Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Geração semi-automática de *Testbenches* para Circuitos Integrados Digitais

Isaac Maia Pessoa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Elmar Uwe Kurt Melcher
(Orientador)

Campina Grande, Paraíba, Brasil

P475g Pessoa, Isaac Maia
Geracao semi-automatica de Testbenches para circuitos integrados digitais / Isaac Maia Pessoa.- Campina Grande, 2007.
57 f. : il.

Dissertacao (Mestrado em Computacao e Ciencias do Dominio da Engenharia Eletrica) - Universidade Federal de Campina Grande, Centro de Engenharia Eletrica e Informatica.

1. Circuitos Integrados 2. Testbenches 3. EDA 4 - VLSI 5 - I. Universidade Federal de Campina Grande - Campina Grande (PB) II. Título

CDU 621.315.616(043)

"GERAÇÃO SEMI-AUTOMÁTICA DE *TESTBENCHES* PARA CIRCUITOS INTEGRADOS DIGITAIS"

ISAAC MAIA PESSOA

DISSERTAÇÃO APROVADA EM 10.04.2007

PROF. ELMAR UWE KURT MELCHER, Dr.
Orientador

PROFª JOSEANA MACÊDO FECHINE , D.Sc
Examinadora

PROF. JOSÉ ANTONIO GOMES DE LIMA, D.Sc
Examinador

CAMPINA GRANDE – PB

Resumo

A complexidade da verificação funcional tende a crescer exponencialmente com relação ao tamanho do hardware a ser verificado. O contínuo avanço da complexidade de circuitos integrados está seguindo a lei de Moore e tem criado uma grande pressão no engenheiro de verificação para que este continue certo de que não existem falhas funcionais ao final da fase de verificação.

O tempo e dinheiro necessários neste processo aumentam ainda mais a pressão, pois o processo de verificação consome a maior parte dos recursos em um projeto de hardware.

Assim, uma abordagem que possua uma ferramenta flexível e que consiga auxiliar o engenheiro de verificação em suas tarefas pode ser de grande utilidade.

A metodologia de verificação VeriSC pode ajudar a resolver problemas envolvidos na verificação funcional.

O objetivo deste trabalho é o desenvolvimento de uma ferramenta de suporte à metodologia VeriSC que seja útil na automatização do processo de construção de ambientes de simulação (*testbenches*) e desta forma consiga aumentar, através de um mecanismo flexível, a velocidade em que as tarefas de verificação são executadas.

Abstract

Functional verification complexity tends to increase exponentially with design size. The Moore's law places an ever growing demand on today's verification engineer to continue to ensure that no bug is missed in the verification process.

The time necessary and money spent on the verification process increases the demand because it consumes most of the resources of a hardware project. Thus, an approach that has a flexible tool and helps the verification engineer in his tasks can be very useful in the verification process.

The verification methodology VeriSC can help to solve several problems involving funcional verification.

This work's objective is a supporting tool for VeriSC methodology useful for automated construction of simulation environments (*Testbenches*) enabling a flexible way to speed up verification tasks.

Agradecimentos

Inicialmente agradeço aos meus pais, José Nildo (em memória) e Maria de Fátima, os criadores da criatura que aqui apresenta este trabalho.

Em especial, agradeço a uma pessoa que consigo admirar desde sempre: Elmar Melcher, mais conhecido entre seus alunos como Alemão.

Aos eternos membros do cafa clube: George, Henrique, Matheus, Osman ,Wilson e Zurita.

Ao ar feminino do LAD[5]: Daniella, Joseana, Karina e Karinina.

Sem esquecer ainda, agradeço à ajuda extra da professora Joseana Fechine e ao professor Roberto Faria.

Sem deixar de lado os botecos de Campina Grande, lugares de grande contribuição para desenvolvimento deste trabalho.

Conteúdo

1 Introdução **1**

1.1 Objetivos 2

1.1.1 Objetivos Específicos 2

1.2 Estrutura da Dissertação 3

2 Fundamentação Teórica **4**

2.1 Conceitos Básicos 4

2.2 Verificação 4

2.3 Verificação Funcional 5

2.3.1 Linguagens de Descrição de Hardware 6

2.3.2 *Transaction Level Netlist* (TLN) 6

2.3.3 *Register Transfer Level* (RTL) 7

2.3.4 Metodologia de Verificação Funcional 7

2.3.5 O *Testbench* 9

2.3.6 A Metodologia de Verificação Funcinal VeriSC 12

3 A Ferramenta eTBc **14**

3.1 Funcionamento Geral da Ferramenta eTBc 14

3.2 A linguagem *eTBc-Design-Language* (eDL) 15

3.3 A linguagem *eTBc Template Language* (eTL) 19

3.4 Trabalhos Relacionados 25

4 Exemplos de Funcionamento da Ferramenta eTBc **27**

4.1 Exemplos de Geração de Protótipos 31

5 Resultados e Sugestões para Trabalhos Futuros **41**

5.1 Resultados 41

5.2 Sugestões para Trabalhos Futuros 42
5.3 Considerações Finais 43

A Gramáticas usadas no eTBc **46**

B *A Methodology aimed at Better Integration of Functional Verification and RTL Design* **52**

Lista de Símbolos

ASIC - *Application Specific Integrated Circuit*

DUV - *Design Under Verification*

EDA - *Electronic Design Automation*

eDL - *eTBc Design Language*

eTL - *eTBc Template Language*

FIFO - *First-in First-out*

FPGA - *Field Programmable Gate Array*

HDL - *Hardware Description Language*

IC - *Integrated Circuit*

IP-core - *Intellectual Property of Hardware Project*

MPEG - *Motion Pictures Experts Group*

OSCI - *Open SystemC Initiative*

RTL - *Register Transfer Level*

SCV - *SystemC Verification Library*

SoC - *System on Chip*

TLD - *Transaction Level Data*

TLM - *Transaction Level Model*

TLN - *Transaction Level Netlist*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very-High-Speed Integrated Circuit*

Lista de Figuras

2.1 Características do Projeto 8
2.2 Representação de um *testbench* na metodologia VeriSC 10

3.1 Representação arquitetural da ferramenta eTBc 15
3.2 Representação do *IP-core* yuv2rgb 17

4.1 Representação do DPCM 27

Lista de Tabelas

3.1 Palavras reservadas da linguagem eDL 15
3.2 Comandos de controle da linguagem eTL 20
3.3 Comandos de substituição na linguagem eTL 21
3.4 Listas em eTL 22

Lista de Códigos Fonte

3.1 TLN para representação do *IP-core* yuv2rgb em eDL 17
3.2 Exemplo do uso de eTL para criar um arquivo 23
3.3 Código resultante do molde 3.2 23
3.4 Exemplo de laço 23
3.5 Arquivo exemplo_foreach.txt 23
3.6 Exemplo mais completo 24
3.7 Arquivo exemploMaior.txt 24
4.1 Arquivo dpcm.design, TLN para o DPCM escrito na linguagem eDL 28
4.2 Sintaxe do eTBc 31
4.3 Sintaxe do eTBc para a geração do Source 31
4.4 Arquivo source.h gerado pelo eTBc 32
4.5 Arquivo de molde para o source.h 33
4.6 Arquivo structs.h gerado pelo eTBc 36
4.7 Arquivo de molde sc_structs 37
4.8 Arquivo dpcm.v gerado pelo eTBc 39
4.9 Arquivo de molde v_duv 39
A.1 Gramática da Linguagem eDL 46
A.2 Gramática da Linguagem eTL 48

Capítulo 1

Introdução

A criação de um circuito integrado (CI) envolve muitas etapas. Cada etapa requer tempo e dinheiro do projeto. O fluxo inteiro requer muita atenção em cada um estágio de cada etapa pois um erro detectado no início é mais barato de ser corrigido a um detectado no final do fluxo. Para o mercado, um CI que apresente falhas funcionais é inaceitável.

A tentativa de detectar erros funcionais do CI durante o início do fluxo é o maior desafio no projeto, visto que um erro no final pode comprometer o *time-to-market* do produto.

Algumas técnicas são usadas para esse propósito. A maior parte dessas técnicas detém uma grande fatia de tempo do projeto. Essas técnicas são chamadas de processo ou metodologia de verificação funcional.

A metodologia de verificação funcional tem o objetivo de detectar erros funcionais no projeto lógico do CI, chamado de *IP-core*. Esse processo pode estar presente desde o inicio do fluxo até o sua concretização e portanto é o processo mais demorado e mais caro do projeto.

A maior parte do tempo de um projeto de hardware, cerca de 70% , é gasto na verificação funcional [12].

Uma metodologia que consiga minimizar esse tempo é sempre interessante do ponto de vista de recursos humanos a serem economizados. Desta forma, ela deve dispor de recursos tais como ferramentas de apoio ao seu fluxo de trabalho.

A idéia conceitual da metodologia de verificação funcional, juntamente com as ferramentas de apoio a essa metodologia, auxiliam o projeto a atingir os seus objetivos com relação a tempo, recursos e requisitos.

O objetivo desse trabalho é atuar na área de ferramentas computacionais de apoio ao processo de verificação funcional VeriSC [22].

1.1 Objetivos

O objetivo principal do trabalho aqui apresentado é a criação de uma ferramenta de apoio ao processo de verificação funcional VeriSC [22] que possa ser útil no desenvolvimento de projetos de hardware e que contribua na melhoria do processo de desenvolvimento de *IP-cores*.
A ferramenta tem também como objetivo a geração automática de protótipos de cada um dos elementos de um *testbench* além das ligações entre os mesmos.
Os elementos que a ferramenta se dispõe a gerar são: *Source*, *TDriver*, *Checker* e *TMonitor*, que serão discutidos posteriormente neste documento. Além disso, poderão também ser geradas variações de cada um desses elementos. Essas variações dependem da metodologia de verificação funcional empregada. Neste caso a abordagem desse trabalho está relacionada com a metodologia VeriSC que tem como foco a geração da maior parte desses elementos na linguagem SystemC [23] e Verilog [20].
A ferramenta faz uso de moldes para a geração de cada protótipo individualmente. Desta forma, os moldes possuem flexibilidade para a construção de protótipos em outras linguagens de hardware.
Após a geração de todos os protótipos de cada um dos elementos necessários a um *testbench*, o engenheiro de verificação funcional deverá ser capaz de compilá-los e rodá-los sem erros estruturais.
O nome da ferramenta deste trabalho é *Easy Testbench Creator* (eTBc).

1.1.1 Objetivos Específicos

Os objetivos específicos do trabalho estão relacionados ao processo de construção da ferramenta eTBc.

- Definição da linguagem a ser usada pelo usuário da ferramenta. Essa linguagem chama-se *eTBc Design Language* (eDL) e tem como objetivo a modelagem de um *IP-core* em nível de transação.
- Definição da linguagem a ser usada nos moldes (*templates*). Essa linguagem chama-se *eTBc Template Language* (eTL).
- Criação das gramáticas referentes às duas linguagens anteriores.
- Implementação dos analisadores léxico, sintático e semântico referentes às linguagens eDL e eTL;
- Implementação do gerador de código. Esse módulo tem como função a geração de código de protótipos.

- Elaborar testes de unidade e de aceitação para usar como auxílio no desenvolvimento da ferramenta.

1.2 Estrutura da Dissertação

O resto deste trabalho está estruturado da seguinte maneira :

- **Capítulo 2:** Este capítulo está direcionado aos fundamentos teóriocos e conceitos necessários ao leitor para que o mesmo consiga ler e entender o restante do trabalho.
- **Capítulo 3:** Neste capítulo abordados os detalhes internos da ferramenta eTBc e alguns trabalhos relacionados.
- **Capítulo 4:** Exemplos do uso e funcionamento da ferramenta eTBc.
- **Capítulo 5:** Apresentação dos resultados, trabalhos futuros e considerações finais.

Capítulo 2

Fundamentação Teórica

2.1 Conceitos Básicos

Um *Intellectual Property core* (*IP-core*) [13][12] pode ser visto como a lógica ou os dados necessários para construir um dado projeto de hardware. Idealmente ele é "reusável" e pode ser adaptado a vários tipos de dispositivos de hardware. Pode-se entender o *IP-core* como a implementação de um dado projeto de hardware em uma linguagem específica para esse objetivo.

