
- The TDriver and TMonitor operate in different levels of abstraction. The DUV side interface operate at the same abstraction level as the simulation model of the DUV

and the testbench side might operate at a higher level of abstraction (transaction level). Both act as a bridge between these two levels.

- The TDrivers and TMonitors are also responsible to check the interface protocol from the DUV. This is essential to VeriSC2 methodology to assure the absence of errors in the testbench. There are no portability problems, because one can reuse this TDriver and just change the interface protocol.
- The format is SystemC Language.

B.1.4 Monitors

There is no Monitors definition in VeriSC2 methodology testbench. In order to monitoring the DUV, VeriSC2 methodology has the Sniffer concept. The Sniffers are responsible for monitoring the RTL code in the interface from the DUV in each clock cycle, transform it in transactions and record every transaction. There are some rules from VSIA that are not improved by Sniffers from VeriSC2 Methodology:

- There is only one kind of sniffers, which is contradictory with the VSIA rule, which establishes that each interface monitor should be split in environment monitors and simulation-specific-monitors. There is no environment Monitor.
- The Sniffers do not check interfaces errors in the simulation. It could be checked by TDrivers and TMonitors.

B.1.5 Assertions

Assertions are not used in VeriSC2 Methodology. According to VSIA specification, Assertions are no mandatory to Functional Verification, although they add significantly to the quality of the verification.

B.1.6 Functional Coverage

According to VSI Alliance Specification, the coverage models should be organized in two dimensions: Input, output and internal data functional coverage and input, output, internal and input/output temporal functional coverage.

VeriSC2 methodology improves Functional Coverage by means of BVE_COVER library. This library is responsible for measure Functional Coverage in input, output data functional coverage. There are some not covered sides using BVE_COVER:

- The Functional coverage does not measure internal data or temporal functional coverage. These internal data can only be measure in the interfaces from the DUV. Internally VeriSC2 methodology can only use Sniffers like an internal monitor.

The Functional Coverage format for delivery is Multi-dimensional structure of attributes.

VeriSC2 tool provides a functional coverage mechanism (BVE_COVER) that monitors the progress of the verification process. With functional coverage monitoring, one knows at any moment which percentage of a specified full coverage has already been achieved. With the BVE_COVER it is possible to define buckets and the number of samples that are necessary to be reached during simulation. When all buckets reach the specified values, the simulation automatically stops. There is a progress bar that shows the progress of the simulation, based on the implemented buckets.

The BVE_COVER functional coverage is composed by 4 basic tasks:

- Coverage bucket: measure the status and progress of the simulation and report if some specified functionality has not been tested.

- Coverage illegal: reports when some illegal specified functionality occurred.

- Cross-coverage: Cross all specified functionalities and reports which of this crossed functionalities has been tested. The objective is to find hole coverage.

- Hole-coverage: identifies parts of the functional verification that has not been tested for some reason.

B.1.7 Code Coverage

VeriSC2 do not improve Code Coverage. According to VSIA specification, Code Coverage is not mandatory to Functional Verification, but they help to assess the quality of the verification.

B.1.8 Formal Methods

Formal Methods are not used in VeriSC2 Methodology, because it is a Functional Coverage Methodology.

B.1.9 Documentation

VSI Alliance has specifications about some documentation that should be provided. This documentation list is shown below.

The methodology VeriSC2 establishes some VSIA documentation specifications:

- Documentation of a verification plan.
- Documentation of the testbench files and structure used (using svn).
- Documentation of stimulus generator (verification plan).
- Documentation of response checker (verification plan).
- Documentation of all standard test suites used (verification plan).
- Documentation of the directory structure used (using svn).
- Documentation of the environment used for DUV verification (specification of VeriSC2 Methodology).
- Documentation of the Design languages used (SystemC Manual).

The other documentation are not provided by VeriSC2 methodology

- Functional Coverage documentation for the provider verification test suite or other process.
- Documentation of all third-party test suites used to determine compliance to a standard.
- Documentation of the modules (memory or other functions) used.
- Documentation of the bus functional model (BFM) for on-chip bus (OCB) VCs.
- Documentation of all the software drivers used.

- Documentation of the design parameterization options and how they were checked.
- Documentation of the tool and platform configuration used.

The documents format is in Microsoft word or PDF format.

There are Functional Verification Deliverable Documents that should be provided.

Some of these documents are provided by VeriSC2 methodology:

- Documentation of the directory structure used and setup process (using svn).
- A manifest of all the deliverables and a description of the deliverables, file names (using svn).
- Documentation of how to compile the verification environment.
- Documentation of how to use, set up, and debug the testbench.
- Documentation of the verification steps to be performed by the VC integrator.

There is another kind of documentation to the final user that is not provided by VeriSC2 methodology:

- Documentation of how to interpret the results of the deliverable tests.
- Documentation of how certain deliverables could be reused for system-level verification.
- Documentation of any provider tests that cannot be reproduced using the functional verification.

B.1.10 Behavioural Models

VeriSC2 Methodology uses a Reference Model to be a golden model in a simulation. It will allow comparing the output of the system with the waited answer. This Reference Model could be a Behavioural Model, since it is a representation of the function and timing of a Design Under Verification.

B.1.11 Behavioural Models for Memory

This Behavioural Models for Memory could be used in VeriSC2 Methodology.

B.1.12 Detailed Behavioural Models for I/O Pads

This kind of Detailed Behavioural Models for I/O Pads could be used in VeriSC2 Methodology.