Um *System-On-Chip* (SoC) [13] é um circuito integrado composto por vários *IP-cores*.

Uma simulação no contexto de projetos de hardware é representada por um conjunto de cenários nos quais o *IP-core* deve ser submetido. A simulação então é satisfeita se o dado *IP-core* consegue se comportar da forma como foi especificado no projeto.

A verificação funcional é o processo usado para verificar que um dado *IP-core* obedece aos seus requisitos funcionais.

Um *Design Under Verification* (DUV) pode ser entendido como a parte ou módulo específico do *IP-core*, que está em fase de verificação funcional. Um DUV também pode ser visto como todo o *IP-core*. É uma questão apenas de referência.

2.2 Verificação

Verificação é o processo usado para mostrar que um dado sistema obedece a sua especificação. Existem alguns tipos de verificação [13]:

- Estática ou formal

- Dinâmica ou funcional
- Híbrida

Os três tipos de verificação propõem "verificar" se um dado modelo ou sistema está equivalente ao outro. Tipicamente, os dois modelos (formal e funcional) possuem níveis de abstração distintas.
A verificação formal [13] no contexto de hardware está relacionada à prova ou refutação da corretude de um certo sistema. A fundamentação do sistema é feita com respeito a uma certa especificação formal ou propriedades que podem ser definidas através de métodos formais ou matematicamente.
A verificação dinâmica ou funcional também é usada para provar a corretude de sistemas com relação à especificação, porém a diferença é que a especificação para este tipo de verificação não precisa ser formal. Além disso, este tipo de abordagem faz o uso de simulações para mostrar que o modelo está de acordo com as especificações.
Como já foi mencionado, o *Design Under Verification*(DUV) é a implementação em alguma linguagem de hardware de uma dada funcionalidade específica do *IP-core*.
O processo de verificação funcional mostra através de simulação que um dado DUV está de acordo com a especificação. Esse método é vantajoso porque não sofre limitações em termos de complexidade do *IP core* a ser verificado. Por outro lado, apenas a simulação não é o bastante para mostrar que um *IP-core* está livre de erros. Por isso, se faz necessário o uso de cobertura funcional que proporcionará um limiar ou uma meta a ser atingida na verificação.
Para as verificações formal e funcional, é possível fazer a prova da presença de erros, porém não é possível provar a ausência de erros.
A verificação híbrida é a união da verificação formal com a verificação funcional. Isso quer dizer que em alguns casos se faz o uso da verificação funcional em para outros casos se faz o uso da verificação formal.
Neste trabalho será apenas usado o conceito de verificação funcional.

2.3 Verificação Funcional

Em projetos de *IP-core* ou SoC um processo bastante usado e que absorve uma das grandes dificuldades enfrentadas é o de verificação funcional que consiste em certificar se os requisitos estão sendo obedecidos[12][21]. Tais requisitos podem ser estabelecidos com relação a funcionalidades que devem ser cobertas de acordo com as necessidades da aplicação.
Este trabalho está voltado para o processo de verificação funcional.

2.3.1 Linguagens de Descrição de Hardware

A sigla que representa as linguagens de hardware é HDL que em inglês significa *Hardware Description Language*. As HDL são usadas para criar modelos de dispositivos de hardware.
A diferença fundamental entre uma linguagem de hardware e uma linguagem convencional de software é que o código não é linear, ou seja, um comando não é executado após o outro comando. Não há uma seqüência temporal de execução das instruções. Para modelar o comportamento de um dispositivo eletrônico, a linguagem de hardware precisa ter um mecanismo que consiga simular paralelismo e tempo. As principais linguagens para descrição de hardware são: Verilog [20], VHDL [10], SystemC [23] e SystemVerilog [3]. Além dessas linguagens, existem muitas outras com diferentes sintaxes, porém com a mesma idéia fundamental de tentar modelar um sistema de hardware.

2.3.2 *Transaction Level Netlist* (TLN)

É bastante comum se enxergar um *IP-core* como um conjunto de módulos dispostos de tal forma que cada um é visto como uma caixa preta. Sendo assim, uma das tarefas consiste em saber quais são as estruturas de dados usadas nas entradas e saídas de cada um dos módulos a serem verificados.
Estas estruturas de dados que trafegam entre os módulos, são chamadas de transações.
Desta forma define-se neste trabalho o conceito de *Transaction Level Netlist* (TLN) que são as descrições dos dados em nível de transação. Isso implica dizer que quando alguém está usando tal conceito para projetar o *testbench*, na verdade está pensando nos módulos do sistema e a comunicação entre os mesmos. Essa comunicação é vista em nível de dados que trafegam em forma de transações entre tais módulos. Esse nível de abstração será usado de agora em diante para o restante do trabalho e será referenciado pela sigla TLN. Dentro de uma *Transaction Level Netlist*(TLN) deverão existir definições de:

- **Módulos:** São entidades hierárquicas e funcionais que definem o sistema como um conjunto caixas pretas, nas quais se observam apenas entradas e saídas.
- **Transações:** São os dados que trafegam entre os módulos. As operações que ocorrem nas interfaces de E/S[13]. Uma transação pode por exemplo ser representada por um pacote ethernet.
- **Ligações entre módulos:** São as conexões existentes entre módulos.

2.3.3 *Register Transfer Level* (RTL)

Há uma outra forma de enxergar um *IP-core*. Essa outra forma é chamada RTL [19]. Diferente de TLN, RTL é um termo bastante usado no contexto de *IP-cores* e não é uma definição que está apenas no escopo do trabalho aqui descrito. No ponto de vista de abstração, RTL é considerado um nível mais baixo que TLN.

RTL é de uma forma de descrever o *IP-core* através do uso de registradores. Ou seja, o *IP-core* é visto através do fluxo(transferência) de dados entre os seus registradores.

Para que um *IP-core* se transforme em um dispositivo real de hardware se faz necessário a sua implementação em nível RTL através de alguma linguagem de descrição de hardware (HDL).

O nível RTL é muitas vezes usado no processo chamado de síntese lógica [19] que consiste em um outro nível de abstração em que todo o modelo RTL é transformado em um outro modelo em nível de portas lógicas.

Na verificação funcional, usam-se os dois níveis de descrição: RTL e *Transaction Level Modelling* (TLM). O TLM [15] pode ser representado por um *testbench*, pois esse não possui descrições em nível de registradores, mas apenas em nível de transações.

2.3.4 Metodologia de Verificação Funcional

A metodologia de verificação funcional é o processo que rege as regras de todas as etapas do processo de verificação funcional.

Existem três aspectos importantes a serem observados para a realização da metodologia de verificação funcional, que são: especificação, intenção do projeto e implementação [21]. Esses aspectos são apresentados na Figura 2.1.

A visão que corresponde à intenção do projeto é tudo aquilo que o cliente imagina antes mesmo de sua especificação. A visão que corresponde à especificação é aquela em que o projetista estando consciente da intenção do projeto, consegue elaborar toda a análise e documentação de maneira a dispor a uma especificação funcional do sistema a ser implementado. A última visão é a da implementação, que corresponde ao que foi implementado daquilo que foi especificado. O conjunto D representa o comportamento não pretendido, não especificado e não implementado.

O ideal seria que os conjuntos intenção do projeto, especificação e implementação fossem iguais. Desta forma, são definidos os seguintes subconjuntos :

- **Subconjunto E:** É a parte da intenção e da especificação do projeto que não foi implementada;
- **Subconjunto F:** É tudo aquilo que foi especificado e implementado, mas não era a intenção

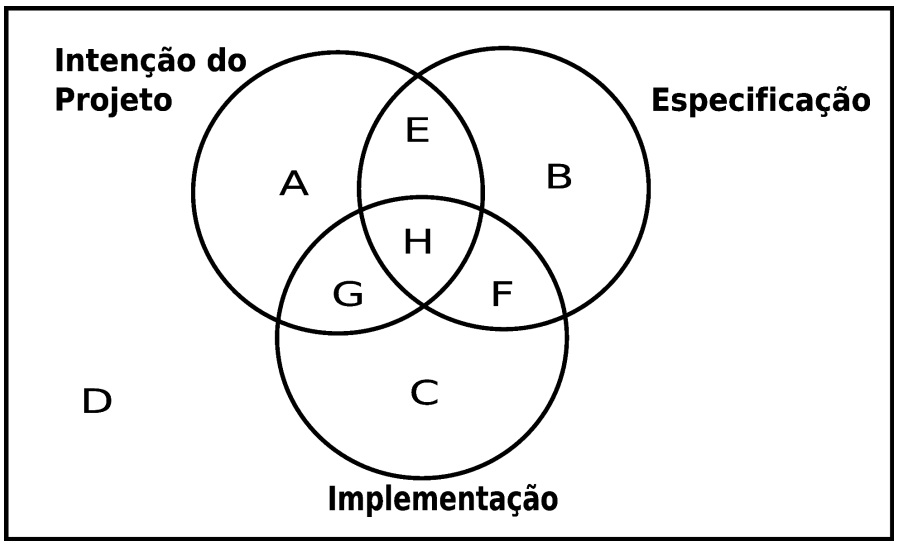

Figura 2.1: Características do Projeto

do projeto;

- **Subconjunto G:** Representa a parte que é intenção do projeto e que foi implementada, porém não fazia parte da especificação;
- **Subconjunto H:** É tudo aquilo que foi implementado e que faz parte da especificação e da intenção do projeto.

Existe um esforço para maximizar os subconjuntos **H** e **F** e minimizar os subconjuntos **B** e **C**. A idéia de verificação funcional é fazer com que casos nas áreas **A** e **B** sejam apontados provocando a redução das mesmas e como consequência realizando o benefício para as áreas **H** e **F**.

A verificação funcional é a fase mais importante de projetos de hardware, pois se algum erro não for detectado nesta fase então o mesmo será repassado mais adiante no processo de criação do circuito integrado e posteriormente na fabricação, de forma a tornar a sua comercialização inviável.

Estima-se que cerca de 70% do projeto de um hardware é gasto na verificação funcional sendo essa a fase mais importante em termos de dinheiro, recursos humanos e tempo [21][12].

Desta forma, o *time-to-market* de um projeto de hardware também está muito relacionado ao processo de verificação funcional, pois este é o responsável por deter a maior parte de tempo no processo de desenvolvimento de um projeto de hardware[24].

A necessidade de uma metodologia de verificação funcional que consiga otimizar o tempo de desenvolvimento de um projeto de hardware torna-se essencial[22].

Outro ponto importante a se observar são os possíveis cenários para um dado projeto, tais como, quais são todas as possíveis combinações de entradas para um dado *IP-core*.

Porém, gerar todos os possíveis cenários tem custo inviável para módulos maiores da mesma forma que para testar muitas unidades menores também é inviável visto que pode levar um tempo muito grande.
Assim, é necessário fazer escolhas tais como :

- Verificar situações mencionadas na especificação;
- Verificar situações extremas;
- Utilizar estímulos reais;
- Criar situações aleatórias.

Nesse contexo, situações aleatórias são especialmente importantes porque são capazes de gerar cenários que seriam esquecidos.

2.3.5 O *Testbench*

As simulações em projetos de hardware são concretizadas ou implementadas através de elementos chamados *testbenches*. Um *testbench* é um artefato escrito em linguagem formal, usado para criar simulações para o modelo do *Design Under Verification*(DUV) que é representado em alguma linguagem de descrição de hardware. O *testbench* cria estímulos que consiguem ativar as funcionalidades desejadas no DUV. Por exemplo, se um DUV tem uma funcionalidade de fazer soma de números inteiros, um estímulo para ele pode ser representado por dois números inteiros. O *testbench* é a "montagem" em volta do DUV que confrontará a sua funcionalidade com o modelo de referência que pode ser até mesmo um software escrito em alguma linguagem de alto nível como C, C++, Java, Python, etc.
O modelo de referência representa as funcionalidades que foram especificadas no projeto. Esse modelo pode ser visto como um projeto de software já testado funcionalmente que está sendo usando neste caso para ser confrontado com o modelo de hardware.
As características desejáveis de um *testbench* são [12]:

- Escrito em alguma linguagem de verificação de hardware;
- Não possui entradas nem saídas;
- Um modelo do universo em volta do projeto;
- Cria estímulos e verifica a resposta;

- Imprime mensagens e cria histórico(*log*) quando o DUV apresenta comportamento inesperado;
- Baseado em Transações. É modelado através do conceito de TLM [14].
- Auto-checagem;
- Dirigido por cobertura funcional;
- Uso de estímulos aleatórios;

Na Figura 2.2 é apresentada uma representação de um *testbench* da metodologia VeriSC[22][16].

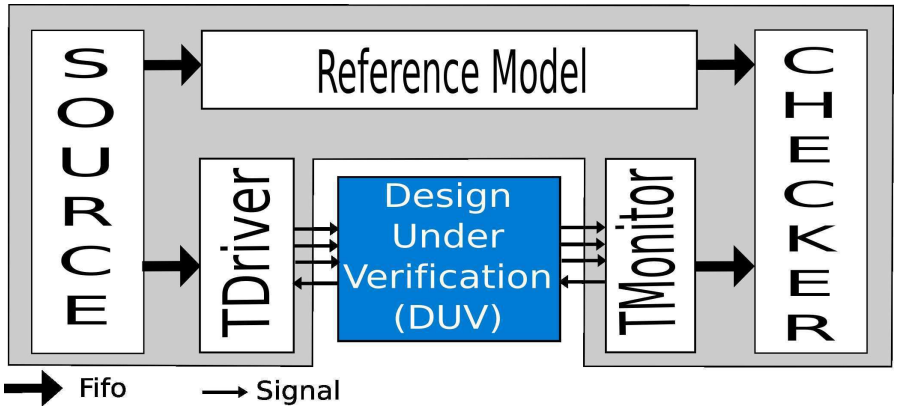

Figura 2.2: Representação de um *testbench* na metodologia VeriSC

O *testbench* é a parte cinza claro da figura em forma de U invertido que envolve os seguintes elementos : *Source*, *TDriver*, *TMonitor*, *Checker* e modelo de referência (*Reference Model*).
O *testbench* está em nível de transações, pois é um modelo feito através de TLM [15]. O DUV não é um modelo em nível de transações, pois está em nível RTL [19] e possui sinais de comunicação. Então, são necessários dois elementos para realizar a conversão entre os dois níveis : TDriver e TMonitor. O TDriver faz a conversão de transações em TLM para sinais em nível RTL. O TMonitor realiza a conversão inversa que vai de sinais novamente para transações. Os sinais na Figura 2.2 são representados pelas setas mais finas.

Os elemtos apresentados na Figura2.2 são:

- **FIFO:** Alguns elementos do *testbench* são ligados por FIFOs, representadas na figura por setas largas. As FIFOs exercem uma tarefa muito importante no *testbench*, pois são responsáveis por controlar o seqüenciamento e o sincronismo dos dados das transações. O *testbench* é implementado em nível de transações. Essa modelagem é chamada de *Transaction Level Modeling* (TLM)[15].

- **DUV:** O *Design Under Verification* (DUV) é o projeto que está sendo verificado. Este deve ser implementado no nível RTL [19]. Por isso, se faz necessário algum mecanismo para a comunicação com o *testbench* que está em TLM. Essa comunicação se faz com a ajuda do *TDriver* e *TMonitor* que traduzem o nível dos sinais que estão em RTL para transações e vice-versa.

- **Source:** O *Source* é responsável por criar estímulos para a simulação. Esses estímulos devem ser cuidadosamente escolhidos para satisfazer aos critérios de cobertura especificados. Todos os estímulos criados pelo *Source* são transações. O *Source* envia então, as transações diretamente para o modelo de referência (na figura representado pelo retângulo *Reference Model*) e *TDriver*. Os estímulos são enviados através das FIFOs, como é mostrado na Figura 2.2, e devem exercitar todas as funcionalidades especificadas, para saber se as respostas do DUV estão corretas. Os estímulos inseridos no *testbench* podem ser ajustados de acordo com a cobertura funcional medida durante a simulação.

- **TDriver:** O *testbench* possui um *TDriver* para cada interface de entrada do DUV (as interfaces de entrada são as comunicações por onde o DUV recebe dados). O *TDriver* é responsável por converter as transações, recebidas pelo *Source*, em sinais e submetê-los para o DUV. O *TDriver* também executa o protocolo de comunicação(*handshake*) com o DUV.

- **TMonitor:** O *testbench* possui um *TMonitor* para cada interface de saída do DUV (as interface de saída são as comunicações por onde o DUV envia dados). O *TMonitor* executa o papel inverso do *TDriver*. Este converte todos os sinais de saída do DUV para transações e os repassa para o *Checker*, via FIFO. Além disso, o *TDriver* também executa um protocolo de comunicação com o DUV, através de sinais.

- **Checker:** É o responsável por comparar as respostas resultantes do DUV e do modelo de referência, para saber se são equivalentes. O *Checker* compara esses dados automaticamente e, no caso de encontrar erros, emite uma mensagem mostrando quando aconteceu o erro. Como a metodologia adotada neste trabalho é *black-box*, isto significa que ela não permite a visão dos componentes internos do DUV. A cobertura deve indicar a todo instante o quanto dos critérios especificados já foram alcançados e deve parar a simulação quando forem alcançados todos os critérios especificados.

- **Reference Model:** O modelo de referência (*Reference Model*) é, por definição, a implementação ideal (especificada) do sistema. Por isso, ao receber estímulos, esse modelo deve produzir

respostas corretas. Para efeito de comparação dos dados, após a conclusão das operações, o modelo deve passar os dados de saída para o módulo *Checker* via FIFO(s). O modelo de referência pode ser implementado em software em alguma linguagem de alto nível.

2.3.6 A Metodologia de Verificação Funcinal VeriSC

A metodologia de verificação com a qual este trabalho está relacionado é chamada VeriSC [22][16] e é direcionada para a verificação funcional de *IP-cores* de propósito geral, que sejam síncronos e utilizem um único sinal de relógio. Essa metodologia propõe a criação do *testbench* antes da implementação do DUV. Dessa forma, o engenheiro de projeto tem acesso a um ambiente de simulação antes de começar a implementação de um DUV específico, podendo saber no decorrer da implementação se a implementação do DUV que está sendo realizada está correta ou não.

O engenheiro de verificação deve ser o responsável por criar o *testbench* e o engenheiro de projeto deve implementar o DUV. Os dois engenheiros devem decidir o nível de granularidade do projeto e criar uma especificação contendo informações necessárias para a criação do *testbench*. Esses devem ser implementados antes ou em paralelo ao projeto do DUV.

Essa metodologia de verificação funcional foi usada na construção de um *IP-core* desenvolvido no Laboratório de Arquiteturas Dedicadas(LAD) [5] da Universidade Federal de Campina Grande (UFCG) com recursos do projeto Brazil-IP [7]. Trata-se de um *IP-core* para decodificação de vídeo no formato MPEG-4 [11]. A mesma metodologia de verificação foi usada também por outros parceiros do projeto Brazil-IP, na UFPE, Unicamp, USP, UnB, UFMG.

O *IP-core* MPEG-4 citado acima, foi concluído e tem as seguintes características:

- 120.000 linhas de código distribuídos em *testbenches*, código RTL e scripts.
- Ocupação de aproximadamente 35.000 elementos lógicos de uma FPGA da série Stratix II da Altera.
- Desenvolvido usando a metodologia VeriSC [22].
- 75% do desenvolvimento gasto em verificação funcional.
- 25% do desenvolvimento gasto em desenvolvimento do RTL.
- Layout de IC(Integrated Circuit) com cerca de 50.000 portas lógicas.
- Silício com área de aproximadamente 22,6mm^2

Hoje, o MPEG-4 representa o maior e mais complexo *IP-core* já desenvolvido por uma universidade Brasileira e teve sua primeira validação depois de um período de dois anos e meio.
Outro projeto também desenvolvido através da metodologia é o *DigiSeal* que tem como objetivo fazer a detecção de violação em dispositivos de medição de energia instalados em postes de iluminação pública.
O projeto foi feito em parceria com as seguintes entidades: Companhia de Energia Elétrica do Rio de Janeiro (LIGHT), Centro de Estudos e Sistemas Avançados do Recife (CESAR), Universidade Federal de Pernambuco (UFPE) e Universidade Federal de Campina Grande (UFCG). O DigiSeal possui as seguintes características :

- Protótipado em um dispositivo FPGA usando 6.176 elementos lógicos;
- Aproximadamente 10.000 linhas de código RTL e *testbench*;
- Silício com área de aproximadamente 2,4mm^2

A metodologia VeriSC[22] está sendo usada na *Design House* (DH) do Centro de Tecnologias Estratégicas do Nordeste (DH-CETENE)[6] que faz parte do programa CI-Brasil e é um centro de desenvolvimento e transferência de tecnologias consideradas estratégicas para a região Nordeste criado em 2005 pelo Ministério da Ciência e Tecnologia.
O trabalho aqui proposto representa um recurso computacional para a verificação de *IP-cores* que sejam ou venham a ser desenvolvidos através da metodologia de verificação funcional VeriSC que é um trabalho de doutorado desenvolvido pela aluna Karina Rocha [22][16] na UFCG.
O trabalho "*A Methodology aimed at Better Integration of Functional Verification and RTL Design*"[22] será apresentado no apêndice B deste documento. Trata-se de um artigo explicativo da metodologia VeriSC que é a base deste trabalho.

Capítulo 3

A Ferramenta eTBc

Esta seção apresentará detalhes arquiteturais e funcionais específicos da ferramenta eTBc além de alguns trabalhos relacionados.

Usuários de eTBc são tipicamente engenheiros de verificação funcional. Esses podem desenvolver *testbenches* de forma semi-automática através do uso da ferramenta. Após isso, será necessário a complementação no código gerado. Essa complementação consiste na implementação do plano de cobertura funcional, incluindo a geração de estímulos de acordo com as especificações do projeto.

A geração de código no eTBc é dita semi-automática pelo fato de o mesmo necessitar de alguns detalhes a serem completados como por exemplo, a cobertura funcional. Além disso, o usuário precisará implementar protocolos de comunicação (*handshake*). Após a implementação desses detalhes, o *testbench* estará pronto de acordo com as especificações do projeto em questão.

3.1 Funcionamento Geral da Ferramenta eTBc

A ferramenta eTBc funciona como um gerador de código. Esse recebe como entrada dois arquivos: Um *Transaction Level Netlist* (TLN) descrito pelo engenheiro de verificação funcional (usuário da ferramenta), e um molde (*template*) de um dos elementos de *testbench* a ser gerado, esse molde pode ser construído pelo responsável da metodologia de verificação funcional ou pelo desenvolvedor da ferramenta. Na Figura 3.1 é apresentada uma representação da ferramenta.

Na Figura 3.1, os moldes (*templates*) são criados através da linguagem *eTBc-Template-Language* (eTL) e os TLNs são criados através da linguagem *eTBc-Design-Language* (eDL). Internamente, a ferramenta usa dois compiladores, um deles para interpretar o código dos TLNs, e o outro para interpretar código dos moldes, e por fim, existe um gerador de código.

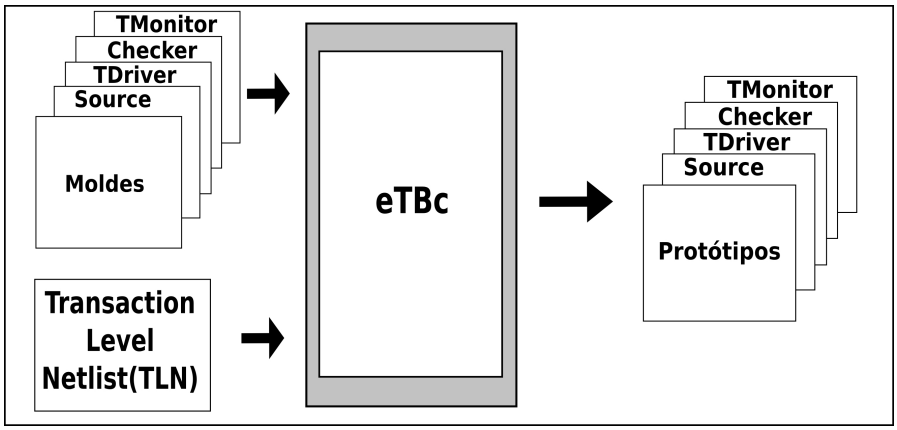

Figura 3.1: Representação arquitetural da ferramenta eTBc

3.2 A linguagem *eTBc-Design-Language* (eDL)

A linguagem eDL é usada para que o usuário da ferramenta possa elaborar os TLNs referentes ao seu projeto de verificação funcional. Para mais detalhes sobre TLN, consultar o tópico 2.3.2 do Capítulo 2. Essa linguagem possui um conjunto de dez palavras reservadas descritas na Tabela 3.1. Após a definição das palavras reservadas, será apresentado um exemplo com o uso das mesmas. No Apêndice A deste documento será apresentado a gramática referente à esta linguagem.