B.1.13 Stimulus

The stimulus follows the recommendation of VSI Alliance. The entire stimuli are created in high level of abstraction.

B.1.14 Scripts

There are no rules for using scripts in VeriSC2 Methodology. The regression tests, which depend on these scripts, are implemented using a version control tool (SVN).

B.1.15 Stub Model

There is no specification about Stub Model.

B.1.16 Functional Verification Certificate

There is no specification about Functional Verification Certificate.

B.2 Reuse of Functional Verification Deliverables in SoC Verification

The VSIA imposes some rules to be applied to the provider of the DUV: This rules are necessary to be used for the receiver of the DUV, to verify the DUV again if is necessary. VeriSC2 methodology does not have such rules.

B.2.1 DUV Re-Verification

There is no specification about DUV Re-Verification.

B.2.2 DUV Re-Verification in a SoC

There is no specification about DUV Re-Verification in a SoC.

B.3 Functional Verification Deliverables Rules

B.3.1 Drivers

There are some VSIA established Rules that should be followed for each Driver, or TDriver and TMonitor in our methodology.

The VeriSC2 have most of this rules. There is few rules that are not observed in VeriSC2 methodology and one rule that is established in VeriSC2 methodology but not in VSIA.

The common rules between VSIA and VeriSC2 methodology are shown following:

1. A TDriver and TMonitor must not assign values to an interface signal more than once in the same time step.
2. The DUV must be driven only by self-contained TDrivers and TMonitors.
3. Each TDriver and TMonitor must stimulate and read only one interface.
4. TDrivers and TMonitors must be self-contained.
5. TDrivers and TMonitors must drive all transactions that the interface can perform.
6. TDrivers and TMonitors must have a procedural interface with input and output arguments.
7. Global signals must not be used to configure TDrivers and TMonitors.
8. TDrivers and TMonitors should be partitioned for granularity of control.
9. Clocks should be used only to sample or product data synchronously with a clock.

The rules that are not implemented by VeriSC2 methodology are:

1. TDrivers and TMonitors must not check the interface protocol.

2. Inputs must be driven with legal values only for the duration that they are valid.

There is one more rule without established in VLSI, but used in VeriSC2 methodology:

1. TMonitors should verify the stability of the data during the time the handshake signal valid is active.

B.3.2 Monitors

VeriSC2 methodology does not have the concept of Monitors from VSIA. These Monitors could be partially replaced by the Sniffers from VeriSC2. Some rules from VSIA are not applied to Sniffers and some are applied.

Applied rules to Sniffers from VeriSC2:

- Sniffers must monitor only one interface.
- Sniffers must not drive design inputs.
- Sniffers must check or observe all transactions on the interface.
- Sniffers must be self-contained.
- Sniffers must continuously monitor the interface.
- Sniffers must not determine if a transaction should be in progress on an interface.
- Sniffers must only sample signals that will be preserved after synthesis.
- Sniffers must be reusable by all DUVs that connect to the interface.
- Sniffers output should be kept to a minimum in the default configuration.

Not applied rules to Sniffers from VeriSC2:

- The interface Sniffers should be split into two types: environment monitors and simulation-specific Sniffers.

- Sniffers must verify the protocol on the external interface of a DUV.
- Unrecognized interface behaviour must be flagged as an error.
- Sniffers must be capable of being established and disabled.
- Sniffers should provide abstractions of interface activity.

B.3.3 Assertions

There are no assertions in the VeriSC2 methodology.

B.3.4 Functional Coverage

Input Data Functional Coverage

VSI Alliance has a rule that imposes that the measurements of input functional coverage must be documented in a manner that permits the DUV integrator to assess the quality of the standalone DUV verification.

VeriSC2 methodology can perform this rule measuring the functional coverage in the Source and performing Cross_coverage.

Output Data Functional Coverage

VSI Alliance has a rule that imposes that the measurements of output functional coverage must be documented in a manner that permits the DUV integrator to assess the quality of the standalone DUV verification.

VeriSC2 methodology can perform this rule measuring the functional coverage in the Checker and performing Cross_coverage.

Internal Data Functional Coverage

VSI Alliance has a rule that imposes that the measurements of internal functional coverage must be documented in a manner that permits the DUV integrator to assess the quality of the standalone DUV verification.

VeriSC2 methodology does not measure internal data functional coverage.

Input Temporal Functional Coverage

VeriSC2 methodology does not measure input temporal functional coverage.

Output Temporal Functional Coverage

VeriSC2 methodology does not measure output temporal functional coverage.

Internal Temporal Functional Coverage

VeriSC2 methodology does not measure internal temporal functional coverage.

Input/Output Temporal Functional Coverage

VeriSC2 methodology does not measure input/output temporal functional coverage.

B.3.5 Code Coverage

There is no code coverage in the VeriSC2 methodology.

B.3.6 Formal Methods

There are no formal methods in the VeriSC2 methodology.

B.3.7 Documentation

We do not have pattern documentation for the Functional Verification.

B.3.8 Behavioural Models

The only rule proposed by VSIA is followed by the VeriSC2 methodology:

- The detailed behavioural model must be configured independently of the DUV.

B.3.9 Scripts

We do not have rules for the scripts.

B.3.10 Stub Model

We do not have rules for the stub model.

B.3.11 Functional Verification Certificate

We do not have rules for the Functional Verification Certificate.