Tabela 3.1: Palavras reservadas da linguagem eDL

Palavra Reservada	**Significado**
struct	Palavra reservada que permite a definição de uma transação e de seus sinais. Esses dois tipos de dados representam o TLD(*Transaction Level Data*).
trans	Usada no escopo da palavra reservada "struct" e serve para declarar transações. No escopo de transações, os dados podem ser definidos com a mesma sintaxe que a declaração de dados em linguagem C, com os seguintes tipos primitivos: int, char , short, long, float, double, bool e unsiged.
signals	Esta palavra reservada é usada no escopo da palavra reservada "struct" e serve para declarar sinais em nível RTL(Register transfer Level).

Tabela 3.1: Palavras reservadas da linguagem eDL(continuação)

Palavra Reservada	**Significado**
signed	Usado para a declaração de um identificador com sinal numérico. Esta palavra reservada é usada no escopo de "signals".
unsigned	Usado para a declaração de um identificador sem sinal numérico. Esta palavra reservada é usada no escopo de "signals".
bool	Usado para a declaração de um identificador com um bit. Esta palavra reservada é usada no escopo de "signals".
invert ou **inv**	Usado no escopo de "signals" e serve para inverter o sentido do fluxo normal do sinal. É usado para criar fios de protocolo, onde há sinais em sentidos opostos.
module	Usado para declaração de blocos funcionais.
input	Usado para criar as interfaces de entrada de um módulo. Não confundir com entrada de sinal. Neste caso, as interfaces de entradas são vias por onde entram transações em um módulo.
output	Usado para criar as interfaces de saída de um módulo. Não confundir com saída de sinal. Neste caso, as interfaces de saída são vias por onde saem transações de um módulo.
channel	Usado para fazer a comunicação entre dois módulos. Em linhas gerais um *channel* tem função de ser o veículo das transações que trafegam entre dois módulos.

Para a melhor explicação do funcionamento das palavras reservadas, será mostrado o exemplo de um *IP-core* que realiza a operação de conversão do modelo de cor YUV[11] para o modelo RGB[11]. O modelo de cor YUV é amplamente usado em sistemas de TV e está presente em muitos projetos de codificadores e decodificadores de vídeo. O modelo RGB é usado no mercado em monitores do tipo CRT e LCD além de outros dispositivos.

Muitas vezes os dois modelos de cor são usados em conjunto no mesmo sistema. Isso acontece porque a representação interna do modelo YUV pode ser feita por um conjunto menor de informação e assim, tendo como consequência a economia de recursos. No caso, o RGB geralmente é usado na saída do sistema quando trata-se de vídeo. Então, dessa forma, o sistema como um todo usa o modelo YUV internamente e faz a conversão para RGB em sua saída economizando assim alguns recursos.

O modelo YUV é composto pelos seguintes elementos:

- Y : Componente de luminância (*luma*) que representa o brilho da imagem em termos de porção de cor acromática (preto e branco)
- U,V : Componentes de crominância que representam a porção de cor da imagem.

Já o modelo de cor RGB é representado pelos seguintes elementos:

- R : Componente vermelho.
- G : Componente verde.
- B : Componente azul.

Na Figura 3.2 é feita a representação do *IP-core* que será chamado neste exemplo de yuv2rgb. No caso deste exemplo, cada componente de cada um dos sistemas de cores, será representado com oito bits de precisão.

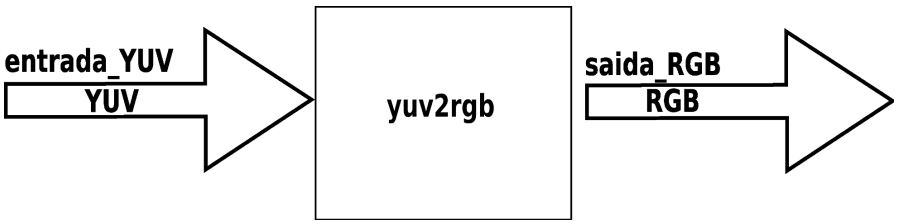

Figura 3.2: Representação do *IP-core* yuv2rgb

O código fonte 3.1 faz a representação desse *IP-core* usando a linguagem eDL do eTBc.

Código Fonte 3.1: TLN para representação do *IP-core* yuv2rgb em eDL

```
1  struct yuv {                //definicao de uma estrutura chamada yuv
2    trans {                   //definicao de transacoes da estrutura yuv
3      int y;                  //um dado do tipo inteiro
4      iny u;
5      int v;
6    }
7    signals {                 //definicao de sinais da estrutura yuv
8       signed   [8] canal;    //um sinal de oito bits
9       bool         valid;    //um sinal de um bit
10      bool  inv    ready;    //um sinal de um bit com seu sentido invertido
11   }
```

```
12 }
13
14 struct rgb {              //definicao de uma estrutura chamada rgb
15   trans {                 //definicao de transacoes da estrutura rgb
16     int r;               //um dado do tipo inteiro
17     int g;               //um dado do tipo inteiro
18     int b;               //um dado do tipo inteiro
19   }
20   signals {               //definicao de sinais da estrutura rgb
21      signed   [8] canal; //um sinal de oito bits
22      bool         valid; //um sinal de um bit
23      bool  inv    ready; //um sinal de um bit com seu sentido invertido
24   }
25 }
26
27 //Definicao de um bloco funcional chamado yuv2rgb.
28 //Eh o modulo principal do IP-core
29 module yuv2rgb {
30
31    input   yuv entrada_yuv; //definicao de uma interface de entrada
32                             //cujo tipo eh a estrutura yuv
33
34    output  rgb saida_rgb;   //definicao de uma interface de saida
35                             //cujo tipo eh a estrutura rgb
36 }
```

Neste exemplo, existem duas estruturas: rgb e yuv. Essas estruturas representam os dados nos níveis de transação e RTL. Para a estrutura rgb, os dados em nível de transação estão dentro do escopo de "*trans*"começando na linha 2 e terminando na linha 6. As variáveis y,u,v estão definidas representam os dados de transação.

Já para o nível RTL, que está no escopo de "signals"da linha 7 até a linha 11 foram definidos alguns nomes de sinais. Esses não serão usados em nível de transações, mas apenas em nível RTL.

Portanto, as variáveis em nível de transação, serão em nível RTL, transformadas no sinal "canal" no escopo de "signals". Os sinais "valid"e "ready"são elementos à parte que serão usados para criar protocolos de comunicação ou mais comumente chamados de *handshake*.

A estrutura "rgb"definida na linha 14 segue a mesma idéia da estrutura yuv, pois possuem dados

semelhantes. Por final, é definido um módulo chamado "yuv2rgb"na linha 29. Esse módulo representa o *IP-core* do exemplo. Como interface de entrada, foi definida uma variável com o nome "entrada_yuv"cujo tipo é a estrutura "yuv". Como interface de saída, foi definida uma variável "saida_rgb"cujo tipo é a estrutura "rgb".

3.3 A linguagem *eTBc Template Language* (eTL)

A linguagem *eTBc Template Language* (eTL) é usada para criar moldes (*templates*) dos códigos a serem gerados. Esses moldes serão usados então pela ferramenta para construir os protótipos de cada elemento de um *testbench*. Essa linguagem dos moldes possui uma característica flexível para permitir a criação de estruturas sintáticas variadas. Isso possibilita que a equipe de verificação funcional construa moldes para as suas necessidades de acordo com a linguagem escolhida para o plano de verificação do projeto.

No caso específico deste trabalho, foram construídos moldes para as linguagens SystemC, Verilog e a make-language [1]. No caso da primeira linguagem, foram construídos moldes para elementos de *testbench*, no caso da segunda linguagem, foi construído um molde para o DUV, no caso da terceira foram construídos moldes para scripts de compilação.

A linguagem funciona como uma estrutura para auxiliar a geração de código. Dessa forma, quando a ferramenta eTBc encontra alguma palavra reservada, é feita uma análise para identificar a devida geração de código a ser realizada no lugar da mesma. Todas as palavras reservadas nessa linguagem são definidas sintaticamente na seguinte forma: $$(palavra-reservada).

Essa estrutura sintática para palavras reservadas foi criada com intenção de não confundir o código eTL com o código de outras linguagens que estão sendo usadas dentro dos moldes como por exemplo, SystemC ou Verilog.

eTL não permite definição de variáveis. Desta forma, existem duas variáveis pré-definidas: i e j. Elas são usadas para controlar índice de laço. A linguagem permite dois níveis de aninhamento de laços, que são controlados pelas variáveis “i” e “j”. O índice “i” indica o laço mais interno, enquanto o índice “j” indica o laço mais externo. Assim, as palavras reservadas que forem indexadas com “i” estão se referindo ao laço mais interno, e as que forem indexadas com a variável “j” estão se referindo ao laço mais externo. Quando se tem apenas um laço sem aninhamento, deve-se usar o índice “i”.

Na linguagem eTL existem três tipos de palavras reservadas:

- Comandos de Controle: São usados para gerenciar o código gerado. Os comandos de controle são: $$(file), $$(endfile), $$(foreach), $$(endfor), $$(if), $$(endif). No caso do comando $$(if) as condições possíveis são: $$(isnotlast), $$(isinv) e $$(isarray).

- Comandos de Substituição: Causam a sua substituição por um certo código. São eles: $$(i.type), $$(j.type), $$(i.name), $$(j.name), $$(i.size) e $$(j.size).

- Listas: São usadas como parâmetros para o comando $$(foreach) e retornam uma lista de elementos a serem iterados. Os tipos de lista são: $$(struct), $$(module.in), $$(module.out), $$(var) e $$(signal).

A Tabela 3.2 apresenta os comandos de controle da linguagem eTL. Depois da apresentação dos três conjuntos de palavras reservadas, serão apresentados alguns exemplos fazendo o uso das mesmas.

Tabela 3.2: Comandos de controle da linguagem eTL

Palavra Reservada	**Significado**
$$(file)	Comando usado para criar um arquivo. Deve ser sucedido de um nome de arquivo.
$$(endfile)	Comando que indica fim de arquivo. Usado em conjunto com $$(file), para indicar onde deverá ser o início e o fim de um arquivo a ser criado.
$$(foreach)	Comando que indica o início de uma estrutura de laço. A estrutura de laço na linguagem eTL é usada para iterar em listas. As listas permitidas como parâmetro de $$(foreach) são as seguintes: $$(struct), $$(module.in) , $$(moule.out),$$(var) , $$(i.sigal) e $$(channels)
$$(endfor)	Comando que indica o fim de uma estrutura de laço. Usado em conjunto com $$(foreach) para criar um laço.
$$(if)	Comando que indica o início de uma estrutura condicional. São permitidos as seguintes condições : $$(isnotlast), $$(isinv) , $$(j.isnotlast) e $$(j.isinv).
$$(endif)	Comando que indica o fim de uma estrutura condicional iniciada com $$(if).

Tabela 3.2: Comandos de controle da linguagem eTL(continuação)

Palavra Reservada	**Significado**
$$(isnotlast)	Esse comando é uma das condições permitidas em $$(if). Serve para indicar que o laço está ou não na ultima iteração. Retorna verdade caso o comando $$(foreach) não esteja na ultima iteração.
$$(isinv)	Retorna verdade caso um sinal na lista de sinais tem o seu sentido invertido. A lista de sinais é declarada no escopo da palavra reservada "signals"na linguagem eDL para descrever TLNs.
$$(isarray)	Retorna verdade caso uma variável é um array. A lista de variáveis é declarada no escopo da palavra reservada "trans"na linguagem eDL para descrever TLNs.

Agora, na Tabela 3.3 serão apresentados os comandos de substituição. Eles estão relacionadas com a linguagem eDL discutida na seção 3.2 e serão usados para serem substituídos por declarações realizadas em um código de alguma TLN. Um exemplo de TLN foi descrita no exemplo yuv2rgb do código 3.1.

Tabela 3.3: Comandos de substituição na linguagem eTL

Palavra Reservada	**Significado**
$$(i.type)	Causa sua substituição por um tipo associado a um elemento. Esse tipo associado a um elemento é declarado no escopo de “trans” da linguagem eDL. O índice “i” se refere ao laço mais interno da estrutura $$(foreach).
$$(j.type)	Tem o mesmo significado de $$(i.type), mas neste caso para o índice “j” que é usado para o laço mais externo.
$$(i.name)	Causa a sua substituição por um elemento ou um sinal.
$$(j.name)	Tem a mesma função de $$(i.name) mas neste caso para o índice “j” que é usado para o laço mais externo.
$$(i.size)	Essa palavra reservada causa a sua substituição pelo tamanho em número de bits de um sinal.

Tabela 3.3: Comandos de substituição na linguagem eTL(continuação)

Palavra Reservada	**Significado**
$$(j.size)	Tem a mesma função do comando $$(i.size), mas neste caso para o índice “j” que é usado para o laço mais externo.
$$(i.port)	No caso de uma declaração do tipo .x(y) , $$(i.port) retorna x.
$$(j.port)	Mesmo que $$(i.port) , mas neste caso para o índice “j”.
$$(i.link)	No caso de uma declaração do tipo .x(y) , $$(i.port) retorna y.
$$(j.link)	Mesmo que $$(i.link) , mas neste caso para o índice “j”.

Na Tabela 3.4 serão apresentadas as palavras reservadas que representam listas em eTL. As listas são estruturas que armazenam informações das TLNs e podem ser iteradas através do comando $$(foreach).

Tabela 3.4: Listas em eTL

Palavra Reservada	**Significado**
$$(struct)	Essa lista armazena o conjunto de todas as estruturas que foram declaradas em uma TLN através da palavra reservada "struct".
$$(module.in)	Essa lista armazena todas as interfaces de entrada de um dado módulo declarado em uma TLN através da palavra reservada "input".
$$(module.out)	Essa lista armazena todas as interfaces de saída de um dado módulo declarado em uma TLN através da palavra reservada "output".
$$(var)	Essa lista armazena todas as variáveis declaradas no escopo de algum "trans" de uma TLN. Armazena variáveis de transações.
$$(signals)	Essa lista armazena todos os sinais declaradas no escopo de algum "signals" de uma TLN.
$$(channels)	Essa lista armazena todos os *channels* declaradas no escopo de algum "module" de uma TLN.

A seguir serão apresentados alguns exemplos mostrando o uso da linguagem eTL discutida anteriormente. Para esses exemplos será considerado o Código Fonte 3.1 da TLN que descreve o *IP-core* yuv2rgb já discutido anteriormente.

Código Fonte 3.2: Exemplo do uso de eTL para criar um arquivo

```
1 $$file $$(exemplo-etl.txt)
2 Esse comando cria um arquivo com o nome exemplo-etl.txt
3 $$(endfile)
```

Esse trecho de Código Fonte 3.2 produzirá um arquivo de saída chamado "exemplo-etl.txt"com o conteúdo mostrado no Código Fonte 3.3 a seguir.

Código Fonte 3.3: Código resultante do molde 3.2

```
1 Esse comando cria um arquivo com o nome exemplo-etl.txt
```

Agora será apresentado um exemplo com laço no Código Fonte 3.4 a seguir.

Código Fonte 3.4: Exemplo de laço

```
1 $$file $$(exemplo_foreach.txt)
2  exemplo demonstrativo do comando "foreach"
3
4 $$(foreach) $$(struct)
5     Cada struct da TLN sera visitada neste laco
6     $$(i.name)
7 $$(endfor)
8
9 $$(endfile)
```

Esse exemplo cria um arquivo chamado "exemplo_foreach.txt" através do comando na linha 1. O laço da linha 4 faz uma iteração na lista de estruturas do exemplo da TLN 3.1 do yuv2rgb. A lista de estruturas é então iterada e o laço através do índice i é percorrido. Na linha 6 do exemplo a palavra reservada $$(i.name) resgata o nome da estrutura que está sendo iterada no momento. No caso a iteração se repetiu duas vezes, pois no exemplo da TLN 3.1 existem duas estruturas.

O código resultante para o exemplo 3.4 é o arquivo exemplo_foreach.txt do Código Fonte 3.5:

Código Fonte 3.5: Arquivo exemplo_foreach.txt

```
1  exemplo demonstrativo do comando "foreach"
2
3
4     Cada struct da TLN sera visitada neste laco
5     yuv
6
```

```
7     Cada struct da TLN sera visitada neste laco
8     rgb
```

Considerando o próximo exemplo do Código Fonte 3.6 que é mais completo,

Código Fonte 3.6: Exemplo mais completo

```
1  $$file $$(exemploMaior.txt)
2  Um outro exemplo mais completo
3
4  $$(foreach) $$(module.in)
5    $$(foreach) $$(i.signal)
6      Cada sinal de cada estrutura da TLN sera visitada neste laco
7      $$(j.name) <- O nome de cada interface de entrada - Laco mais externo
8      $$(i.name) <- O nome de cada sinal - Laco mais interno
9      $$(if) $$(i.isnotlast) Esta nao eh a utima iteracao $$(endif)
10   $$(endfor)
11 $$(endfor)
12
13 $$(endfile)
```

Tem-se o arquivo apresentado no Código Fonte 3.7 com o nome "exemploMaior.txt" resultante do molde 3.6 anterior.

Código Fonte 3.7: Arquivo exemploMaior.txt

```
1  Um outro exemplo mais completo
2
3
4
5      Cada sinal de cada estrutura da TLN sera visitada neste laco
6      entrada_yuv <- O nome de cada interface de entrada - Laco mais externo
7      y <- O nome de cada sinal - Laco mais interno
8        Esta nao eh a utima iteracao
9
10     Cada sinal de cada estrutura da TLN sera visitada neste laco
11     entrada_yuv <- O nome de cada interface de entrada - Laco mais externo
12     u <- O nome de cada sinal - Laco mais interno
13       Esta nao eh a utima iteracao
14
```

```
15    Cada sinal de cada estrutura da TLN sera visitada neste laco
16    entrada_yuv <- O nome de cada interface de entrada - Laco mais externo
17    v <- O nome de cada sinal - Laco mais interno
18      Esta nao eh a utima iteracao
19
20    Cada sinal de cada estrutura da TLN sera visitada neste laco
21    entrada_yuv <- O nome de cada interface de entrada - Laco mais externo
22    valid <- O nome de cada sinal - Laco mais interno
23      Esta nao eh a utima iteracao
24
25    Cada sinal de cada estrutura da TLN sera visitada neste laco
26    entrada_yuv <- O nome de cada interface de entrada - Laco mais externo
27    ready <- O nome de cada sinal - Laco mais interno
```

No código do exemplo 3.6 observa-se o uso de laços aninhados. Os laços aninhados definidos nas linhas 4 e 5 do molde 3.6 produziram o código entre as linhas 5 e 27 do arquivo gerado 3.7.

Nas linhas 6,11,16,21 e 26 do arquivo resultante exemploMaior.txt(código 3.7) observa-se a ação do comando da linha 7 do molde 3.6. Essa linha do molde faz chamada ao comando $$(j.name) que pertence ao laço mais externo e está iterando na lista de interfaces de entrada do módulo yuv2rgb do código 3.1. Como só existe uma interface de entrada, essa foi repetida durante todo o ciclo até o final do laço.

No molde 3.6 , na linha 8 é usado o comando $$(i.name) que por estar associado ao índice i faz iteração no laço mais interno declarado na linha 5. Esse laço está sobre a lista de sinais do laço mais externo declarado na linha 4. No caso a linha 8 do molde produziu as linhas 7,12,17,22 e 27 do código resultante 3.7.

Existe também o comando $$(if) na linha 9 do molde 3.6. Esse comando produziu as linhas 8,13,18 e 23 do arquivo resultante 3.7. No caso da última iteração, não foi produzido código para a linha 9 do molde 3.6 pois o comando $$(if) desta linha realizou a condição de não produzir código para a última iteração do laço.

3.4 Trabalhos Relacionados

Esta seção apresentará alguns trabalhos relacionados com esta dissertação e também fará análises com relação às vantagens e desvantagens de cada uma das ferramentas apresentadas.

No trabalho *Automatic Testbench Generation for Simulation-based Validation* [18] a ideia principal

é fazer a criação de estímulos para que os mesmos consigam cobrir toda a funcionalidade do *IP-core* que está sendo verificado. A criação é realizada através de um algoritmo genético que por sua vez faz o cálculo dos estímulos mais significativos ao domínio do problema a partir da descrição das entradas do *IP-core*. Após a criação, os mesmos podem ser inseridos nos *testbenches*. A vantagem dessa abordagem [18] é o fato de se fazer uma análise detalhada do domínio do problema e assim criar um conjunto de entradas (estímulos) mais adequados. A desvantagem é o fato da ferramenta não gerar código efetivo de *testbenches*. Por um lado isso é interessante, pois supre uma parte da necessidade de criação de estímulos, mas por outro lado pode deixar o desenvolvedor (engenheiro de verificação) com um trabalho extra de adequar os estímulos gerados aos seus *testbenches*.

Em *An automatic testbench generation system* [17] a idéia é o uso de uma ferramenta para a geração automática de *testbenches* para modelos comportamentais em VHDL [10]. Trata-se de uma ferramenta dotada de interface gráfica capaz de abstrair modelos em VHDL através de um artifício que o autor chama de *Process Model Graph*(PGM). Depois do modelo descrito em PGM, é então usado um gerador automático de *testbenches* chamado *Testbench Generator*(TBG).

Outros trabalhos [9][8] têm o propósito de fazer geração automática de *testbenches* simplificados em VHDL. Trata-se de geradores automáticos de código que criam um arquivo de saída baseado no modelo VHDL. No caso do *VHDL-Online* [9] a vantagem é o fato desse gerar um código simples baseado no modelo de entrada do usuário. A desvantagem é a falta de flexibilidade para a geração de outros tipos de *testbenches* pois a ferramenta não oferece nenhuma opção auxiliar.

A outra ferramenta *VHDL Testbench Creation Using Perl* [8] apresenta comportamento similar em que o usuário fornece um modelo VHDL e recebe um *testbench* simplificado com estímulos básicos. Em todos os casos foi observado que não há geração de código para uma linguagem de escolha do usuário. Ou seja, não é possível que este possa escolher em qual linguagem o seu *testbench* deverá ser criado. Essa característica foi observada como importante em um dos requisitos deste trabalho. Assim, foi criado um mecanismo que o usuário possa criar *testbench* em outras linguagens e não fique limitado apenas a uma linguagem específica.

Outra característica observada nos trabalhos citados é que os mesmos não consideram a metodologia de verificação usada fazendo a flexibilidade ao uso da ferramenta mais limitado. Desta forma, outra característica importante é o fato de o que este trabalho leva em consideração a metodologia de verificação funcional podendo até mesmo adaptar-se a uma metodologia específica.

Capítulo 4

Exemplos de Funcionamento da Ferramenta eTBc

Este Capítulo apresentará exemplos completos do fluxo de funcionamento da ferramenta. Além disso, serão apresentados exemplos de protótipos gerados de alguns elementos do *testbench* nas linguagens SystemC e Verilog. Para isso foi usado o módulo *Differencial Pulse Code Modulation*(DPCM) [2]. Ele foi escolhido como exemplo por ser simples e fácil de modelar através do eTBc. Alguns exemplos de código gerado serão apresentados.

O DPCM tem como função básica mudar a representação de um sinal digital. A funcionalidade é simples, o valor atual da entrada é subtraído do valor anterior e o resultado saturado é colocado na saída. Portando, o DPCM terá dois sub-módulos internos:

- dif: Tem função de calcular a diferença entre o valor atual e o anterior
- sat: Faz a saturação do valor resultante da diferença. A saturação é o cálculo dos limiares superior e inferior.

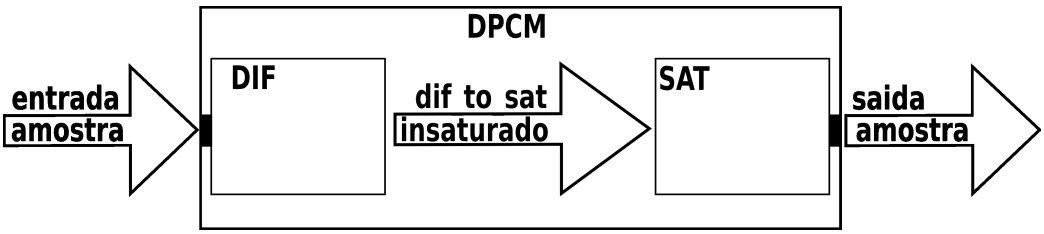

Figura 4.1: Representação do DPCM

Na Figura 4.1 é possível observar a seguinte situação :

Para o DPCM existem duas interfaces que são “entrada” e “saida” , de entrada e de saída do módulo, respectivamente. Através da interface “entrada” o tipo de dado é representado pela estrutura “amostra”. Na interface de saída, o tipo de dado também é a estrutura “amostra”, pois o tipo de dado que entra no módulo é igual ao tipo de dado que sai do mesmo. O dado que entra é então passado para o sub-módulo "dif" para que este calcule a diferença entre o valor atual e o anterior. O resultado da diferença é então repassado para sub-módulo "sat" para que esse calcule a saturação. Observando que o tipo de dado que sai do módulo dif é "insaturado"e assim o tipo de dado que entra no módulo "sat" é também insaturado. O dado que sai do módulo "sat"é por sua vez do tipo "amostra"que é repassado para a saída do DPCM.

O Código Fonte 4.1 a seguir do representa a TLN que modela o DPCM na linguagem eDL.

Código Fonte 4.1: Arquivo dpcm.design, TLN para o DPCM escrito na linguagem eDL

```
1  //definicao de estrutura de dados amostra
2  struct amostra{
3   trans {
4    //declaracao de variavel para transacao
5     int valor;
6   }
7   signals {
8     //sinal de tres bits em nivel RTL
9     signed [3] valor;
10    //sinal para realizar protocolo(handshake)
11    bool       valid;
12    //sinal para realizar protocolo(handshake)
13    bool inv   ready;
14  }
15 }
16
17 //definicao da estrutura de dados unsaturated
18 struct insaturado{
19  trans {
20    int valor;
21  }
22  signals {
23    //sinal de quatro bits em nivel RTL
```

```
24      signed [4]  valor;
25      //sinal para realizar protocolo(handshake)
26      bool        valid;
27      //sinal para realizar protocolo(handshake)
28      bool inv    ready;
29    }
30  }
31
32  //modulo dif usado para fazer a diferenca
33  module dif{
34    //interface de entrada do tipo amostra
35    input amostra     dif_in;
36    //interface de saida do tipo unsaturated
37    output insaturado dif_out;
38  }
39
40  //modulo sat usado para realizar a saturacao
41  module sat{
42    //interface de entrada do tipo unsaturated
43    input insaturado sat_in;
44    //interface de saida do tipo amostra
45    output amostra   sat_out;
46  }
47
48  //modulo principal
49  module dpcm{
50    //interface de entrada do tipo amostra
51    input  amostra entrada;
52    //interface de entrada do tipo amostra
53    output amostra saida;
54
55    //fifo do tipo insaturado para realizar a ligacao
56    //entre os modulos
57    fifo insaturado dif_to_sat;
58
59    //ligacao entre os modulos atraves da fifo dif_to_sat
60    dif dif_i( .dif_in( entrada )   , .dif_out( dif_to_sat )  );
```

```
61    sat sat_i( .sat_in( dif_to_sat) , .sat_out(saida)          );
62 }
```

No Código Fonte 4.1 é possível observar os seguintes características :

- **linhas 1 a 15:** Definição da estrutura amostra.
  - **linhas 3 a 6:** Definição de transação para a estrutura amostra.
  - **linha 5:** Declaração de variável em nível de transação.
  - **linhas 7 a 14:** Declaração de sinais em nível RTL.
- **linhas 18 a 30:** Definição da estrutura "insaturado". Segue a mesma idéia que a definição da estrutura amostra nas linhas de 1 a 15 com a diferença do sinal valor que agora tem quatro bits para conseguir realizar operação de diferença sem perda de precisão.
- **linhas 33 a 38:** Definição do módulo dif que tem a função de realizar a diferença.
  - **linha 35:** Definição de uma interface de entrada para o módulo dif do tipo "amostra" e nome "dif_in".
  - **linha 37:** Definição de uma interface de saída para o módulo dif do tipo "insaturado" e nome "dif_out".
- **linhas 41 a 46:** Definição do módulo sat que realizará a saturação.
  - **linha 43:** Definição de uma interface de entrada para o módulo sat do tipo "insaturado" e nome "sat_in".
  - **linha 45:** Definição de uma interface de saída para o módulo sat do tipo "amostra" e nome "sat_out".
- **linhas 49 a 62:** Definição do módulo dpcm que é o módulo principal.
  - **linha 51:** Definição de uma interface de entrada para o módulo dpcm do tipo "amostra" e nome "entrada".
  - **linha 53:** Definição de uma interface de saída para o módulo sat do tipo "amostra" e nome "saida".
  - **linha 57:** Definição de um channel chamada dif_to_sat do tipo "insaturado" e que terá a função de fazer a ligação entre o sub-módulo dif e o sub-módulo sat.

– **linhas 60 e 61:** Essas definições realizam a função de acoplar todos os módulos. São instâncias dos módulos dif(dif_i) e sat(sat_i) e são usadas para realizar a ligação entre suas interfaces.

A interface de entrada do "dif" está ligada à interface de entrada do módulo dpcm.

Da mesma forma, a interface de saída do "dif" está ligada à interface de entrada do "sat" através do *channel* "dif_to_sat". Finalizando, tem-se a interface de saída do "sat" ligada à interface de saída do "dpcm".

4.1 Exemplos de Geração de Protótipos

Com base na TLN do Código Fonte 4.1, que representa o DPCM, serão apresentados alguns exemplos de funcionamento da ferramenta eTBc.

Serão mostrados os protótipos gerados e seus moldes associados. Após a definição do TLN, pode-se então gerar qualquer protótipo de um dos elementos de um *testbench*, bastando que para isso, o usuário faça chamadas à ferramenta eTBc. Para usar o eTBc o usuário deverá obedecer a sintaxe do Código Fonte 4.2 em um terminal de linhas de comando:

Código Fonte 4.2: Sintaxe do eTBc

```
1 eTBc [tln] [molde] [modulo]
```

De acordo com o Código Fonte 4.2 que exibe a sintaxe de chamada ao eTBc, são necessários três parâmetros para executá-lo:

- **tln:** O nome de TLN que representa o *IP-core*.
- **molde:** O nome do molde(*template*) que será usado para a geraçaõ do protótipo (Source, TDriver, TMonitor, Checker ...) desejado.
- **modulo:** O nome da instância ou do módulo do TLN para a qual os protótipos serão gerados.

Para efeito ilustrativo, no caso do código da TLN 4.1 do DPCM, será usado apenas o módulo dpcm para a geração de código.

Para a geração de um protótipo do elemento "Source" por exemplo deve-se usar a sintaxe exibida no Código Fonte 4.3:

Código Fonte 4.3: Sintaxe do eTBc para a geração do Source

```
1 eTBc dpcm.design sc_source dpcm
```

Sendo:

- **dpcm.design:** O nome do arquivo que representa a TLN do *IP-core* DPCM.
- **sc_source:** O nome do arquivo de molde SystemC do Source.
- **dpcm:** O nome do módulo para o qual o código do protótipo Source está sendo gerado.

Como resultado o arquivo de saída gerado chamado source.h é apresentado no Código Fonte 4.4.

Código Fonte 4.4: Arquivo source.h gerado pelo eTBc

```
1  class entrada_constraint_class: public scv_constraint_base {
2    //  scv_bag<int> ?_distrib;
3   public:
4    scv_smart_ptr<amostra> amostra_sptr;
5     SCV_CONSTRAINT_CTOR(entrada_constraint_class) {
6    //  ?_distrib.push(?, ?);
7    // amostra_sptr->?.set_mode(?_distrib);
8    //  SCV_CONSTRAINT(amostra_sptr->?() > ?);
9       }
10 };
11
12 SC_MODULE(source) {
13
14     sc_fifo_out<amostra_ptr> entrada_to_refmod;
15     sc_fifo_out<amostra_ptr> entrada_to_driver;
16     entrada_constraint_class entrada_constraint;
17
18 void entrada_p() {
19      ifstream ifs("entrada.stim");
20      string type;
21      amostra stim;
22
23      //Static stimuli
24      while( !ifs.fail() && !ifs.eof() ) {
25        ifs >> type;
26        if ( type == "amostra" ) {
27          ifs >> stim;
28          entrada_to_refmod.write(new amostra( stim ) );
```

```
29          }
30          ifs.ignore(225, '\n');
31      }
32
33      //Random stimuli generantion
34      while(1) {
35          entrada_constraint.next();
36          stim =  entrada_constraint.amostra_sptr.read();
37          entrada_to_refmod.write(new  amostra( stim ) );
38          entrada_to_driver.write(new  amostra( stim ) );
39      }
40  }
41
42  SC_CTOR( source ):
43          entrada_constraint("entrada_constraint")
44      {
45          SC_THREAD(entrada_p);
46      }
47  };
```

O Código Fonte 4.4 usou como molde o Código Fonte 4.5.

Código Fonte 4.5: Arquivo de molde para o source.h

```
1  $$(file) $$(source.h)
2  $$(foreach) $$(module.in)
3  class $$(i.name)_constraint_class: public scv_constraint_base {
4    //  scv_bag<int> ?_distrib;
5   public:
6    scv_smart_ptr<$$(i.type)> $$(i.type)_sptr;
7     SCV_CONSTRAINT_CTOR($$(i.name)_constraint_class) {
8    //  ?_distrib.push(?, ?);
9    // $$(i.type)_sptr->?.set_mode(?_distrib);
10   //  SCV_CONSTRAINT($$(i.type)_sptr->?() > ?);
11      }
12 };
13 $$(endfor)
14
15 SC_MODULE(source) {
```

```
16
17 $$(foreach)  $$(module.in)
18     sc_fifo_out<$$(i.type)_ptr> $$(i.name)_to_refmod;
19     sc_fifo_out<$$(i.type)_ptr> $$(i.name)_to_driver;
20     $$(i.name)_constraint_class $$(i.name)_constraint;
21 $$(endfor)
22
23 $$(foreach) $$(module.in)
24 void $$(i.name)_p() {
25      ifstream ifs("$$(i.name).stim");
26      string type;
27      $$(i.type) stim;
28
29      //Static stimuli
30      while( !ifs.fail() && !ifs.eof() ) {
31        ifs >> type;
32        if ( type == "$$(i.type)" ) {
33          ifs >> stim;
34          $$(i.name)_to_refmod.write(new $$(i.type)( stim ) );
35        }
36
37        ifs.ignore(225, '\n');
38      }
39
40      //Random stimuli generantion
41      while(1) {
42          $$(i.name)_constraint.next();
43          stim =  $$(i.name)_constraint.$$(i.type)_sptr.read();
44          $$(i.name)_to_refmod.write(new  $$(i.type)( stim ) );
45          $$(i.name)_to_driver.write(new  $$(i.type)( stim ) );
46      }
47 }
48 $$(endfor)
49      SC_CTOR( source ):
50      $$(foreach) $$(module.in)
51        $$(i.name)_constraint("$$(i.name)_constraint") $$(if) $$(i.
              isnotlast) , $$(endif)
```

```
52      $$(endfor)
53      { $$(foreach) $$(module.in)
54          SC_THREAD($$(i.name)_p);
55        $$(endfor)
56      }
57 };
58 $$(endfile)
```

Para realizar a geração do Código Fonte 4.4 o eTBc usou o Código Fonte 4.1 da TLN do DPCM juntamente com o Código Fonte 4.5 do molde.
Com relação ao Código Fonte 4.5 do molde usado para gerar o Código Fonte 4.4 é possível observar o seguinte comportamento:

- **linha 1:** Definição do arquivo a ser gerado.
- **linhas 2 a 13 :** Laço que itera na lista de interfaces de entrada do módulo dpcm. O dpcm possui apenas uma interface de entrada.
  - **linha 6:** O comando $$(i.type) retorna o nome do tipo da interface de entrada e produz a linha 4 no Código Fonte 4.4 do protótipo gerado.
  - **linha 7:** O comando $$(i.name) retorna o nome da interface de entrada e produz a linha a linha 5 no Código Fonte 4.4 do protótipo gerado.
- **linha 15:** Essa linha produziu diretamente a linha 12 no arquivo 4.4 de saída pois a mesma não está regida por nenhum laço.
- **linhas 17 a 21:** Definição de outro laço iterando na lista de interfaces de entrada do módulo dpcm da TLN. Essas linhas usam o mesmo princípio de geração de código observado anteriormente, pois os mesmos comandos de substituição foram usados.
- **linha 51 :** O comando $$(if) desta linha do molde produziu a linha 43 do protótipo gerado. O caractere "," não apareceu no arquivo pois a condição do $$(if) não foi satisfeita e logo o código no escopo do comando não foi gerado.

Após a geração do Código Fonte 4.4 o engenheiro de verificação funcional precisa criar estímulos dentro desse código para que o mesmo esteja pronto para a verificação. Esses estímulos serão usados na tentativa de provocar erros funcionais nos DUVs que estão sendo estimulados. Esses erros indicarão o que precisa ser modificado para que a verificação seja satisfatória.

O Código Fonte 4.6 é o arquivo "structs.h" que também é um arquivo na linguagem SystemC gerado pelo eTBc através do molde apresentado no Código Fonte 4.7 juntamente com a TLN 4.1.

Código Fonte 4.6: Arquivo structs.h gerado pelo eTBc

```
1  //Data Types
2  #ifndef STRUCTS_H
3  #define STRUCTS_H
4  #include <iomanip.h>
5
6  // struct for amostra
7  struct  amostra {
8     int  valor;
9     inline bool operator== (const amostra& arg) const {
10       return(
11                ( valor == arg.valor)
12            );
13    }
14 };
15
16 typedef amostra *amostra_ptr;
17
18   inline ostream& operator << (ostream& os, const amostra& arg){
19       os << "amostra=("
20         << "valor=" << arg.valor
21         << ")";
22        return os;
23    }
24
25    inline istream& operator>> (istream& is, amostra& arg) {
26         is >> arg.valor;
27         return is;
28    }
29
30
31 // struct for insaturado
32 struct  insaturado {
33    int  valor;
34    inline bool operator== (const insaturado& arg) const {
```

```
35        return(
36                  ( valor == arg.valor)
37              );
38     }
39 };
40
41 typedef insaturado *insaturado_ptr;
42
43   inline ostream& operator << (ostream& os, const insaturado& arg){
44        os << "insaturado=("
45          << "valor=" << arg.valor
46          << ")";
47         return os;
48     }
49
50     inline istream& operator>> (istream& is, insaturado& arg) {
51           is >> arg.valor;
52           return is;
53     }
54 #endif
```

O molde usado para criar o arquivo 4.6 é mostrado no Código Fonte 4.7 a seguir.

Código Fonte 4.7: Arquivo de molde sc_structs

```
1  $$(file) $$(structs.h)
2  //Data Types
3  #ifndef STRUCTS_H
4  #define STRUCTS_H
5  #include <iomanip.h>
6
7  $$(foreach)  $$(struct)
8  // struct for $$(i.name)
9  struct  $$(i.name) {
10   $$(foreach) $$(i.var)
11    $$(i.type) $$(i.name);
12   $$(endfor)
13    inline bool operator== (const $$(i.name)& arg) const {
14       return(  $$(foreach) $$(i.var)
```

```
15                ( $$(i.name) == arg.$$(i.name)) $$(if) $$(i.isnotlast) &&
                      $$(endif) $$(endfor)
16            );
17    }
18 };
19
20 typedef $$(i.name) *$$(i.name)_ptr;
21   inline ostream& operator << (ostream& os, const $$(i.name)& arg){
22       os << "$$(i.name)=("
23       $$(foreach) $$(i.var)
24         << "$$(i.name)=" << arg.$$(i.name) $$(if) $$(i.isnotlast) << " "
               $$(endif) $$(endfor)
25         << ")";
26        return os;
27    }
28
29    inline istream& operator>> (istream& is, $$(i.name)& arg) {
30       $$(foreach) $$(i.var)
31         is >> arg.$$(i.name);
32       $$(endfor)
33         return is;
34    }
35 $$(endfor)
36 #endif
37 $$(endfile)
```

Com relação ao Código Fonte 4.7 é possível observar o seguinte comportamento:

- **linhas 7 a 35 :** Cria um laço em cima da lista de estruturas definidas na TLN do DPCM.
  - **linhas 10 a 12 :** Cria outro laço aninhado com o laço anterior. Esse novo laço está agora iterando entre todas as variáveis de transações de uma dada estrutura. Essas linhas produzem o código referente às linhas 8 e 33 do Código Fonte 4.7 gerado.
  - **linha 24 :** Essa linha é responsável por produzir o código das linhas 20 e 45 do arquivo gerado. O comando $$(if) faz a restrição de gerar código apenas caso não esteja na última iteração do laço.

Agora será mostrado um código Verilog gerado pelo eTBc. O Código Fonte 4.8 foi gerado a partir do código do molde 4.9.

Código Fonte 4.8: Arquivo dpcm.v gerado pelo eTBc

```
1  //DUV
2  module dpcm (
3     clk,
4     reset,
5     entrada_valor,
6     entrada_valid,
7     entrada_ready,
8     saida_valor,
9     saida_valid,
10    saida_ready
11 )
12    input clk;
13    input reset;
14    input [3-1:0] entrada_valor;
15    input [3-1:0] entrada_valid;
16    input [3-1:0] entrada_ready;
17
18    output [3-1:0] saida_valor;
19    output [3-1:0] saida_valid;
20    output [3-1:0] saida_ready;
21
22 endmodule
```

Os princípios usados para gerar o código Verilog foram os mesmos usados para gerar código SystemC.

Código Fonte 4.9: Arquivo de molde v_duv

```
1  $$(file) $$(module.name.v)
2  //DUV
3  module $$(module.name) (
4     clk,
5     reset,
6     $$(foreach) $$(module.in)
7     $$(foreach) $$(i.signal)
```

```
8     $$(j.name)_$$(i.name),
9     $$(endfor)
10    $$(endfor)
11    $$(foreach) $$(module.out)
12    $$(foreach) $$(i.signal)
13    $$(j.name)_$$(i.name) $$(if) $$(i.isnotlast) ,  $$(endif)
14    $$(endfor)
15    $$(endfor)
16
17  )
18    input clk;
19    input reset;
20    $$(foreach) $$(module.in)
21    $$(foreach) $$(i.signal)
22    input [ $$(i.size)-1 :0] $$(j.name)_$$(i.name);
23    $$(endfor)
24    $$(endfor)
25    $$(foreach) $$(module.out)
26    $$(foreach) $$(i.signal)
27    output [ $$(i.size)-1 :0] $$(j.name)_$$(i.name);
28    $$(endfor)
29    $$(endfor)
30
31  endmodule
32  $$(endfile)
```

O Código Fonte 4.9 do molde para geração de DUV possui algumas características a serem observadas :

- **linhas 6 a 10:** Esse trecho de código possui um aninhamento de laços. O laço mais externo na linha 6 está iterando na lista de interfaces de entrada do módulo dpcm da TLN 4.1. O laço mais interno da linha 7 está iterando na lista de sinais de cada um dos elementos da laço mais externo. A linha 8 então do molde produz as linhas 5 a 10 do arquivo dpcm.v gerado mostrado no código 4.8.

Este capítulo apresentou de forma detalhada como a geração de código é feita no eTBc. Esse processo é feito usando-se as linguagens eDL e eTL. Através das TLN é possível modelar o sistema desejado e através de moldes é possível modelar os arquivos de saída.

Capítulo 5

Resultados e Sugestões para Trabalhos Futuros

5.1 Resultados

No decorrer do trabalho, uma equipe de aproximadamente vinte pessoas que estavam em processo de aprendizagem da metodologia [22] usaram duas abordagens para criação *testbenches* nessa metodologia. A primeira, foi a criação de todos os protótipos dos *testbenches* manualmente. A segunda, foi através do uso da ferramenta eTBc.

A diferença entre o método manual e o semi-automático é que, no primeiro método, todo o código dos *testbenches* são feitos sem a ajuda de nenhum gerador de código, isso quer dizer que o código é gerado linha a linha manualmente. O segundo método usa o gerador automático de código que no caso é o eTBc. Usando a primeira forma, a equipe teve um tempo de aproximadamente 72h corridas para criar todos os protótipos de um *testbench* em SystemC. Além disso, para poder compilar o *testbench*, era necessário a criação de um arquivo com regras de compilação.

O *testbench* estava sendo construído para realizar o processo de verificação funcional do *IP-core* DPCM [2].

Depois da construção manual, o processo foi recomeçado do ponto inicial, mas agora usando o segunda alternativa através do eTBc. Nesta abordagem, as pessoas precisaram entender o funcionamento da linguagem *eTBc Design Language* (eDL) para a criação de modelos de *IP-cores* em nível de transação. Depois disso, foi criado o modelo do *IP-core* DPCM na linguagem eDL. Esse processo de aprendizagem até a criação do modelo do DPCM teve uma duração de aproximadamente três horas.

Após isso, todos os protótipos de cada um dos elementos dos *testbenches* puderam ser gerados em quinze minutos. Nesse tempo de geração de elementos, está também incluso o tempo de aprendizagem do uso da ferramenta.

Depois da geração de todos os protótipos dos elementos dos *testbenches*, foi necessário criar um arquivo de regras de compilação também através de eTBc.

Uma vez compilado o *testbench*, as pessoas passaram a implementar os seus planos de verificação incluindo a cobertura funcional e definição de estímulos nos elementos gerados. O processo inteiro demorou aproximadamente 15h. Com isso, o uso da ferramenta em alternativa ao processo comum que é manual teve um significativo ganho com relação ao tempo e esforço adotado na criação do *testbench* para o *IP-core* DPCM.

A porcentagem de código gerada pela ferramenta eTBc é de 85% para um *testbench* completo. Isso significa que de um *testbench* que já está pronto, rodando e com sua cobertura funcional completa, 85% desse código foi produzido pela ferramenta, e o restante, 15% foi resultado de um trabalho manual.

A expectativa é que eTBc seja agora usada na *Design House* CETENE [6] como ferramenta de apoio no processo de verificação funcional.

5.2 Sugestões para Trabalhos Futuros

eTBc ainda não foi usada massivamente para construir um grande sistema. Desta forma existe um conjunto de características que precisam ser desenvolvidas com a finalidade de refinar a ferramenta:

- Uso da ferramenta para construção de um *IP-core* complexo para a identificação de faltas e bugs ainda não encontrados.
- Refinamento das linguagens eDL e eTL.
- Construção de uma Interface Gráfica.
- Refinamento dos moldes (*templates*) existentes.
- Criação de moldes para VHDL e SystemVerilog.
- Criação de um manual do usuário.
- Uso do eTBc para criação de estímulos pseudo-aleatórios.

5.3 Considerações Finais

Este trabalho esteve diretamente ligado ao trabalho de desenvolvimento de uma metodologia de verificação funcional chamada VeriSC [22]. A idéia principal foi desenvolver uma abordagem que conseguisse acelerar o processo de construção de *testbenches*, pois o trabalho de se criar manualmente testbenches geralmente requer um esforço muito grande por parte do engenheiro de verificação. Com isso a idéia inicial de poder construir *testbenches* de forma semi-automática foi bem aceita, o trabalho então foi desenvolvido e a ferramenta foi implementada com esse intuito.

A elaboração da idéia precisava ser ampla ao ponto de conseguir adaptar-se as freqüentes mudanças na metodologia VeriSC. Assim, foi proposto o uso de moldes de forma a flexibilizar a solução. Dessa forma, a ferramenta foi construída também com o foco na geração de código em outras linguagens de hardware.

A idéia de moldes foi fazer com que o gerente de verificação pudesse construí-los de forma a adaptarem-se as suas necessidades sem que seja preciso fazer modificações na ferramenta.

A experiência adquirida com o desenvolvimento desse trabalho foi de grande utilidade na aprendizagem tanto da área de hardware quanto em software. Em hardware foi adquirido o conhecimento no fluxo de desenvolvimento de um projeto de *IP-core* focando nas metodologias de verificação funcional. Em software, o conhecimento na área de construção de compiladores foi bastante usado no desenvolvimento deste trabalho.

Bibliografia

[1] Disponível em: <http://en.wikipedia.org/wiki/make> . acessado em: 05 mar. 2007.

[2] Disponível em: <http://en.wikipedia.org/wiki/pulse-code_modulation> . acessado em: 05 mar. 2007.

[3] Disponível em: <http://en.wikipedia.org/wiki/systemverilog> . acessado em: 05 mar. 2007.

[4] Disponível em: <http://gals.sourceforge.net> . acessado em: 05 mar. 2007.

[5] Disponível em: <http://lad.dsc.ufcg.edu.br> . acessado em: 05 mar. 2007.

[6] Disponível em: <http://www.bergbrandt.com.br/cetene/asp/home.asp> . acessado em: 05 mar. 2007.

[7] Disponível em: <http://www.brazilip.org.br> . acessado em: 05 mar. 2007.

[8] Disponível em: <http://www.doulos.com/knowhow/perl/testbench_creation/> . acessado em: 05 mar. 2007.

[9] Disponível em: <http://www.vhdl-online.de/tb-gen/ent2tb1.htm> . acessado em: 05 mar. 2007.

[10] *IEEE standard VHDL language reference manual.* IEEE Standards Office, New York, NY, USA, 1993.

[11] ISO/IEC 14496-2:2001(E). *Coding of Audio-Visual Objects - Part 2: Visual*, Dec 2001.

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

[13] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] D. S.;COX S.;GALLO J.;GRUNDMANN W. ;IP C. N.;PAULSEN W. ;PIERCE J. L. ;ROSE J.; SHEA D.; WHITING K. BRAHME. The transaction-based verification methodology. Technical report CDNL-TR-2000-0825, Cadence Berkeley Labs, August 2000.

[15] L. Cai and D. Gajski. Transaction level modeling in system level design. Technical report, Center for Embedded Computer Systems, University of California, 2003.

[16] 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. ACM Press, 2004.

[17] KAPOOR et al. An automatic testbench generation system. pages pp. 8–17, 1994.

[18] M. ;LAVAGNO L. ;REBAUDENGO M. LAJOLO. Automatic testbench generation for simulation-based validation.

[19] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[20] CORPORATE Inc. Institute of Electrical and Electronics Engineers. *IEEE Standard Description Language Based on the VERILOG Hardware Description Language, 1364-1995*. IEEE Standards Office, New York, NY, USA, 1996.

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

[22] K. R. G. da; MELCHER E. U. K. ; PESSOA I. M ; CUNHA H. N. SILVA. A methodology aimed at better integration of functional verification and rtl design. *Design Automation for Embedded Systems*, A Special SystemC Edition, 2006.

[23] Community. SYSTEMC. Disponível em: <http://www.systemc.org/> , acessado em: 05 mar. 2007.

[24] Andreas Vörg, Natividad Martínez Madrid, Wolfgang Rosenstiel, and Ralf Seepold. Ip qualification of reusable designs. Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany, 2001.

Apêndice A

Gramáticas usadas no eTBc

Nesta seção serão apresentadas as duas gramáticas livres de contexto usadas no eTBc. As gramáticas foram escritas na linguagem da ferramenta gals[4]. A primeira gramática é referente à linguagem eDL e a segunda é referente à linguagem eTL.

Código Fonte A.1: Gramática da Linguagem eDL

```
1  #Definicoes Regulares
2  D : [0-9]
3  L : [a-z A-Z _]
4  A : [\"]
5
6  //caracteres desprezados
7  WS : [\ \t\n\r]
8  //comantarios
9  COMMENT : ( "/*" [^ "*" ]* "*/" ) | ("/" ("/")  .* )
10
11 ID : {L} ( {L} | {D} )*
12
13 string : {A} ( . )* {A}
14 const : {D}+
15
16 #Tokens
17
18 : {WS}*
19 :! {COMMENT}
20
```

```
21 #palavras reservadas
22 module  = ID : "module"
23 fifo    = ID : "fifo"
24 struct  = ID : "struct"
25 signals = ID : "signals"
26 trans   = ID : "trans"
27 return  = ID : "return"
28 main    = ID : "main"
29 inv     = ID : "invert"
30
31 input   = ID : "input"
32 output  = ID : "output"
33 int     = ID : "int"
34 char    = ID : "char"
35
36 signed  = ID : "signed"
37 unsigned= ID : "unsigned"
38
39 short   = ID : "short"
40 long    = ID : "long"
41 float   = ID : "float"
42 double  = ID : "double"
43 bool    = ID : "bool"
44
45
46 #Gramatica
47 //declaracoe do programa principal
48 <program>       ::=     <declarations> #28 ;
49 <declarations>  ::=     <struct><modules>;
50
51 <struct>        ::=     struct  ID #1  "{" <trans><signals> "}" #19 <
       struct> | i;
52 <trans>         ::=     trans  "{" <tvardecl> "}";
53 <signals>       ::=     signals "{" <sgvardecl> "}";
54
55
56 //Declaracao de variaveis no escopo TRANS
```

```
57 <tvardecl>       ::=     <ttype> ID #3 <tlvar> | i ;
58 <tlvar>          ::=     "[" const #4 "]"  <tlvar>  | "," #20 ID #3 <tlvar
      > | ";" #20 <tvardecl> | i ;
59 <ttype>          ::=     int #2 | char #2 | double #2 | bool #2 | unsigned
       #2 <ttype1> #5 | signed #2 <ttype1> #5 | short #2 <ttype2> | long #2
      <ttype2> ;
60 <ttype1>         ::=     int #5 | char #5;
61 <ttype2>         ::=     int #6 | i ;
62
63
64 //Declaracao de variaveis no escopo SIGNALS
65 <sgvardecl>     ::=     <inv> <sgtype> "[" const #8 "]" ID #9 <sglvar> |
      <sgtype1> ID #9 <sglvar1> | i ;
66 <sglvar>        ::=     "," #21 ID #9 <sglvar> | ";" #21 <sgvardecl> | i;
67 <sglvar1>       ::=     ID <sglvar1> | "," ID #9 <sglvar1> | ";" #21 <
      sgvardecl>;
68 <sgtype>        ::=     signed #7 | unsigned #7 ;
69 <sgtype1>       ::=     bool #7;
70 <inv>           ::=     inv #29 | i;
71
72 //module
73 <modules>       ::=     module  ID #10  "{" <mdecls><fifos><links> "}"
      #27 <modules> | i;
74
75 //declaracao de variaveis no escopo de module
76 <mdecls>        ::=     <iotype> ID #12 ID #13 <mlvar> | i;
77 <mlvar>         ::=     "," ID #13  <mlvar> | ";" <mdecls> | i;
78
79 <fifos>         ::=     fifo ID #14 ID #15 <fifos1> | i;
80 <fifos1>        ::=     "," ID #15 <fifos1> | ";" <fifos> | i;
81
82 <links>         ::=     ID #16  ID #17  "(" "." ID #18 "(" ID #24 ")" ","
       "." ID #25 "(" ID #26 ")" ")" ";" #23<links> | i;
83
84 <iotype>        ::=    input #11 | output #22;
```

Agora a gramática da linguagem eTL :

Código Fonte A.2: Gramática da Linguagem eTL

```
1  #Definicoes Regulares
2  D : [0-9]
3  L : [a-z A-Z]
4  A : [\"]
5
6  SS: $$
7  SYMBOLS : [ %/,=<>;:!@#&\""\""|""*""+""?""("")""["]""{""}"".""^""-"]
8
9  keyword          : {SS} "(" (\ )* ( {L} | _  ) ( {L} | {D} | _ | "." )*
       (\ )* ")"
10
11 text             : $? ( {L} | {D} | _ |  {WS}* | {SYMBOLS}* )*
12
13
14 #Palavras Reservadas
15 file             =keyword        : "$$(file)"
16 endfile          =keyword        : "$$(endfile)"
17
18 foreach          =keyword        : "$$(foreach)"
19 endfor           =keyword        : "$$(endfor)"
20
21 if               =keyword        : "$$(if)"
22 endif            =keyword        : "$$(endif)"
23
24 //argumentos de controle de laco
25 i_var            =keyword        : "$$(i.var)"
26 j_var            =keyword        : "$$(j.var)"
27
28 i_signal         =keyword        : "$$(i.signal)"
29 j_signal         =keyword        : "$$(j.signal)"
30
31 module           =keyword        : "$$(module)"
32 module_in        =keyword        : "$$(module.in)"
33 module_out       =keyword        : "$$(module.out)"
34 struct           =keyword        : "$$(struct)"
35
```

```
36
37 //argumentos de controle de if
38
39 i_isnolast      =keyword        : "$$(i.isnotlast)"
40 j_isnolast      =keyword        : "$$(j.isnotlast)"
41
42
43 //argumentos de substituicao
44 i_name          =keyword        : "$$(i.name)"
45 j_name          =keyword        : "$$(j.name)"
46
47 i_type          =keyword        : "$$(i.type)"
48 j_type          =keyword        : "$$(j.type)"
49
50 i_size          =keyword        : "$$(i.size)"
51 j_size          =keyword        : "$$(j.size)"
52
53 module_name     =keyword        : "$$(module.name)"
54
55
56
57 #Gramatica
58 <program>       ::=     text #2 <cmds> | <cmds>;
59 <text>          ::=     text #2 <text> | i;
60 <cmds>          ::=     file    #22     <text> #1 <file_args>   |
61                         foreach         #23     <text> #1 <foreach_args>
                                    |
62                         if      #8      <text> #1 <if_args>     |
63                         i_name  #9      <text>  <cmds>  |
64                         j_name  #10     <text>  <cmds>  |
65                         i_type  #15     <text>  <cmds>  |
66                         j_type  #16     <text>  <cmds>  |
67                         structname #17  <text>  <cmds>  |
68                         module_name #19 <text>  <cmds>  |
69                         i;
70
```

```
71 <file_args>      ::=     keyword          #3      <text>  <cmds>  endfile
       #19 <end_program> |
72                          module_name #3  <text>  <cmds>  endfile #19 <
                                end_program>;
73
74 <foreach_args>  ::=     module  #29      <text>  <cmds>  endfor  #20 <
       end_program>|
75                          module_in  #4    <text>  <cmds>  endfor  #25 <
                                end_program>|
76                          module_out #5    <text>  <cmds>  endfor  #26 <
                                end_program>|
77                          struct  #30      <text>  <cmds>  endfor  #31 <
                                end_program>|
78                          i_signal         #6      <text>  <cmds>  endfor
                                #27 <end_program>|
79                          i_var   #24      <text>  <cmds>  endfor  #28 <
                                end_program>;
80
81
82
83 <if_args>        ::=     i_isnolast #7 <text> <cmds>      endif #21 <
       end_program>|
84                          j_isnolast #7 <text> <cmds>      endif #21 <
                                end_program>;
85
86 <end_program>    ::=     text #2 <cmds> | i ;
```

Apêndice B

A Methodology aimed at Better Integration of Functional Verification and RTL Design